Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

GLuRay: Ray Tracing in Scientific Visualization Applications using OpenGL Interception

Carson Brownlee[1,2], Thomas Fogal[2], and Charles D. Hansen[1,2]

[1]School of Computing, University of Utah
[2]SCI Institute, University of Utah

Abstract

Ray tracing in scientific visualization allows for substantial gains in performance and rendering quality with large scale polygonal datasets compared to brute-force rasterization, however implementing new rendering architectures into existing tools is often costly and time consuming. This paper presents a library, GLuRay, which intercepts OpenGL calls from many common visualization applications and renders them with the CPU ray tracer Manta without modification to the underlying visualization tool. Rendering polygonal models such as isosurfaces can be done identically to an OpenGL implementation using provided material and camera properties or superior rendering can be achieved using enhanced settings such as dielectric materials or pinhole cameras with depth of field effects. Comparative benchmarks were conducted on the Texas Advanced Computing Center's Longhorn cluster using the popular visualization packages ParaView, VisIt, Ensight, and VAPOR. Through the parallel rendering package ParaView, scaling up to 64 nodes is demonstrated. With our tests we show that using OpenGL interception to accelerate and enhance visualization programs provides a viable enhancement to existing tools with little overhead and no code modification while allowing for the creation of publication quality renderings using advanced effects and greatly improved large-scale software rendering performance within tools that scientists are currently using.

Categories and Subject Descriptors (according to ACM CCS): K.6.1 [Computer Graphics]: Petascale visualization, Flow visualization, Space visualization—

1. Introduction

Massively parallel ray tracing has proven to be an effective method for rendering large polygonal models which scales well on increasingly parallel architectures [WSB01]. Ray tracing implementations are rare in scientific visualization programs, which instead often rely on brute-force OpenGL implementations for rendering. As such, they often scale poorly and become sub-interactive with increasing geometry counts. Furthermore, many common scientific visualization programs do not provide enhanced rendering effects such as indirect lighting which give better approximations to the rendering equation than direct lighting alone and provide better depth cues to users and more realistic images. To achieve such effects, scientists must use custom visualization software which provides ray tracing capabilities [PSL*98, BSP06]. End users are reluctant to learn new tools which are non-typical in their workflows. Integrating ray tracing into common visualization tools through custom programming can offer faster and higher quality renderings for scientists in the same programs used for data exploration and analysis, however creating and maintaining such integrated renderers across multiple programs and versions is often a daunting task for developers. We therefore implement ray tracing across many existing visualization programs by intercepting OpenGL calls and instead of rasterizing the scene, we render those calls with the software ray tracer Manta [BSP06]. Our approach is therefore largely program agnostic for any visualization application using the fixed function OpenGL pipeline and has been tested to work with ParaView, VisIt, VAPOR, and EnSight as of the time of this writing [CGM*06, LLN10, UCA09, CEI10].

Achieving interactive performance in scientific visualization is an important goal for data exploration and analysis. When rendering on machines lacking graphics hardware acceleration, programs such as VisIt and ParaView use single-threaded Mesa rendering by default which often fails to achieve interactive performance even for modestly sized polygonal models. Furthermore, even when specialized rendering clusters are used with graphics processing units, sub-interactive performance is often observed for large data sizes due to a lack of occlusion culling in many programs. To improve performance, GLuRay implements acceleration structures to achieve logarithmic performance scaling with geometry sizes while the ray tracing algorithm gains occlusion and frustum culling implicitly through the nearest-hit algorithm. As many of the programs tested use brute-force algorithms and our implementation is a translation between rasterization and ray tracing, this paper is not meant solely to compare the performance of ray tracing and rasterization, but rather a real-world solution with existing tools which do not necessarily present ideal rasterization implementations. The main contribution of this paper is the presentation of a ray tracing implementation using OpenGL interception with interactive performance and advanced rendering effects for scientific visualization applications. Scalability using the parallel rendering tool ParaView is demonstrated across 64 nodes of a cluster.

In this paper we describe previous work with massive model rasterization and ray tracing in distributed systems in Section 2. The implementation of GLuRay is then given in Section 3 describing OpenGL interception, asynchronous rendering, and generating high quality images. To understand the trade-offs of our system for dealing with extremely large datasets, we have employed an in-depth timing study in Section 4 of three different rendering methods for large polygonal data: software-based ray tracing, software-based rasterization, and hardware-accelerated rasterization. We use four different datasets: one synthetic, and three scientific. Through these studies we show that our system can handle large datasets of various types across multiple applications with vastly improved interactive rendering times, scalability, and enhanced quality over their built-in rendering engines. Finally, the limitations of the system and future work are discussed in Section 5.

2. Background

Many tools have been developed for rendering large-scale scientific data using polygonal representations. VAPOR was developed as visualization tool by Clyne et al. for atmospheric scientists to explore data using isosurfaces, streamlines and volumetric representations [UCA09]. Programs such as ParaView, VisIt and EnSight present visualization solutions across large heterogeneous cluster environments for data which are often too large to fit or be rendered efficiently on a single node [Kit10, LLN10, CEI10]. Many of these systems utilize a client/server architecture with a single client interface utilizing multiple server nodes made up of render or data servers for rendering and analysis.

Data is commonly distributed in a data-parallel fashion in large distributed memory systems. This balances both the data loading and processing but also the rendering work which is composited together using sort-last compositing algorithms to combine each node's portion of the rendered scene. ParaView redistributes work further by sending generated geometry to render servers in an additional step in cases where fewer rendering nodes may be needed compared to analysis or vice-versa [CGM*06]. Once geometry has been distributed, rendering relies on hardware acceleration to achieve interactive frame rates and often defaults to Mesa for software rasterization on machines lacking graphics processing hardware. Single-threaded Mesa performance fails to utilize increasing numbers of cores on cluster nodes or large distributed memory systems.

Mitra and Chiueh [MC98] developed a parallel Mesa software rasterization implementation by running Mesa in parallel in the background through a serial interface. In order to provide scalability they utilized compositing operations for each running instance of Mesa. Nouanesengsy et al. [NAWS11] explored current performance of Mesa software rasterization on large shared memory machines using various compositing methods. Their tests showed that nearly-linear speedups could be achieved with the number of threads using a hybrid sort-first and sort-last compositing step, however running multiple instances of ParaView by spawning additional MPI processes failed to scale well in their tests. Howison et al. [MHC10] found that running with a single MPI process with 6 threads in a hybrid parallelism setup was significantly faster on a 12 core node than an MPI only approach. We therefore focus our testing with a single process scaling over multiple threads for all tests to demonstrate the per-process optimizations of ray tracing. Additionally, a scaling study is presented which demonstrates GLuRay performance scaling over multiple nodes with ParaView using data-parallel work distribution.

Ray tracing performance has been shown to be embarrassingly parallel [WSB01]. This makes it an ideal algorithm for software rendering on compute clusters with increasing core counts per node. Current implementations of ray tracing for scientific visualization employ a custom rendering framework as opposed to using standard visualization tools [BSP06, PSL*98, PPL*99]. Marsalek et al. [MDG*10] recently implemented a ray tracing engine into a molecular rendering tool for the purpose of generating advanced rendering effects. Their implementation showed real-time performance up to 56 thousand triangles on a single machine but it is unknown how well it performs with millions of triangles, how it compares with OpenGL performance, and was only implemented for a single application.

The Manta real-time ray tracing software has proven scaling performance on shared-memory machines [SBB*06]. Thus, combining CPU rendering using Manta with other cluster based visualization tools which handle data distribution, geometry creation, and image compositing provides a working distributed rendering solution. Ize et al. [IBH11] developed a version of Manta which ran in real-time over distributed shared memory systems by using a paging scheme to swap data between nodes and we implemented a similar scheme using replicated data, however we chose to focus this paper on using existing tools' data distribution and image composition. Parker et al. developed RTSL [PBBR07], a shading language for Manta which was largely a superset of GLSL. This provides the potential to translate GLSL shaders for use with GLuRay, however this was not explored for this paper. Wald et al. introduced a ray tracing API called OpenRT which gave an API interface similar to OpenGL and was shown to scale well using commodity PC clusters [WBDS03]. Instead of creating another API which requires developers to rewrite their visualization implementation however, we intercept OpenGL API calls from existing programs.

OpenGL interception is a common method used for debugging and profiling OpenGL applications with programs such as glTrace [SM97]. WireGL and Chromium took OpenGL interception a step further by modifying the behavior of the OpenGL calls into a stream processing framework to support functions such as distributed rendering with sort-first and sort-last compositing operations [HEB*01, HHN*02]. The main contribution of this paper is the presentation of a ray tracing implementation using OpenGL interception with interactive performance and advanced rendering effects for scientific visualization applications.

3. Ray Tracing OpenGL

GLuRay operates as a ray tracer which runs with existing OpenGL programs. Implementing GLuRay required creating a false OpenGL library and dynamically linking it with a host program at run-time using LD_PRELOAD or dlopen. This library maps calls from the rasterization algorithm present in OpenGL into a ray tracer.

3.1. Intercepting OpenGL calls

In order to capture function calls, an OpenGL implementation was created based on the official OpenGL specification. The OpenGL debugging tool SpyGLass was used as a basis for the program [Mag11]. Some calls are ignored with no direct mapping such as clearing the depth buffer, some are passed on to an actual OpenGL implementation such as GLX calls, and some are sent to GLuRay's ray tracing implementation for tracking state or rendering. Function calls which are mapped to ray tracing include calls to modify transformation matrices, material parameters, light properties, geometry information, and rendering attributes. Calls which are passed to another OpenGL specification include calls such as glDrawPixels and glXSwapBuffers. When tracking these calls, it helps to think of OpenGL as a state machine. Each call either affects some given state or returns information about the given state. In OpenGL, this state affects how a polygon is drawn each time a draw call is made using either glVertex calls in immediate mode, glCallList for display lists, or a more modern glDrawArrays or equivalent call. When ray tracing, multiple draw calls need to be avoided as much as possible as the rendering time is $k \cdot O(log(n))$, where n is the amount of geometry and k is the screen size. For each draw call, the time complexity roughly linearly increases as k expands. Therefore each draw call is recorded and not rendered until the system determines a draw is required for the entire scene. This is determined through calls to glXSwapBuffers, glFlush, glFinish, or glClear depending on the application. This has the potential to break certain behaviors such as depth-ordered blending modes, however common uses of this are for transparency which can be handled by using transparent material properties with ray tracing and has not posed a problem for our current implementations.

In the serial implementation of GLuRay, rendering occurs as soon as a draw is required. Acceleration structures are built as needed which are instanced with their transforms, lighting information, material parameters, and geometry. When a render is requested by the host program, the rendered scene is drawn into the OpenGL context, and data is cleared as shown in Figure 1.

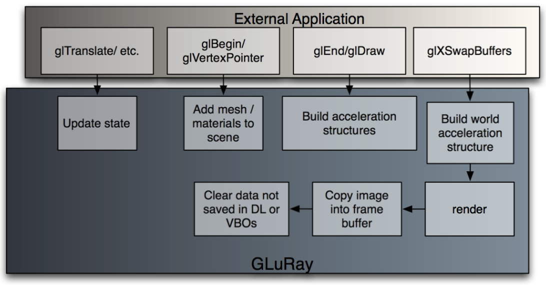

External Application
glTranslate/ etc.
glBegin/ glVertexPointer
glEnd/glDraw
glXSwapBuffers
Update state
Add mesh / materials to scene
Build acceleration structures
Build world acceleration structure
Clear data not saved in DL or VBOs
Copy image into frame buffer
render
GLuRay

Figure 1: *Sequential architecture of GluRay.*

3.2. Asynchronous Rendering

To speed up interactive rendering, the option to add a one frame lag between rendering calls was added. This alleviate the idle time for GLuRay waiting for rendering calls from single-threaded applications. When a draw call is made, the previous frame is copied to the framebuffer and returns. While the next batch of OpenGL calls are being made, the multithreaded system is rendering and building acceleration structures for the previous frame. For this we use a packet based Bounding Volume Hierarchy, BVH [WBS07]. To further decrease the time to build acceleration structures, an approximate BVH can be used which builds faster but

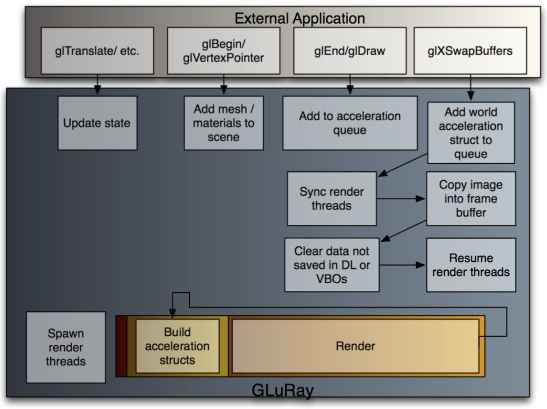

External Application
glTranslate/ etc.
glBegin/ glVertexPointer
glEnd/glDraw
glXSwapBuffers
Update state
Add mesh / materials to scene
Add to acceleration queue
Add world acceleration struct to queue
Sync render threads
Copy image into frame buffer
Clear data not saved in DL or VBOs
Resume render threads
Spawn render threads
Build acceleration structs
Render
GLuRay

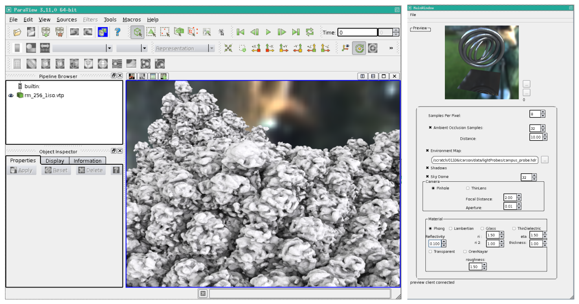
Figure 2: *Parallel architecture of GluRay.*

gives moderately slower runtime performance. This effectively gives a variable which can be changed depending on whether a system needs to be more interactive for updates to the underlying geometry or faster for changes to the camera. This system is shown in Figure 2.

3.3. High Quality Rendering

Ray tracing allows for advanced effects such as global illumination, accurate reflections, depth of field, soft shadows, transparency, and refraction to name a few. These techniques can be handled through other means, however our implementation provides an intuitive implementation of the rendering equation using light rays which can be used for publication quality images. Manta supports path tracing, however it was not used in our testing. Figure 3(a) shows GLuRay running within the visualization tool ParaView rendering a Richtmyer-Meshkov Instability with ambient occlusion. Camera manipulations operate just as they would with OpenGL and material and light properties are updated whenever OpenGL state changes are made in the host program. Not all material properties can be provided through OpenGL alone, such as refractive indices of glass objects or ambient occlusion options. Such additional material properties proprietary to GLuRay are exposed through an external GUI application shown in Figure 3(b). Changes are applied globally to all objects in the current scene. Modifications are broadcast to running programs through TCP sockets over the localhost.

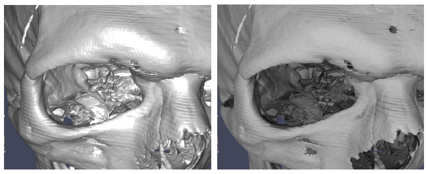
Figure 4 shows the comparison between a phong shading of a human skull and the same rendering with ambient occlusion added. Ambient occlusion provides depth cues not present in the phong shaded image by occluding light blocked by nearby geometry. Psychological user studies have validated that more realistic lighting using approximations of global light can aid comprehension of complex



(a) (b)

Figure 3: *GLuRay running within ParaView (a) and an external GUI (b).*

features in data [GP06] when compared to purely local lighting algorithms such as phong shading. Figure 5 shows a side-by-side comparison of an OpenGL rendering within ParaView of an aluminum ball hitting an aluminum plate and GLuRay rendering the same dataset within the same program using additional effects. Data loading, interaction, and rendering calls, were all done within the host program without noticeable differences to the user until special rendering modes were selected. Figure 6(b) shows an astrophysics simulation of magnetic reversal in a solar-type star rendered within VAPOR using GLuRay [BMB*11]. Ambient occlusion enhances streamlines while reflections and soft shadows add to the realism of the rendered image compared to only using local lighting as shown in Figure 6(a). Secondary rays are only supported on shared-memory systems or GLuRay's ray-parallel distributed mode which was implemented similar to Ize et al. [IBH11] but as of this writing this implementation only works with replicated data on each node.



(a) (b)

Figure 4: *Shading effects such as ambient occlusion shown in (b) add more depth cues compared to a standard phong shading technique (a).*

4. Results

We evaluated the rendering performance of GLuRay compared to hardware-accelerated OpenGL and software OpenGL rendering using Mesa on a single node of the rendering cluster Longhorn, an NSF XD visualization and data analysis cluster located at the Texas Advanced Computing

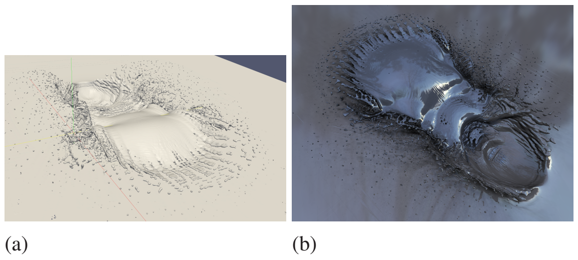


Figure 5: *Rendering of an impact dataset with ParaView using OpenGL (a) and a rendering using GLuRay showing reflections, soft shadows and ambient occlusion (b).*

Center (TACC). Longhorn has 256 4X QDR InfiniBand connected nodes, each with 2 Intel Nehalem quad core CPUs (model E5540) at 2.53 GHz and 48-144 GB of RAM. Each node of Longhorn also has 2 NVidia FX 5800 GPUs. We used datasets of varying sizes, including a synthetic wavelet dataset, a dataset from Los Alamos's plasma simulation code VPIC, a simulation of magnetic reversal in a solar-type star, and a time step from a Richtmyer-Meshkov Instability (RM) simulation rendered with two different views. All images were rendered at 1024x1024 resolution with the same settings and views across VisIt and ParaView where applicable. ParaView, VisIt and Ensight are built with Mvapich2 1.4 which is provided on Longhorn. ParaView was run on Longhorn using $tacc_x run$, pvbatch, and offscreen rendering for the GPU and Mesa render timings. GLuRay was run using vglrun, except for the scaling study in which case pvbatch, $tacc_x run$ and offscreen rendering were used. All benchmarked timings for GLuRay have the same illumination model as OpenGL with local lighting only and no shadows computed. Benchmarks were conducted with up to 6000 frames and an initial warmup period. This shows an expected frame rate from camera exploration of an isosurface which is the focus of our study, but not necessarily what may be achieved when exploring isosurface values or other updates which require rebuilding acceleration structures each frame.

Datasets

- **Astrophysics** The astrophysics dataset shows a Sun-like star visualized with 48000 streamlines representing magnetic field lines [BMB*11]. Figure 6(a) shows a rendering of this dataset in VAPOR.
- **Wavelet** The wavelet triangle dataset is a computed synthetic dataset source released with ParaView. We generated a 201^3 dataset and then calculated as many isosurfaces as needed to produce a certain quantity of triangles. The isosurfaces are nested within each other. Images produced with 16 million triangles are shown in Figure 6(c).
- **VPIC Visualization** Using a singe time-step from the VPIC plasma simulation, we calculated an isosurface and extracted streamtubes that combined totaled 102 million polygons. A view of this dataset rendered in ParaView can be seen in Figure 6(d).
- **Richtmyer-Meshkov Instability** The Richtmyer-Meshkov Instability simulation, RM, presents a commonly used scientific dataset. We created a polygonal representation with an isosurface from a single time-step resulting in 316 million triangles. To understand the behavior of the ray tracer we have rendered both a zoomed out view of the RM dataset, RMO, seen in Figure 6(f) and a closeup view in Figure 6(e), RMI. The closeup view shows a smaller portion of the overall data, however it also takes up more screen space.

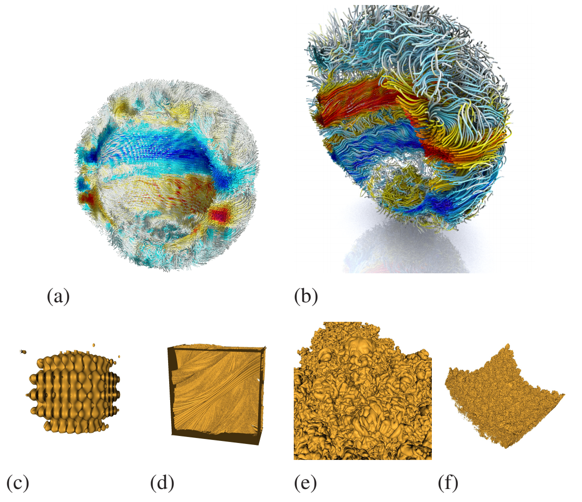


Figure 6: *Magnetic fields from an astrophysics simulation colored with streamlines (a), the astrophysics dataset rendered with advanced effects (b), a wavelet contour (c), VPIC Plamsa simulation (d), RM zoomed in (e), and RM zoomed out (f) datasets used for benchmarking.*

Scientific Visualization Programs

- **VAPOR** VAPOR is a visualization program developed by NCAR and designed for oceanic, atmospheric, and solar research focusing on isosurfaces, volumes and streamlines. Rendering is done through vertex arrays and display lists. Version 2.0.2 was used in our timing study.
- **EnSight** EnSight is an in-depth visualization package featuring volume rendering, streamlines, glyphs, and contours to name a few of the rendering modes supported. Rendering uses immediate mode rendering or display lists. Version 9.2 was used for our benchmarks.
- **ParaView** ParaView is a distributed visualization program built around VTK and designed for use on large cluster environments. Rendering is done through immediate mode rendering in OpenGL or display lists. For our tests we used the most recent available version when our tests were conducted, 3.11.0.

- **VisIt** VisIt is a distributed visualization program built around VTK for use on large clusters similar to ParaView. We utilized visit 2.4.0 for our tests.

For this paper we tested four visualization programs with three different rendering modes: software ray tracing using GLuRay, OpenGL software rasterization using Mesa, and hardware-accelerated OpenGL. Mesa is not multi-threaded nor the fastest available software rasterization package, however it is the only one supported as build options with ParaView and VisIt and is commonly used when GPUs are not available. The OpenGL implementation within these programs is a brute-force implementation with no advanced acceleration methods used. This comparison is therefore not a comparison of the ultimate potential of rasterization vs. ray tracing algorithms but rather a real-world study of their existing performance in commonly utilized tools with real world problems.

To test the scaling of these methods, the synthetic wavelet dataset was scaled from 1 to 256 million triangles in ParaView and VisIt. The dataset is shown with 16 million triangles in Figure 6(c). Mesa manages over one fps in ParaView only when the triangle count remains under 2 million triangles. The hardware-accelerated OpenGL implementation retains interactive performance at one to two million triangles in ParaView and VisIt, however performance degrades roughly linearly with triangle count. Slow GPU performance may be due to ParaView's rendering code which was built around immediate-mode rendering and accelerated through a display list. This leads to a large number of function calls for the initial build and updates compared with using vertex buffers, which can specify large numbers of vertices with a single function call. We found rendering performance to be roughly a tenth of what it could be by using vertex buffer objects or multiple display lists in our tests. Another issue is that display lists on Longhorn crash after about 32 million triangles. This could be fixed by splitting up the data across multiple display lists in a custom implementation, however immediate-mode rendering was used above 32 million triangles for the hardware-accelerated runs in our tests as this method that worked with an unaltered code base. VisIt was slower than ParaView at lower geometry counts in our tests due to increased overhead for each frame which accounts for the slightly slower GLuRay performance with VisIt. Additionally, VisIt uses a textured colormap which was set to interpolate between two identical colors for our tests whereas ParaView uses a single solid color resulting in fewer OpenGL calls. GluRay shows sub-linear performance degradation when the triangle count increases, scaling well into the hundreds of millions of triangles and only dropping below 5 fps past 128 million triangles in ParaView. Performance sometimes decreases with increased geometry counts. This may be a caching issue with some data sizes exhibiting better caching behavior than smaller triangle counts and differences in background space around the datasets which can be easily culled by the ray tracer.

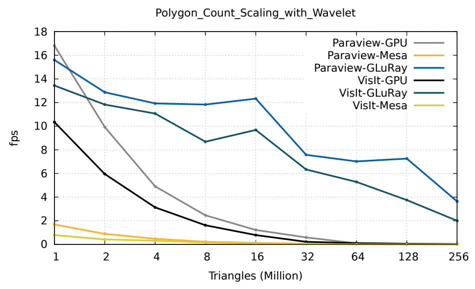


Figure 7: *Performance timings for increasing triangle counts for a wavelet dataset as seen in Figure 6(c).*

Table 1 shows timings for various datasets over multiple applications. All times are averages of several render frame times after a few initial warmup frames and include all host program overhead which is the most accurate view of performance for an asynchronous renderer. Seven render threads were used on an 8-core node, leaving one main thread for processing OpenGL state changes and image display. GLuRay is faster in all cases we tested with and achieves better than 300x speedup over Mesa for the RMO dataset and a 62x speedup over the GPU implementation for VPIC in ParaView. GLuRay has at least a 90x performance increase over Mesa while speedup over the GPU ranges from 3.57x speedup rendering the astrophysics dataset in VAPOR, up to a 123.85x speedup over the GPU rendering the RMO dataset when zoomed out in ParaView. The RM dataset zoomed in and out achieved similar performance for each view when rendered with the GPU. GLuRay and Mesa, however, have differing results between the two views of the RM dataset. This is because the geometry took up a larger portion of the scene in GLuRay when zoomed in, and in Mesa this is likely showing that clipping triangles outside of the viewport gave a greater speedup for Mesa than for the GPU. GLuRay is the only rendering method to achieve above 5-10 fps for interactive rendering for the 16M triangle wavelet dataset with 12.33 fps in ParaView and 9.69 fps in VisIt. GLuRay achieved 5.02, 6.94, and 2.55 fps rendering the VPIC datasets in ParaView, EnSight, and VisIt which was as much as a 62.75x performance improvement over OpenGL. Timings of the same dataset are different across different programs as each program incurs its own overhead as well as differing OpenGL calls per frame which affects the GPU, GLuRay, and to a lesser extent the Mesa performance. In the case of the astrophysics dataset with VAPOR, only a 3.57x speedup was achieved over the GPU. This is likely due to VAPOR's use of vertex buffers instead of the glVertex calls used by the other programs which greatly accelerated the GPU rendering.

The time to render the first frame is usually a combination of data loading, geometry generation, passing data to GLuRay, and finally acceleration structure construction and rendering. Approximate acceleration structure builds decrease

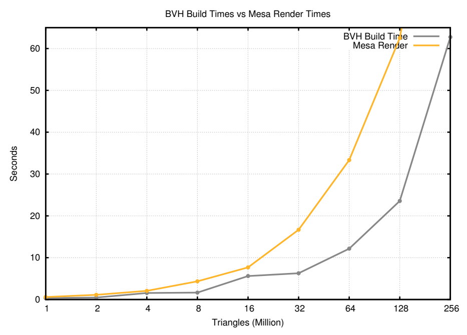


Figure 8: *BVH build times for the wavelet dataset compared to Mesa rendering time for a single frame.*

rendering performance but speed up build times. The overhead from the acceleration structure builds using approximate builds for the wavelet dataset is shown in Figure 8. This overhead varies from less than a second at 1 million triangles to over a minute with 256 million triangles, however this time is still less than rendering a single frame with software Mesa in all cases. Users of visualization programs typically generate an isosurface of data which is then explored through transformations to the camera. A duration of 10 seconds moving the camera with a render time of less than 0.1s per frame thus produces over a hundred rendered frames, making the time to process a single acceleration structure insignificant in overall runtime. Cases where geometry is animated, however, could need to build acceleration structures for each rendered frame which could limit performance for our program. When textures or color arrays are used in OpenGL, updating colormaps does not require rebuilding geometry. However, when colors are built into display lists those display lists must be rebuilt, which requires acceleration structures to be updated.

The overall build and runtime behavior of GLuRay is shown through Gantt charts in Figure 9 which illustrate a run from program start to end in ParaView with an 8 million triangle wavelet dataset and a closeup of rendering behavior from the same run at the bottom. There are 9 rows in total, one for the main thread at the top and 8 render threads below. The empty space at the beginning shows program startup and idle GLuRay threads waiting on the host program to send data through OpenGL function calls. The brown line after data loading shows the host program making millions of OpenGL calls such as glVertex and glNormal, which copy geometry into GLuRay. The following blue bars display the construction of acceleration structures and then turn gold for rendering. A lazy system was used for BVH construction, resulting in some of the rendering time being used for additional BVH processing which can be seen in the initial setup phase. As shown in the Gantt chart, the time to build acceleration structures and setup the first frame in the rendering threads is roughly equivalent to the time the program spends specifying the geometry through OpenGL. Grey denotes time spent waiting for the render threads to finish in the main thread. In between each render call, global acceleration structures must be updated and images copied to the framebuffer. The global acceleration structure takes into account the transforms applied to each stored set of geometry, such as changes to the ModelView matrix applied to the geometry from glCallList in OpenGL. The closeup of rendering performance at the bottom shows that the rendering threads are well utilized with very little overhead for building acceleration structures in between rendering or downtime waiting for updates.

In order to benchmark strong scaling across multiple nodes, the 316 million triangle RMO dataset was rendered on one to 64 nodes using the Longhorn visualization cluster and the parallel visualization tool ParaView using the same camera position shown in Figure 6(f). For parallel rendering, ParaView uses sort-last compositing through the IceT library, which introduces an additional compositing step at the end of every frame. Howison et al. [MHC10] used a sort-last compositing algorithm on the Jaguar supercomputer, where they found that compositing was their biggest bottleneck for a high resolution image. Their maximum achieved frame rate was 2 fps for a 21 million pixel image over 216,000 cores. Assuming this performance scales down to a 1 million pixel image, the maximum frame rate from compositing would be approximately 42 fps. We therefore aim to approach real-time rendering rates using our method for image sizes of 1024^2. Render times are reported from hardware-accelerated OpenGL and GLuRay in Figure 10. In comparison to the single node timings shown in Table 1, the scaling runs for ParaView use the parallel version of ParaView, pvbatch, with offscreen rendering enabled and use Longhorn's batch configuration script which decreases overhead from image display. To generate the correct image when needed by the compositor, rendering for GLuRay was modified to render upon calls to glCallList or glReadPixels, which IceT uses to gather the rendered scene for compositing. Enabling GLuRay's frame lag would result in a frame rate which is approximately equal to the maximum of the render time and the compositing time instead of the aggregate of the two. Therefore, in order to time just rendering, GLuRay was run without a frame lag such that asynchronous rendering was not utilized.

In our strong scaling study, GPU-accelerated average render times drop from 29.46 seconds to 0.28 seconds from 1 to 64 nodes respectively as the average triangle count drops from 316 million triangles to about 5 million triangles per node. Rendering times for GLuRay start at 0.21 seconds on a single node and decrease to 0.037 seconds for 64 nodes, which resulted in an overall frame rate of 18.47 fps on 64 nodes with compositing and other overhead within ParaView. The GPU render times at 64 nodes do not reach the performance of a single node with GLuRay, while GLuRay continues to increase in performance with each node

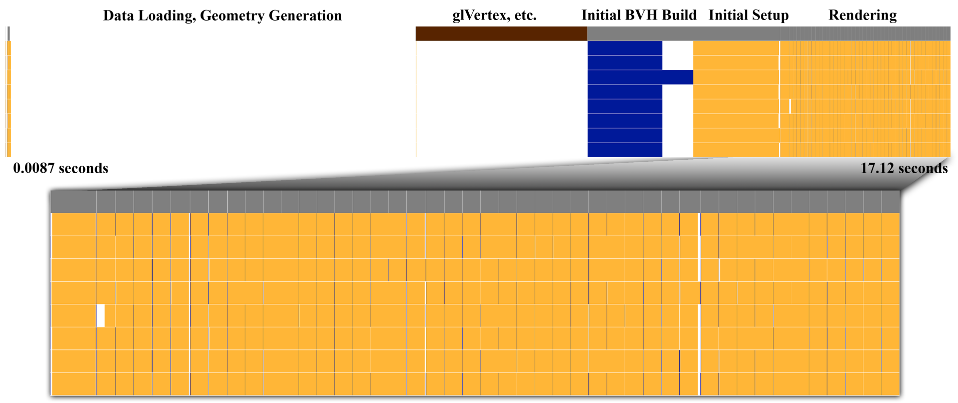


Figure 9: *Gantt charts showing a rendering of a wavelet dataset with 8 million triangles. An overall view of setup time, geometry transfer, acceleration builds and rendering is shown in the top runs while a closeup of the rendering is shown in the bottom. The gold color displays the efficiency of the asynchronous renderer while the blue displays the cost of BVH builds in relation to data loading and geometry specification through OpenGL shown in brown.*

DataSet	Triangles (M)	FPS	Mesa	GPU	Speedup vs. Mesa	Speedup vs. GPU
PV-Wavelet	16	12.33	0.13	1.22	94.85	10.11
VI-Wavelet	16	9.69	0.085	0.79	113.99	12.27
PV-VPIC	102	5.02	0.026	0.08	193.08	62.75
VI-VPIC	102	2.55	0.012	0.069	212.49	39.96
Ensight-VPIC	102	6.94	0.02	0.23	347.15	30.19
PV-RMI	316	2.53	0.001	0.03	180.71	93.70
PV-RMO	316	3.22	0.009	0.03	357.78	123.85
VAPOR-Star	86	2.39	0.03	0.67	95.60	3.57

Table 1: *Performance timings for various datasets across different applications with varying amounts of triangles specified in the millions. PV signifies ParaView and VI refers to VisIt. RMI and RMO are the Richtmyer-Meshkov datasets zoomed in and out respectively. GLuRay achieves significant speedups in all runs tested with large polygon counts.*

added. GLuRay render times do not decrease as dramatically as the GPU render times with each node added because the acceleration structures used scale well with increasingly large amounts of geometry as seen in Figure 7. Therefore, decreasing the triangle count per node does not impact performance as much as OpenGL. In order to achieve better work distribution with GLuRay, view-dependent distribution of the data would be needed. Running GLuRay over programs which utilize sort-first data distribution or a hybrid technique such as the one used by Nouanesengsy et al. [NAWS11] could provide better scaling behavior for ray tracing. Ize et al [IBH11] achieved up to 100 fps for a two megapixel rendering of the RM dataset using 60 nodes of a cluster using ray-parallel work distribution.

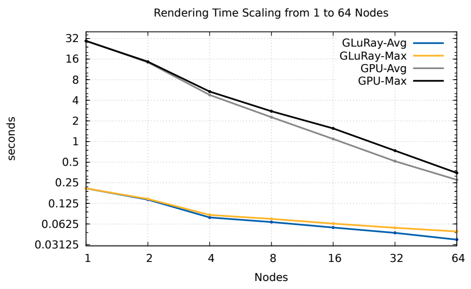


Figure 10: *Rendering times for the RMO dataset in ParaView using the Longhorn visualization cluster. "Avg" denotes the averaged render times across every node while "Max" designates the maximum render time across all nodes.*

5. Limitations and Future Work

The interception nature of GLuRay and the ray tracing engine produces a few limitations. GLuRay usually relies entirely on the data distribution of the host program and since that host program typically relies on data-parallel distribution, but with this method there is no way to access other portions of the scene from remote nodes for secondary effects such as shadows at run-time. Adjacent areas can be duplicated across nodes for some effects such as distance-limited ambient occlusion, however this is not common in the programs we have tested with. Data distribution in distributed-memory systems may be solved by implementing distribution through GLuRay in the background similar to such programs as Pomegranate or Chromium [EIH00, HHN*02]. Currently a ray-parallel work distribution similar to Ize et al. [IBH11] was implemented for GLuRay but as of this writing only supports replicated data on each node and has been tested with ParaView, where data distributed by ParaView is sent to each node and only node zero sends image data to ParaView. An out-of-core solution where nodes can page in data as needed is in developement. Stephens et al. showed Manta scaling very well on a large shared-memory system using transparency and other effects [SBB*06]. The main limitation of GLuRay is the memory overhead incurred by storing geometry and building acceleration structures. For a dataset with n polygons a typical BVH will be bounded by $2(n-1)$ BVH nodes. In clusters where memory is at a premium and compute is cheap, a slower but less memory intensive implementation may be ideal, however system memory is often much larger than that found on GPUs. The additional time to build acceleration structures is also a concern, but in exploratory visualization, a user will typically generate an isosurface to be interactively viewed resulting in many renderings for each update to geometry.

GLuRay is not a full mapping of OpenGL. Shaders are not supported yet and multi-pass rendering can significantly slow down the running system. Multiple passes are often used for effects such as shadows. In our testing many of the scientific visualization packages do not use such techniques and if they did, such systems could likely be turned off and their intended purpose replicated through the ray tracer in a single pass for performance considerations. There are many operations within OpenGL which may break the current program architecture and are not currently supported such as blending functions or state changes beyond geometry, texture or materials within display lists. Programs which use OpenGL to render out GUI elements could prove problematic, however none of the production level visualization tools we tested with use this method. None of these shortcomings have proven problematic for the generally simplistic rendering implementations within the tools we have tested with.

There is significant future work which could benefit ray tracing through OpenGL. Knoll et al. [KTW*11] recently demonstrated that direct volume rendering through CPU ray casting presents a very viable approach for volume and isosurface rendering with large speed advantages compared to out-of-core GPU rendering. Supporting volume rendering and shaders would be highly beneficial but is beyond the scope of this paper and is left for future work. Much work still needs to be done to accommodate visualization including maximizing single node performance for other parts of the visualization pipeline such as IO, isosurfacing, calculator operations, and implicit geometry rendering without using additional memory for geometry generation and storage. GPU accelerated ray tracing is another avenue of research which was not considered for this paper but worth further study. Although we have shown the capability of GLuRay to scale when running ParaView on a cluster, an in-depth study of render times in cluster environments would be worthwhile to determine the compositing and data-distribution impact of programs intended for rasterization.

6. Conclusion

We have shown that current rendering algorithms utilized in many scientific visualization tools do not always achieve sufficient performance to interactively handle ever increasing data sizes. With GLuRay, we have proven that by intercepting OpenGL calls and using an optimized software ray tracer, we can achieve significant improvements in rendering performance in some of our tests using millions of polygons over several common scientific visualization programs which otherwise fail to achieve interactivity on the Longhorn visualization cluster. We have shown that ray tracing with OpenGL interception presents an efficient method of ray tracing over programs running the fixed-function OpenGL pipeline with display lists and single-pass rendering. Additionally, we have shown that GLuRay performance scales on a distributed-memory cluster environment using ParaView's data-parallel work distribution and sort-last compositing. Since interactive rendering for gigatriangle sized datasets is already possible using GLuRay with current systems, we believe frame rates will improve on future machines as the number of cores per node increase. For users who do not need increased performance, we have also presented advanced rendering for publication quality images and enhanced insight within existing tools without the need for learning additional rendering programs. Limitations of our approach include decreased performance from building acceleration structures each frame with dynamic data and the currently limited domain of the four visualization programs using the fixed function OpenGL pipeline we have tested with.

7. Acknowledgements

This research was sponsored by the National Nuclear Security Administration under the Advanced Simulation and Computing program through DOE Cooperative Agreement #DE-NA0000740, and by Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST), and DOE SciDAC:VACET, NSF OCI-0906379, NIH-1R01GM098151-01. Special thanks to Hank Childs, Kadi Bouatouch, Thiago Ize, Ben Brown, and Brad Whitlock.

References

[BMB*11] Brown B. P., Miesch M. S., Browning M. K., Brun A. S., Toomre J.: Magnetic Cycles in a Convective Dynamo Simulation of a Young Solar-type Star. *apj 731* (Apr. 2011), 69. arXiv:1102.1993, doi:10.1088/0004-637X/731/1/69. 4, 5

[BSP06] Bigler J., Stephens A., Parker S.: Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium on* (Sept. 2006), pp. 187 –196. doi:10.1109/RT.2006.280230. 1, 2

[CEI10] CEI: Cei-creators of ensight visualization software, 2010. http://www.ensight.com/. 1, 2

[CGM*06] Cedilnik A., Geveci B., Morel K., Ahrens J., Favre J.: Remote large data visualization in the paraview framework. In *Eurographics Parallel Graphics and Visualization 2006* (May 2006), pp. 162–170. 1, 2

[EIH00] Eldridge M., Igehy H., Hanrahan P.: Pomegranate: a fully scalable graphics architecture. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), SIGGRAPH 00, ACM Press/Addison-Wesley Publishing Co., pp. 443–454. URL: http://dx.doi.org/10.1145/344779.344981, doi:http://dx.doi.org/10.1145/344779.344981. 9

[GP06] Gribble C. P., Parker S. G.: Enhancing interactive particle visualization with advanced shading models. In *Proceedings of the 3rd symposium on Applied perception in graphics and visualization* (New York, NY, USA, 2006), APGV '06, ACM, pp. 111–118. URL: http://doi.acm.org/10.1145/1140491.1140514, doi:http://doi.acm.org/10.1145/1140491.1140514. 4

[HEB*01] Humphreys G., Eldridge M., Buck I., Stoll G., Everett M., Hanrahan P.: Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 129–140. URL: http://doi.acm.org/10.1145/383259.383272, doi:http://doi.acm.org/10.1145/383259.383272. 3

[HHN*02] Humphreys G., Houston M., Ng R., Frank R., Ahern S., Kirchner P. D., Klosowski J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 693–702. URL: http://doi.acm.org/10.1145/566570.566639, doi:http://doi.acm.org/10.1145/566570.566639. 3, 9

[IBH11] Ize T., Brownlee C., Hansen C. D.: Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 61–69. 3, 4, 8, 9

[Kit10] Kitware: *Paraview - Open Source Scientific Visualization*, 2010. http://www.paraview.org/. 2

[KTW*11] Knoll A., Thelen S., Wald I., Hansen C. D., Hagen H., Papka M. E.: Full-resolution interactive cpu volume rendering with coherent bvh traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium* (Washington, DC, USA, 2011), PACIFICVIS '11, IEEE Computer Society, pp. 3–10. URL: http://dl.acm.org/citation.cfm?id=2015551.2015627. 9

[LLN10] LLNL: *VisIt Visualization Tool*, 2010. https://wci.llnl.gov/codes/visit/. 1, 2

[Mag11] Magallŭn M. E.: spyglass: an opengl call tracer and debugging tool, 2011. 3

[MC98] Mitra T., Chiueh T.-C.: Implementation and evaluation of the parallel mesa library. In *Parallel and Distributed Systems, 1998. Proceedings.* (dec 1998), pp. 84 –91. doi:10.1109/ICPADS.1998.741023. 2

[MDG*10] Marsalek L., Dehof A., Georgiev I., Lenhof H.-P., Slusallek P., Hildebrandt A.: Real-time ray tracing of complex molecular scenes. In *Information Visualization: Information Visualization in Biomedical Informatics (IVBI)* (2010). 2

[MHC10] M. Howison E. B., Childs H.: Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (May 2010). 2, 7

[NAWS11] Nouanesengsy B., Ahrens J., Woodring J., Shen H.: Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 31–40. 2, 8

[PBBR07] Parker S., Boulos S., Bigler J., Robison A.: Rtsl: a ray tracing shading language. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (sept. 2007), pp. 149 –160. doi:10.1109/RT.2007.4342603. 3

[PPL*99] Parker S., Parker M., Livnat Y., Sloan P.-P., Hansen C., Shirley P.: Interactive ray tracing for volume visualization. *Visualization and Computer Graphics, IEEE Transactions on 5*, 3 (jul-sep 1999), 238 –250. doi:10.1109/2945.795215. 2

[PSL*98] Parker S., Shirley P., Livnat Y., Hansen C., Sloan P.-P.: Interactive ray tracing for isosurface rendering. In *Visualization '98. Proceedings* (oct. 1998), pp. 233 –238. doi:10.1109/VISUAL.1998.745713. 1, 2

[SBB*06] Stephens A., Boulos S., Bigler J., Wald I., Parker S. G.: An application of scalable massive model interaction using shared memory systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 19–26. 2, 9

[SM97] SGI, Miles J.: gltrace, 1997. http://reality.sgi.com/opengl/gltrace/. 3

[UCA09] UCAR: Vapor, 2009. http://www.vapor.ucar.edu/. 1, 2

[WBDS03] Wald I., Benthin C., Dietrich A., Slusallek P.: Interactive ray tracing on commodity pc clusters. *LECTURE NOTES IN COMPUTER SCIENCE* (2003), 499–508. 3

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan. 2007). 3

[WSB01] Wald I., Slusallek P., Benthin C.: Interactive distributed ray tracing of highly complex models. In *Proc. of Eurographics Workshop on Rendering* (2001), pp. 274–285. 1, 2