Ray Tracing Visualization Toolkit

Christiaan Gribble* Jeremy Fisher Daniel Eby Ed Quigley Gideon Ludwig
Department of Computer Science
Grove City College

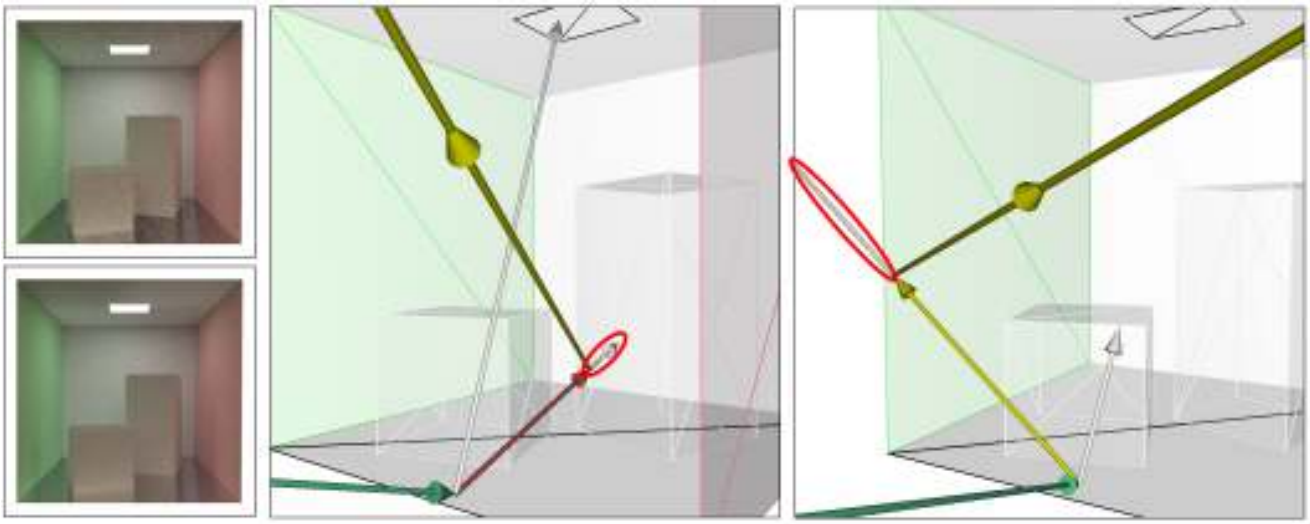

Figure 1: *Ray Tracing Visualization Toolkit. An extensible GUI, together with a ray state recording library and a collection of visualization components, integrate existing client renderers via a flexible plug-in architecture to support visual analysis of ray-based rendering algorithms. Here, rtVTK is used to visually debug an ill-behaved path tracer. The previously undiscovered bug is not obvious in either a low-quality image with relatively few samples per pixel (top left) or a high-quality image with many more samples per pixel (bottom left). However, when viewing the ray state directly with rtVTK (middle, right), the problem is immediately obvious: some shadow rays originating from diffuse surfaces (circled) are not directed toward any light source in the scene. Once identified, the correction was trivial; however, in its previous state, the renderer was incorrect, and likely would have remained so without the ability to visualize the ray tracing process itself.*

Abstract

The Ray Tracing Visualization Toolkit (rtVTK) is a collection of programming and visualization tools supporting visual analysis of ray-based rendering algorithms. rtVTK leverages layered visualization within the spatial domain of computation, enabling investigators to explore the computational elements of any ray-based renderer. Renderers utilize a library for recording and processing ray state, and a configurable pipeline of loosely coupled components allows run-time control of the resulting visualization. rtVTK enhances tasks in development, education, and analysis by enabling users to interact with a visual representation of ray tracing computations.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Stand-Alone Systems; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray tracing; I.6.9 [Simulation, Modeling, and Visualization]: Visualization—Visualization Systems and Software

Keywords: ray-based rendering, ray tracing, visualization

*e-mail: cpgribble@gcc.edu

1 Introduction

We introduce the Ray Tracing Visualization Toolkit (rtVTK), a collection of programming and visualization tools supporting visual analysis of ray-based rendering algorithms. rtVTK comprises a library for recording and processing ray state, together with a flexible software architecture for visualization components, integrated via an extensible GUI. rtVTK enables an investigator to inspect, interrogate, and interact with the computational elements of the ray tracing algorithm itself, thereby promoting a deeper understanding of how computation proceeds.

Our goal is to employ real-time ray tracing for applications in fields as diverse as science, engineering, history, and the arts. Many of these applications require predictive images, or those in which computer-generated results are identical to the photo- and radiometric values obtained by measuring a scene in the physical world. Ray-based rendering algorithms are ideally suited to this task.

Typically, these algorithms simulate the behavior of photons as they interact with objects in an environment according to the laws of geometric optics. These interactions are often very complex, and depend on the spatial arrangement of objects in a scene, their material properties, and the optical effects captured by the particular algorithm in use. Moreover, generating a high-fidelity result requires computing many millions (if not billions) of ray/object interactions. Thus, the complexity encountered in predictive rendering applications limits the practicality of current approaches for real-time image synthesis. Even with recent advances targeting highly parallel platforms [van Antwerpen 2011], many seconds of computation are required for results to converge. As such, rendering predictive images at real-time frame rates is not yet feasible.

Importantly, predictive applications also require that advanced ray-based rendering algorithms be physically correct for the results to be effective—much more so than applications requiring simply plausible or visually convincing results. Here, too, the complexity of scene geometry, material properties, and optical effects used in predictive rendering leads to difficulties in ensuring program correctness: traditional software debugging tools are not designed to leverage the inherent visual nature of computer graphics computation, and thus lead to a cumbersome debugging process.

We believe that effective visualization of ray tracing state will promote deeper understanding of how computation proceeds, addressing a wide variety of problems in ray-based rendering. For example, a subtle and long-standing bug related to secondary ray generation in a batch renderer has been exposed as a result of visualization with rtVTK, as illustrated in Figure 1. Similarly, anecdotal evidence from an undergraduate computer graphics course suggests that students of ray tracing are better able to grasp the algorithm's details by interacting with a visual representation of the computation. Moreover, visualization with rtVTK may enhance tasks in ray tracing performance analysis by enabling insights beyond the summary statistics provided by traditional analysis tools. Finally, because of its flexibility and extensibility, we believe that rtVTK will be well-suited to new, possibly unforeseen, problem domains and application areas as well.

2 Related Work

A large number of systems are designed to visualize various aspects of algorithms or programs, including general run-time behavior [Brown and Sedgewick 1984], advanced CPU state [Stolte et al. 1999], memory system performance [Choudhury et al. 2008; Aftandilian et al. 2010; Choudhury and Rosen 2011], and parallel performance [Nagel et al. 1996; Shende and Malony 2006], among others. However, the existing tools are either too low-level or too specific for our purposes—focusing on particular algorithms, data structures, or hardware components—and do not provide the flexibility we require for tasks in development, education, and analysis.

Surprisingly, very few systems target computer graphics algorithms generally, or ray tracing in particular. One system, described by Russell [1999], is a web-based application that executes as a collection of Java Applets. The system renders a three-dimensional view in the spatial domain of the ray traced image, and shows the state of visibility and secondary rays. In a similar manner, rtVTK visualizes computational elements in the spatial domain of the scene. However, Russell's system is intended primarily for students of ray tracing, and does not offer users a sufficient level of control over the resulting visualization process for our purposes.

Other systems provide more general mechanisms for visualizing computer graphics algorithms, often employing ray tracing for illustrative purposes. Goldman et al. [1996] implement a prototype algorithm animation system that uses the so-called *vector-guided view* to produce visualizations of three-dimensional graphics computation. The work focuses on automatic determination of an ideal view for visualizing such algorithms as a means to address three specific problems: limited perception of movement, object interference, and view scale inadequacy. Whereas overcoming these issues automatically is an important feature for offline visualization, rtVTK addresses these problems via interactivity: online visualization modes give a user comprehensive control over all aspects of the resulting image in real-time—a difficulty for their system, due in part to the lack of necessary processing power on contemporary computing systems.

Briggs and Bergeron [1998] provide an object-oriented environment for rendering in which a library of C++ support classes provide hidden functionality so that objects visualize themselves. With this approach, a programmer can focus on the development of an algorithm using derived classes, while the inherited functionality handles visualization. In particular, objects communicate with a visualization process that executes in parallel with the rendering process and provides an interactive environment in which the user can alter the view of rendering state. rtVTK also leverages interactivity to display rendering state and provides control of the corresponding visual elements. Similarly, rtVTK requires a programmer to instrument client renderers with the calls to a ray state recording API; however, visualization of that state is exposed to the user, which provides a more flexible visualization process.

The AlgoViz environment [Ullrich and Fellner 2004] focuses on visualization of fundamental computer graphics algorithms and geometric modeling concepts. AlgoViz provides a collection of reusable components for common graphics algorithm visualization tasks. Using these components, applications are constructed in a visual programming environment, which simplifies a wide range of algorithm visualization tasks. In a similar spirit, rtVTK utilizes a component-based visualization framework in which each component contributes to the final image. However, rtVTK focuses on ray tracing specifically, giving users finer control of the result by employing algorithm-specific visualization components.

3 Ray Tracing Visualization

Effective visualization of rendering state will promote deeper understanding of ray tracing computations and help to ensure program correctness. However, the sheer number of rays involved in predictive rendering, as well as the intricacies of ray/object interactions, lead to issues with scale and visual clutter. Such issues necessitate a flexible, interactive visualization environment in which users control results at run-time. Additionally, the wide variety of ray-based rendering algorithms have different requirements and, therefore, different features of interest. This issue necessitates a flexible process for recording ray state from within client renderers.

rtVTK satisfies these constraints via interactive visualization coupled with a ray state recording and processing API and an extensible, loosely coupled plug-in architecture. The programming tools and rendering components promote flexibility with user-controlled features including arbitrary ray state payloads and online visualization modes, while an interactive GUI enhances a user's ability to perform debugging and analysis tasks by enabling easier navigation and exploration of the data in real-time. The key components of the rtVTK system architecture are depicted in Figure 2.

3.1 System Architecture

rtVTK provides a ray tracing visualization process that is functional, flexible, and extensible. The design of rtVTK leverages the following concepts:

- **Plug-in architecture.** rtVTK is built around a set of configurable rendering components that follow a specific design pattern to create a flexible infrastructure in which to implement ray tracing visualization. We provide a set of core components to perform common tasks, but the plug-in architecture enables a programmer to create new components and extend the core facilities with arbitrary visualization functionality.

- **Pipelined rendering.** rtVTK uses a pipeline model for rendering, coupled with lazy evaluation for necessary values to avoid recomputation in later stages. The pipeline model leads to a layered visualization approach in which results of individual components are combined, under control of the user and at run-time, to achieve the desired result.

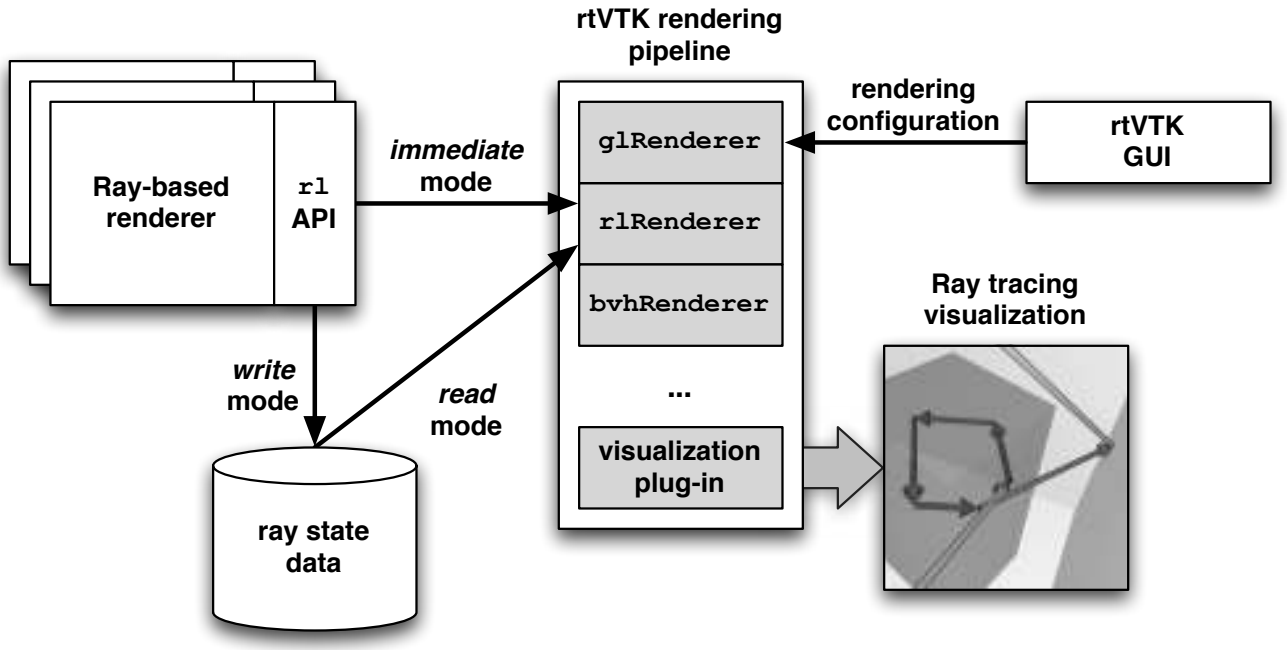

Figure 2: *rtVTK system architecture. Interactive visualization coupled with a ray state recording and processing API and a loosely coupled plug-in architecture creates a functional, flexible, and extensible ray tracing visualization system. A set of core components perform common ray tracing visualization tasks, but the plug-in architecture enables a programmer to extend the core facilities with arbitrary functionality. Rendering components are integrated via an interactive GUI that enables comprehensive control of the entire visualization process.*

- **Extensible GUI.** The visualization components are integrated via an extensible GUI to enable comprehensive control of the entire visualization process. Visualization components can tailor their user interface, exposing rendering parameters in a manner consistent with their functionality. These features provide fine-grained control of the resulting visualization, making rtVTK ideally suited to many visualization tasks.

This design enables layered ray state visualization within the spatial domain of computation. The visualization depicted in Figure 3 illustrates this idea. In this image, the Cornell Box scene is rendered interactively via a GPU path tracer. Using a 2D/3D element composition component, a ray state renderer layers one of the ray trees generated during rendering over the path traced result. Additionally, a BVH visualization component shows the acceleration structure used by the renderer to improve performance. rtVTK enables ray tracing visualizations such as this one by layering results of several components to generate the final image.

3.2 Ray State Recording

Ray state from a client renderer is captured via an API called `rl`. Existing renderers are instrumented with calls to the `rl` engine, and per-ray data—including arbitrary client payloads—are recorded throughout rendering. The resulting state is then explored using rtVTK.

`rl` supports both depth-first and breadth-first ray tree traversal, it enables simultaneous recording of multiple trees, and it is designed to minimize storage requirements for both in-memory and on-disk structures. `rl` currently supports three modes of operation: *write* mode, for clients that capture rendering state for later processing; *read* mode, for data visualization and other post-processing tasks; and *immediate* mode, for online renderers that export the rtVTK plug-in interface. Common high-level operations such as ray tree traversal are implemented easily with a collection of utility functions that aggregate low-level `rl` operations. Currently, `rl` functionality is exposed to client renderers via a procedural, C-style interface, as well as via a wrapper class that provides C++ bindings for seamless integration with object-oriented clients.

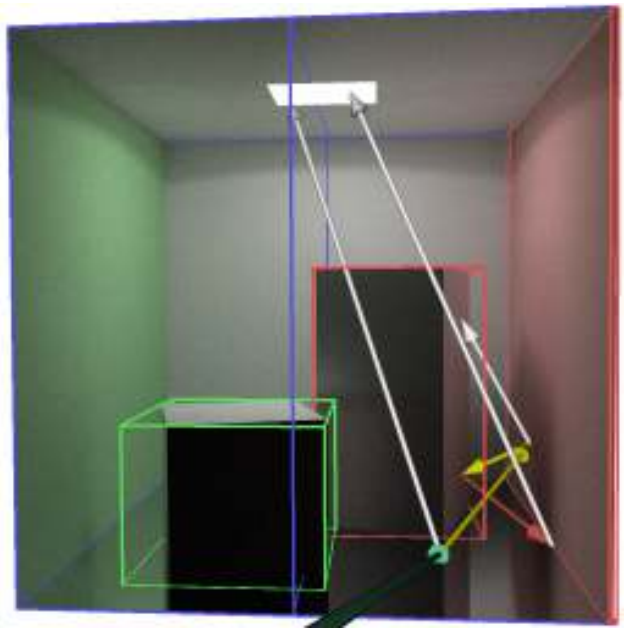

Figure 3: *Layered visualization. The pipeline rendering model adopted by rtVTK enables sophisticated ray tracing visualization by layering results of several components to generate the final image. In this example, an interactive path tracer generates an image of the computational domain, while the 2D/3D element composition, ray state, and BVH visualization components combine to render the other visual elements present in the final result.*

rl pseudocode

```
// loop over pixels
for (uint y = 0; y < height; ++y)
  for (uint x = 0; x < width; ++x)
    // generate visibility ray and trace
    rlBeginTree(x, y);
    trace(visibilityRay, ... );
    rlEndTree();

trace(const Ray& r, ... )
  // perform ray tracing computations
  //   and recurse
  rlAddRay(r.o, r.d, r.t, ray.type,
           &my_data, sizeof(MyData));
  rlDescendTree();
  trace(nextRay, ... );
  rlAscendTree();
```

Figure 4: *Basic `rl` example. This pseudocode demonstrates the use of `rl` functionality in a recursive ray tracer. In write mode, the ray state is written to disk for later processing, including visualization with rtVTK. In immediate mode, ray state is recorded to memory, and the visualization pipeline layers ray state primitives on top of the computational domain.*

Write mode. Using `rl` *write* mode, a client renderer can capture ray state to disk in a compact file format for later processing. As illustrated in Figure 4, client renderers simply instrument their core ray tracing functionality with calls to a small collection of straightforward functions in the `rl` API.

In particular, `rl` state is initialized to prepare the in-memory and on-disk data structures necessary to capture a session. To record a ray tree, a renderer simply invokes `rlBeginTree` with the current pixel coordinates. Then, for each ray, `rlAddRay` records common properties and arbitrary per-ray data, which enables a renderer to extend the basic `rl` state to suit its needs. In depth-first mode, the tree is traversed using `rlDescendTree` and `rlAscendTree`, as in Figure 4.

Read mode. To process previously recorded ray state, `rl` client applications utilize *read* mode. As in write mode, a small collection of straightforward calls are used to modify the `rl` engine state. For example, `rlReadTree` returns data for the current tree, while `rlReadRay` returns the current ray data and automatically advances to the next ray. Calls to reset read state, to seek among and within currently loaded trees, and to query various statistics are also available.

Immediate mode. Any ray-based renderer that exposes the rtVTK plug-in interface can interact with rtVTK via `rl` immediate mode. Using a special keyboard and mouse control sequence (for example, *ctrl-click*), rtVTK instructs such a renderer to generate the ray tree corresponding to a particular pixel. The client captures rendering state using `rl` write mode functions, which record the data to in-memory structures. Finally, rtVTK renders the recorded state for a comprehensive, online view of the ray tracing process.

3.3 Visualization Facilities

The combination of a plug-in architecture and pipelined rendering enable layered visualization of ray tracing state. The configurable pipeline enables users to create many different visualizations simply by modifying the order in which components execute. Moreover, lazy evaluation in the pipelined model enables efficient implementation by avoiding recomputation of necessary values.

Figure 5: *Rendering scene geometry. The `glRenderer` plug-in provides a view of the computational domain, using wireframe (left), alpha-blended (middle), or opaque (right) surfaces. Additional visualization elements can then be layered over these results.*

Visualization plug-ins. Existing renderers and new visualization components integrate by exporting the rtVTK rendering API:

```
// Core rendering functionality
void idle();
void render();
void resize(uint, uint);
void traceRay(uint, uint);
```

These functions serve to synchronize the plug-ins' operation with each other and with the rtVTK application. For example, `render` updates the component's rendering state in response to interactive changes by the user—including mouse motion and changes to other parameters exposed via the GUI—and re-renders its target.

Some components, such as a path tracer, may require several passes to render their results. In these cases, performance would suffer by having other components re-render from scratch when the rendering state to which they respond has not, in fact, changed. To overcome this issue, a component can implement the `idle` function to update its rendering state and (possibly) its output, without requiring invocation of the `render` function for the entire pipeline.

The `traceRay` function implements direct support for `rl` immediate mode as discussed above, and is invoked in response to a particular keyboard and mouse control sequence. This function generates a ray tree through the corresponding pixel and updates the `rl` immediate mode structures for online visualization.

In our experience, the interface outlined above enables visualization of a wide range of computational elements, including scene geometry, ray state, and acceleration structures. Additional functionality can be exposed via the GUI, enabling fine-grained control of a plug-in's run-time behavior.

The core rtVTK visualization components include a basic rasterization component for scene geometry, an `rl` component for ray state, and a 2D/3D element composition component. We also implement several additional plug-ins to demonstrate the flexibility of rtVTK, including an interactive GPU path tracer, a single ray CPU path tracer, and a BVH visualization component.

Scene geometry. A key feature of the rtVTK approach is that visualization occurs in the spatial domain of the actual computation. As such, we require the ability to render a view of the scene geometry, as well as to explore interesting features of the scene interactively. Toward this end, a basic OpenGL viewer, called `glRenderer`, renders the geometry. This component exposes a number of options to the user, including the three surface rendering modes depicted in Figure 5.

`glRenderer` forms the basis for most interactive visualization tasks; however, more advanced visualization components, such as the interactive path tracer discussed below, can be used to visualize the computational domain as well.

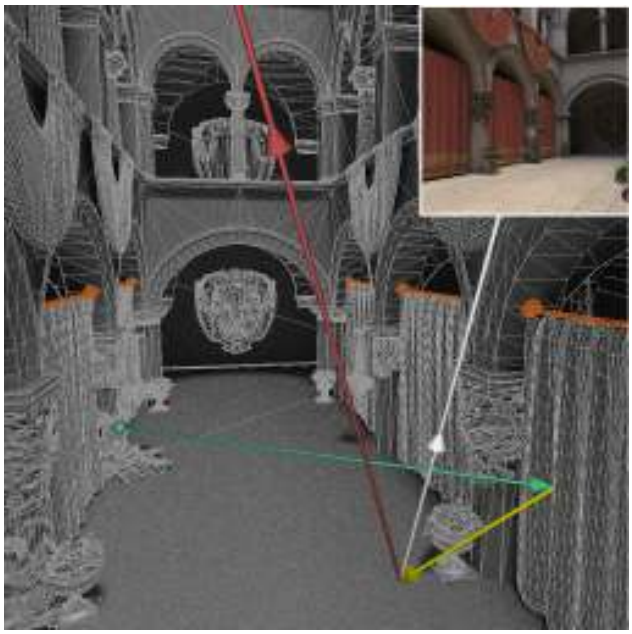

Figure 6: *Ray state visualization. Existing renderers integrate with rtVTK by exporting the rendering plug-in interface. Here, a path tracing plug-in generates a high-quality image of the Crytek Sponza scene (inset). A ray tree generated in the original view is then explored interactively:* `glRenderer` *layers a wireframe view of the scene geometry over a path traced result, while* `rlRenderer` *adds ray state visualization. (Textures disabled for clarity.)*

Ray state. We implement a visualization component specifically for ray state gathered by a client renderer utilizing the `rl` API. This component, called `rlRenderer`, exposes a number of options, including the ability to navigate among ray trees; to selectively filter rays by type and depth; to control the visual properties of ray state primitives; and, in `rl` read mode, to animate the ray tracing process using timed events provided by the GUI.

`rlRenderer` is a core component of rtVTK: it reveals the ray tracing process using a visual representation of rendering state, as depicted in Figure 6. In read mode, `rlRenderer` renders previously recorded state, allowing users to review that state by seeking forward or backward using GUI controls. `rlRenderer` also collaborates with the `rl` engine to display ray state generated by an online client renderer using immediate mode.

2D/3D element composition. Ray tracing components, such as the path tracers highlighted below, render a two-dimensional image, and depth information is not typically retained. As a consequence, layering other visual elements in a manner that retains proper depth relationships is problematic. One way to resolve this issue is simply to force client renderers to retain per-pixel depth values in an auxiliary buffer. However, this approach is unnecessarily intrusive, imposing potentially significant modifications to such renderers. Instead, we implement three-dimensional composition over a two-dimensional image by rasterizing a depth-only version of the scene [Wachowicz 2011]. The output of this component gives sufficient information to resolve depth relationships between ray traced results and other, typically rasterized, visual elements.

This component, the pipelined rendering model, and lazy evaluation are critical to the layered visualization approach. When combined with the components described below—and indeed with any components a user might develop—rtVTK can be used to gain a deeper understanding of computational elements beyond ray state alone.

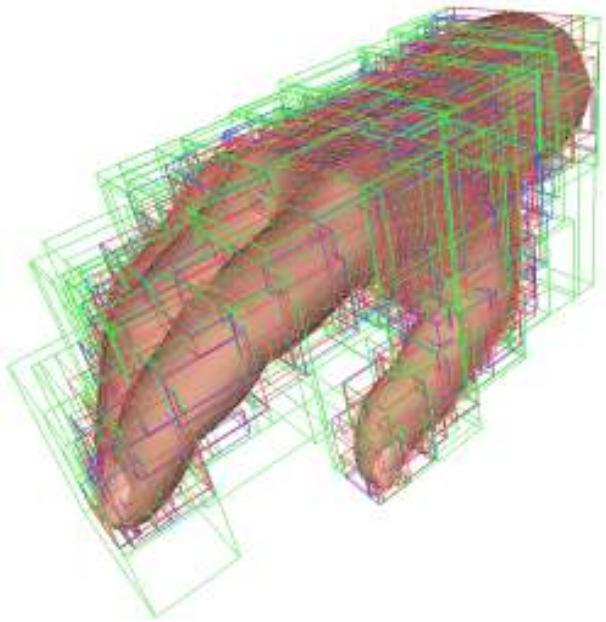

Figure 7: *BVH visualization. The flexibility of the rtVTK plug-in interface enables advanced features such as acceleration structure visualization. Here, bounding volumes in levels 13-15 of the BVH used to generate this image are layered over a ray traced result using the* `bvhRenderer` *component.*

Interactive path tracing. Using OpenCL, we implement an interactive path tracing component that supports triangle-based models, uses a BVH with stackless traversal [Hapala et al. 2011] for improved performance, and integrates `rl` immediate mode functionality. Although the current implementation is essentially a brute-force method, performance on modern GPUs already exceeds ten frames per second for simple scenes rendered with several important visual effects, including glossy reflection and transparency. More importantly, as shown in Figure 3, this component demonstrates the flexibility of the pipelined rendering model and layered visualization approach employed by rtVTK.

Batch-mode path tracing. Similarly, we integrate an existing CPU batch-mode path tracer by wrapping its core functionality in the rtVTK plug-in interface. This renderer is a simple recursive single ray system that uses a BVH [Wald et al. 2007] for improved performance and is instrumented to support `rl` immediate mode according to the pattern outlined in Figure 4. Various GUI elements are exposed to control its configuration, demonstrating that an existing renderer can be integrated with relative ease.

BVH visualization. We also implement a component to display a BVH over the current geometry, as shown in Figure 7. This component, called `bvhRenderer`, exposes options for controlling the visual appearance of BVH nodes, including the starting and ending levels of the hierarchy for which bounding volumes are rendered.

The `traceRay` function of this component is currently unused; however, one could imagine an implementation that interacts with an online client renderer to visualize only those BVH nodes that are visited by the rays in a particular tree, for example. In this case, even the seemingly unnecessary functionality of `traceRay` can be leveraged for advanced ray tracing visualization results. In addition to its primary function, this component demonstrates the advanced visualization features enabled by rtVTK's flexible design.

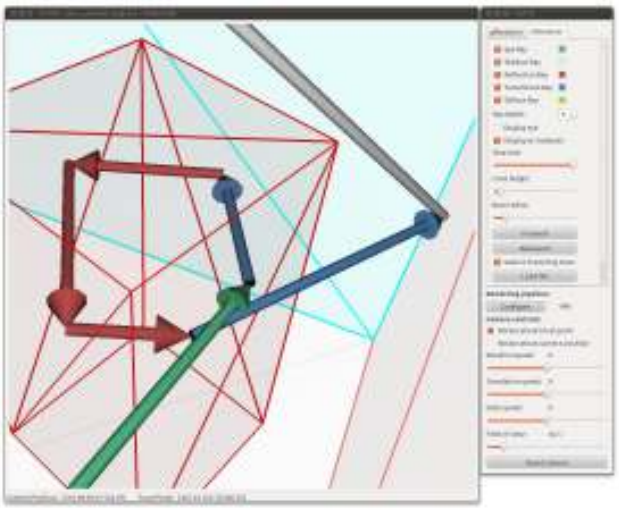

Figure 8: *rtVTK GUI. A render window (left) provides interaction with the visualization, while an application window (right) provides GUI elements for controlling rtVTK itself. To expose a larger set of functionality, each rendering component dynamically populates a tab with appropriate GUI elements, permitting fine-grained, run-time control of the resulting visualization.*

3.4 Graphical User Interface

The visualization components described above are integrated via an extensible GUI that provides interactive control of the visualization process. As illustrated in Figure 8, a render window provides interaction with the visualization—including control of viewing parameters, image resolution, and so forth—while an application window provides GUI elements for controlling rtVTK itself. In addition, the application window acts as a liaison among pipeline components and the render window, connecting exposed functionality of the corresponding objects at run-time.

Additional features include timers that can be connected to the visualization components, enabling a wide range of advanced functionality. For example, rather than navigate a collection of ray trees manually, a user can instead connect a timer event to the *Advance tree* functionality exposed by `rlRenderer`. When triggered, the event invokes the registered function at user-defined intervals, providing an animated view of the entire ray tracing process.

4 Applications

The primary goal of rtVTK is to provide tools supporting visual analysis of ray-based renderers, thereby enabling deeper understanding of how computation proceeds. We highlight the utility of ray tracing visualization generally, and rtVTK specifically, with two example applications:

- discovering a previously unknown bug in a batch renderer based on Kajiya-style path tracing [Kajiya 1986], and
- teaching recursive ray tracing [Whitted 1980] in an undergraduate computer graphics course.

rtVTK enhances these tasks by enabling users to identify and explore the salient features of the ray tracing algorithm interactively. We also highlight the potential role of rtVTK in performance analysis tasks, for example, as a means to identify and exploit ray coherence on current and future hardware architectures.

Code development. While code development ideally refers to creating new functionality, the ability to debug ill-behaved implementations is an extremely important consequence of ray tracing visualization, particularly for the predictive rendering applications in which we are interested: in this context, results must be correct if they are to be effective.

Debugging graphics algorithms is an inherently visual process, but traditional software debugging tools are not designed to leverage this characteristic. As a result, diagnosing ray tracing bugs is tedious at best, and extremely difficult at worst. However, by visualizing rendering state in the spatial domain of computation, rtVTK provides a clear advantage over traditional debugging tools.

We have used rtVTK to identify, and then correct, a previously unknown bug related to secondary ray generation in an offline path tracer. The bug remained undiscovered until the ray state, recorded with `rl`, was used to test the visualization components of rtVTK. As illustrated in Figure 1, the bug, which under certain conditions generates incorrect directions for shadow rays, is not noticeable in either a low-quality image with relatively few samples per pixel or a high-quality image with many more samples per pixel. However, when viewing the ray state directly, the problem is immediately obvious: the circled shadow rays are not directed toward any light source in the scene. Once identified, the correction was trivial; however, in its previous state, the renderer was incorrect, and likely would have remained so without the ability to interact with the computational elements in the spatial domain of the scene.

Education. Algorithm animation has long been thought to help students' understanding of how a problem is solved. As noted in Section 2, several existing systems have been designed with computer science education in mind. In this context, rtVTK helps students to better understand how recursive ray tracing works.

Anecdotal evidence from an undergraduate computer graphics course indicates that students appreciate the ability to interact with visual representations of key computational elements. In particular, the interactive environment provided by rtVTK allows students to quickly identify optimal views in which the most pertinent features of the ray tracing process are visible.

For example, Figure 3 clearly illustrates the recursive ray tracing algorithm. Initially, a visibility ray (green) intersects the glossy floor. The shader spawns a shadow ray (white) and a diffuse reflection ray (yellow) for direct and indirect illumination, respectively. This process continues by generating additional shadow rays and reflection rays (diffuse $\rightarrow$ yellow; specular $\rightarrow$ red) until the termination criteria are met. Interestingly, we observe that several rays intersect the bounding volume of the tall box and are thus tested against the enclosed geometry. However, not all of these rays actually intersect the box, which results in false positives. These unnecessary tests are are an important consequence of using axis-aligned bounding volumes in a BVH and represent a potential source of inefficiency with this implementation choice.

Similarly, Figure 9 clearly depicts the complicated ray behavior associated with total internal reflection in dielectric materials. In this version of the Cornell Box scene, the visibility ray (green) refracts into the tall glass box (transmission ray $\rightarrow$ blue) and undergoes several total internal reflection events (reflection ray $\rightarrow$ red) before finally escaping the medium and emitting a shadow ray (white) to compute the illumination. By exploring the resulting state interactively, students can better understand the complexities arising from ray/object interactions such as the one depicted here.

Moreover, the ability to animate the process—to literally watch the generation and tracing of rays throughout an environment—helps students to understand implications of design decisions such as pixel traversal order. Likewise, the ability to see the ray distributions of various reflectance models gives students a concrete rep-

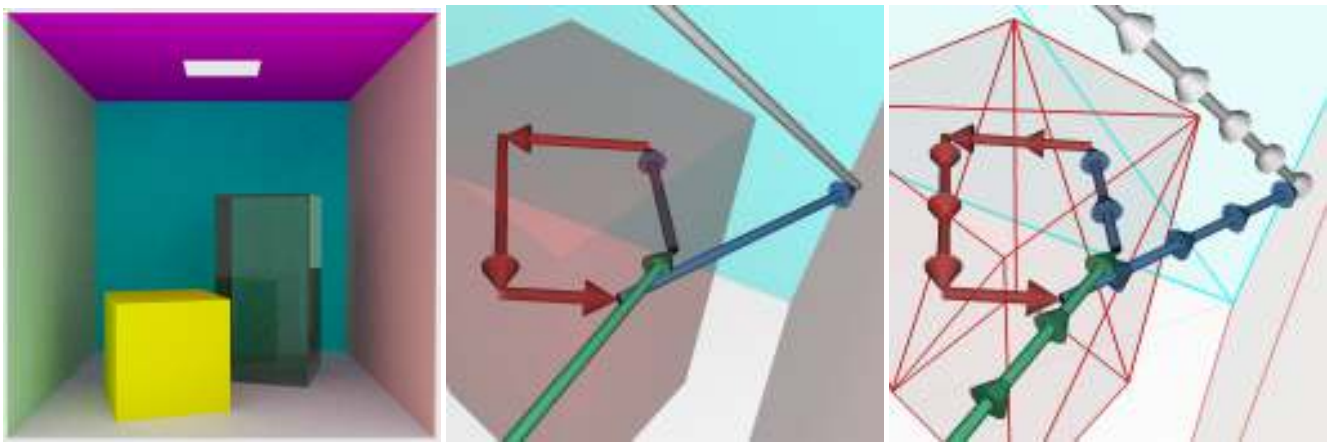

Figure 9: *Understanding recursive ray tracing. rtVTK integrates a batch-mode path tracer to render a version of the Cornell Box scene (left). The resulting state is then explored interactively, enabling run-time control of the resulting visualization (middle, right). Here, rtVTK reveals interesting behavior along the depicted path: rays undergo several total internal reflection events before finally escaping the medium.*

resentation of the abstract physical and mathematical concepts behind ray tracing. Additionally, `rl` immediate mode, together with the simple but powerful rtVTK plug-in interface, enables students to design, implement, and debug their own online client renderers, enabling visualization of not only scene geometry, but the ray state used to generate images as well.

Performance analysis. Over the past several years, numerous techniques based on coherent ray tracing [Wald et al. 2001] have become attractive alternatives for interactive rendering [Benthin 2006; Boulos et al. 2007; Günther et al. 2007; Wald et al. 2007; Boulos et al. 2008; Overbeck et al. 2008]. Identifying and tracing coherent rays is critical to achieving high performance with these techniques. However, understanding the behavior of rays, either individually or in aggregate, is confounded by the sheer number of rays involved. Additionally, other properties, such as nodes visited during traversal of an acceleration structure; geometry tested for intersection; textures queried during shading; and indeed the execution paths for traversal, intersection, and shading, are often shared among many rays, even those that are not spatially similar. These properties have been used to improve performance in various ray-based renderers [Pharr et al. 1997; Navratil et al. 2007; Aila and Karras 2010], so any use of ray coherence as a means to improve performance must also include such properties.

Reasoning about coherence, particularly non-spatial coherence, can be difficult: intuition alone can fail to identify all such properties, and trial-and-error is time-consuming and often ineffective. An approach that enables programmers to define, extract, and exploit coherence [Gribble and Ramani 2008] may help, but such an approach is only beneficial if the anticipated coherence actually exists.

Although a comprehensive analysis of ray coherence is beyond the scope of this work, our hope is that ray tracing visualization will succeed where intuition may fail. Just as rtVTK enhances debugging and learning tasks, we believe that visualization with rtVTK may expose new opportunities to exploit ray coherence in the broadest sense, further demonstrating the utility of ray tracing visualization across a variety of problems in computer graphics.

5 Conclusions and Future Work

The Ray Tracing Visualization Toolkit provides an interactive visualization environment coupled with a flexible, extensible system architecture to create effective visualizations of ray tracing state. In particular, layered visualization within the spatial domain of computation enables users to visualize elements of any ray-based rendering algorithm, while a configurable pipeline allows users to control the resulting visualization at run-time. Moreover, rtVTK enhances tasks in ray tracing development, education, and analysis by enabling users to visually identify and explore key computational elements of the algorithm.

The current bottleneck in the rtVTK visualization process is data collection for offline ray-based renderers, such as the one described in Section 4. Although `rl` is explicitly designed to minimize the storage required by in-memory and on-disk structures, the resulting state still consumes significant storage for predictive image synthesis applications. Using data compression techniques and an appropriate on-disk file structure that enables random access to both ray tree metadata and actual ray state may alleviate data management problems imposed by visualization for a wide range of applications.

The extensibility of the rtVTK programming tools and visualization components enables a number of techniques not yet implemented. Of particular interest is the addition of a component that provides a so-called *source code orientation* [Brown and Sedgewick 1984; Choudhury et al. 2008], wherein lines of code corresponding to current ray tracing operations are highlighted. Such a component may better orient a user within the overall ray tracing algorithm by combining the familiar environment of traditional software debuggers with the visual process enabled by rtVTK.

Finally, immediate mode visualization of ray state generated by client renderers is likely to be of value in many development, learning, and analysis tasks. As such, even the relatively simple requirements that client renderers expose the rtVTK plug-in interface and utilize `rl` to record ray state may be unnecessarily intrusive. Methods that increase the transparency of collecting, storing, processing, and ultimately analyzing ray state from arbitrary ray-based renderers are of particular interest.

Its role as a visual debugger for advanced ray-based rendering algorithms that must be physically correct to be effective, and as an educational tool for new generations of graphics students, make rtVTK an important addition to the collection of tools supporting modern computer graphics research. In addition, we expect that the flexibility of rtVTK as a framework for ray tracing visualization will permit application to more advanced rendering and visualization problems—for example, real-time Monte Carlo path tracing and meaningful representations for large quantities of ray tracing state. Using interactivity, rtVTK enables both thorough and rapid investigation of problems such as these.

Acknowledgments

This work was funded by grants from the II-VI Foundation and the Grove City College Swezey Research Fund. The GPUs used in this research were donated by NVIDIA through their Professor Partnership Program. We are indebted to A.N.M. Imroz Choudhury (SCI Institute, University of Utah) and the anonymous reviewers for insightful and detailed comments regarding this work.

References

AFTANDILIAN, E. E., KELLEY, S., GRAMAZIO, C., RICCI, N., SU, S. L., AND GUYER, S. Z. 2010. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th ACM Symposium on Software Visualization*, 53–62.

AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of High Performance Graphics 2010*, 113–122.

BENTHIN, C. 2006. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., WALD, I., AND SHIRLEY, P. 2007. Packet-based Whitted and distribution ray tracing. In *Graphics Interface 2007*, 177–184.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive ray packet reordering. In *2008 IEEE/Eurographics Symposium on Interactive Ray Tracing*, 131–138.

BRIGGS JR., E. S., AND BERGERON, R. D. 1998. A self-visualizing rendering support environment. *Computers and Graphics 22*, 4, 547–555.

BROWN, M. H., AND SEDGEWICK, R. 1984. A system for algorithm animation. *Computer Graphics 18*, 3, 177–186.

CHOUDHURY, A.N.M. I., AND ROSEN, P. 2011. Abstract visualization of runtime memory behavior. In *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 22–29.

CHOUDHURY, A.N.M. I., POTTER, K. C., AND PARKER, S. G. 2008. Interactive visualization for memory reference traces. *Computer Graphics Forum 27*, 3 (May), 815–822.

GOLDMAN, D. A., ECKERT, R. R., AND COHEN, M. S. 1996. Three-dimensional computation visualization for computer graphics rendering algorithms. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, 358–362.

GRIBBLE, C. P., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *2008 IEEE/Eurographics Symposium on Interactive Ray Tracing*, 59–66.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*, 113–118.

HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. 2011. Efficient stack-less BVH traversal for ray tracing. In *27th Spring Conference on Computer Graphics (SCCG 2011)*.

KAJIYA, J. T. 1986. The rendering equation. In *Siggraph 1986*, 143–150.

NAGEL, W. E., ARNOLD, A., WEBER, M., HOPPE, H.-C., AND COLCHENBACH, K. 1996. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12*, 1, 69–80.

NAVRATIL, P., FUSSELL, D., LIN, C., AND MARK, W. R. 2007. Dynamic ray scheduling for improved system performance. In *2007 IEEE Symposium on Interactive Ray Tracing*, 95–104.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time Whitted ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 41–48.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics 31*, Annual Conference Series, 101–108.

RUSSELL, J. A., 1999. An interactive web-based ray tracing visualization tool. Undergraduate Honors Program Senior Thesis, Department of Computer Science, University of Washington.

SHENDE, S. S., AND MALONY, A. D. 2006. The Tau parallel performance system. *International Journal of High Performance Computing Applications 20*, 287–331.

STOLTE, C., BOSCH, R., HANRAHAN, P., AND ROSENBLUM, M. 1999. Visualizing application behavior on superscalar processors. In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, 10–17.

ULLRICH, T., AND FELLNER, D. W. 2004. AlgoViz – a computer graphics algorithm visualization toolkit. In *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications*, 941–948.

VAN ANTWERPEN, D. 2011. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proceedings of High Performance Graphics 2011*, 41–50.

WACHOWICZ, P. 2011. *Accelerating Photon Mapping with Photon Flipping and Invalidity Photons*. Master's thesis, University of Amsterdam.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (September), 153–164.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (January), 6.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.