# Design for Parallel Interactive Ray Tracing Systems

James Bigler Abe Stephens Steven G. Parker*

Scientific Computing and Imaging Institute, University of Utah

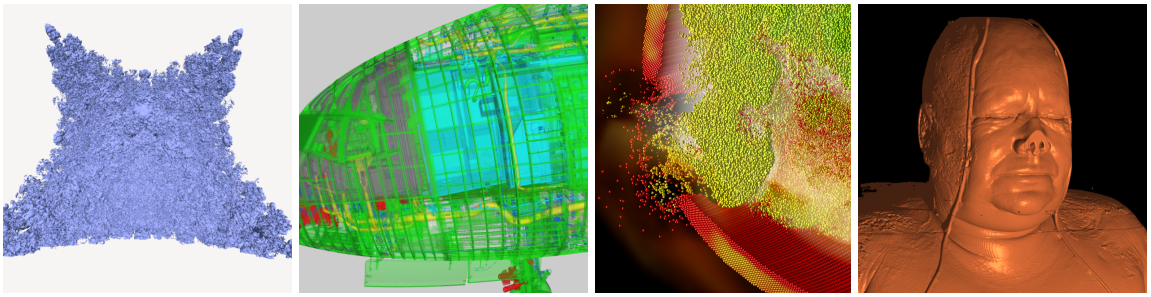

Figure 1: Images generated using interactive ray tracing. From left to right, time step 225 of a Richtmyer-Meshkov instability simulation from Lawrence Livermore National Labs (Image courtesy of Aaron Knoll), Boeing 777 (Data courtesy The Boeing Company), 2.8 million particle MPM simulation with direct volume rendered fire, and iso-surface of the Visible Female.

## Abstract

We describe the software architecture of the Manta interactive ray tracer and describe its application in engineering and scientific visualization. Although numerous ray tracing software packages have been developed, much of the traditional design wisdom needs to be updated to provide support for interactivity, high degrees of parallelism, and modern packet-based acceleration structures. We discuss situations that are normally not considered when designing a batch ray tracer, and present methods to overcome those challenges. This paper advocates a forward looking programming model for interactive ray tracing that uses reconfigurable components to achieve flexibility while achieving scalability on large numbers of processors. Manta employs data structures motivated by modern microprocessor design that can exploit instruction-level parallelism. We discuss the design tradeoffs and the performance achieved for this system.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.8 [Computer Graphics]: Applications D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

**Keywords:** ray tracing, interactive ray tracing, software architecture

## 1 Introduction

Almost ten years after the first interactive ray tracer was introduced on a massive super computer, the necessary compute power to solve the same visualization problems will be available on a workstation sized system. Early interactive systems [11, 13] were built similarly to batch renderers [7], which traced a single ray at a time and employed simple mechanisms for parallelism. Modern ray tracing systems [27] have focused on raw performance for single data structures but sacrificed generality for performance. Based on our experience with interactive ray tracing over the past several years, we set out to define an architecture that achieves performance while maintaining flexibility and accommodates the demands of interactivity in a highly concurrent system.

There are many driving objectives for the ray tracing system described in this paper, the most important two are *interactivity* and *flexibility*. The renderer must deliver interactive performance and at the same time its design and implementation must be flexible enough to apply to a number of different graphics and visualization problems. Other objectives include portability, embedability and maintainability.

To deliver interactive performance it is necessary for software to fully leverage the underlying processor and system architecture. Interactive ray tracing was traditionally performed on large multiprocessor systems, but due to trends in single processor design, the landscape of multi-processor, multi-threaded systems is shifting from supercomputers to processors that contain a large number of separate processing cores. Instead of a supercomputer running threads across several cabinets in a machine room, a multi-core system running the same number of threads will fit in a workstation chassis.

This shift towards multi-core is accompanied by processor design decisions that affect how programs must be written to achieve high performance. Multi-core processor designs rely on thread-level parallelism and explicit instruction stream parallelism, such as SIMD instruction sets, and less on hardware mechanisms to discover such parallelism automatically.

We propose a two-piece programming model to take advantage of current and upcoming hardware design. The first piece is a multi-threaded scalable parallel pipeline. The second is a collection of software mechanisms and data structures, based on ray packets, for exploiting parallelism and performance optimization from component-based rendering code. This joint model provides the ray tracer with the opportunity to scale on growing mainstream multi-core platforms while at the same time taking advantage of the hardware designs that enable higher degrees of parallelism.

Rendering systems must be flexible enough to solve a variety of different problems while individual rendering techniques usually solve a single problem. Manta is a ray tracing system that is applied to a number of graphics and visualization problems, from triangle mesh rendering to time-varying multi-modal sphere glyph and volume rendering. Ray tracing allows for a more direct implementation of these techniques than is possible using a textured triangle

*{bigler|abe|sparker}@cs.utah.edu

based rendering system. In addition the ray tracer implementation is able to use the same parallel performance model while rendering different types of primitives. Manta's programming model provides this flexibility and has permitted the software to be used in a number of different ray tracing applications. Manta is available freely as an open-source software project [1].

The remainder of this paper discusses the challenges encountered by interactive ray tracing systems and our design philosophy for interactive ray tracers, discusses the details of Manta's design and tradeoffs we encountered, and finally describes several applications of the architecture to graphics and visualization problems.

## 2 Hardware Trends

Ray tracing one frame containing only static scenery with a precomputed acceleration structures is an embarrassingly parallel task. This property has provided a straightforward response to most performance issues: divide the frame into more parallel pieces and increase the number of processors.

Both cluster and shared memory system configurations may be scaled in this way. Early interactive ray tracers were implemented on shared memory supercomputers [14]. This type of system is a highly tuned cluster of individual systems, usually with special-purpose processors, connected by a high performance network. The topology of the network varies by configuration as does memory-access latency. These systems provide a single-system image to the application, which simplifies the programming model but still requires attention to memory locality. Cluster systems composed of individual commodity workstation or servers have also demonstrated success in interactive ray tracing [23, 6, 4], but require explicit communication to be performed at the application level.

Rendering performance may also be increased by improving single processor performance. Many single core processors are designed to exploit instruction level parallelism and deep instruction pipelining within a single thread of execution. These processors increase serial instruction stream performance through increased clock speed, branch prediction, and various instruction issue strategies. While Moore's Law continues, the number of transistors available on each die increases, but the ability of conventional mechanisms to extract parallelism from a single thread of execution is diminishing [2, 8].

Processor designs are turning towards explicit multi-threading and are using increased die area for multiple cores instead of more complex units. At the same time single instruction multiple data (SIMD) instruction sets are enabling more explicit instruction level parallelism declaration in the instruction stream. For example a four wide SIMD add operation would combine four sets of two numbers and produce four results. Depending on the design of the underlying processor all operations may or may not be executed in parallel. The instruction only indicates that the vector operations may be performed in any order. These instructions are a method to explicitly specify opportunities for instruction-level parallelism (ILP). Making use of SIMD instructions requires a specific alignment and ordering of data members, and impacts other aspects of the architecture. It is reasonable to expect that wider SIMD instruction sets might be available in the future.

## 3 Software Design Philosophy

The hardware trends described in the previous section guide our software architecture. Matching anticipated hardware trends, Manta is designed to provide scalable thread-level parallelism on a multi-core system and allow for instruction stream optimization on those cores. Instead of requiring the use of a particular acceleration structure or sampling technique to achieve interactive performance, we advocate building an interactive ray tracer using a set of configurable software components that follow a specific design pattern. These components form the infrastructure in which acceleration structures, primitives, material shaders and rendering techniques can be implemented. This provides flexibility but imposes overhead and presents challenges for parallelism. We overcome these challenges by utilizing wide ray packets and a pipeline model for parallelism.

Manta achieves scalable performance on a variety of configurations through the use of a pipeline model for thread-level parallelism. Scalable performance means that the rendering speed increases near-linearly as the number of processors increases. Scalable performance is important to systems that achieve good performance on one or two core systems today because the presence of bottlenecks will ultimately prevent these systems from taking advantage of larger systems or upcoming dense multi-core systems.

The parallel pipeline model divides rendering tasks into stages based on load characteristics of each task. Certain tasks, like sending the image to the graphics card for display, are load imbalanced (often performed by a single processor), and other tasks, like ray tracing the frame, can be dynamically load balanced. Synchronization between all threads, allowing one thread to change global state, is constrained to occur only at specific places in the pipeline.

Parallel scaling addresses one important consideration for high performance ray tracing. Individual thread performance is equally important, especially on modern multi-core processor designs that de-emphasize instruction level parallelism extraction. Single thread throughput is affected by two considerations; first the size of the instruction stream (number of operations required) by the acceleration structure traversal, shading, and other rendering techniques, and second, the throughput of the core executing that stream. Since our architecture targets a wide range of rendering techniques, we provide several mechanisms based on wide ray packets that exploit instruction stream optimization.

Wide ray packets are data structures that contain basic ray information and other derived values for many rays. In addition, ray packets contain flags to indicate certain properties about all rays in the packet, such as indicators that the rays share a common origin or that have normalized direction vectors. Using this data structure allows for several types of single thread performance optimization including: software pipelining, use of SIMD instructions, special case code, and coherence optimization. Wide ray packets are typically larger than SIMD vectors and are processed in smaller pieces with SIMD instructions in tight loops.

Ideally, ray packets contain coherent sets of rays with shared attributes like direction, common origin, etc. These shared attributes can allow special case optimizations. Wide ray packets also help amortize the cost of virtual function overhead over a large number of rays. Virtual function lookup is particularly useful for connecting loosely coupled rendering components with a common ray packet based API.

Another high-level design philosophy is the lazy evaluation of expensive values. If two components both require a derived value, that value can be computed once lazily in the first component, and a flag is set in the ray packet. The second component can check the flag before recomputing the derived value.

## 4 Related Work

Examples of interactive ray tracing have existed for many years. This section describes several interactive ray tracing systems as well as contemporary ray tracing techniques that influence software design. Our work builds on research from two areas; parallel software design for interactive ray tracing and general interactive ray tracing techniques that must be implemented on top of those designs.

Software systems for interactive ray tracing have, until quite recently [18], always used parallel hardware. Large multiprocessor systems, although not widely available, were used to predict future workstation performance [11, 13]. Modern hardware is becoming highly parallel, with multi-core designs dominating future processor road maps.

The Star-Ray rendering system was the first versatile interactive system and was used to visualize several different types of data [14, 15]. The renderer was designed to run on large shared memory super computers and scaled up to 1024 processors on one configuration. Although it was applied to a number of different visualization problems, employed an object model, and defined an extendable programming interface, its development predated the wide spread adoption of ray packets.

The Star-Ray system was later partially adapted to cluster systems in the DIRT rendering system by DeMarle et al. [4]. This renderer managed its own distributed memory providing the programmer with the illusion of a shared memory programming environment while running on a cluster [5] and was applied to similar visualization problems.

Interactive ray tracing systems have been designed from the ground up for cluster systems. OpenRT, an API and rendering system for interactive ray tracing, is an example of this approach [23, 6, 26]. OpenRT based systems feature extendable shaders and have been deployed successfully in a variety of industrial visualization capacities [22].

Other interactive ray tracing systems are designed without a focus on massive scalability. The Razor rendering system achieves nearly interactive frame rates for rendering subdivision surfaces [20]. This system's fundamental design addresses subdivision surface intersection rather than scalable parallelism.

Some ray tracing systems are highly modular yet lack interactivity. The authors of "Physically Based Rendering" released the renderer implemented in their book, called PBRT, as an extensible physically based ray tracer [16]. There are numerous extensions making use of this renderer. The open source ray tracer POV-Ray is also widely used and extensible [12], but far from interactive.

Along with efficient kd-tree acceleration structures, OpenRT first made use of ray packets in order to take advantage of the SSE SIMD instruction set [27]. Ray packets in this system consisted of only four rays.

In principle, two solutions have been chosen for building dynamic scene structures. Some approaches use single threaded data structure build routines[25, 24], others use a separate parallelization scheme for building and rendering[9]. Lazy or on-the-fly structure building creates another difficulty since the (ideally) parallel build would occur in small pieces during ray tracing, when the threads have already been load balanced for rendering.

Non-general purpose hardware is used for ray tracing as well. Two examples are Cell processors [10] and GPUs [17]. Low level software models must be adapted for use on these high throughput but quite constrained platforms.

## 5 Software Architecture

One of Manta's primary design objectives is flexibility. This is achieved through the use of modular components. Manta is composed of two groups of modular components, a pipeline and a rendering stack. Components in the pipeline group form a front-end that implements our parallel pipeline model and drives a group of back-end components arranged in a stack. The rendering stack samples pixels, generates rays, computes intersections and shaded colors, and finally produces colors for each pixel. In addition to these two component groups the architecture includes scene geometry, surface shaders and utility libraries.

Application programming interfaces (APIs) of each component define how data moves though the pipeline and between components in the rendering stack. Component APIs are defined in pure-virtual C++ classes. Context structures are used to communicate pointers to other components, thread information and other system configuration information between components. Ray packets are the primary structure used to communicate ray tracing data between components in the rendering stack, scene intersection, texturing, and shading.

### 5.1 Wide Ray Packets

Wide ray packets contain individual rays plus all of the data needed to allow intersection routines, shaders, and other components to take advantage of instruction stream optimizations. Packets are a container of rays with similar properties. These structures allow for many types of optimizations including software pipelining, SIMD operations, special case code, and sharing derived values between loosely coupled components. Ray packets also increase ray coherence during intersection and shading.

Ray packets have a maximum size determined at compile time and unlike other ray tracing systems are not limited to SIMD width on the target system or load balancer tile size. We have found that ray packets with a maximum size of 64 achieves the best performance. At this size, the entire ray packet data can fit in L1 cache.

One design decision we encountered was determining what to do when rays in a ray packet are not perfectly coherent. Some rays may hit one object while others hit a second. In the worst case, each ray does something entirely different. However, most scenes enjoy some type of coherence for rays and Manta tries to optimize for those situations. Ray packets are split into coherent sections at each phase of the algorithm, such as between intersection and shading. Splitting the ray packet is accomplished by finding a sequential run of rays in the packet with a common characteristic, like hitting the same material shader, and creating a new smaller packet containing only these rays (see Figure 6). Splitting a ray packet creates challenges for code that employs SIMD operations, since these instructions require a specific operand alignment, so SIMD code usually has a preloop phase to handle the unaligned portion of the ray packet (always 3 rays or fewer for 4-way SIMD). Splitting ensures that coherence is maintained as the ray packet propagates through different components.

#### 5.1.1 Data Layout for SIMD

SIMD instruction sets permit explicit declaration of instruction-level parallelism that may be exploited by the processor when scheduling operations or using functional units in parallel. Ray packet data fields are arranged in memory to be loaded into SIMD registers. This data organization also has other performance benefits on platforms that don't rely on these instructions.

Instruction sets such as Intel's SSE family [21] require an non-intuitive data layout. Instructions in SSE have two sets of four operands and produce four results. Each set of operands and the result must appear sequentially in memory and be 16-byte aligned. This requires a vertical (structure of arrays) layout of the data structure, instead of a horizontal (array of structures) layout.

Manta accommodates code requiring both layouts. Horizontal accessor methods for ray packet fields gather vector components from each array in the native vertical layout. This way, one could ask for the origin and direction and get them as a C++ geometric vector object. These accessors provide a gentle migration from existing code that assumes horizontal data layout with only a slight performance penalty.

#### 5.1.2 Software Pipelining

One by-product of vertical data layout is that it prevents use of most C++ overloaded vector operators. On in-order processors like the Itanium2 vertical layout decreased dependencies between neighboring instructions in the instruction stream and increased software pipelining in the compiled code. Although not as dramatic, decreasing data dependency, even without explicit SIMD instruction utilization, produced a performance improvement on mainstream out-of-order processors.

#### 5.1.3 Special Case Code

Manta defines fourteen ray packet properties such as common origin, constant direction sign, and whether or not derived values have been computed and stored in the packet. These properties may be used to select optimized code paths in special cases.

During certain scientific visualizations, sphere intersection is a common operation (see section 6.3). If all the rays in a packet have a common origin, then intermediate values, like the vector from the ray origin to sphere center, may be computed once for the entire packet. This instruction stream optimization reduces the number of operations necessary to intersect the sphere with the entire ray packet.

#### 5.1.4 Loosely Coupled Components

Loosely coupled components are neighboring code modules with no dependencies on the others' implementation. These components increase flexibility in Manta because they increase code reuse by allowing software from one configuration to be placed in another. Since loosely coupled components cannot make any assumptions about each other, it is difficult to eliminate redundant calculations or perform other optimizations between components.

Consequently, Manta contains a mechanism to reduce potential redundancy. Along with the ray data, derived values are stored in the ray packet. These quantities such as the inverse direction, intersection location, and surface normals are computed the first time they are needed and stored in the ray packet. For example, surface normals might be used by local shaders and shaders that generate reflection rays. The ray packet accessors use packet properties to insure that the qualities are only calculated once. Since the cost of storing and checking this flag is amortized over all of the rays in the packet, the cost does not become burdensome.

### 5.2 Parallel Pipeline

Manta's parallel pipeline components are responsible for controlling thread activity and synchronization. The pipeline is used to overlap asynchronous image display with image rendering to reduce the overhead of single-threaded tasks. Manta achieves good scalability with large numbers of rendering threads by constraining thread synchronization to certain points in the pipeline.

The pipeline consists of several stages executed by each thread. Stages are organized by dividing program tasks based on their load balance characteristics. Tasks that are inherently load balanced are executed first, followed by imbalanced tasks (such as image display), and lastly tasks that are dynamically load balanced (like rendering). (Figure 3 contains pseudo code for a simple pipeline.) This organization reduces overhead introduced by single-threaded tasks like image display by allowing them to be performed asynchronously with rendering.

One frame buffer is associated with each stage in the pipeline. In a two stage pipeline with image display and rendering, this results in a one-frame latency between the time a frame is rendered and the time it is displayed. In the most common two-stage configuration (see Figure 2), image display will typically be executed by a single thread, that will then join the rendering once the display has completed. Manta also supports rendering multiple viewpoints simultaneously in the same load balance mechanism.

```
void Pipeline::inner_loop( int frame, int proc ) {
  // Inherently load balanced.
  parallel_animation_callbacks();

  // Imbalanced.
  if (proc == display_proc)
    image_display->displayImage( buffer[frame-1] );

  // Dynamically balanced.
  image_traverser->render_image( buffer[frame], proc );
}
```

Figure 3: Pseudo code for a pipeline loop. Load balanced tasks are executed first. Imbalanced tasks are followed by dynamically load balanced tasks. This organization prevents high latency single-thread operations, like image display, from causing all threads to stall.

Thread synchronization occurs at a barrier at the end of the pipeline. This barrier provides an opportunity for the pipeline to modify the renderer's state safely using transactions and to invoke callbacks. Modifications to the scene graph are also performed at this time.

### 5.3 Transactions and Callbacks

Interaction requires that the renderer's state be safely updated after user input or animation. For example, unsafe updates from user interface threads can cause visual tearing artifacts if the camera configuration is changed part way through rendering the frame. More serious race conditions between user interface threads and Manta can occur if scenery or acceleration structures are altered, potentially resulting in memory access violations.

Instead of double buffering every piece of state that might need to be changed during rendering (potentially the entire scene for some applications), Manta performs user input state changes using transactions, or short callback functions, to execute code that modify state at a *safe* time. The rendering pipeline maintains a queue of transactions that are invoked during thread synchronization at the pipeline barrier for rendering threads. The GUI thread, or other application level threads, are not explicitly synchronized as part of the engine. Transactions are just one type of callback, other types may be invoked at different points during the rendering pipeline depending on their load balance characteristics.

Animation or other scene graph changes can be performed through transactions. Although the nature of these changes is application dependent, the transaction model provides several mechanisms to allow updates to be propagated. A parallel in-place change of the scene graph can utilize the rendering threads (before they start of a new frame) with a parallel animation callback. Alternatively, a separate thread can work on a copy of the scene graph and through a transaction change the renderer to use this new copy.

Callbacks are structures that contain a pointer to a class instance, pointer to a class method and a static copy of all arguments for the method. Usually the target method is implemented in code outside of Manta such as a GUI or other third party code. When callbacks are invoked by a Manta thread, the thread calls the function contained in the structure along with the copied arguments and any call time bound parameters. The callback structure itself is heavily templated so that a wide range of functions with varying numbers of arguments may be invoked.

Figure 4 contains example code for sending a transaction to Manta. After an asynchronous GUI thread detects user input, a

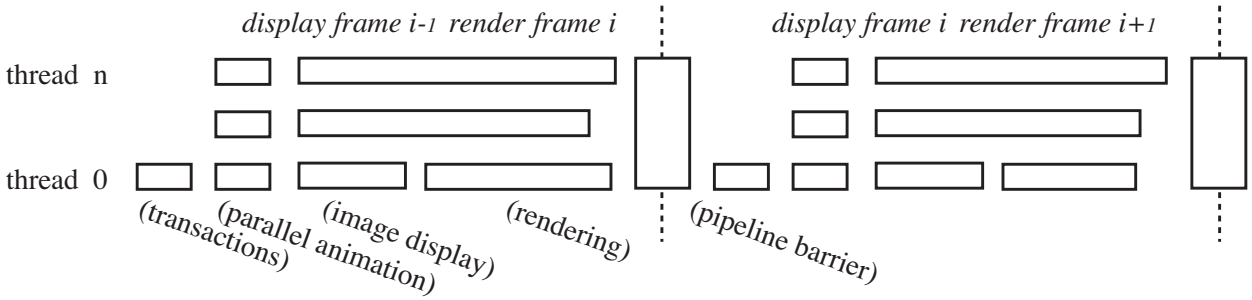


Figure 2: Two stage parallel pipelines consisting of image display and rendering are the most common Manta configuration. In this diagram, threads 0 through $n$ execute transactions and animation callbacks, then thread 0 performs image display while threads 1 through $n$ begin rendering. After finishing display, thread 0 joins rendering. After all of the threads finish rendering the frame, at approximately the same time, the synchronize at the pipeline barrier.

```
void Keyboard::onWKey( const Event &event ) {
  float dolly;
  ...
  manta->addTransaction(Callback::create(this,
                          &Keyboard::mantaDolly, dolly));
}

void Keyboard::mantaDolly( float amount ) {
  manta->getCamera()->dolly( amount );
}
```

Figure 4: Pseudo code for an example transaction to move the camera. First the GUI thread calls the `onWKey` method and adds a transaction callback to `manta` passing the method `mantaDolly`. Later when all the rendering threads are synchronized, one rendering thread will invoke the callback and execute the `mantaDolly` method modifying the camera.

callback to a GUI class method is created and added to Manta's transaction queue. When the renderer reaches the next pipeline barrier, the transaction queue is flushed, causing the Manta thread to invoke the GUI class method. This subtle handoff is important, although most of the code in the GUI class is executed by the the GUI thread, the callback function will be executed by a Manta thread and it can safely alter any of the renderer's state.

Semantics for state update using transactions limit the number of places that rendering threads must synchronize. This constraint minimizes the amount of time any thread spends blocked at a mutex or waiting at a barrier and enables the system to scale up to a large number of threads.

### 5.4 Rendering Stack

In the ray tracing stage of the parallel pipeline each thread asynchronously traverses components of the rendering stack. These components are responsible for dividing the frame up into tiles, determining sample points and finally performing ray tracing. There are three modular components in the rendering stack; an image traverser, pixel sampler, and renderer. See Figure 5.

The first component in the rendering stack is the image traverser. The image traverser is responsible for dividing the frame into regions and assigning these regions to threads. Since the image traverser is invoked at the end of the rendering pipeline, it uses a dynamic load balancer to even out the work load of each thread enabling them to finish rendering at approximately the same time. This is accomplished by means of a work queue. When a thread needs more work it requests the next block of work until the queue

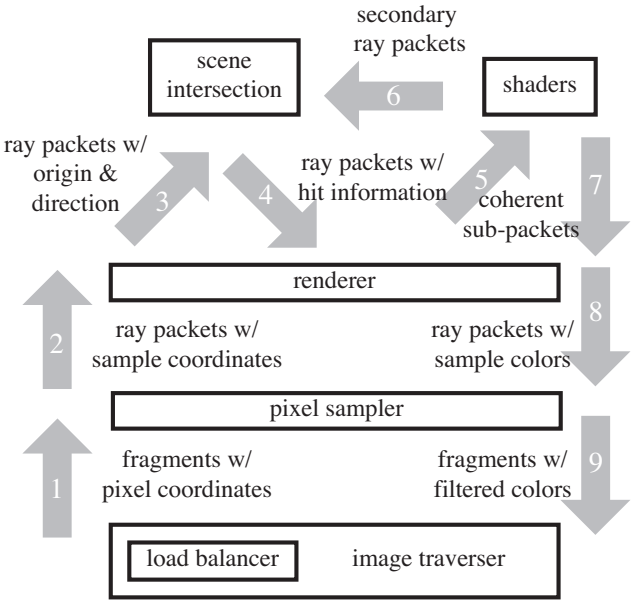


Figure 5: Each thread asynchronously executes the modular rendering stack during the rendering stage of the pipeline.

is empty. The queue divides the work into many assignments with progressively increasing granularity. The larger assignments help reduce communication costs while the smaller assignments help even out the work distribution at the end. This minimizes the amount of time any thread must wait at the pipeline barrier and insures that the parallel pipeline will keep all threads as busy as possible. The load balancer is a modular component of the Manta engine and can be extended or replaced with different load balancing schemes.

Ray packets are introduced to the rendering stack by the pixel sampler component. The image traverser passes a collection of pixels to the sampler. The pixel sampler determines the location of samples for each pixel and maps a ray to each sample. Ray packet data is allocated on the program stack by the pixel sampler and passed up the rendering stack to the renderer component for ray tracing.

The modular renderer component is responsible for tracing rays through a scene graph and then invoking material shaders on coherent sub-packets of rays that strike the same shader. Figure 6 contains actual code for a renderer module that performs ray trac-

```
void Raytracer::traceRays(const Context& context,
                          RayPacket& rays) {
  context.camera->makeRays(rays);

  rays.resetHits();
  context.scene->getObject()->intersect(context, rays);

  for(int i = rays.begin();i<rays.end();){
    if(rays.wasHit(i)){
      const Material* hit_matl = rays.getHitMaterial(i);
      int end = i+1;
      while(end < rays.end() && rays.wasHit(end) &&
            rays.getHitMaterial(end) == hit_matl)
        end++;
      RayPacket subPacket(rays, i, end);
      hit_matl->shade(context, subPacket);
      i=end;
    } else {
      int end = i+1;
      while(end < rays.end() && !rays.wasHit(end))
        end++;
      RayPacket subPacket(rays, i, end);
      context.scene->getBackground()->shade(context,
                                            subPacket);
      i=end;
    }
  }
}
```

Figure 6: Code for the implementation of the renderer interface that performs ray tracing. After tracing a ray packet and finding intersection points the packet is split into smaller sub-packets containing rays that strike the same material shader.

ing. This code splits the incoming packet into smaller sub-packets whose contents all strike the same material shader. Creating a sub-packet does not copy data; it merely selects a range of rays in the parent packet. Splitting packets into sub-packets and masking out rays in a packet are two mechanism available to shaders in Manta to address the diminishing coherence of secondary rays. Sections of the code that require recursive secondary rays, such as shadows or reflections, create new ray packets on the stack and incorporate the results after tracing the rays. The issue of coherency of these secondary rays is an ongoing area of research.

The Manta libraries contain several implementations for many of the components in the rendering stack. This also allows extensions to be built that make use of the rest of the infrastructure without building an entire ray tracer from scratch. Example rendering stack and configuration changes for a specific application are described by Stephens et al. [19].

## 6 Applications

The Manta software architecture has been applied to several graphics and visualization problems. This section describes various applications from large triangle model visualization to volume rendering.

### 6.1 Massive Model Visualization

The data structures used for accelerating static scenes make ray tracing especially well suited to massive triangle model rendering. Traversal time of kd-trees and other structures scale sub-linearly, thereby enabling models consisting of hundreds of millions of triangles to be rendered interactively. Since visibility is computed on-the-fly by tracing rays, effects such as cutting planes, transparency and ambient occlusion are implemented in a straightforward manner.

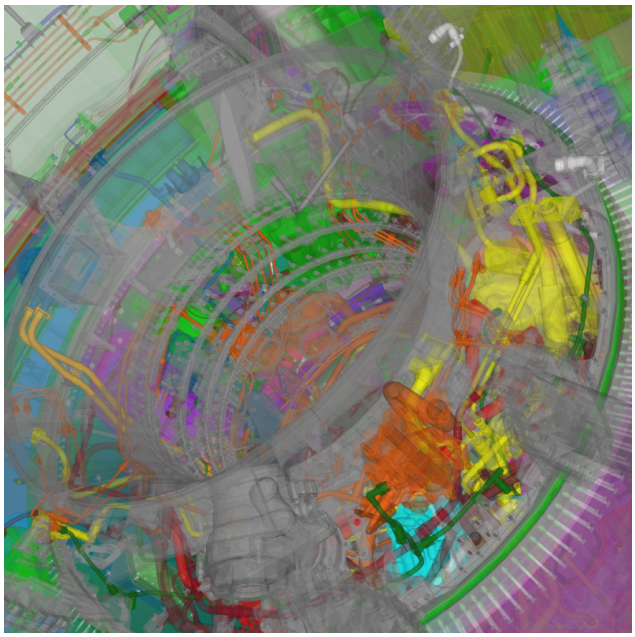

Figure 7: Transparent rendering of a Boeing 777 aircraft engine. The model has been rendered on SGI Itanium2 based systems with between 12 and 512 processors.

Massive model visualization in Manta has concentrated on the Boeing 777 dataset consisting of approximately 350 million triangles [19]. Manta uses a static kd-tree built offline from this data. Transparent rendering is accomplished by modifying the kd-tree traversal to blend sorted ray triangle intersections 7. Modular components in Manta's rendering stack are also able to be replaced to perform interleaved sampling and rendering ambient occlusion based shading.

Manta scaled to 126 processors running on an SGI Itanium2 system. In this configuration rendering performance scaled to 92% of linear on 64 processors and 82% on 126 processors. (See Figure 8.) More common configurations contain between 12 and 64 processors, and often Manta is used in a collaborative visualization environment. In one instance the Manta was run on a 512 processor system to produce a 600 megapixel rendering of the Boeing 777 in several seconds.

### 6.2 Direct Iso-surface Rendering

In addition to rendering millions of triangles, specialized data structures can be used to provide direct iso-surface rendering. Rather than computing a triangulated mesh with a method such as Marching Cubes and rendering the resulting geometry, the surface is instead computed directly during intersection. This eliminates the intermediate geometry producing a view dependent representation of the iso-surface. Since the surface is recomputed each frame as part of the intersection phase, the user has the ability to interactively change the iso-surface value. This provides an interactive framework for data exploration. In addition, the memory that would be needed to store triangulations of complex iso-surfaces is no longer required. Only the original data needs to remain resident in memory[15].

The Visible Female data set contains a high resolution CT scan of the subject. Using Manta we are able to interactively view iso-surfaces of the data using two cores of an Opteron 880 processor

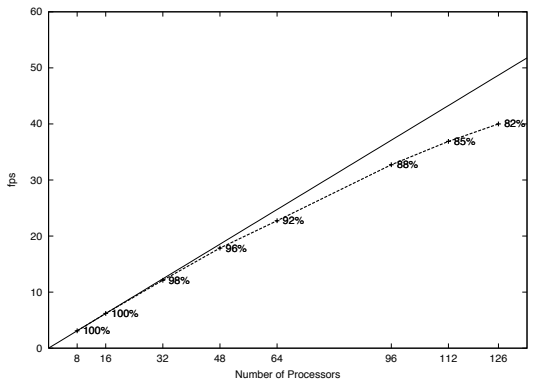


Figure 8: Scaling performance of Manta rendering the Boeing 777 dataset with Phong shaded surfaces and shadows. Resolution 1024x768.

(3 frames per second for a 512x512 image without shadows or approximately 800K rays per second). An example of the rendering (with shadows) can be seen in Figure 9.

Figure 9: Iso-surface rendering of the Visible Female data set (512x512x1734). Bone is shown with rendered shadows.

Time varying data can also be rendered interactively. Figure 10 depicts an iso-surface representing the boundary between two fluids. The fluid in the center is moving quickly through the outside material, developing a turbulent boundary layer that is shown by the iso-surface.

### 6.3 Multi-modal Visualization

Manta is capable of rendering a rich set of primitives in addition to triangles. The Center for the Simulation of Accidental Fires and Explosions (C-SAFE) utilizes ray tracing technology to visualize millions of particles as spheres at interactive rates using specialized acceleration structures. Figure 11 depicts a rendering of C-SAFE data that contains both geometric and volumetric data. Scientists

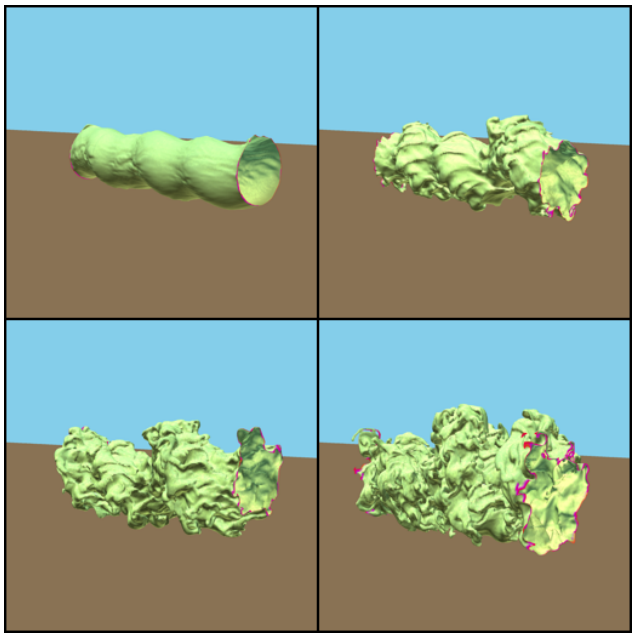

Figure 10: Four time steps from a visualization of an iso-surface. The iso-surface represents the boundary between a fast flowing fluid though the middle of a stationary fluid. Friction between the two materials causes turbulence to occur, which is visible as the time steps progress. As the complexity of the surface increases, so would the time and memory to compute and store the geometry. Using a specialized ray tracing primitive, no intermediate geometry is needed and interactive frame rates for tens of time steps is possible.

can load as many time steps as allowed by main memory (450 time steps for a total of 55 GB of RAM for this case) and can interactively render the data as it is animating. The acceleration structure used also makes it possible to interactively crop the particles by values associated with the particles. Direct volume rendering is facilitated by a specialized data structure that minimizes memory accesses to improve performance. This system first enabled application scientists to view the simulation with both particle data and volumetric fire in the same interactive visualization. Ray tracing also makes it possible to include advanced lighting effects such as ambient occlusion [3].

## 7 Conclusions and Future Work

This paper presented an overview of the Manta software architecture and discussed its design philosophy. Manta's design evolved from Star-Ray, a single ray interactive ray tracer developed in the late nineties. The inflexibility of this system coupled with other developments in interactive ray tracing led Manta's design to focus implementing a high performance interactive ray tracer using wide ray packets in a highly modular software environment.

Manta was originally implemented on SGI Origin and Itanium2 based multi-processor systems. The Origin predated the widespread adoption of SIMD units, and on the Itanium2 these units did not provide a compelling performance improvement. As a result, Manta's implementation experienced a large change when workstations and x86 multi-processor systems with SIMD units became widely available. Ray packet data had to be organized vertically for SIMD instructions while new accessors were added to accommodate legacy horizontal code. Although work on Manta continues, many important application domains can be addressed by the current system.

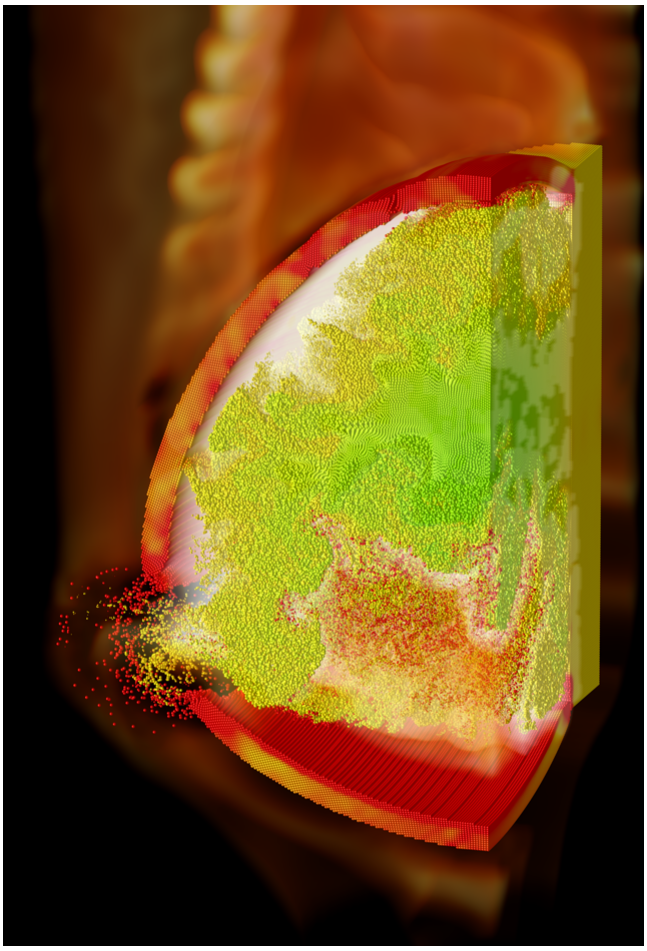

Figure 11: Image of C-SAFE data. The container is composed of 2.8 million particles represented as spheres and color-mapped by temperature. The container is enveloped by pool fire rendered using direct volume rendering of the temperature field (82x322x82). With acceleration structures and data, each time step of data takes 122 MB of main memory. Users are able to interact with the data by changing the color map, cropping particles, and adjusting the transfer function for the volume rendering, all at interactive frame rates. For this application using 15 cores of 8 Opteron 880 processors we achieved approximately 3.6 to 4.8 million rays per second.

Manta is a much more flexible then many single purpose academic renderers. There is a perception in the community that this flexibilty is achieved at the expense of high performance. Preliminary experiments with an implementation of Wald et al.'s dynamic BVH acceleration structure [24] inside Manta, suggest that the performance difference is in the neighborhood of 20 percent. Much of the difference can be attributed to the compiler's inability to optimize across modular component boundaries during intersection and shading in Manta. Other modular parts of Manta's implementation such as the rendering stack, frame buffer access, and image display have produced identical performance characteristics. Quantifying and closing the performance gap between the two systems is a topic of current research.

Today many interactive ray tracing techniques are implemented on single thread processors or small multi-core workstations [18, 9, 20], as the number of available cores increases these renderers will need to adopt a scalable parallelization scheme in order to fully take advantage of new hardware. We propose that interactive ray tracers should be built from the ground up using a parallel pipeline model. The model we describe constrains thread synchronization and has been shown to scale on multi-processor systems in a variety of applications. Further we advocate using wide ray packets to take advantage of special case optimizations, software pipelining, SIMD instructions, and ray coherence to optimize code for current and future processor designs. We also advocate using software-based transactions to maintain consistent state changes in a multi-threaded environment rather than multi-buffering state.

We hope that the discussion of our experience is valuable to others who undertake a similar design task. We also hope that future researchers can provide additional techniques for achieving scalability, flexibility and performance in the context of a modular and maintainable system.

## Acknowledgments

The Manta Interactive Ray Tracer is an open source project and many people have made contributions to it. This work was supported by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions, under grant W-7405-ENG-48, and the Utah Center of Excellence for Interactive Ray-Tracing and Photo Realistic Visualization, and the National Science Foundation.

Hansong Zhang and Rocky Rhodes of Silicon Graphics Inc. contributed an initial kd-tree implementation and funded an intern position during the summer of 2005. Jim Hurley of Intel Corporation funded an intern position during the summer of 2006. Abhinav Dayal, Ben Watson, and David Luebke worked on adaptive frame-less rendering, that stressed the software architecture in new directions. We also acknowledge the members of the Scientific Computing and Imaging (SCI) Institute Rendering Group that have participated in countless discussions and have contributed software and ideas to the Manta architecture, including Solomon Boulos, Deb Ghosh, Christiaan Gribble, Thiago Ize, Tom Johnson, Andrew Kensler, Aaron Knoll, Vincent Pegoraro, Peter Shirley and Ingo Wald.

Jet data courtesy of the University of Chicago flash center, Boeing 777 data courtesy of The Boeing Company via David Kasik, visible female data courtesy of the NIH Visible Human project, and explosion data courtesy of Jim Guilkey, Todd Harman, and the Center for Simulation of Accidental Fires and Explosions.

We would also like to thank our reviewers for their thoughtful and insightful recommendations.

## References

[1] Manta wiki. http://code.sci.utah.edu/Manta.

[2] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 2000.

[3] James Bigler, James Guilkey, Christiaan Gribble, Charles Hansen, and Steven G. Parker. A case study: Visualizing material point method data. In *Proceedings of Euro Vis 2006*, pages 299–306,377, May 2006.

[4] David DeMarle, Steve G. Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG*, pages 87–94, 2003.

[5] D.E. DeMarle, C.P. Gribble, S. Boulos, and S.G. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 31(2):221–242, 2005.

[6] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, Darmstadt, Germany, 2003. Eurographics Association.

[7] Andrew Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989. ISBN 0-12286-160-4.

[8] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[9] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *Technical Report, University of North Carolina at Chapel Hill*, 2006.

[10] B. Minor, G. Fossum, and V. To. TRE : Cell broadband optimized real-time ray-caster. In *Proceedings of GPSx*, 2005.

[11] Michael Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, 1995.

[12] Persistence of Vision Pty. Ltd. Persistence of vision raytracer (version 3.6). [Computer software]., 2004. Retrieved from http://www.povray.org/download/.

[13] Steven G. Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian E. Smits, and Charles D. Hansen. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[14] Steven G. Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Chuck Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, 1999.

[15] Steven G. Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[16] Matt Pharr and Greg Humphreys. *Physically Based Rendering, From Theory to Implementation*. Elsevier Science & Technology Books, July 2004.

[17] Timothy Purcell, Ian Buck, William Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*, pages 703–712, 2002.

[18] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *(Proceedings of SIGGRAPH*, pages 1176–1185, 2005.

[19] Abe Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven G Parker. An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualization*, pages 19–26, 2006.

[20] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. RAZOR: An Architecture for Dynamic Multiresolution Ray Tracing. *University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-21*, 2006.

[21] Shreekant (Ticky) Thakkar and Tom Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, 3(2):8, 1999.

[22] Ingo Wald, Carsten Benthin, Alexander Efremov, Tim Dahmen, Johannes Guenther, Andreas Dietrich, Vlastimil Havran, Philipp Slusallek, and Hans-Peter Seidel. A ray tracing based framework for high-quality virtual reality in industrial design applications. (submitted for publication), 2005.

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

[24] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version). *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023 (conditionally accepted at ACM Transactions on Graphics)*, 2006.

[25] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM SIGGRAPH 2006*, 2006.

[26] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[27] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).