Considering Rendering Algorithms for In Situ Visualization

Matthew Larsen
Department of Information and Computer Science, University of Oregon

Keywords—**Rendering, In Situ, Scientific Visualization**

Abstract—**A diverse set of goals drive rendering algorithm research. The gaming industry, hardware manufacturers, and the film industry all contribute to this research. The field of scientific visualization uses methods from both as well developing its own. However, a portion of these algorithms may no longer be applicable as we move to exascale. The I/O gap is forcing scientific visualization to in situ based systems, and the in situ environment, at scale, is one of limited resources (i.e., time and memory). As a result, design decisions for different rendering algorithm implementations that were appropriate today may not be for in situ systems. In this paper, we survey the current research in rendering, then present a qualitative analysis based on in situ exascale use cases.**

1 Introduction

Rendering research is driven by diverse goals. The rendering environments targeted by the gaming industry, hardware manufacturers, and the film industry all differ, but each contribute to rendering's body of research. Some algorithms use more memory to achieve higher performance, and other algorithms use more memory to renderer higher quality images. Scientific visualization uses a subset of rendering research, in addition to contributing its own, to explore and analyze data from physical simulations. Traditionally, using rendering algorithms from all sources has worked well for scientific visualization, but the computing environment will change as we move to exascale.

Scientific visualization has traditionally been a post-hoc process. Scientists ran simulations and saved the data to disk, where they would analyze the results. However, the growth in computational power enables simulations to produce data at rates far exceeding our ability to save it, forcing us to save data at ever sparser intervals. Coarse temporal resolution hampers a scientist's ability to analyze and explore the results of a costly simulation, and the gap between the amount of data simulations generate and our ability to save it is pushing the development of in situ visualization systems, which analyze data and render images while the simulations are running.

mlarsen@cs.uoregon.edu

The current set of production visualization systems (e.g., VisIt and ParaView) have been retrofitted for in situ, but they still use algorithms developed for post-hoc analysis. At exascale, memory will become a constrained resource as architectural trends continue to reduce the memory per core. Additionally, there is a tension between the time and frequency of visualization. Simulations have a fixed time budget, and performing visualization reduces the amount of time steps a simulation can execute. Since both memory and time will be limited resources in an exascale environment, algorithms designed for post-hoc analysis may no longer be appropriate. Consequently, we need to re-examine rendering in terms of exascale in situ; figure 1 illustrates the unknown subset of current algorithms that will be viable in an exascale environment.

In this paper, we examine several classes of rendering techniques in the context of limited resources. These classes are summarized in table I. If we are given a time and memory budget, then can we render one image? We currently cannot answer this question. As a first step, we survey current rendering research, and present a qualitative analysis of rendering methods by discussing each with regards to in situ use cases representative of a three axis spectrum defined by time, memory, and image quality.

Organization

In section 2, we provide background on topics necessary to discuss our central rendering questions: high-performance computing, in situ visualization, parallel programming models, and basic rendering methods.

Rendering algorithms loop over pixels (i.e., image order) or objects (i.e., object order). Within these algorithms, the techniques can be placed two categories: surface rendering and volume rendering. Section 3 covers images order algorithms and section 4 covers object order algorithms.

In section 5, we discuss distributed-memory parallel considerations for rendering. In section 6, we conclude

Name	Order	Rendering Type	Mesh Type	Section Described
Ray Tracing	Image	Surface	Unstructured	3.2.1
Ray Casting	Image	Volume	Structured/Unstructured	3.3, 3.4
Rasterization	Object	Surface	Unstructured	4.1.1
Sampling	Object	Volume	Structured/Unstructured	4.2, 4.3.2
Splatting	Object	Volume	Unstructured	4.3.1

TABLE I
The rendering techniques surveyed in this paper.

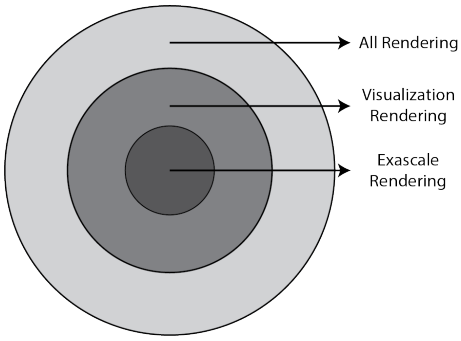

Fig. 1: The outer circle represents all rendering research, the middle circle represents the portion of rendering used by visualization, and the inner circle represents the rendering techniques that are viable at exascale. Which techniques belong in the inner circle is still an open question. With this survey, we are trying to inform which techniques belong in the inner circle, i.e., which are suitable for an environment where resources are limited.

by discussing the algorithms from section 3 and 4 in the context of three factors: render time, image quality, and memory usage.

2 Background: Rendering on Tomorrow's Supercomputers

The current HPC environment is moving to exascale, and the increase in computing power is exacerbating the I/O gap. Current in situ visualization solutions are working on todays machines, but the push to exascale may stress the current systems to the breaking point as memory and time become scarce resources. In this section, we cover the relevant background related to rendering on the next generation of supercomputers. Additionally, we define the two parallel programming models of the future machines, and finally, we discuss aspects of rendering that apply to all types of surface and volume rendering algorithms.

2.1 HPC Environment

Exascale

Todays supercomputers are capable of 10^{15} (.i.e, Petascale) floating point operations per second (FLOPS) and, at Exascale, computers will be capable of computation on the order of 10^{18} FLOPS. The diverse set of current computer architectures are a result of the explosive growth in computation power as we move from Petascale to Exascale computing, and the next step on the way to Exascale are machines like Summit (Oak Ridge National Laboratory) and Aurora (Argonne National Laboratory), scheduled to be deployed in 2018 at DOE leadership class computing facilities. These machines will have approximately an order of magnitude increase in FLOPS over the supercomputers we use today. However, I/O systems are not keeping pace with computational power, and, we cannot save all the data generated by simulations running on these machines.

I/O Bottleneck

In the past, the standard scientific work flow was to run a simulation and save out the data. The data would then be analyzed as a post-process with scientific visualization tools. Today, simulation code can run for 100,000s of time steps [1], and they generate more data than can possibly be saved. To mitigate the storage limitations, only sparse intervals of time steps are saved out to disk, making the temporal resolution coarse [2]. Simulation "hero" runs, which use all the resources of a supercomputer, are even more problematic. In fact, it would take 53 minutes to write out all available memory on Sequoia [3], a supercomputer at Lawrence Livermore National Laboratory.

As writable FLOPs continue to decline with every generation of supercomputer, the temporal resolution of

the time steps will continue to decrease. This is a major problem because scientists miss all of the information between time steps, which possibly limit the effectiveness of the simulations themselves. In situ visualization techniques are a solution to this problem. In this work flow, visualization runs along with the simulation code.

2.2 In Situ Visualization

By coupling simulation and visualization code, scientists have access to all time steps of the simulation by bypassing the I/O bottleneck. With in situ techniques, we can visualize data while the simulation is running, or we can perform analysis and save a subset of the original data. In situ analysis is not a new concept [4], but simulation designers were reluctant to use it until the I/O bottleneck became a serious consideration. Not only would the visualization code consume compute resources otherwise meant for the simulation, but the visualization would have to be integrated into the simulation, introducing more complexity into an already intricate system [1]. However, the ever growing gap between computation and I/O has drawn major attention to in situ visualization.

There are two types of in situ paradigms: tightly coupled and loosely coupled. Tightly coupled in situ runs concurrently with a simulation on the same resources. A simulation will stop at regular intervals within the main loop and turn over control of the program to the visualization infrastructure. At this point, the visualization algorithms have direct access to simulation memory, bypassing the I/O bottleneck.

Loosely coupled in situ visualization runs with the simulation code but on a different set of nodes. Data is sent over the network to the nodes where visualization processing takes place. While the speed at which data can be sent is limited by network bandwidth, supercomputer interconnects have throughput higher than that of the I/O systems.

Despite the choices, there is no one-size-fits-all solution to in situ visualization. The choice of how to perform visualization depends on the constraints of the simulation and the available resources. Further, there are many open questions on the best way to perform this task. In this section, we detail some important solutions to date. For a comprehensive survey of in situ, see the recent paper by Bauer[5] et al.

2.2.1 Tightly Coupled In Situ

Tightly coupled methods run on the same compute nodes as the simulation code, and they have direct access to simulation memory. While the visualization has full access to the simulation data, there are some drawbacks. First, the simulation must stop while the visualization is processed. Second, the visualization can consume a large amount of memory, and the compute node must have enough memory for both [6]. Finally, simulations often decompose data across compute resources in ways that are not ideal for visualization algorithms [1]. Despite these drawbacks, there are several successful approaches including Catalyst [7] and Libsim [3].

2.2.1.1 ParaView Catalyst

The ParaView Catalyst library, also known as the ParaView co-processing library, runs in conjunction with simulation [7]. Catalyst is a general visualization framework, based on ParaView, that is designed to work with any simulation. At each time step, the simulation code relinquishes control of the program to the library, where it checks to see if any visualization task needs to be performed. If there are no tasks, Catalyst will immediately return control of the program back to the simulation. Both a memory adapter and visualization pipeline must be created in order to couple the two systems.

The memory adapter bridges the gap between the simulation code data representation and the VTK data objects used by the visualization pipeline. It is possible to provide direct pointers to memory if the data representation is already in VTK format. Direct memory access greatly reduces the memory footprint, but this is rare since the simulation is optimized for its own algorithms. When the data layout is incompatible, the data must be copied into VTK format, increasing the memory footprint [7][2]. However, the computational cost of the copy outweighs the cost of saving the data to disk. With the memory adapter in place, the visualization pipeline proceeds.

With Catalyst, pipelines must be created in advance of the simulation. This requires domain knowledge of the simulation to decide what kind of operations need to be performed (e.g., isosurfacing, slicing, etc), values for those operations, visualization frequency, and the subregions of the data set to visualize [7]. The pipeline setup can be done in two different ways. First, the pipeline can be created programatically in the simulation code with all of the above mentioned knowledge. The other

way is to run the simulation for a brief period of time and save a small output file containing a representative snapshot of the data [2]. With this information, the user generates and exports an appropriate pipeline using the ParaView client.

There are many visualization filters that will scale along with the simulation, but there are some that will not scale. In this case, the Catalyst client-server infrastructure can send the data to a separate visualization cluster where it can be processed [7]. The trade-off is that the data must be sent over the network, but this cost is still less than the cost of I/O.

2.2.1.2 VisIt and Libsim

The Libsim library interfaces simulations with VisIt [8], providing a general visualization framework to the simulation code. Libsim offers two features that make it distinct from Catalyst. First, the Libsim code contains two separate components that enables demand driven visualization. That is to say, once the Libsim code has been integrated into the simulation code main loop, scientists can connect to the simulation with a VisIt client at any point or not at all. Second, visualization pipelines do not have to be known in advance [3].

Libsim contains two components: the "front-end-library" and the "runtime" library. The front-end-library is the main interface into the simulation, and it is compiled with the simulation. In this code, metadata objects are populated that tell Libsim what data the simulation contains. For example, the simulation registers meshes, variables, and other information that it wants to have available to visualize[6]. By creating data callbacks, Libsim loads only the data it needs for the visualization. Additionally, the front-end-library has a listener that waits for incoming connections from a VisIt client, and, when a client connects during execution, the second runtime library containing the VisIt server is loaded. By separating the two components, the Libsim has a minimal impact on the simulation code when the in situ capabilities are not used.

When the VisIt client connects to the server, the server sends the metadata to the client, exposing the variables and meshes it has access to, and, with this information, the user can define a visualization pipeline on-the-fly. Using the client, the pipeline is created and sent to the servers to be executed. As with Catalyst, Libsim will use the data directly if it is in a compatible layout or copy the data into VTK format when necessary [3]. Then, the results are finally sent back to the client.

2.2.1.3 Imaged-Based Approaches

Recently, imaged-based approaches have been developed as a data reduction strategy. The idea is that two-dimensional images require many orders of magnitudes less data than the entire simulation data [9]. Many images, which require far less memory than simulation data, are rendered and saved to disk. The results can then be analyzed after the simulation finishes.

One notable implementation is the work by Ahrens et al. [9] called Cinema. They believe that a petabyte is a reasonable amount of data to save from a simulation running at extreme scale, and with that space budget, they can produce a billion images. Each image contains RGB pixel data, a depth buffer, camera parameters, and other metadata describing the content of the image. The collection of all images are stored in a database that can be interacted with via a client.

As with Catalyst, the types of filters that create the images must be known in advance. In the user interface, a domain scientist defines what kinds of visualizations will take place, the range of parameters for those visualizations, at what time intervals the images are generated, and how many camera positions to render from. To make the scientists aware of the impact of their choices, the interface provides a rudimentary cost estimate (e.g., total data size and render time), so they can manage the trade offs. Next, the simulation runs and the image database is created.

Cinema allows the user to interactively explore the database. The viewer achieves simulated camera movement by loading a stream of images from the database with an interactive response rate of 12 FPS. The user can move through time steps or change isosurface values. Additionally, the user can query the metadata associated with the images to search for specific values or camera positions. The main contribution of the work is the ability to create new visualizations by compositing together multiple images from different filters.

2.2.2 Loosely Coupled In Situ

Loosely coupled in situ systems run on a subset of nodes while the simulation is running. Data is transfered over the network to the visualization nodes. While the content is limited to data sent over the network, visualization cannot crash the simulation and does not have

to share the memory space. Loosely coupled solutions generally act as middle-ware.

2.2.2.1 ADIOS

The Adaptable IO System (ADIOS) is an I/O middleware layer for simulations [10]. The API allows simulation code to perform I/O functions with a number of back-ends without being aware of the final representation. ADIOS uses metadata to describe the contents of the data, and the metadata allows an easy interface between simulation code and I/O systems.

One of the back-ends to ADIOS is a data staging area, known as ActiveSpaces, where the data can be processed [11]. The idea is to have a number of nodes that can hold the data where consumers of the data can further process it separately from the simulation code, also know as hybrid in situ [12]. Saying it another way, we must "bring the code to the data." On these dedicate I/O nodes, data can be reduced through visualization operations or data compression techniques. The staging areas have shown scalability up to hundreds of nodes, and each node can hold on the order of 20 time steps [13].

On the staging nodes, the ADIOS metadata contains descriptions of the data that are compatible with VTK data requirements. As such, both VisIt and ParaView have ADIOS file format readers that allow visualization to occur on the staging nodes [13][12]. The data can be visualized via traditional post-processing work flows.

2.2.2.2 ICARUS

ICARUS is a ParaView plug-in that uses the HDF5 I/O library to share data between the simulation and visualization [6]. The simulation sends data to HDF5 and then to Paraview, through the use of the Distributed Shared Memory (DSM) file driver. Basically, both applications communicate by accessing a shared virtual file. The simulation sends a message through the file to indicate that there is new data that can be visualized. When ICARUS reads this message, it processes the new data in the file.

Limited Resources

Simulations use problem sizes that consume almost all available memory on a machine. Sharing the resources of a supercomputer between simulation and visualization means memory is limited for both tightly coupled and loosely coupled in situ methods. In a tightly coupled use case, either the simulation must give up memory, which is unlikely, or visualization algorithms must execute with less memory than they accustomed to. In the loosely coupled use case, the visualization nodes must process the data from the entire simulation, pushing the memory limits. Thus, memory usage for rendering, along with all other algorithms, is a major concern in the in situ setting.

Time is also limited resource. Simulations must pause to turn over control of the node for visualization in the tightly coupled use case, and holding up the simulation limits advance of the simulation. In the loosely couple use case, visualization must be able to keep pace with the incoming data, otherwise, important information could be lost. If execution time is limited, rendering algorithms must ensure that they can execute within a given time allotment.

Image quality is also a point of tension between time and memory. Poor image quality hampers the communication and exploration value of the images. In some cases, the image quality of an algorithm has a relatively fixed value, and in other cases, we can produce higher quality images by using more time and memory. These tensions create a complex space full of trade-offs.

2.3 Parallel Execution Models

In this section, we examine the two main parallel execution models used by rendering algorithms on modern architectures: single instruction multiple threads (SIMT) and single instruction multiple data (SIMD). Both SIMT and SIMD perform the same instruction on multiple data elements, but they contain subtle differences that influence algorithm implementations targeting specific architectures.

2.3.1 SIMT

The SIMT model performs the same instruction on multiple data elements in different hardware threads. GPU threads are organized into groups, called warps or wavefronts, that operate in lockstep, and the typical sizes are 32 (NVIDIA) and 64 (AMD). For clarity, we will refer to these groups of threads only as *warps*. Each thread in a warp has access to a set of dedicated registers and instruction addresses [14]. Depending on the data a thread operates on, SIMT allows each thread to take different branching paths through the code. When branch divergence occurs within a warp, all threads execute the instructions of all branches and throw away the results

of the irrelevant paths. Saying it another way, branch divergence reduces efficiency.

SIMT architectures oversubscribe processors with warps to hide the latency of memory requests. When threads request memory not already in the cache, the thread's warp must wait until the memory arrives before continuing execution, and, by having many warps assigned to a processor, the scheduler can swap in a warp that is ready to continue executing, keeping the processor busy. Additionally, SIMT hardware coalesces memory accesses. If all threads in a warp load or store contiguous aligned memory, then the hardware can fulfill the operation in a minimal number of transactions, otherwise, the instruction must be replayed until all threads are serviced.

2.3.2 SIMD

The SIMD model operates by performing the same instruction on multiple data elements in vector registers. Each thread of execution loads n data elements from memory into the vector registers of a single processing core. The processor simultaneously performs the same instruction on all data elements, until the calculation is complete, and the path each data elements takes through the vector registers is called a *vector lane*. Unlike SIMT where each thread has dedicated registers, each vector lane contains no state other than the current value in the vector register.

The vector widths of modern processors have been steadily increasing, and current CPUs have vector widths of 2, 4, and 8, depending on the instruction set architecture (ISA). The most common ISAs are Streaming SIMD Extensions (SSE) and Advance Vector Extensions (AVX), and there are multiple version of both SSE and AVX (e.g., SSE2, SSE4, and AVX2) which support different vector widths. While not all problems map well to the SIMD programming model, most rendering algorithms are embarrassingly parallel and can take advantage of the vector units.

There are two main ways to take advantage of the vector units. First, compilers using aggressive optimizations will attempt to auto-vectorize loops whenever possible, but the programmer must be aware that pointer aliasing, memory alignment, and branching can prevent the compiler from vectorizing the code. Second, programmers can explicitly vectorize code though compiler intrinsics, which allow low-level access to vector instructions. Intrinsics, however, are inherently non-portable, and algorithms require separate implementations for each ISA.

Recently, ISPC, an open source compiler supported by Intel, was released to make vectorization more accessible to programmers [15]. ISPC vectorizes code containing branching by masking off vector lanes, essentially implementing SIMT in software on top of the SIMD hardware. Using ISPC, each vector lane behaves as a thread. Additionally, ISPC allows programs to be compiled for all modern ISAs. While divergent code suffers from the same penalties as in SIMT (i.e., throwing away results from the vector lanes not in the branch), ISCP allows a wider range of problems to be mapped to vector hardware, and a number of rendering frameworks are using ISCP to maximize performance.

2.4 Rendering Methods

Rendering methods fall into two categories: surface rendering and volume rendering. Here we give an overview of the different methods to provide the foundations for a more detailed discussion in sections 3 and 4.

2.4.1 Surface Rendering

In scientific visualization, surfaces are created from scalar fields through analytical operations. For example, an isosurface creates continuous surface where the field evaluates to a single value. There are two main algorithms for rendering images of surfaces: rasterization and ray tracing.

2.4.2 Volume Rendering

Volume rendering algorithms visualize participating media (e.g., fog and haze). Light attenuates as it travels through participating media, dimming to differing degrees depending on the material absorption properties. In scientific visualization, volume rendering is used to visualize scalar fields such as temperature, density, or pressure. Since scalar fields have no inherent color, a mapping called a *transfer function* determines a specific color for each scalar value, and colors are composited together into a final pixel value. Figure 2 is an example of a volume rendering of a density scalar field from a cosmological simulation.

Levoy [16] created the first CPU ray caster for volume rendering, but it was far from interactive. Until recently, interactive volume rendering was limited to GPU

implementations that perform ray casting or exploit rasterization hardware, but modern CPU architectures with wide SIMD vector widths opened the door for interactive CPU volume rendering. Now, algorithms exist for both CPUs and GPUs, operating on structured and unstructured data.

Both rasterization and ray tracing methods are used to render volumetric data. With rasterization, transparency is supported, but objects must be sorted in a view dependent order, since compositing must take place in depth order. A ray tracing variant called ray casting is well suited for volume rendering, and rays are cast from the camera into the volume. As the ray marches through the volume, ray casting samples the scalar field at regular intervals in front-to-back order. Additionally, there are sample based approaches that fill buffers with samples, then use rasterization or implicit ray casting methods for compositing.

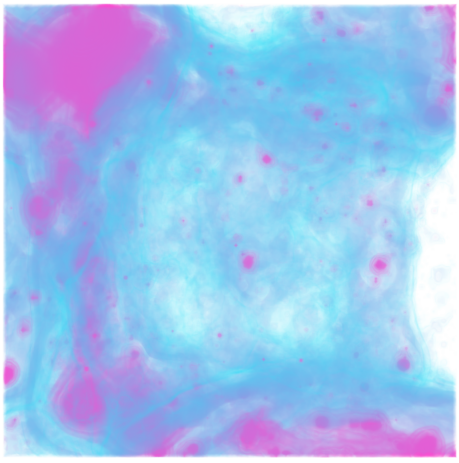

Fig. 2: A volume rendering of a cosmological data set from the Enzo simulation code.

Volume rendering is used to visualize both structured (e.g., regular grids) and unstructured meshes (e.g., tetrahedral meshes), and current volume rendering algorithms fall into one of these two categories. Structured volume renderers exploit the regular nature of the grid to sample the volume, and each cell is easily located in memory. Conversely for unstructured data, there is no implicit location in memory for any given cell, and either acceleration structures like BVHs or cell-to-cell connectivity information must be used to locate sampling points. In this survey, we will examine both structured and unstructured volume rendering algorithms. There are several concepts that apply to all types of volume rendering: lighting and pre-integration.

2.4.2.1 Lighting

For better image quality, volume rendering performs shading calculations at each sample. Traditional surface shading models use the normal of the surface, camera direction, and light direction to adjust the color of the sample. However, there is no surface in a volume, so the scalar gradient is used as the normal in shading calculations. The gradient can either be pre-computed or calculated on-the-fly. If sampled at every point, a pre-computed gradient takes up three times the space of the original scalar field. Alternatively, calculating the gradient on-the-fly can be expensive and causes redundant memory loads.

For the CPU based implementations, caching methods exist for more efficient on-the-fly gradient calculations. On the GPU, implementations load stencils into shared memory since multiple rays access the same memory locations more than once. Shared memory lowers the time needed for threads to access shared stencil locations and reduces register usage [17], with the greatest benefit coming from larger stencil sizes.

2.4.2.2 Pre-Integration Tables

Sampling at regular intervals can generate visual artifacts. These artifacts look like ridges on an elevation map, where the ridge lines identify areas of the same elevation. This can be mitigated by changing the transfer function, but this might not be possible since they are designed to identify specific features in the data.

Max [18] et al. created pre-integration tables to minimize this issue and create high quality volume visualizations. The idea is to create a table that contains the pre-computed integral between all possible sample points based on the transfer function [19]. More recently Lum [20] et al. showed a more compact representation and included pre-calculated lighting tables as well. Pre-integration tables create a trade-off between higher quality images and memory usage for all volume rendering variants.

3 Image Order Techniques

Image order techniques loop over pixels in the image. With image order algorithms, each pixel can be thought of a ray starting that starts from the camera and extends into the scene through the image plane at a some *(i,j)* location. Any object in the scene that intersects with

the ray can contribute to the color of the pixel. In this section, we cover image order algorithms.

3.1 Bounding Volume Hierarchies

Image order techniques involve a search problem. A renderer must find a cell (e.g., surfaces) or list of cells (e.g, volumes) that contribute to the color of each pixel. In scientific visualization, data sets consist of millions to billions of cells, and iterating over all cells in the data set is far too costly. Thus, renders use acceleration structures to make the search cost reasonable. Consequently, the performance of image order rendering techniques is tightly coupled to acceleration structures, since total frame time is the sum of the acceleration structure construction and the time to render.

Octrees [21], kd-trees [22], and BVHs are all used as acceleration structures for interactive rendering In the past, octrees and kd-tress had been favored because they can be constructed quickly, and while the time to locate cells in a BVH is lower, BVH construction time outweighed the benefits. However, recent research has significantly decreased BVH construction time, and today BVHs are the preferred acceleration structure since they are fast, flexible, and memory efficient.

A BHVs is a object partitioning of geometric primitives, and typical BHVs have a branching factor of 2, although some BVHs have branching factors of 4, 8, and 16. In the tree, the inner nodes contain the *axis-aligned bounding boxes* (AABBs) of each child along with pointers to each of its children Leaf nodes contain a variable number of primitives, usually in the form of pointers to vertex data. The AABB in the tree root contains the bounds of the entire data set. To traverse a BVH, rays are intersected with the child nodes AABB stored by each parent. If the ray intersects more than one child's AABB, then the closest subtree is traversed first, even though the closest primitive intersection may actual be contained in the another subtree.

BVH's are an object partitioning, and much research has been performed to determine the best way to partition these objects. The surface area heuristic (SAH) [24] [25] was first introduced by Goldsmith and Salmon, and SAH is a means for evaluating the best object partitioning during construction. The SAH says that for a given set of objects and some split point, the cost of that node is relative to the surface area of the AABBs and the number of object for each side of the split. Figure 4 shows the equation for the SAH. The total cost of

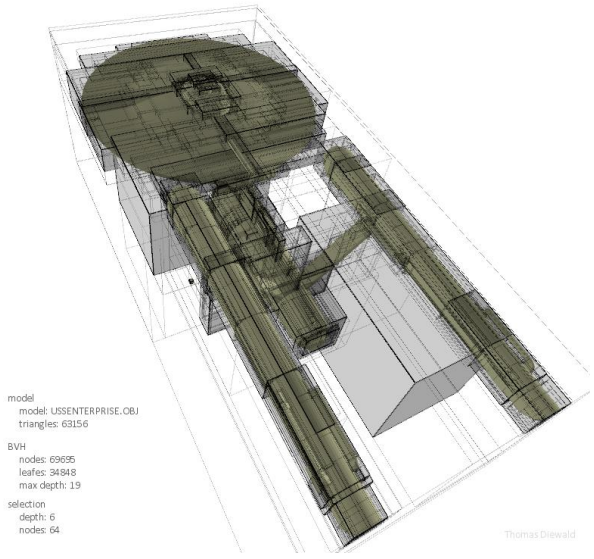

Fig. 3: This image shows the AABBs of all BVH nodes over a triangle mesh [23].

$$C_t + C_i * (\frac{SA(L)}{SA(L+R)} * N_l + \frac{SA(R)}{SA(L+R)} * N_r)$$

Fig. 4: The equation for the SAH where $SA(L)$ and $SA(R)$ are the surface area of the left and right nodes, and N_l and N_r are the number of objects in the left and right nodes. C_i and C_t are implementation specific constants that represent the cost of intersection and the cost traversal.

the tree is the sum of the cost of all nodes. Trees with a lower cost are traversed faster than those with a higher cost. Thus, it is not possible to know the true cost of a tree until all other splits are made, but the heuristic provides a effective guide for making split decisions. In general, considering a larger set of possible partitions results in a lower cost tree, but ultimately, there is a speed versus quality trade off for BVH construction.

BVHs types can be categorized by their construction methods:

- Top-Down
- Bottom-Up
- Hybrid

In this section, we consider these various construction methods and their applicability to in situ rendering. When we refer to the **quality** of a particular BVH type, we are making an approximate assertion about how ef-

ficiently the tree can be traversed. The absolute quality of a BVH depends on the objects in the scene.

3.1.1 Top-Down

3.1.1.1 Median

A median BVH builds a tree from the top-down by splitting each level of the hierarchy at some median [26][27]. At each level, all the primitives are sorted along some axis and split according to the middle element or the spacial median. Leaves are created when the number of elements in each subtree reaches some threshold. Median builders can create trees quickly because of their simplicity, but the low quality of the trees makes tracing a large number of rays through them undesirable.

3.1.1.2 Full Sweep

SAH builders find spacial subdivisions based on the SAH cost function, and they are the best known methods for high quality BVHs. The cost function creates higher quality trees at the cost of longer build times. The algorithm uses a greedy top-down approach to build the BVH by evaluating the cost of dividing primitives into two groups. A full sweep SAH builder sorts all primitives according to one coordinate (e.g., x, y, or z) of the object's AABB centroid, then the buikder sweeps from one side to the other, considering every split point by calculating the SAH of the current partition. The process is repeated for all three coordinates axes, and the lowest cost split is performed. Splits are performed recursively, until a specific threshold is reached.

3.1.1.3 Binner

SAH binners use the same concept as full sweep builder but consider far less potential split points. The number of possible splits is large, and to reduce the number of splits considered, SAH binning builders randomly consider split planes or choose split planes at regular intervals (i.e., binning). This reduces build time dramatically, and the trees still maintain good quality. Popov et al. [28], noted that sampling the split positions at regular intervals on each axis, in their case 1024 samples per axix, provided a "very good approximation of the best spit position," and each every primitive falls into one of the spatial bins. While Popov et al. used this method for k-d tree construction, the method is applicable to BVHs as well. They also noted that adaptive sampling could be used to create more bins in places where the SAH cost is low, but in practice, SAH binners discretized the space at regular intervals, often using as little as 16 or 32 bins [29][30]. SAH binners produce trees close to the quality of a full sweep builder and with lower build times. Consequently, SAH binners are an active area of research with variants for all modern architectures.

Wald [29] adapted the binning construction algorithm to take advantage of task-based parallelism. He observed that there were not a sufficient number of tasks at the top level of the hierarchy to take full advantage of the available parallelism, so Wald proposed a system that used all threads to process large tasks at the top of the tree. When the number of tasks matched the amount of available parallelism, they switched to one task per thread. Using this approach, Wald eliminated the lack of parallelism at the top of the tree.

On the GPU, Lauterbach et al. [31] used a similar tasked-based system where large and small tasks are treated differently. For large tasks, they use k bins and use $3k$ threads to calculate the SAH of all the bins. Smaller tasks use a single warp to process the each subtree, and all data for each subtree is loaded into shared memory to enable fast access to the data and minimize memory bandwidth.

Later, Wald [32] adapted the SAH binned builder for an early MIC architecture which ran 128 concurrent threads and had 512-bit wide vector lanes. For SIMD friendly memory alignment, the builder transformed each primitive into an AABB and packed it into a hybrid *structure-of-arrays* format. That is, an array-of-structures (AoS) in which each element represents groups of 16 AABBs are stored in a *structure-of-arrays* (SoA) format. Further, most data structures were in a 16-wide (i.e., 16 32-bit floating point values fit into a 512-bit vector registers) SoA packets to match the vector register width, including the number of bins. In this format, the builder loaded all 16 AABBs into the SIMD registers and binned all 16 AABBs in parallel. Similarly, SAH evaluation of the bins were computed in SIMD registers.

The Embree ray tracing framework includes a number of SAH binners for both the CPU and the Xeon Phi [33], and they achieve impressive build times. Rates of millions to hundreds of millions of triangles per second are achieved on desktop CPUs and Xeon Phis respectively. The Xeon Phi builders are similar to [32] where the number of bins corresponds to the width of

the SIMD vector registers, however the MIC architecture has evolved since the work in 2012. At the top levels of the hierarchy, both the CPU and Xeon Phi use all threads to build each level of the tree until the task size reaches a threshold value. Once the threshold is reached, the Xeon Phi builder maps each medium-sized task to four threads per core, and once tasks become small enough, tasks map to a single thread. On the CPU, medium-sized and small-sized tasks map to a single thread.

Building on SIMD parallelism, Wald implemented a general tasking system with priority, dependency, and sub-tasking features. Sub-tasking allowed for a task to contain multiple jobs that could execute concurrently, and using sub-tasking, larger jobs at the top of the hierarchy were broken up in many smaller parts to fill the MIC with work. To remove the overhead of context switching from the task system, the worker pool mapped only one thread per core, so there were a total of 128 threads drawing from the task queue. The algorithm for SAH binning was highly tuned to the specific architecture to maximize device utilization.

3.1.1.4 Spatial Splits

A split BVH considers object splitting to decrease the total cost of the BVH. Large objects can produce significant overlap in the AABBs of neighboring bins, and overlap leads to additional computation since both subtrees are potentially traversed to find the nearest object. To minimize the overlap, split BVHs create two references to the same object by splitting the object's AABB along the plane that separates two neighboring bins. In terms of the SAH formula in figure 4, the number of objects in each partition are increased while the surface area of the two partitions are possibly decreased, which would lead to a lower SAH cost (i.e., higher quality tree).

The splitting references to object was first proposed by Ernst and Greiner [34]. In there work, they defined the technique of *Early Split Clipping* (ESC) in which AABBs were subdivided before the BVH is constructed, until the surface area was below a user defined threshold. In there study, they found that their methods led to two a three fold increase in performance over traditional methods in scenes with large triangles. Similar to Ernst, Dammertz [35] et al. proposed an algorithm for subdividing the objects themselves in order to reduce the empty space in the AABB.

Stich [36] et al. created a hybrid method using both reference splitting and object partitioning called the Split BVH (SBVH). The first phase of their method performs a full SAH sweep of object partitioning, then performs a chopped spacial binning. During the binning phase, references that straddle two spacial bins are chopped at the spacial boundary, creating two references in each bin. Finally, the best split is performed based on the lowest SAH found from either phase. Stich et al. also created a mechanism for controlling the number of spacial splits by only considering splits where the ratio of the surface area of the overlap between the two partitions and that of total surface area of the scene's AABB was beyond a user-defined constant. This constant assured that only meaningful splits are considered. The final representation can consume more memory, but it produces a high quality tree.

Embree [33] uses a similar technique, but only considers one spacial split at the object center of each axis, reducing both quality and build time. The Embree builders also use fine-grained SAH binning at the top levels of the hierarchy, then coarsen the binning grain at the lower levels. Additionally, Embree enforces a limit to the number of spacial splits that can be preformed by creating a fixed splitting budget, and once the budget is exhausted, the builder no longer considers spacial splits. This altered method uses the best of both techniques and is both fast and creates high quality trees, while not increasing memory usage beyond a pre-configured bound.

3.1.1.5 Linear BVH

The linear bounding volume hierarchy (LBVH) is a fast parallel construction method. The algorithm was introduced by Lauterbach et al. [31] in 2009, and the construction method demonstrates impressive build times on the GPU. The LBVH works by sorting the primitives by their centroid along a space filling Morton curve into a linear list. Morton codes work by interleaving the bits of the centroid coordinates into a singe value, and the length of the shared prefix bits between two codes determine how close the two primitives are to each other along the space filling curve. The key observation was that the Morton codes contain all of the spacial subdivision information needed to construct the BVH. Since no SAH is used to determine the spacial splits, the resulting BVH quality is less that optimal, but the loss of quality is offset by fast build times.

Each Morton code can be generated in parallel inde-

pendently, exposing significant parallelism. After the calculation of the Morton codes, the keys are sorted using a parallel radix sort which is capable of sorting 1 billions keys per second on modern GPUs, 100s of millions keys per second on the CPU, and 800 million keys per second on the Xeon Phi (MIC) [33] [37]. Between two Morton codes, the first differing bit determines the existence of a split, and the LBVH builder proceeds to sort all bits of the morton code in parallel. The method proceeds from the first bit (i.e., the root node) where all codes with bits of zero are on one side of the split and one bits on the other side.

On the CPU and Xeon Phi, the Embree ray tracing framework also uses fast BVH builders leveraging Morton codes. All threads cooperate to generate Morton codes in parallel. Then using a task based system, all threads cooperate to build levels of the tree, until a sufficient number of tasks exist for individual threads to process the tasks on their own [33]. Finally, subtrees terminate at leaves of four primitives each.

3.1.1.6 Hierachical Linear BVH (HLBVH)

The Hierachical Linear BVH was first introduced by Lauterbach [31] et al. by combining the strengths of both the LBVH and SAH binning algorithms. They used the LBVH method to construct the first several levels of the tree, since the top-down SAH binning algorithm lacked sufficient parallelism upper levels of the hierarchy. After the top levels of the hierarchy are created, the SAH binning algorithm is used to create the lower levels. Garanzha [38] simplified the hierarchy generation and implemented a GPU builder that used work queues. During the LBVH phase of the algorithm, they mapped individual threads to tasks, and, instead of looping through all primitives in the task to find the split plane, they used a simple binary search. Other versions of the HLBVH use SAH binning at the top levels since more branches can be ruled out at the top of the tree.

3.1.2 Bottom-Up Builders

3.1.2.1 Fully Parallel LBVH (FP-LBVH)

Expanding on the original LBVH, Karras [39] observed that LBVH could be viewed as a binary radix tree. In a binary radix tree, each shared prefix is represented by an inner node and each Morton code is a leaf node. Additionally, for n leaf nodes, there are always $n-1$ inner nodes [39]. Karras found that by representing the leaf nodes and inner nodes in two separate arrays, an implicit hierarchal structure exists. Basically, the entire hierarchy is generated in a single data-parallel step.

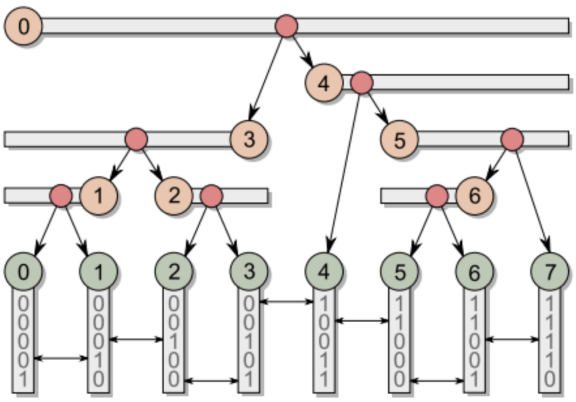

Fig. 5: An example node layout containing common prefix ranges for each inner node in a binary radix tree [39].

For example, each inner node at index i corresponds to a shared prefix beginning, or ending, at a leaf node with the same index i, and each inner node can find the index of its child pointers by performing a binary search on the Morton codes in the leaves. Figure 5 demonstrates an example of the node layout structure. Since all inner nodes can be calculated at the same time, the amount of parallelism exposed for n primitives is $n-1$. The final step is to calculate the AABBs using a bottom up traversal, where the amount of available parallelism is limited by the number of nodes on the current level of the tree.

Apetrei [40] recently improved on the Karras method by combining the hierarchy generation and bottom-up AABB propagation. Instead of looking for the length of the largest common prefix in neighboring Morton codes, they look for the index of the first differing bit which represents the distance between two Morton codes. This observation simplifies the hierarchy generation logic and produces an equivalent tree in a reduced amount of time.

3.1.3 Hybrid Builders

3.1.3.1 Treelet Restructuring BVH (TRBVH)

The treelet restructuring BVH adds several post processing steps after constructing an initial LBVH. The idea is to build a fully parallel LBVH quickly, then, us-

ing the SAH, optimize the hierarchy in place by restructuring small neighborhoods of nodes. By performing several optimization passes in succession, the quality of the tree can be improved up to 90-95% of the gold standard, and the BVH can be constructed at interactive rates [41]. However, the optimization takes a significant amount of computational power and the only published implementation is on the GPU.

After the LBVH is built, the TRBVH traverses the tree from the bottom up using atomic counters. The first worker to reach a node increments the node counter, then the worker exits. By traversing the tree in this fashion, each worker is assured that there are no other workers currently executing below them in the tree, and the worker can freely modify the treelet rooted at the current node. To limit the GPU utilization bottleneck in the top levels of the tree, Karras and Aila use an entire warp as the basic working unit.

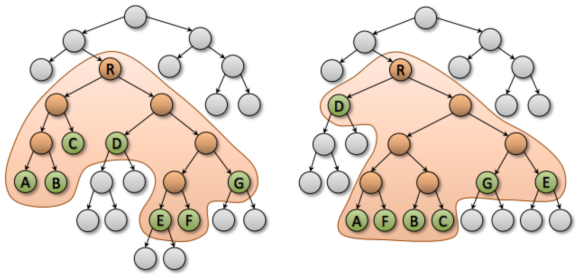

Fig. 6: Left: the initial treelet with seven leaf nodes. Right: the SAH optimized restructured treelet [41].

Each warp processes a treelet to find the optimal organization. Starting from an inner node in the tree, the warp expands the treelet by adding the leaf node with the largest surface area, until the treelet contains 7 leafs nodes. Figure 6 shows an example treelet formed by the expansion. Next, Karras and Alia use dynamic programming to find the optimal SAH configuration of nodes by solving all the subproblems contributing to the solution. Each thread in the warp solves a specific subproblem using a precomputed schedule. To keep memory usage down and GPU occupancy at full capacity, some values are stored as bitmasks and compiler intrinsics are used to share registers between threads. After several passes of treelet restructuring completes, a final pass merges each primitive in the leaf nodes into four primitives per leaf. To our knowledge, there only exists GPU implementations of the TRBVH, since solving the optimal treelet problem is computationally expensive.

3.1.3.2 Approximate Agglomerative Clustering (AAC)

Approximate agglomerative clustering is a method introduced by Gu [42] et al. that refines an initial LBVH structure. ACC uses the LBVH method to generate a set of local clusters at the bottom of the hierarchy. Then, AAC rebuilds the tree by clustering nodes at the bottom and proceeds up to the root node. Each cluster performs a constrained greedy search to find the best cluster to merge with, where the merged cluster represents the parent node. When two suitable clusters merge, the algorithm uses the SAH to determine if flattening the current subtree into a leaf would improve tree quality. ACC moves from the bottom-up and terminates at the root node. The resulting tree is constructed quickly and has better quality than the original LBVH.

3.1.4 Alternative Construction Methods

There are several alternatives to building a complete BVH for every time step. In this section, we briefly consider the two alternative methods.

3.1.4.1 BVH Restructuring

In animation or simulations that deform a mesh over time, the BVH is typically rebuilt each time step. If the coordinates of the primitives change over time, then refitting the BVH can be quickly calculated by updating the node's AABB, but doing so steadily decreases the quality of the BVH with each refit. One approach is to use a heuristic to determine when to rebuild the BVH once the quality has dropped below a threshold. Another approach is to restructure the BVH to maintain the quality.

Some early work on BVH restructuring was done by Larsson [43] et al. and Teschner [44] et al. in the context of collision detection, another use for BVHs. However, these methods focused on restructuring the entire BVH or large subtrees, but restructuring can be computationally expensive for large scenes and not cost effective for ray tracing. Then, Larsson [45] et al. applied the methods to ray tracing with an emphasis on making the algorithm as fast as possible.

Lauterbach [46] et al. considers the effects of a dynamic scene by comparing the surface area of a node to the surface area of its children. If the ratio becomes too high (i.e., the distance between the two children becomes too high), then the subtree is restructured. Alternatively, Yoon [47] used two metrics to selectively

identify a small set of subtrees that needed to be restructured. The first metric is based on the SAH, and the second method uses a probabilistic model to evaluate the possible gains from restructuring the subtree.

Garanzha [48] et al. use several methods to find the cheapest possible action that has the largest performance increase. They look at three possible actions: refit the subtree, relocate it somewhere else in the tree, or rebuild the subtree using a SAH binning approach. Their algorithm uses a set of heuristics to determine which is the most beneficial. Another approach was proposed by Kopta [49] et al. that uses tree rotations to perform local optimizations that incrementally improves the quality of the BVH. This approach is similar to the TRBVH builder except that tree quality has been degraded due to deformation of the mesh and the number of potential rotations is much smaller than the TRBVH.

3.1.4.2 Lazy Building

The idea of a lazy BVH is to build the acceleration structure on-demand. Initially with no BVH built, a ray is tested against the AABB of the scene, and if it misses, then nothing else is computed. On the other hand, if the ray does hit the AABB, then the first two nodes are constructed at that moment. By building the hierarchy on-demand, it is possible that the tree contains many unfinished subtrees. No specific build type is associated with a lazy BVH, and it can be built by using any method.

The idea was first proposed by Hook et al. [50] by annotating the nodes of the tree to indicate their state. Nodes could be marked as an inner node, leaf node, or as a split-table node, and when the a split-table node is encountered, the node is further partitioned. Watcher [51] et al. used a similar scheme for a bounding interval hierarchy (BIH), which is a simple tree split at planes and does not contain AABBs. In their scheme, a node simply contains a flag indicating it is unfinished.

More recently, Vinkler [52] et al. demonstrated a GPU ray tracer system that utilizes a lazy BVH builder. In their system, a task-based system is used to process both ray tracing and BVH building simultaneously. When a ray encounters an unfinished portion of the BVH, the kernel adds a build task to the queue and suspends itself, and when the dependent task finishes, it relaunches the ray. Additionally, Vinkler uses speculative construction of nodes in the tree to keep the GPU filled with work– thus, a node may already be constructed when visited for the first time.

3.2 Surface Rendering

3.2.1 Ray Tracing

Ray tracing simulates light transport by tracing rays from the camera into the scene. Rays intersect with geometry and reflect in new directions, and the fundamental task in a ray tracer is finding ray triangle intersections. Rendering shadows, depth of field, and refraction maps more naturally to ray tracing than rasterization. While achieving these effects are possible with rasterization, they often require multiple rendering passes and add complexity to the rendering algorithm. Impressive degrees of realism (see figure 7) can be achieved through ray tracing variants such as path tracing, which uses Monte Carlo methods to evaluate the outgoing and incoming radiance at every intersection point. Realism makes ray tracing the preferred rendering method of the film industry, but the high degree of realism comes at the expense of real-time rendering.

In general, one or more rays are created for each pixel in the screen, and they have an initial direction defined by the camera parameters. Once created, the rays interact with the geometry they hit by accumulating the color along the path of the ray. If a ray misses all geometry in the scene, then it terminates. Otherwise, the ray can continue bouncing off other surfaces until it misses or some other termination criteria is met. With each change of direction, a ray tracer finds the nearest intersection point by checking the ray origin and direction against all scene geometry. The idea behind ray tracing is conceptually simple, but computationally intensive.

For years, ray tracing has been extensively used in the movie industry, and more recently ray tracing has made headway into the field of scientific visualization. Thus, finding efficient ray tracing techniques is an active area of research in both industry and academia.

Using acceleration structures, intersection tests are only performed on a small number of primitives relative to the total number in the scene. The most performant ray tracers use BVHs (see section 3.1) to reduce the number of ray-triangle intersection tests. By using acceleration structures, ray tracers are bound by the number of rays per frame (i.e., image order), unlike rasterization which is bound by the number of objects (i.e., object order).

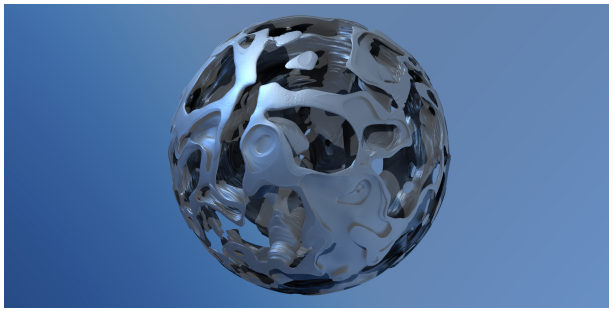

Fig. 7: An image created using path tracing.

Traversal and Intersection

Ray tracers perform two main calculations: traversal and intersection. Most ray tracing kernels are structured as a loop that alternates between testing the AABBs of the BVH nodes and intersecting primitives referenced in the leaf nodes. Of the two types of calculations, the latter dominates.

Each inner node of the BVH contains the AABBs of all of its children, and the ray is tested against each one of the AABBs. If the ray misses an AABB, that subtree is not traversed. Otherwise, nodes are traversed from the nearest to the furthest. Once a leaf node is encountered, the kernel performs intersection tests with all primitives the leaf contains.

All rays are not created equal. *Primary rays* proceed from the camera through each pixel in the image plane, and since primary rays have similar origins and directions, tracing primary rays exhibits coherent memory access patterns as they proceed through the scene. If an image is rendered using only primary rays then the resulting image would be equivalent to a rasterized image.

Secondary rays add more realistic effects to the image and have different memory access patterns. After the primary rays reflect off surfaces, they no longer have similar directions and exhibit incoherent memory accesses during traversal. Diffuse reflections are essentially random and have the worst memory access patterns. Another example of a secondary ray is a shadow ray, where a ray is cast towards the light source for every ray intersection.

There are several different algorithmic variants that take into account the memory access patterns of primary rays and secondary rays. Single ray traversal is best at handling secondary rays, and packet traversal is best for tracing primary rays. Additionally, hybrid variants attempt to use the strengths of both traversal methods.

3.2.2 Single Ray Traversal

Single ray traversal traces a single ray through the BVH. Each ray moves though the BVH independently by performing a stack-based traversal. To trace rays in parallel, each thread traces a single ray until an intersection is found, but achieving maximum performance is difficult. Between CPUs, GPUs, and Xeon Phis there exist subtle hardware differences that force architecture specific implementations to vary.

In the ray tracing world, there are two clear performance leaders on their respective platforms. On the GPU, the leader is NVIDIA's Optix Prime [53], and on the CPU and MIC, the leader is Intel's Embree [33] [54]. Both frameworks contain ray tracing kernels that are highly optimized for their target architectures.

GPU Variants

The OptiX ray tracing framework provides a robust infrastructure that enables programmers to build full rendering engines around it. Additionally, OptiX provides access to a low-level interface, OptiX Prime, which is specifically geared towards performance. Although the source code is not public, the kernels are based on public research by Alia and Laine [55] [56]. Alia and Laine were the first to publish thorough results examining the efficiency of ray tracing on GPUs. They observed that existing methods were around 2x slower than the theoretical upper-bound on performance, and that the limiting factor was work distribution not memory bandwidth. That is to say, tracing rays is a heterogeneous workload, and a single long running ray will make the entire warp unavailable for reassignment until that one ray has completed.

To combat the problem of long running threads, a global task pool was also implemented to further increase traversal efficiency. The idea is to launch enough threads to fill the GPU with work, then replace terminated rays from a global work pool. The highest benefits are received by replacing rays when the number of active rays drops below 60% [56]. Persistent threads further increases the performance of secondary rays.

To maximize the warp efficiency (i.e., reduce divergence), Aila and Laine implemented *speculative traversal* [55]. Threads in a warp coordinate their actions so that threads are all processing inner nodes or all processing leaf nodes by using a compiler intrinsic to poll a

boolean value in each warp. If some threads are ready to proceed to the leaf nodes but the others are not, then they keep traversing the tree accumulating more leaf nodes to process once the warp votes to do so. By increasing the SIMT efficiency, this method boosts performance for incoherent rays. Gupta [57] et al. showed that using work queues on the GPU can be beneficial to other types of workloads.

On the GPU, there are a finite number of registers available to each thread, and if kernels take up too many registers, then fewer warps can execute simultaneously. The OptiX Prime kernels uses a ray-triangle intersection test specifically to keep register usage to a minimum. The Woop unit triangle intersection test [58] stores an affine transform which converts a triangle into the unit triangle. Instead of loading the vertices of the triangle, OptiX loads the transform and converts the ray into unit triangle coordinate space. This method takes up more space but offers better performance.

CPU/MIC Variants

Until recently, many perceived the CPU as an off-line rendering target [59], but in the last several years, real-time ray tracers and volume renderers are emerging for CPUs and MICs. Embree adapted the traversal algorithm for vector instruction sets, yielding an impressive amount of performance. By creating an intrinsics framework, Embree compiles for SSE2, SSE4, AVX, AVX-2, and AVX-512. The broad ISA support enables Embree to use SIMD widths of 4, 8, and 16, allowing Embree to execute on desktop workstations and Xeon Phi co-processors.

Unlike the OptiX, Embree uses BVH branching factors higher than 2. Embree creates branching factors of 4, 8, and 16 to match the SIMD vector register width. When processing an inner node, the AABBs of all children are loaded into the vector registers and tested in parallel. By storing node data in packets of AoS format, Embree uses vector load instructions to fetch node data efficiently while minimizing cache misses. Since the BVH is created and used exclusively by the ray tracer, customizing the memory layout does not increase the memory overhead.

Embree includes two ray-triangle intersection methods [60][61] for different use cases, one that is watertight (i.e., rays will not miss when hitting the border between two triangles) and one that is not. Both triangle intersection variants are vectorized. Similar to the BVH implementations, the triangle intersectors simultaneously test a number of triangles at once. For the Xeon Phi, only four triangles [62] are tested at once, and up to eight at once on the CPU.

3.2.3 Packet Traversal

Packet traversal is a traversal variant that uses packets of rays instead of a single ray. When an inner node is fetched, the AABBs are tested against all rays at once in the SIMD registers, moving the parallelism from multiple AABBs per ray to multiple rays per AABB. Instead of maintaining one stack per ray, a single stack is used for all rays in the packet. Packet traversal enables a significant performance enhancement for primary rays, which tend to have similar paths through the BVH, but the same is not true for incoherent rays. Wald[63] et al. first proposed packet tracing as a way to increase performance. In Wald's system, one thread traced four rays at once using SSE vector instructions, and he observed that the overhead of packet tracing was small for primary rays. Additionally, the memory bandwidth was significantly reduced.

On the GPU, Garanzha [64] et al. created a system that uses packet tracing for both primary and secondary rays. They use a frustum to represent the collective directions with the packet instead of testing individual rays against AABBs. After the rays are reflected, they create a four-dimensional hash (i.e., the rays origin and direction) for each ray, then a radix sort groups rays together into packets. This method was shown to be effective for primary rays and area light sampling, but their results did not include any tests using global illumination workloads. NVIDIAs production ray tracer has likely not adopted this approach since it adds additional code complexity and is unproven for the types of rendering workloads they target.

As previously mentioned, CPU and MIC packet tracing methods map rays to vector registers, and the size of packets are determined by the width of the vector registers. Frustum based methods can map more rays per thread, but they tend to suffer performance hits for incoherent secondary rays [65][66]. Embree includes packet tracing kernels all common vector withs on modern architectures [33].

Hybrid Traversal

Hybrid ray traversal uses both single ray and packet traversal to get the benefits of both. Bethin [62] et at.

successfully demonstrated a hybrid method on the Xeon Phi. Since then, Embree adopted the same approach that switches between packet traversal and single ray traversal once efficiency drops below a specific threshold, giving Embree a throughput 50% higher than the other methods alone [33]. While we are not aware of any GPU systems using this approach, it seems likely that the GPU would also benefit.

3.3 Structured Volume Rendering (Ray Casting)

Structured grids are either regular or rectilinear. Regular grids have equal spacing of field data on each axis, and rectilinear grids can have variable spacing on each axis. Due to the inherent structure of the data in memory, volume rendering algorithms can easily find the sample points by location alone, significantly simplifying the algorithms. Ray casting is used by state-of-the-art CPU and GPU renderers.

The method is straight-forward. Cast a ray from the camera, and sample the scalar field at regular intervals. For each interpolated sample value, look up the associated scalar value in a color map, and composite the sample colors from front-to-back order. The ray terminates when the next sample point is outside the mesh, the number of samples per ray is met, or the color reaches full opacity (i.e., early ray termination).

Structured ray casting is a well studied area. Since the technique is straight-forward, most work centers around methods to increase memory efficiency, and this leads to trade-offs between quality, time, and memory. The majority of these techniques are applicable to both CPU and GPU variants, but generally GPU ray casters are more sensitive to memory usage issues.

CPU Variants

Most CPU volume rendering focuses on using SIMD vector registers, and this can be characterized in the works by Peterka [67] and Knoll [68]. Knoll et al. created a ray casting volume renderer, inspired by the lessons of SIMD ray tracing, which used SSE intrinsics. The authors are able to highly optimize all of the code through clever uses of SSE intrinsics. Interpolation of a sample points is optimized by using SSE register swizzling to re-use values, resulting in a 15% performance increase. Further, SSE-specific optimizations increase performance on almost every operation including gradient, shading, and compositing. Knoll's resulting code is compact, under 200 lines of code, but almost completely written in SSE intrinsics, and thus requires expertise to understand, not to mention extend.

Even on a modest 8-core Intel CPU, the implementation reaches interactive frame rates (i.e., grater than 10 frames per second) on large data sets (e.g, 7 GB) and could achieve even better performance on machines with higher core counts such as those found on supercomputers. Additionally, the CPU-based approach does not suffer the memory limitations of GPUs, where data sets that exceed physical memory would have to be processed by an out-of-core renderer.

GPU Variants

Ray casting in CUDA is straight-forward, and additionally, the CUDA API exposes hardware features that are not available in the OpenGL pipeline, namely shared memory. CUDA volume renderers have three main kernels: ray generation, intersection, and ray marching. Ray generation outputs origins and directions based on the camera parameters, and it orders rays along a space-filling curve to improve cache usage [69]. Then, the rays are intersected with the volume's bounding box, finding the entry and exit points, and finally, the ray marching kernel samples and composites. The scalar field is stored in a texture to use the spacial caching hardware with the dedicated texture units.

GPU ray casting has access to GPU hardware that provides some advantages over CPU variants. The first is shared memory, which allows blocks to efficiently share (e.g., gradient calculations) data across threads within a block. The second is texture memory. The entire scalar field can be loaded into a texture, which is optimized for three-dimensional spatial access and includes access to hardware interpolation.

Wang et al. [70] introduced a ray casting method that moves away from the one thread per ray parallelization strategy. Instead, they use the combined power of a warp to process a single ray to provide an algorithm that uses a cache-aware sampling strategy called "warp marching." Each warp gathers the next 32 samples along the ray to maximize data locality. To maintain high warp efficiency, warp polling is used to determine if all 32 samples are empty space (i.e., transfer function opacity is zero), skipping the rest of the calculations. Otherwise a warp-wide reduction is calculated to find the composited color value of all the samples. Instead of using shared memory for intra-warp communication, register shuffling intrinsics are used to share memory between

warp members. By leveraging the hardware specific techniques not available in OpenGL, Wang et at. created a memory efficient ray casting variant that exhibits constant texture cache hit rates.

3.3.1 Acceleration Structures

Transfer functions impact rendering performance because they are designed to highlight specific features in the data. For example, doctors may only be interested in viewing parts of a CT scan with a specific range of densities, so the transfer function may only have a small range of values that map to any color. In this case, the majority of samples are wasted since they do not contribute to the final image. To minimize wasted sampling, ray casters use acceleration structures to ensure that samples contribute to the final image.

Octrees [71], kd trees [72], and more recently, BVHs [68], have been used to classify the scalar ranges of spacial regions. These acceleration structures, called min-max trees, store the minimum and maximum scalar ranges for each subtree. Combined with the transfer function, a sample point can easily be skipped if there is no contribution. Using an acceleration structure costs more memory (e.g., 10% [71]), but can be a performance boost if the transfer function is sparse.

3.4 Unstructured Volume Rendering (Ray Casting)

Unstructured data consists of cells that are polyhedral (e.g., tetrahedrons and hexahedrons). Volume rendering of unstructured data involves additional calculations since the cell containing a sample point cannot be implicitly found from the sample coordinates. For unstructured data, every sample point must be located which makes it more complicated than ray casting structured data. There are two main methods for image order unstructured volume rendering: connectivity-based and point location-based methods.

3.4.1 Connectivity

Compact data set representations store cells in two different arrays. There is an array of all the points in the data set and an array of cell indices. Adjacent cells share faces, and shared faces contain the same indices into the coordinate arrays. Thus, many unstructured volume renderers use mesh connectivity to traverse through the cells. The face-to-face connectivity must be calculated and stored, but this is a one time cost.

Garrity [73] first introduced this method in 1990, and he used an acceleration structure to keep track mesh entry. The connectivity method has been incrementally improved over the years, including GPU adaptations [74]. Additionally, compacting the connectivity has been a more recent area of research [75][76].

3.4.1.1 Location

Location-based methods maintain an acceleration structure to locate sample points along a ray. As with many algorithms that require spatial data structures, there are several options. The two most popular choices are BVHs and kd trees. BVHs require less memory, but they do not guarantee that nodes will be traversed in an absolute front to back order.

Rathke et al. [77] recently introduced a fast CPU unstructured volume renderer based on the Embree ray tracing framework [33]. The ray caster renders general polyhedral meshes by providing Embree shape intersectors for each supported type. Additionally, they replaced Embree's BVH builder with a custom min-max BVH [68], where each node contains the range of scalar values in the sub-tree. The main contribution of the work is providing a SIMD travseral method using the Intel ISPC compiler [15]. The ISPC traveral method uses packet traversal since all the rays are cast in a uniform manner from the camera. Since a BVH is used, they buffer out-of-order samples until there are enough to composite together in vector registers.

4 Object Order Techniques

Object order algorithms loop over objects in the scene. For each object, an algorithm finds the pixels that it contributes to by sampling the object, and object order algorithms must keep track of visibility ordering to produce the correct image.

4.1 Surface Rendering

4.1.1 Rasterization

Rasterization is the process of converting the vertices of geometric primitives, most commonly triangles, into a two-dimensional array of pixels. Rasterization can be performed on either a GPU or a CPU [78]. In modern computer graphics, the most prevalent rasterization method is the scanline algorithm. The scanline algorithm operates on triangle vertices that have been transformed into a coordinate system defined by the camera

position, view direction, and up direction. The triangle is sampled at pixel locations, each of which is given a color and a depth value. Rasterization maintains both the image buffer and the z-buffer. The z-buffer contains the depth of the nearest pixel to the camera, and each incoming pixel must be compared with the depth value in the z-buffer to determine visibility. That is, we either throw away the pixel or update the z-buffer and image color.

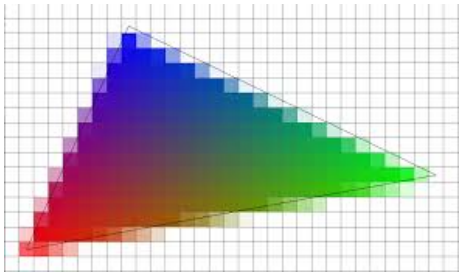

Fig. 8: An example of the scanline algorithm rasterizing a traingle into pixels [79].

While rasterization could be performed on the CPU, GPUs were developed to perform rasterization in hardware using a fixed function pipeline. The OpenGL API was created in 1992 as an interface to vendor specific hardware implementations, and in 2001, hardware vendors began exposing parts of the pipeline with an API The full featured API allowed programmers to build algorithms using rasterization hardware. There are several different algorithms that perform rasterization, and we briefly cover them here.

4.1.1.1 Scan Line

The scan line algorithm was the first rasterization algorithm created in the 1960s [80]. The idea is to transform each primitive into screen space and calculate the pixels that they cover. For a triangle, the scan line algorithm finds the y-intercepts of each row of pixels, and it interpolates the vertex colors and depth values. Then, the algorithm interpolates in the x direction for each covered pixel. This method is efficient for triangles that cover a large amount of pixels.

4.1.1.2 Barycentric

The barycentric algorithms calculate the barycentric coordinate system of an object, which is most often a triangle. Then, each pixel in the screen space AABB is tested to see if it is inside the triangle by simply checking if the sum of the barycentric coordinates is greater than or equal to zero and less than or equal to one. With this method, interpolation is virtually free since the barycentric coordinates represent the distance from each of the three vertices. One drawback to this approach is that it tests many samples that are outside large and degenerate triangles.

Pineda [81] created a parallel rasterization algorithm based on this method. Since the barycentric coordinates change linearly in each direction, it is trivial to parallelize it. CPU implementations can use integer only representations to speed up calculations.

The typical rasterizer is implemented as a pipeline. The first phase transforms the vertices into screen space and removes triangles that are outside the camera view. Next, the triangles are binned into tiles that segment the screen. After that, each thread serially rasterizes tiles. The resulting fragments are shaded to produce the final image.

OS Mesa [78] is an example of a software based rasterizer. Currently, it is being modernized to use threading and SIMD registers to rasterize multiple triangles at once. Intel is leading the effort with OpenSWR [82], which can render at interactive rates. Additionally, CUDA-base rasterizers have been created to explore the algorithm on programmable GPUs [83].

4.2 Structured Volume Rendering

The only object order structured volume rendering technique we are aware of was recently published by Schroots [84] and Ma. Schroots adapted Larsen [**?**] et. al's unstructured volume rendering methods for structured data.

Conceptually, a large buffer is created. The size of the buffer is the image's dimensions multiplied by the number of samples. With a large buffer, each object is processes and the contributing samples are deposited in the buffer. To reduce the memory usage, the buffer is broken up into several passes which process the volume from front-to-back.

4.3 Unstructured Volume Rendering

4.3.1 Splatting

Projected tetrahedra (PT) is a clever method of volume rendering unstructured tetrahedral meshes. The idea of PT is to decompose the tetrahedron into a number of semi-transparent triangles and rasterize, or splat, them onto the screen, but the triangles must be rendered

in a view dependent ordering. Triangle decomposition takes place inside a programmable stage of the graphics hardware pipeline called a geometry shader, and, depending on the view direction, the shader classifies each tetrahedron into one of several cases (see figure 9 for an example). Geometry shaders take a single input, in this case tetrahedons, and can output an arbitrary number of shapes into the parallel pipeline. Early implementations of PT sorted tetrahedrons by their centroid to determine visibility ordering, however, this was an approximation and led to inaccurate images. Two modifications to PT — hardware-assisted visibility (HAVS) [85] and later hardware-assisted projected tetrahedra (HAPT) [86] — implement versions of PT that partially and fully corrected the errors produced by the sorting order.

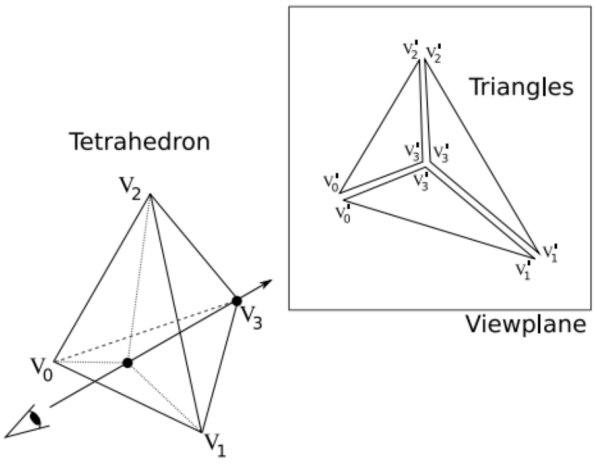

Fig. 9: An example projected tetrahedra triangle decomposition case [85].

HAVS sorts tetrahedrons by centroid but partially corrects the ordering on-the-fly in a hardware buffer exposed by OpenGL. Sorting is performed by an initial rendering pass that uses OpenGL draw calls, and the second pass renders the triangles to the screen. As triangles are rasterized, pixel fragments are deposited into the *k-buffer* which has a configurable size. As fragments are inserted into the buffer, the partial ordering is corrected based on the fragment depth values before the hardware compositing stage. Using the k-buffer improved fragment ordering but did not completely correct it.

HAPT uses both OpenGL and CUDA to order the cells correctly at the cost of greater sorting time. The HAPT implementation includes faster sorting methods that do not completely correct the visibility ordering. By using CUDA, HAPT renders the volume in a single pass, leading to increased frame rates. The *meshed polyhedra visibility ordering* [87] method produces the correct ordering for rasterization but requires data structures that cannot be implemented in OpenGL.

4.3.2 Sample-Based

Sample-based volume rendering creates a large sample buffer, and by using a buffer, it trades a larger memory footprint for the ability to process cells in an arbitrary order, eliminating the need for any pre-process. Further, the algorithm maps naturally to both structured and unstructured data sets, since the buffer frees it from the constraints of other volume rending methods. The execution consists of several phases. First, tetrahedra vertex coordinates are transformed into screen space, a coordinate system defined by the camera position and view direction. In screen space, each integer coordinate corresponds to a location in the sample buffer. Then, the renderer iterates, in parallel, over each tetrahedron and deposits the contributing samples the buffer. Finally, samples in the buffer are composited to produce the final image.

Using a large buffer creates a large memory footprint. For example, a 1024^2 image with a 1000 samples per pixel will result in a billion samples (i.e., 4GB of memory). However, when used in a distributed memory environment, the algorithm scales well since the buffer is spread out across a number of nodes and cells can be sampled independently.

Childs et al. [88] introduced the sample-based volume renderer which was targeted for CPUs, the dominate supercomputing architecture in 2006. The algorithm used the Message Passing Interface (MPI) library to implement the renderer in VisIt, a popular open source visualization tool. Since 2006, supercomputing architecture has become increasingly varied, and current architectures use CPUs, GPUs, and MICs for computation.

Larsen et al. [89] extended the algorithm to use the available node-level parallelism. They implemented a variant of the sampling volume rendering algorithm in EAVL [90], a library providing abstract data-parallelism over both CPU and GPU architectures. The new algorithm breaks the buffer up into multiple passes along the viewing direction, then samples sections of the buffer at a time. This method not only reduces the memory footprint of the buffer, but also supports early ray termi-

nation which can drastically lower the number of samples per frame. Using EAVL, the algorithm uses both OpenMP and CUDA for thread-level parallelism on a compute node. Further, the inner sampling loop uses auto-vectorization to further speed up execution on the CPU.

5 Parallel Considerations

The research discussed in sections 3 and 4 focuses on single node performance. In this section, we shift to discuss distribute-memory considerations for rendering

Image Compositing

In distributed-memory rendering, each node has a subset of the data but renders an image at full resolution, and each pixel contains depth value. For surface rendering, image compositing finds the smallest depth value for each pixel, For volume rendering, values for each pixel are sorted from front-to-back and composited together. Scalable image compositing is an important part of distributed-memory rendering.

The direct send [91] was one of the first algorithm created for image compositing. Direct send assigns a set of pixels to one node, and all other nodes sends their pixels to the assigned location. Binary swap [92] created a compositing algorithm that repeatably pairs up nodes and exchanges pixels. After the rounds are complete, each node has a complete subset of the image, then they stitch the image together. Binary swap requires less communication than direct send and is more scalable, but binary swap requires that the number of nodes participating in the exchange be a power of two. Yu [93] et al. generalized binary swap to use any number of processors, called binary 2-3 swap. More recently, radix-k [94] combined the best of both strategies.

All of the previous research was packaged into a single solution called IceT [95]. IceT provides implementations of direct send, binay swap, and radix-k, along with a number of optimizations. This solution is scalable to hundreds of thousands of cores and is in wide use within the visualization community.

Ray Tracing

Ray tracing with primary rays is equivalent to rasterization. If the shading model only considers the intersection point, then each node can render the scene, and, with each nodes data, compositing constructs the final image. Ray tracing offers photo-realistic image quality, but, render photo-realistic rendering requires that a ray be allowed to bounce around the entire scene. The movie industry handles this by distributing the whole scene to each node. However, the entire mesh of a physics simulation cannot fit in the memory of a single node, so we have to send the rays to the data.

Navratil [96][97] et al. described a dynamic ray scheduling system to ray trace data in a distributed-memory setting. As a ray travels through the data set, the system sends rays via MPI to the nodes containing the data for the ray's current location. Sending rays to the data eliminates the need for image compositing strategies. This work has grown into larger system called GraviT, which is an open source effort led by the Texas Advanced Computing Center. This research is in its infancy and is a open area of research.

Volume Rendering

Childs [98] et al. conducted a series of massive scale experiments to exposed the bottlenecks of current visualization systems. Their main goal was to identify the main problems that were likely to occur when concurrency and data sizes increase on the coming generations of supercomputers. Volume rendering was an area that they identified as a bottleneck, using current algorithms, for simulations at scale. The visualization system they tested used the traditional model of parallelism of one MPI task per core.

Howison [69][99][100] explored hybrid parallelism for structured volume rendering (ray casting) on both CPU and GPU supercomputers. They found the hybrid parallelism reduces both the communication and memory overhead that lead to better scalability and faster run times. Unstructured volume rendering is especially challenging since the run times are higher and produce more data for the parallel compositing phase. As the machines get larger and more powerful, we will push the limits of current algorithms, and this will continue to be an active area of research.

6 Conclusion

In this section, we discuss the current space of rendering algorithms presented in the previous sections, highlighting their trade-offs. To motivate the discussion about rendering and in situ, we will define three categories of use cases by its most important consideration:

- Time
- Image Quality

• Memory Usage

Each of these categories represents the "best" end of a three axis spectrum shown in figure 10.

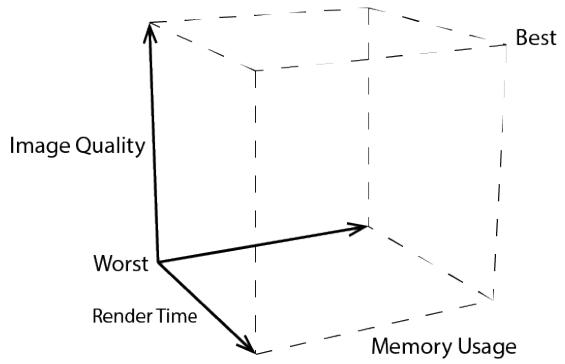

Fig. 10: The three axis spectrum representing the tension between render time, image quality, and memory usage.

Time

There are two main use cases in this grouping. First, image based in situ solutions, like Cinema [9], need to render a large number of images to create an explorable database, so minimizing render time is the most important factor. Creating more images means that the database will be more explorable. Second, this category represents a case where simulation can only spare a small amount of time to perform visualization. In this case, the in situ visualization system must be able to complete rendering within the given time budget.

Image Quality

In this category, image quality is the most important aspect. A visualization of a data set that contains complex features needs to be high quality, so that domain scientists can extract all of the information contained in the visualization. Low quality images are difficult to interpret and hamper understanding. Another example of this use case is when the visualization is meant to communicate the results of the simulation to a broad audience.

Memory Usage

Some simulations use almost all available memory. In these cases, in situ rendering needs to operate on a strict memory budget, so it is important that these rendering algorithms can work within this budget.

6.1 Table Metrics

The tables in this section represent the summary of metrics (i.e., time, quality, and memory) for each category of rendering methods. Each metric has three possible values: W (worst), A (average), and B (Best). If there is a range of values (e.g., A-B), this indicates the presence of a trade-off with one or more of the other metrics. For example if the metric is quality, *A-B (T)* means that we can achieve higher quality images at the cost of render time.

In sections 3 and 4, the top-level organization was image order and object order. In this section, the top-level ordering is grouped by surface rendering and volume rendering. For example, if we are making a choice between methods for surface rendering, then we will choose ray tracing or rasterization.

6.2 Bounding Volume Hierarchies

BVHs are the acceleration structure of choice for image order rendering. We separate the discussion on acceleration structures to simplify the rendering comparisons since there is a significant amount of research on BVHs. In terms of the use cases, build time corresponds to renderer time, and BVH quality corresponds to both image quality and memory usage.

For in situ, there are many choices for BVHs, each with benefits and drawbacks. In table II, we have summarized the relative build times and qualities for all of the BVH types covered in this paper, and these metrics are based on comparisons presented in the surveyed papers. In terms of build time and quality, figure 11 shows the spectrum of where the respective BVHs fall. While actual values may vary based on implementation and architecture, they provide a guide for selecting the best method for a given use case.

The time use case needs a BVH on the higher end of the quality spectrum. The number of rays traced for volume rendering or ray tracing would be high, and efficient traversal would trump the extra cost of the build time. Alternatively, one high quality image may needed, and with image order techniques, higher quality translates to more queries into the acceleration structure.

On the other end of the spectrum for build time, the images could be used to monitor a simulation for correctness and render must complete with a time limit. Since we only need a single image, we would want fast rendering and low build times. The linear BVH or the

Metrics	Median	Sweep	Binner	SBVH	LBVH	HLBVH	FP-BVH	TRBVH	ACC
Build Time	Best	Worst	Ave	Worst	Best	Ave	Best	Best	Best
Quality	Worst	Best	Best	Best	Worst	Best	Worst	Best	Ave
Time-Quality Trade-off	No	No	Yes	Yes	No	Yes	No	Yes	Yes

TABLE II
Summary of BVH build times and qualities. Some BVH types have the ability to adjust the build time at the cost of quality which is indicated in the fourth row.

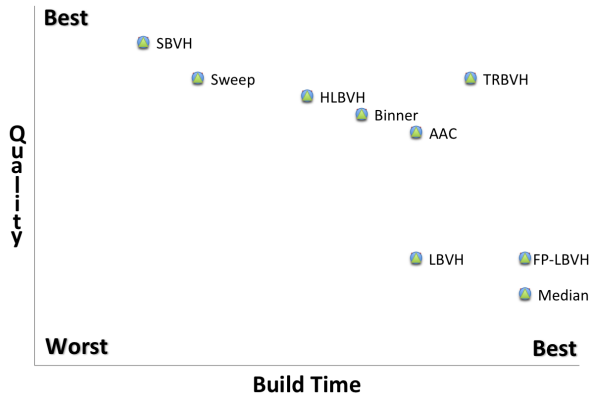

Fig. 11: A table of the relative metrics for the BVH types reviewed. The TRBVH has an astrix since the only implementation we are aware of is on the GPU.

Metrics	Time	Quality	Memory
Ray Tracing (AS)	W-B (Q)	W-B (T)	A
Rasterization	B	W	B

TABLE III
Surface rendering metrics. Image order algorithms are on top and object orders algorithms are on the bottom. (AS) denotes the use of an acceleration structure.

fully parallel BVH are good candidates.

The amount of memory the BVHs take up is also an issue. Top-down builders tend to be higher quality, and they produce shallower hierarchies that take up less memory. With a top-down builder, they have more choices on primitives groupings in leaves, and a linear BVH builder has fewer opportunities to make grouping choices during construction. To make the representation more compact would take additional time. For example, Embree reports that the SAH builders take an order of magnitude more memory than the Morton-based builders. Thus, memory budget would also play a large roll in selecting the right BVH for the in situ use case. Lazy building may mitigate some of the memory issues, but this adds code complexity because lazy building requires increased coupling between builders and renderers.

There are a spectrum of uses cases for in situ visualization that fall in between all three use cases. To have the all the types available in a renderer would involve a significant amount of effort and could add additional complexity to a visualization system. Some of the builders have tunable knobs that can change its characteristics for a given situation, so having only a few tunable builder may be desirable. For example, we could reduce the number of bins for an SAH binner to increase build time. Alternatively, using more bins takes longer, but produces a higher quality tree.

All of these issues are important to an in situ visualization systems, and there is no research into exploring this space. Currently, there are many missing memory statistics from the research results presented in papers. For example, we may know how much memory the final tree consumes but not the peak memory usage during construction, making difficult to evaluate for the memory use case. If we run out of memory, then the whole system will fail.

6.3 Rendering Discussion

6.3.1 Surface Rendering

Table III shows the metrics for our in situ use cases. For surface rendering, there are only two choices: ray tracing and rasterization.

6.3.1.1 Image Order

Rendering time for ray tracing has made significant improvements over the last several years, making it a real-time rendering option. If we assume an exascale machine will have a Xeon Phi architecture comparable

to the current architecture, then we can render at rates of 200-400 million rays per second on each node. For primary rays only, 200 million rays per second means that we could render 200 images at a resolution of 1024^2 or around 100 images at a 1920x1080 (i.e., 1080P).

For GPU ray tracing, the rendering speed is even higher at around 400-600 million rays per second on GPUs typically found on supercomputers. It is thought that the next generations of machines will have multiple GPUs per node, which would only increase the number of images per second or decrease the time to create a single image.

The image quality of ray tracing can is highly tunable between render time and image quality. On one end of the spectrum, primary rays with shading is equivalent to rasterization with the same shading model, and this would likely be the choice for fast render times. On the other end, the image quality use case would need a smaller number of images and allow longer render times. Then, ray tracing can start to add effects such as soft shadows or ambient occlusion. Global illumination algorithms like path tracing can be used for physically based effects if there is an large time budget. Ultimately, there is a tension between the image quality and render time, and these are decisions domain scientist will make on a case-by-case basis.

Considering the memory usage use case, most of the memory usage in a ray tracer is bound to the number of pixels in the image. Each ray minimally uses 56 bytes of memory to store the direction, origin, and index of the primitive that was hit, but several more fields are common. However, we have concerns about the memory usage of ray tracers like Embree and OptiX in an in situ setting.

OptiX uses the unit triangle intersection test, which stores an affine transform maps the ray to the unit triangle. This simplifies the intersection code and increases performance, but doubles the memory overhead of the geometry. With GPU memory already small compared to the size of main memory on a node, OptiX is unlikely to be an acceptable option for the memory use case.

Embree has more options to choose form that impact memory usage. To maximize memory throughput in a SIMD setting, Embree can re-organize the data layout, which doubles memory usage. But for reduced performance, Embree can use the original data layout.

Ray tracers produced by Intel and NVIDIA target the movie and animation industries as a means to convince studios to invest in their hardware. Render farms distribute data very differently than simulations do in a distributed memory setting. In render farms, each node has a copy of the entire scene and is responsible for rendering a small part of the image, but in large scale simulations, the data is too large to fit into the memory of a single node and is distributed across the entire supercomputer. Further, Embree and OptiX cannot currently render data sets directly, and translation between data layouts must occur. Creating a performant ray tracer that uses minimal memory should be goal for the in situ visualization community.

Object Order

Rasterization has been the traditional rendering algorithm of visualization, and there is not any active research other than optimizing rasterization for specific architectures. Rendering times are the fastest of all methods, which makes it attractive for the render time use case, but ray tracing is gaining ground. As more compute power is packed into nodes of a supercomputer, simulations are increasing the numbers of cells per node. More objects means more overhead for processing objects that do not contribute to the final image.

For the image quality use case, quality for rasterization is low and hard to improve. Creating physically accurate effects is complicated but possible. The video game industry trades additional effects at the cost of render time, but these effects are not used in scientific visualization. Thus, there no trade-off for image quality in terms of visualization.

Memory usage is implementation dependent, but it is generally low, supporting memory use case. If all vertices of the data set are transformed into screen space, this doubles the memory footprint of a rasterizer. However, rasterizers are implemented as a pipeline, so memory usage is tied to the width of the pipeline which is relatively low.

6.3.2 Structured Volume Rendering

For structure volume rendering, we have several choices summarized in table IV. Image order algorithms are located above the bold horizontal line. All of the structured volume rendering methods have a complicated set of trade-offs effecting multiple metrics. For structured ray casting, we have the option of using an acceleration structure, which can be beneficial in certain cases. Ray casting methods have trade-offs between

Metrics	Time	Quality	Memory
Ray Casting (AS)	W-A (Q)	A-B (MT)	A-B (Q)
Sampling	W-A (MQ)	A-B (MT)	W-B (QT)

TABLE IV
Structured volume rendering algorithm comparison. Image order algorithms are on top and object orders algorithms are on the bottom. (AS) denotes the use of an acceleration structure.

Metrics	Time	Quality	Memory
Ray Casting (AS)	W-A (Q)	A-B (T)	A
Splatting	B	A	A
Sampling	W-A (MQ)	A-B (MT)	W-B (QT)

TABLE V
Unstructured Volume Rendering algorithm comparison. Image order algorithms are on top and object orders algorithms are on the bottom. (AS) denotes the use of an acceleration structure.

render time versus quality and quality versus memory usage.

For the sample-based, the object order algorithm has three sets of trade-offs that effect all metrics.

Image Order

Structured volume rendering is high quality and fast since the data layout is regular and sample location is not necessary, which supports the render time use case. Faster rendering times are also possible when the transfer function is sparse by using an acceleration structure to skip sampling areas of the volume that do not contribute to the final image. As with all ray casting methods, there is a speed versus quality trade-off. We could increase the sampling rate to render higher quality images but at the cost increased rendering time.

Additionally, there is a tension between quality and memory usage. Pre-integration tables reduce the number of visual artifacts, and increasing the resolution of the integration table increases the quality of the image but uses more memory.

Structured ray casting has three points of tension between render time, image quality, and memory usage, but on the whole, memory usage is much lower than image order unstructured volume rendering and object order techniques. In an in situ setting, the memory overhead is well suited for environments where memory is limited, and since the points of tension are easily adjusted, the algorithmic settings can be explored for a particular in situ use case without much effort.

Object Order

The object order sampling algorithm offers a complex set of trade-offs. Thus, it is flexible for our uses cases, but difficult to assess for the entire spectrum of use cases that fall in between the three use cases.

Render time is slightly higher than image order algorithms. Since every object must be sampled, there is overhead for loading each object, but sampling values inside a cell is fast. Render time is also impacted by the number of samples per cell, which corresponds to image quality. Additionally, memory is reduced by processing chunks of the sample buffer at a time, but the overhead of each chunk increases render time.

Sample based rendering has the same image quality range as the image order methods, since they all sample the volume at regular intervals. Like the other methods, more samples raises the image quality but increases render time. We can increase quality even further using pre-integration, negatively impacting the two other metrics.

Memory usage is where sampling based methods are is most flexible because of the three-way trade-off. On a single node, a conceptually large sample buffer can be broken up to match a memory budget, which is more difficult for the image order methods to achieve. This is not saying that creating versions of other volume rendering algorithms that adhere to a memory budget is impossible, but it is just naturally part of the sample based algorithm. In a distributed memory setting, the size of the buffer is proportional the number of pixels the node contributes to, so the more nodes in a simulation, then the smaller the memory usage.

6.3.3 Unstructured Volume Rendering

Image Order

The render times between these methods depend on the number of samples per pixel. Ray casting for unstructured meshed can either use connectivity-based or location-based traversal methods. Location-based algorithms must traverse the acceleration structure for every sample. Connectivity-based methods only traverse

the acceleration structure to find the entry points into the mesh, otherwise they determine the next cell via a lookup. For a low number of samples per pixel, location-based methods are the fastest since connectivity methods must keep moving through cells when the next sample point is far away. Alternatively when the number of samples per pixel is high, the overhead of the acceleration structure traversal is higher than the next cell lookup. Thus, sampling frequency dictates which method work best for the rendering time use case.

The quality use case, sampling frequency also impacts image quality. The more sample taken translates to a higher quality image. As with structured volume rendering, a low sampling rate can miss important features in the data. On the other hand, using connectivity with pre-integration does not miss any features and results in the high image quality.

The choice of between ray casting variants also depends on the amount of available memory and the data set. The connectivity table consumes less memory than a BVH. The degenerate case is where each cell has no neighbors, and in this case, connectivity is equivalent to location based methods. Finally, pre-integration tables increase memory usage but raises image quality.

Object Order

For a moderate number of cells, render times for splatting are fast which supports the render time use case. The algorithm leverages GPU hardware to render images quickly, and on a CPU, rasterization tends to be slightly faster than ray casting methods. In cases where there are a large number of objects and equal image quality settings, sampling render time are faster because sampling does not need to sort the objects based on camera position.

Sampling is well suited for the image quality use case, and sampling offers a familiar trade-off. By increasing the number of samples, image quality is enhanced at the cost of render times. With splatting, the gradient would have to be pre-calculated each time the data set changed. The image quality of splatting is limited since the algorithm cannot increase the effective sampling rate beyond the number of cells that project onto a pixel. Additionally, artifacts from the approximate view-dependent sort negatively impacts image quality. Splatting and sampling can improve image quality by using pre-integration tables.

For the memory usage use case, splatting's memory usage is a constant factor relative to the number of objects in the scene, and splatting does not offer any trade-offs. Sampling can reduce memory usage by breaking the buffer into segments, but there is a runtime cost for each segment. The sampling approach is also well suited for distributed-memory settings since the memory usage is tied to the number of samples each node it is responsible for.

6.4 Future Work

This section discussed the current space of rendering algorithms in terms three in situ use cases. The use cases represent far ends of a three axis spectrum, although the entire set of use cases will fall somewhere inside these extremes. Additionally, current rendering algorithms offer a variety of trade-offs that can push an algorithm from one side of the space to the other. The intersection of our current rendering algorithm's tunable parameters and the three axis spectrum is unknown and needs exploration.

Further, the intersection may not overlap all three use cases, meaning we cannot render any images under some of the constraints. In this case, either simulation codes would have to relax the constraints, potentially sacrificing science, or the visualization community will have to revisit current algorithms in order to pull them into the three axis spectrum.

To answer these questions, the community needs to define a reasonable set of constraints that can be expected in the exascale in situ environment. Then we must perform a quantitative analysis to establish if our current algorithms will remain viable, or determine if we need new implementations.

REFERENCES

[1] Kwan-Liu Ma, "In situ visualization at extreme scale: Challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, pp. 14–19, 2009.

[2] Benjamin Lorendeau, Yvan Fournier, and Alejandro Ribes, "In-situ visualization in fluid mechanics using catalyst: A case study for code saturne," in *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*. IEEE, 2013, pp. 53–57.

[3] B Whitlock, J Favre, and J Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," 2011.

[4] Christopher Johnson, Steven G Parker, Charles Hansen, Gordon L Kindlmann, and Yarden Livnat, "Interactive simulation and visualization," *Computer*, vol. 32, no. 12, pp. 59–65, 1999.

[5] Andrew C. Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, and E. Wes Bethel, "*In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report," *Computer Graphics Forum, Proceedings of Eurovis 2016*, vol. 35, no. 3, June 2016, LBNL-1005709.

[6] Marzia Rivi, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnic, "In-situ visualization: State-of-the-art and some use cases," *PRACE White Paper*, 2012.

[7] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jan, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 2011, pp. 89–96.

[8] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, Kathleen Bonnell, Mark Miller, Gunther H. Weber, Cyrus Harrison, David Pugmire, Thomas Fogal, Christoph Garth, Allen Sanderson, E. Wes Bethel, Marc Durant, David Camp, Jean M. Favre, Oliver Rübel, Paul Navrátil, Matthew Wheeler, Paul Selby, and Fabien Vivodtzev, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *Proceedings of SciDAC 2011*, Denver, CO, July 2011.

[9] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H Rogers, and Mark Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 424–434.

[10] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[11] Ciprian Docan, Manish Parashar, Julian Cummings, and Scott Klasky, "Moving the code to the data-dynamic code deployment using activespaces," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 758–769.

[12] Hank Childs, Kwan-Lui Ma, Yu Hingfeng, Brad Whitlock, Jeremy Meredith, Jean Favre, Scott Klasky, and Fan Zhang, "In situ processing," 2012.

[13] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, et al., "Examples of in transit visualization," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 1–6.

[14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Ieee Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[15] Matt Pharr and William R Mark, "ispc: A spmd compiler for high-performance cpu programming," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–13.

[16] Marc Levoy, "Display of surfaces from volume data," *Computer Graphics and Applications, IEEE*, vol. 8, no. 3, pp. 29–37, 1988.

[17] Jörg Mensmann, Timo Ropinski, and Klaus Hinrichs, "An advanced volume raycasting technique using gpu stream processing," in *Computer Graphics Theory and Applications, GRAPP 2010*, 2010, pp. 190–198.

[18] Nelson Max, Pat Hanrahan, and Roger Crawfis, *Area and volume coherence for efficient visualization of 3D scalar functions*, vol. 24, ACM, 1990.

[19] Klaus Engel, Martin Kraus, and Thomas Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM, 2001, pp. 9–16.

[20] Eric B Lum, Brett Wilson, and Kwan-Liu Ma, "High-quality lighting and efficient pre-integration for volume rendering," in *Proceedings of the Sixth Joint Eurographics-IEEE TCVG conference on Visualization*. Eurographics Association, 2004, pp. 25–34.

[21] Aaron Knoll, "A survey of octree volume rendering methods," *Scientific Computing and Imaging Institute, University of Utah*, 2006.

[22] Michal Hapala and Vlastimil Havran, "Review: Kd-tree traversal algorithms for ray tracing," in *Computer Graphics Forum*. Wiley Online Library, 2011, vol. 30, pp. 199–213.

[23] "Space partitioning: Octree vs. bvh," Dec. 2015.

[24] Jeffrey Goldsmith and John Salmon, "Automatic creation of object hierarchies for ray tracing," *Computer Graphics and Applications, IEEE*, vol. 7, no. 5, pp. 14–20, 1987.

[25] J David MacDonald and Kellogg S Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990.

[26] Brian Smits, "Efficiency issues for ray tracing," 1999.

[27] Timothy L Kay and James T Kajiya, "Ray tracing complex scenes," in *ACM SIGGRAPH computer graphics*. ACM, 1986, vol. 20, pp. 269–278.

[28] Stefan Popov, Johannes Gunther, Hans-Peter Seidel, and Philipp Slusallek, "Experiences with streaming construction of sah kd-trees," in *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE, 2006, pp. 89–94.

[29] Ingo Wald, "On fast construction of sah-based bounding volume hierarchies," in *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*. IEEE, 2007, pp. 33–40.

[30] Dmitry Sopin, Denis Bogolepov, and Danila Ulyanov, "Real-time sah bvh construction for ray tracing dynamic scenes," in *21th International Conference on Computer Graphics and Vision (GraphiCon)*, 2011, pp. 74–77.

[31] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*. Wiley Online Library, 2009, vol. 28, pp. 375–384.

[32] Ingo Wald, "Fast construction of sah bvhs on the intel many integrated core (mic) architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 47–57, 2012.

[33] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst, "Embree: a kernel framework for

efficient cpu ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 143, 2014.

[34] Manfred Ernst and Günther Greiner, "Early split clipping for bounding volume hierarchies," in *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*. IEEE, 2007, pp. 73–78.

[35] H. Dammertz and A. Keller, "The edge volume heuristic - robust triangle subdivision for improved bvh performance," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, Aug 2008, pp. 155–158.

[36] Martin Stich, Heiko Friedrich, and Andreas Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 2009, pp. 7–13.

[37] Nadathur Satish, Mark Harris, and Michael Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[38] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister, "Simpler and faster hlbvh with work queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2011, pp. 59–64.

[39] Tero Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 2012, pp. 33–37.

[40] Ciprian Apetrei, "Fast and simple agglomerative lbvh construction," 2014.

[41] Tero Karras and Timo Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*. ACM, 2013, pp. 89–99.

[42] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch, "Efficient bvh construction via approximate agglomerative clustering," in *Proceedings of the 5th High-Performance Graphics Conference*. ACM, 2013, pp. 81–88.

[43] Thomas Larsson and Tomas Akenine-Möller, "Efficient collision detection for models deformed by morphing," *The Visual Computer*, vol. 19, no. 2, pp. 164–174, 2003.

[44] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, M-P Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Strasser, et al., "Collision detection for deformable objects," in *Computer graphics forum*. Wiley Online Library, 2005, vol. 24, pp. 61–81.

[45] Thomas Larsson and Tomas Akenine-mÃűller, "Strategies for bounding volume hierarchy updates for ray tracing of deformable models," Tech. Rep., Centre, Maelardalen University, 2003.

[46] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha, "Rt-deform: Interactive ray tracing of dynamic scenes using bvhs," in *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE, 2006, pp. 39–46.

[47] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha, "Ray tracing dynamic scenes using selective restructuring," in *Proceedings of the 18th Eurographics conference on Rendering Techniques*. Eurographics Association, 2007, pp. 73–84.

[48] Kirill Garanzha, "Efficient clustered bvh update algorithm for highly-dynamic models," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE, 2008, pp. 123–130.

[49] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler, "Fast, effective bvh updates for animated scenes," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2012, pp. 197–204.

[50] David Hook and Kevin Forward, "Using kd-trees to guide bounding volume hierarchies for ray tracing," *AUST COMPUT J*, vol. 27, no. 3, pp. 103–108, 1995.

[51] Carsten Wächter and Alexander Keller, "Instant ray tracing: The bounding interval hierarchy," *Rendering Techniques*, vol. 2006, pp. 139–149, 2006.

[52] M. Vinkler, V. Havran, J. Bittner, and J. Sochor, "Parallel on-demand hierarchy construction on contemporary gpus," *Visualization and Computer Graphics, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[53] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al., "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, pp. 66, 2010.

[54] Manfred Ernst and Sven Woop, "Embree: Photo-realistic ray tracing kernels," *White paper, Intel*, 2011.

[55] Timo Aila and Samuli Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the conference on high performance graphics 2009*. ACM, 2009, pp. 145–149.

[56] Timo Aila, Samuli Laine, and Tero Karras, "Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum," *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[57] Kunal Gupta, Jeff A Stuart, and John D Owens, "A study of persistent threads tsyle gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[58] Sven Woop, "A Ray Tracing Hardware Architecture for Dynamic Scenes," Tech. Rep., Saarland University, 2004.

[59] Victor W Lee et al., "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ACM SIGARCH Computer Architecture News*. ACM, 2010, vol. 38, pp. 451–460.

[60] Andrew Kensler and Peter Shirley, "Optimizing ray-triangle intersection via automated search," in *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE, 2006, pp. 33–38.

[61] Sven Woop, Carsten Benthin, and Ingo Wald, "Watertight ray/triangle intersection," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, pp. 65–82, 2013.

[62] Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R Mark, "Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 9, pp. 1438–1448, 2012.

[63] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner, "Interactive rendering with coherent ray tracing," in *Computer graphics forum*, 2001, vol. 20, pp. 153–165.

[64] Kirill Garanzha and Charles Loop, "Fast ray sorting and breadth-first packet traversal for gpu ray tracing," in *Computer Graphics Forum*. Wiley Online Library, 2010, vol. 29, pp. 289–298.

[65] Alexander Reshetov, Alexei Soupikov, and Jim Hurley, "Multi-level ray tracing algorithm," in *ACM Transactions on Graphics (TOG)*. ACM, 2005, vol. 24, pp. 1176–1185.

[66] Carsten Benthin and Ingo Wald, "Efficient ray traced soft shadows using multi-frusta tracing," in *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 2009, pp. 135–144.

[67] Tom Peterka, Hongfeng Yu, Robert B Ross, Kwan-Liu Ma, et al., "Parallel volume rendering on the ibm blue gene/p.," in *EGPGV*. Citeseer, 2008, pp. 73–80.

[68] Aaron Knoll, Sebastian Thelen, Ingo Wald, Charles D Hansen, Hans Hagen, and Michael E Papka, "Full-resolution interactive cpu volume rendering with coherent bvh traversal," in *Pacific Visualization Symposium (PacificVis), 2011 IEEE*. IEEE, 2011, pp. 3–10.

[69] Mark Howison, E Wes Bethel, and Hank Childs, "Hybrid parallelism for volume rendering on large-, multi-, and many-core systems," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 17–29, 2012.

[70] Junpeng Wang, Fei Yang, and Yong Cao, "Cache-aware sampling strategies for texture-based ray casting on gpu," in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*. IEEE, 2014, pp. 19–26.

[71] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Gröller, "Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data," in *Volume Visualization and Graphics, 2004 IEEE Symposium on*. IEEE, 2004, pp. 1–8.

[72] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel, "Faster isosurface ray tracing using implicit kd-trees," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 11, no. 5, pp. 562–572, 2005.

[73] Michael P Garrity, "Raytracing irregular volume data," *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 5, pp. 35–40, 1990.

[74] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. IEEE Computer Society, 2003, p. 44.

[75] Philipp Muigg, Markus Hadwiger, Helmut Doleisch, and Eduard Gröller, "Interactive volume visualization of general polyhedral grids," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 12, pp. 2115–2124, 2011.

[76] Saulo Ribeiro, André Maximo, Cristiana Bentes, Antonio Oliveira, and Ricardo Farias, "Memory-aware and efficient ray-casting algorithm," in *Computer Graphics and Image Processing, 2007. SIBGRAPI 2007. XX Brazilian Symposium on*. IEEE, 2007, pp. 147–154.

[77] Brad Rathke, Ingo Wald, Kenneth Chiu, and Carson Brownlee, "SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids," in *Eurographics Symposium on Parallel Graphics and Visualization*, C. Dachsbacher and P. Navrátil, Eds. 2015, The Eurographics Association.

[78] "Mesa 3d graphics library," June 2015.

[79] "Eecs 487 pa1: Rasterization," Dec. 2015.

[80] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl, "Half-tone perspective drawings by computer," in *Proceedings of the November 14-16, 1967, fall joint computer conference*. ACM, 1967, pp. 49–58.

[81] Juan Pineda, "A parallel algorithm for polygon rasterization," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 17–20, June 1988.

[82] Intel, "Openswr," 1999.

[83] Samuli Laine and Tero Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, New York, NY, USA, 2011, HPG '11, pp. 79–88, ACM.

[84] Hendrik A Schroots and Kwan-Liu Ma, "Volume rendering with data parallel visualization frameworks for emerging high performance computing architectures," in *SIGGRAPH Asia 2015 Visualization in High Performance Computing*. ACM, 2015, p. 3.

[85] Steven P Callahan, Milan Ikits, João Luiz Dihl Comba, and Claudio T Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 11, no. 3, pp. 285–295, 2005.

[86] André Maximo, Ricardo Marroquim, and Ricardo Farias, "Hardware-assisted projected tetrahedra," in *Computer Graphics Forum*. Wiley Online Library, 2010, vol. 29, pp. 903–912.

[87] Peter L Williams, "Visibility-ordering meshed polyhedra," *ACM Transactions on Graphics (TOG)*, vol. 11, no. 2, pp. 103–126, 1992.

[88] Hank Childs, Mark Duchaineau, and Kwan-Liu Ma, "A scalable, hybrid scheme for volume rendering massive data sets," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, Aire-la-Ville, Switzerland, Switzerland, 2006, EGPGV '06, pp. 153–161, Eurographics Association.

[89] M. Larsen, S. Labasan, P. Navrátil, J. S. Meredith, and H. Childs, "Volume rendering via data-parallel primitives," in *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*, Aire-la-Ville, Switzerland, Switzerland, 2015, PGV '15, pp. 53–62, Eurographics Association.

[90] Jeremy S Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros, "Eavl: the extreme-scale analysis and visualization library," 2012.

[91] Ulrich Neumann, "Parallel volume-rendering algorithm performance on mesh-connected multicomputers," in *Parallel Rendering Symposium, 1993*. IEEE, 1993, pp. 97–104.

[92] Kwan Liu Ma, James S Painter, Charles D Hansen, and Michael F Krogh, "A data distributed, parallel algorithm for ray-traced volume rendering," in *Proceedings of the 1993 symposium on Parallel rendering*. ACM, 1993, pp. 15–22.

[93] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma, "Massively parallel volume rendering using 2–3 swap image compositing," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.

[94] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur, "A configurable algorithm for parallel image-compositing applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009, SC '09, pp. 4:1–4:10, ACM.

[95] Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian

Huang, "An image compositing solution at scale," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, SC '11, pp. 25:1–25:10, ACM.

[96] Paul Navrátil, Donald Fussell, Calvin Lin, and Hank Childs, "Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing," in *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, Cagliari, Italy, May 2012, pp. 61–70, *Best paper* award.

[97] Paul Navrátil, Hank Childs, Donald Fussell, and Calvin Lin, "Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 20, no. 6, pp. 893–906, June 2014.

[98] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther Weber, and E. Wes Bethel, "Extreme Scaling of Production Visualization Software on Diverse Architectures," *IEEE Computer Graphics and Applications (CG&A)*, vol. 30, no. 3, pp. 22–31, May/June 2010.

[99] Mark Howison, E. Wes Bethel, and Hank Childs, "Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 18, no. 1, pp. 17–29, Jan. 2012.

[100] Mark Howison, E. Wes Bethel, and Hank Childs, "MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, Apr. 2010, pp. 1–10, One of two *best papers*.