Visionaray: A Cross-Platform Ray Tracing Template Library

Stefan Zellmann* University of Cologne
Daniel Wickeroth† University of Cologne
Ulrich Lang‡ University of Cologne

Figure 1: Ray tracing applications implemented with Visionaray (from left to right): high quality direct volume rendering for medical visualization, scientific visualization in VR, global illumination with path tracing, large point cloud rendering in VR, ambient occlusion rendering.

Abstract

We present the software architecture of the C++ ray tracing template library *Visionaray*, which provides generic algorithms and data structures as building blocks for applications that traverse rays through 3-D space. While many state of the art ray tracing libraries are vendor specific and focus only on acceleration data structure traversal, Visionaray is cross platform compatible and implements several ray tracing-related tasks such as parallel ray generation, shading, Monte Carlo integration, or texture sampling. Visionaray was developed with a strong focus on real-time and interactive rendering on high performance computing platforms. The generic programming approach allows for loose coupling of algorithms and efficient binary code that is optimized for data and instruction cache utilization. We discuss implementation details and software architecture considerations to achieve real-time performance on hardware platforms ranging from single-board mini computers to GPGPUs. We prove the effectiveness of our approach by evaluating and discussing four case studies that focus on real-time rendering, scientific visualization, and virtual reality.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality; D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

1 Introduction

Software libraries for traversing rays through 3-D space in order to integrate radiance along light paths are often optimized exclusively for a specific hardware and typically only focus on core functionality such as efficient construction of acceleration data structures for triangle geometry. While traversing rays through bounding volume hierarchies (BVH) is a core operation in many ray tracing-like algorithms (in this paper we will generally refer to any algorithm that involves traversing rays through 3-D space as "ray tracing algorithm"), this is not generally true for every use case. Algorithms such as direct volume rendering for scientific visualization, or iso surface ray tracing, do not necessarily benefit from a BVH optimized for polygon surfaces, but may e.g. rely on efficient texture filtering or make use of an optimized SIMD vector math library to implement interval arithmetic. Because ray tracing libraries are often developed by hardware vendors, many of them are also not available across platforms. The Visionaray library fills those gaps by providing open source generic C++ algorithms and data structures targeting multiple modern high performance computing platforms that are useful to optimize the whole ray integration pipeline, and not just ray / BVH traversal.

Visionaray makes use of various modern C++ language constructs that were initially introduced with the C++11 standard [17]. Using modern C++ is one possible strategy to achieve cross platform compatibility on high performance computing (HPC) platforms. Alternative strategies may involve using programming languages that conform to the lowest common denominator for the target platforms, such as ANSI C99 or OpenCL 1.2, or devising a dedicated domain specific language (DSL) with compiler backends to manage the cross platform capabilities of this approach. We opted for using C++, since a couple of interesting HPC programming languages or compiler infrastructures, such as Xilinx Vivado HLS for field programmable gate arrays, or AMD ROCm and NVIDIA CUDA for GPUs, nowadays support many modern C++ language features, and because C++, in contrast to e.g. ANSI-C, offers type aware compile time meta programming. We prove the validity of generic programming for real-time interactive systems targeted at virtual reality (VR) by showing that our approach results in applications with competitive runtime performance.

Designing a generic C++ library in the spirit of e.g. the standard template library [3], the Boost C++ libraries [1], or the Thrust GPU compute library [5] results in a general, yet efficient application programming interface. Doing so incurs additional complexity on the library programming level. Obeying a set of common rules can however help the *library programmer* to manage the additional complexity. In this paper we describe major parts of the generic application programming interface of the Visionaray library, and the design decisions that led to this particular interface. We go into further detail about design decisions influenced by runtime performance considerations and platform specific optimization potentials. We then evaluate our approach from the perspective of the *application programmer* and therefore present four case studies that demonstrate how to implement real-time and interactive ray tracing

*e-mail: zellmann@uni-koeln.de
†e-mail:wickeroth@uni-koeln.de
‡e-mail:lang@uni-koeln.de

applications with Visionaray.
The paper is organized as follows. In Section 2 we review related work. Section 3 provides a detailed overview of the Visionaray software architecture and design decisions that led to that architecture. In Section 4 we present four case studies implemented with Visionaray. We briefly discuss our findings in Section 5 and conclude the publication in Section 6.

2 Related Work

The first scientific publications on real-time ray tracing emerged in a time when Intel introduced commodity processors with fourfold single precision floating point SIMD vector operations [21, 20, 19]. In order to achieve real-time performance on contemporary systems, efficient cache utilization was of utmost importance, and was achieved by directly incorporating SIMD compiler intrinsics into algorithmic control flow, and by carefully devising data structures to make efficient use of the large data caches newly introduced by hardware platforms at that time. At roughly the same time, real-time ray tracing on GPUs emerged. Due to the lack of GPGPU programming languages, GPU ray tracing systems were implemented using graphics centric APIs [7]. Real-time ray tracing architectures at that time mandated a low level programming approach which severely differed from object oriented approaches for ray tracing [24, 14, 11] in both maintainability and flexibility.
In 2008, Georgiev and Slusallek proposed the generic C++ ray tracing framework RTfact [15]. At that time, C++ compilers implemented the C++-03 standard, which did not include certain language constructs that are necessary to devise an expressive generic ray tracing architecture, such as *variadic templates* and *lamdba functions*. Because contemporary GPU APIs not yet allowed to use generic C++ language constructs, the RTfact system was also exclusively dedicated to CPU architectures.
Modern ray tracing libraries that are developed by the hardware vendors are typically not generic but provide a fixed-function pipeline with dedicated stages that can be customized by the application programmer [10], or focus on ray tracing kernel implementations like optimized BVH traversal [22]. The kernels are intended to be integrated into larger rendering systems that implements multi threading, shading, texturing, or framebuffer handling. Those systems are also typically only support the platform of the respective hardware vendor.
Domain specific languages (DSL) provide means to program high performance hardware with a high level of abstraction. This is achieved by dedicated compiler backends that translate a narrow set of specific operations to optimized machine code. ISPC [12] and AnyDSL [8] belong in the category of DSLs and are developed by working groups having a designated reputation in the fields of high performance ray tracing and physically based rendering.

3 Visionaray Software Architecture

We envision several usage scenarios for the Visionaray library. The most casual usage scenario would see the application programmer include one or few Visionaray C++ header files in her application code and use a single algorithm, just as one would e.g. with an algorithm such as std::sort() or std::find() from the STL, only that the Visionaray algorithm implements a control flow that is concerned with processing rays. A more complex usage scenario we envision is that the application programmer implements a whole ray tracing-based image generation pipeline with Visionaray and uses a dedicated, generic infrastructure to perform cross platform, SIMD optimized ray traversal. An even more complex scenario would involve other use cases than image generation, so that the application programmer would have to extend the Visionaray architecture with custom ray scheduling mechanisms. These usage scenarios determine the software architecture of Visionaray.
Visionaray's core software components are a SIMD optimized vector math library, a library component for texture management and filtering, a library component for geometrical operations and surface evaluation with Monte Carlo sampling, a component for ray / acceleration data structure traversal, and an image generation component that is concerned with generating and tracing primary rays in parallel, and storing results such as color or depth information in virtual or hardware framebuffers. The vector math library provides coherent ray packets [21] accelerated with SSE2, SSE4.1, AVX, AVX2, AVX-512, and ARM Neon intrinsics on compatible platforms and resorts to a single ray implementation otherwise. The vector math library therefore implements numeric types that behave in concordance with IEE-754 single precision floating point numbers, but wrap up to 16 32-bit floating point values. This basically allows for using SIMD types (*float4*, *float8*, *float16*) interchangeably with builtin C++ language types such as *float* or *double*. Minimal interdependencies between the library components allow for loose coupling of software functionality. In the following we describe major parts of the Visionaray library based on the aforementioned usage scenarios.

3.1 Generic Algorithms and Data Structures

The Visionaray software architecture follows the principles of generic programming and loose coupling of algorithms and data structures. Major design principles that we use follow from the generic programming approach and include *compile time virtual inheritance*, use of *type traits*, and the *substitution failure is not an error* (SFINAE) principle. Every Visionaray algorithm is designed generically. Since Visionaray can be used across platforms, applications may also run on accelerator infrastructures, where compute intensive tasks are offloaded to a coprocessor such as a GPU or an FPGA. Sharing address spaces between host CPU and coprocessor is typically prohibitive due to performance penalties, or even impossible on some systems. Simple data structures that are used with Visionaray algorithms, such as triangles or material properties, are therefore required to be "plain old data" (POD) types that do not contain pointers, do not rely on virtual inheritance, are trivially constructible, and do not have custom destructors. This restriction may seem overly limiting for an expressive ray tracing programming API at first, because ray tracing in general can be naturally described as an object oriented program where virtual inheritance is used to extend abstract shape or material property types. The principle however mandates a programming style that, in addition to generality and cross platform compatibility, ensures *performance portability*. Without runtime polymorphism, branching in inner loops is reduced, and the compiler can apply additional optimizations and static binding. Sets of objects like triangles or materials are not represented by arrays of pointers, but rather by arrays that directly contain the objects themselves. This reduces indirection and maps well to modern architectures, where optimized and cache friendly memory accesses are crucial for good runtime performance [17]. Below we however present abstractions that can alleviate the absence of virtual inheritance with generic programming constructs such as *compile time polymorphism* and *variant data types*.
Visionaray makes rigorous use of the *concept* principal to tie data structures to algorithms. The *closest_hit()* algorithm e.g. implements ray / geometric object traversal and returns a *hit_record* data structure containing information on the particular ray / object interaction. Geometric object types must implement the **Primitive** concept. Two iterators to **Primitive**s, [*first*,*last*), are passed to the algorithm to perform range based traversal. In order to implement the **Primitive** concept, a free function *intersect()* must be added to the *overload set*, which takes a ray and a **Primitive** instance and returns a *hit_record*. The ray type is also generic and must implement the **Ray** concept, which again mandates that the type contains two members that implement the **Vector3** concept (which

Table 1: Visionaray's customization point interface.

Category	Cust. Point	Category	Cust. Point
Primitives	intersect()	Intersection	Custom
	split_primitive()		intersectors
	get_bounds()	Pseudo Random	Sampler types
Traversal	closest_hit()	Numbers	
	any_hit()	Reflection	BRDF types
	multi_hit()		Material types
Hit Records	is_closer()	Lighting	Custom light
	update_if()		types
Textures	tex{1|2|3}D()		sample() at
Shading	get_color()		random position
	get_normal()	Parallel	Schedulers and
	get_tex_coord()	Rendering	Kernels

may e.g. be a *visionaray::vector<3, float>* for single ray traversal, a *visionaray::vector<3, simd::float8>* for coherent ray packet traversal with AVX, or any other type that implements the concept). From that example, one can see how the application programmer can easily extend the Visionaray library with custom types. If she were e.g. to implement a custom primitive type, the application programmer would merely provide the free function *intersect()* with the correct parameter interface, and then the associated type could immediately be used with all Visionaray algorithms that process **Primitive**s. Concepts are also extensible. In order to be able to place the custom primitive in a BVH, the application programmer must e.g. implement the **Primitive** concept, and in addition the free function *get_bounds()*, which returns an axis aligned bounding box (AABB) surrounding the primitive. For compatibility with Visionaray's SBVH implementation [16] the application programmer must additionally implement the free function *split_primitive()*. Note that Visionaray BVHs are also **Primitive**s and are compatible with the traversal algorithms. This allows e.g. for immediately implementing simple real-time animation with non deformable objects. Table 1 provides a (non-comprehensive) list of *customization points* that can be overloaded to extend Visionaray's core functionality.

Visionaray implements the BVH construction strategy from [18] and additionally allows for spatial splits [16]. The construction scheme of the Visionaray BVH with spatial splits is similar to the one proposed in [6]. All algorithms follow the basic principle of being generic customization points that loosely specify compatibility with data types through concepts. Note that concepts are not yet officially part of the C++ standard, so that we emulate this behavior with C++'s SFINAE mechanism. Algorithms are designed to be used in a most casual way and with minimal dependencies. As an example, consider a VR application that does *not* rely on ray tracing for rendering, but has a 3-D GUI with an application menu made up of floating quadrilaterals that may be freely arranged in space and that can potentially overlap. The *closest_hit()* algorithm could then be incorporated to implement interaction with a pointing device to determine which menu item was selected in terms of just a few lines of Visionaray code (cf. Listing 1 in the appendix).

3.2 Considerations for Cross-Platform Compatibility and Real-Time Performance

The question if data is maintained, and potentially copied to an accelerator, by the application itself or by the library is crucial for cross-platform compatibility and for runtime performance. Ray tracing libraries like Embree [22] or Radeon Rays [2] implement a control flow where the application programmer first *commits* the data as a *traversal state*. The ray tracing library then has the opportunity to *relayout* the data for better performance. Only after the traversal state is committed, the application is ready to perform ray traversal itself. After another context switch, the data can be read back to the application for further processing. Because of the loose coupling principle that we follow with Visionaray, one major design decision is that the traversal data is maintained by the application itself and not by the library. This can e.g. be seen in the menu selection example in Listing 1. The menu items are stored in CPU main memory using an STL container. With a stateful API like that of Embree, the ray tracing library would maintain an additional data copy in memory. With a stateful GPGPU ray tracing API, the data would even have to be copied to the accelerator over PCIe. In contrast to that, with Visionaray, if the application programmer has a *device* pointer to the data or some other type of accessor, that may e.g. be obtained from the Thrust GPU compute library, traversing the menu items is immediately possible in a CUDA kernel. In order to work with the *closest_hit()* algorithm, the menu items need then to either be stored as quadrilaterals as indicated in the listing. Alternatively, the application programmer could have promoted the application's internal menu item type to a Visionaray **Primitive** by implementing the *intersect()* customization point. That way, state changes and expensive data copies are reduced to a minimum. Our approach lends itself well to a scenario where data such as geometry is already present on the accelerator, and traversal with Visionaray is implemented as an ensuing stage of a larger processing pipeline. In order to generally pass lists of data, like geometric primitives, material properties, or color buffers to contain the final result from rendering, to compute functions running on accelerators, Visionaray advocates a lightweight referencing approach (data structures that come with Visionaray follow this approach, and the application programmer is advised to follow it when implementing custom data structures that contain large chunks of data). All data structures that store large data chunks, such as acceleration data structures or textures, have a corresponding lightweight reference type, that implements the same public interface as the actual data containing type does, but internally only stores pointers or, more generally, accessors to the data. Note that when porting algorithms to an accelerator platform, this requires the accelerator API to provide means to store accessors to data resident on the accelerator on the host CPU. NVIDIA CUDA and AMD ROCm e.g. allow to store so called device pointers on the host, which are however only valid when accessed in a compute kernel. Consider Visionaray's texture storage objects as an example. The class template *texture* implements the **Texture** concept and stores an array with texture data. If the application involves rendering of textured geometry in real-time with CUDA on an NVIDIA GPU in a compute kernel, the texture data needs to be passed to the kernel through the kernel parameter lists. With lightweight referencing, the application programmer creates a *texture* object once that is resident in GPU DDR memory. In order to pass it to the the CUDA kernel, a lightweight *texture_ref* object is created that contains a device pointer to the data, along with meta information such as the dimensions of the texture. The *texture_ref* class implements the public interface of the *texture* class, but contains only pointers to the actual data. Listing 2 shows C++ pseudo code for this example. The lightweight referencing principal applies to all kinds of data intensive Visionaray storage objects, with further examples being *render_target_ref()* to refer to a virtual framebuffer that is possibly resident as an OpenGL pixel buffer object, or *bvh_ref*, which is used to reference BVH acceleration data structures.

Some hardware specific optimizations are particularly hard to implement generically. An example are 3-D floating point vectors, that benefit from 16 byte alignment on NVIDIA GPUs. Then the CUDA compiler can use dedicated vector load operations to load the vectors from DDR memory. This optimization is performance critical on GPUs, while there is no measurable performance impact

on other platforms. With 16 byte alignment, the size of a 3-D vector however grows from 12 to 16 bytes, resulting in unnecessary memory consumption. As a solution, one could provide two different 3-D vector types for CPUs and GPUs. Then one would however have to convert one type to the other before arrays of 3-D vectors could be copied to the GPU and back. In the particular case, we decided to have only a single 3-D vector type with 16 byte alignment. Such decisions that severely impact runtime performance are few, but require very special care and, as in this specific case, sometimes a compromise. In some cases, we decided to explicitly expose different code paths for different architectures through the API. On architectures utilizing 128 bit SIMD units, the math library e.g. provides a dedicated *simd::float4* type, while this type is inaccessible on GPUs. In such cases, global type aliases can help to reduce application code complexity (see Listing 3 for an example).

Reducing branching in inner loops of algorithms is crucial for high runtime performance on many architectures. One optimization that many ray tracing implementations employ is to support only triangle geometry for BVH traversal to avoid branching in the innermost loop of the ray tracing algorithm. With a generic programming API, this restriction is on the one hand relaxed, because any type of primitive (as long as it implements the **Primitive** concept) is compatible with BVHs. On the other hand, in contrast to object oriented approaches with virtual inheritance to implement primitive types, only a single primitive type can be traversed per template instantiation. There are several ways to mitigate this inflexibility, with one of them being to tesselate any type of primitive to triangles. For cases where this is infeasible, Visionaray uses wrapper templates based on C++ variants, which are basically unions that store a tag along with them to identify which type the concrete instance of the union stores. The type is then retrieved dynamically at runtime. The wrapper class contains an instance of or inherits from the variant type and implements (in the primitive case the **Primitive**) concept. If the application programmer makes use of the wrapper class to e.g. instantiate the *closest_hit()* function template, there may be a performance penalty due to dynamic branching. The general case, where *closest_hit()* is instantiated with a single primitive type is however not affected by this. Since C++ variants are templates with a variadic argument list, the wrapper classes can basically maintain any number of primitive types. Listing 4 shows examples of the variant wrapper classes implemented by Visionaray.

3.3 Schedulers and Kernels for Image Generation

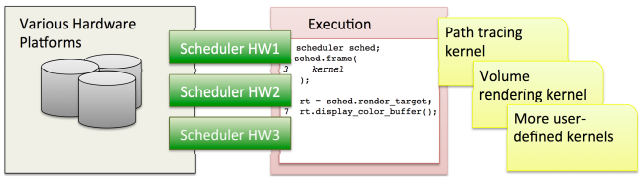


Figure 2: Schedulers generate primary rays, execute kernels that traverse primary rays and write the traversal result to render targets.

Because image generation is the predominant application implemented with ray tracing algorithms, Visionaray provides a special infrastructure for that (see Figure 2). The software abstraction was inspired by high performance parallel computing. **Scheduler**s encapsulate parallel ray generation, initiate ray traversal, and store the traversal result to *render targets*, which are an abstraction for a virtual or physical framebuffer. **Scheduler**s therefore provide a member function *frame()*, which is a template function that is instantiated with a parameter class. The parameter class contains, amongst others, a reference to the render target, and either a pair of camera matrices or reference to a perspective camera object. Additionally, *frame()* takes a "kernel" (a C++ function object that implements ray traversal and is not to be confused with a GPU compute kernel) as function argument. Kernels take a single ray or a coherent ray packet as input, perform ray traversal, and return a result record that was filled during traversal. A typical scheduler's *frame()* function will perform some initialization tasks (e.g. binding a pixel buffer object associated with the framebuffer in the parameters), generate primary rays based upon the camera description and typically in parallel, for each primary ray call the kernel functor with the ray as parameter, write the result from the kernel call to the framebuffer, and, after all primary rays were processed, perform cleanup tasks such as unbinding pixel buffer objects. Both schedulers and kernels are highly customizable. Visionaray provides a couple of default schedulers, e.g. a tile-based task queue scheduler using lock-free atomic synchronization and threads to run on CPUs, and a CUDA compatible scheduler that generates rays and calls Visionaray kernels in a CUDA kernel and writes the result to a GPU device memory buffer. Render targets can be directly displayed to an application window or accessed for further processing on either the CPU or the accelerator. Kernels are however the entity that provide the highest degree of customizability. Kernels are application programmer-defined function objects (C++ lambda functions, free functions, or classes implementing *operator()*) that describe an individual ray traversal function based only on the primary ray as input, and that return a result that is compatible with the render target. In Section 4.4 we demonstrate how to implement a simple kernel. Visionaray comes with a set of kernels that implement popular algorithms such as path tracing (cf. Figure 1).

4 Results

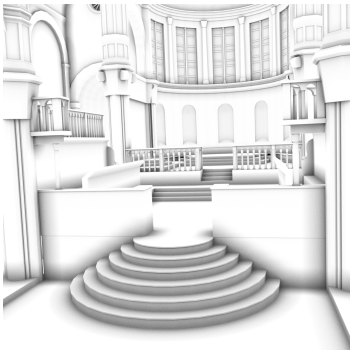
We evaluate our approach using four case studies based on applications that were implemented with Visionaray. We use an ambient occlusion benchmark to evaluate the throughput of our BVH implementation on various hardware platforms. We then implement direct volume rendering with Visionaray. The third case study shows how to combine ray tracing and scientific visualization, and the fourth case study is centered around interactive large point cloud rendering. The latter three applications are published open source as part of the visualization software COVISE [13].

4.1 Case Study: Ambient Occlusion Benchmark

We render ambient occlusion images as a benchmark to prove that our approach is competitive for triangle mesh ray tracing. Good BVH quality and traversal speed are crucial for real-time and interactive rendering applications. We use the same scenes that the authors from [4] used to evaluate their BVH traversal scheme. We evaluate on three HPC systems. We test on a dual socket system with two Intel Xeon E5-2690 CPUs and a total of 16 physical cores. We further test on an NVIDIA Quadro K6000 GPU and on an Intel Xeon PHI processor (codenamed Knights Landing) with 64 physical cores and fourfold simultaneous multithreading (SMT). We also test on a third generation Raspberry PI model B single board computer. We did not choose this platform because we consider it particularly suitable for real-time interactive systems, but rather to demonstrate that Visionaray scales from handheld devices with mobile CPUs (which typically resemble the one employed in the Raspberry PI) to many core systems and GPGPUs. In addition to that, we want to prove that Visionaray performs adequately on the ARM platform.

We empirically determined the parallelization configurations that best suited the respective hardware platforms. We trace packets of four rays with ARM NEON on the Raspberry PI, eight rays with AVX2 on the Xeon system, and sixteen rays with AVX-512 on the Xeon PHI system. We use Visionaray's tile based multi threading scheduler with 4, 16, and 256 threads for the three benchmarks, respectively. We use tiles of 16×16 pixels on the CPU architectures. On the NVIDIA GPU we configure the CUDA scheduler to

Table 2: Ambient Occlusion BVH Throughput (Mrays/s). 1024 × 1024 primary visibility rays, plus eight shadow rays per primary intersection. Results were measured on a Raspberry PI 3 Model B, a 16 core Xeon system, a Quadro K6000 GPU, and a Xeon PHI Knights Landing processor with 64 cores and 256 threads. We show results for binned SAH BVHs, and for binned SAH BVHs with spatial splits (SBVH) in parentheses.



Platform	Conference, 331K tris, 253K nodes	Fairy Forest, 174K tris, 132K nodes	Sibenik, 75K tris, 74K nodes
RPi3	2.0 (2.3)	1.2 (1.3)	1.7 (1.9)
Xeon	102.0 (117.0)	55.0 (57.1)	94.2 (105.6)
K6000	142.4 (189.1)	63.1 (71.1)	113.2 (142.9)
KNL	254.0 (273.2)	130.0 (130.4)	247.7 (268.2)

perform single ray traversal and divide the screen into tiles of 8 × 8 pixels, which provided the highest occupancy on that system. Table 2 shows the results of our benchmark tests. To obtain the images depicted above the result columns, we blend 250 ambient occlusion frames. For each frame we generate 1024 × 1024 primary rays, and eight shadow rays per primary ray, with direction vectors obtained from cosine weighted hemisphere sampling. We measure BVH throughput in *million rays per second* (Mrays/s), which we report for BVH traversal with and without allowing for spatial splits during construction. The results show that our approach is competitive with state of the art ray tracing implementations for surface ray tracing, and that our implementation scales from mobile to HPC processors.

4.2 Case Study: Direct Volume Rendering

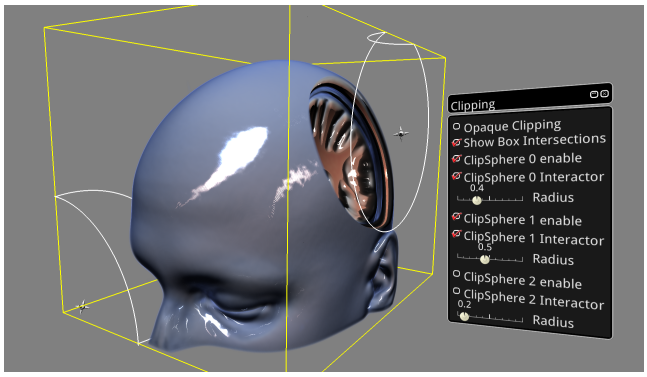


Figure 3: Direct volume rendering and clipping in VR with COVISE.

Our second case study comprises the integration of *direct volume rendering* (DVR) with ray casting in the VR visualization software COVISE. The case study builds upon the work from Zellmann et al. [26] where a two pass DVR algorithm with clipping based on multi-hit BVH traversal was implemented with Visionaray. We use a Visionaray ray tracing kernel to implement DVR with post classification using the *tex3D()* and *tex1D()* Visionaray algorithms to first interpolate a volume sample and then perform a lookup in a 1-D transfer function. For integration with COVISE's VR renderer, which has an OpenGL 3-D user interface, rays are clipped with the OpenGL depth buffer. The DVR ray tracing kernel implements coherent packet traversal on the CPU, and single ray traversal on NVIDIA GPUs. In the latter case, we use CUDA OpenGL interoperability to map the depth buffer in the Visionaray kernel. A graphical user interface was developed to interactively clip the volume with simple parametric surfaces in VR (cf. Figure 3).

4.3 Case Study: Scientific Visualization in VR

COVISE [13] is a data driven scientific visualization software that provides algorithms and tools to implement the visualization pipeline with modules and a flow graph. The rendering module OpenCOVER is specifically targeted towards VR and provides tracking system integration, first person navigation with 3-D tracking devices, and a 3-D VR menu. OpenCOVER uses the OpenSceneGraph [23] library to manage geometry and animation. We implemented a plugin for OpenCOVER that ray traces those parts of the scene graph that are compatible with Visionaray and bypasses OpenGL rendering to display ray traced stereo images instead. We therefore selectively deactivate the respective scene graph nodes for OpenGL rendering traversal. Geometry that is not going to be ray traced, e.g. particle traces that are rendered with line primitives, or the VR menu, are first rendered with OpenGL. We then render the remaining geometry with Visionaray. In order to composite the ray traced images with the OpenGL depth buffer, we bind a Visionaray render target that stores RGBA colors and depth. By binding the depth-enabled render target, and through template argument deduction when instantiating the scheduler with the render target, primary intersection positions are automatically converted to depth pixels in OpenGL window coordinates. Our scene graph rendering plugin has several features to mimic functionality provided by OpenGL, e.g. alpha map textures that are evaluated with custom intersection routines during BVH traversal to determine if a fully transparent surface was hit. We support static and simple dynamic scenes. We therefore a priori generate BVHs for animation sequences, and then pass the currently active BVH to the Visionaray ray tracing kernel using the lightweight references that we described above. The second from left image in Figure 1 shows a time step of a computational fluid dynamics simulation that was rendered with path

tracing. We render shadows from point light sources and specular reflection from mirror materials interactively in VR. We can generate single images with path tracing interactively. Those are however noisy and converge to high quality over time, so that this rendering algorithm is not suitable for VR with head tracking.

4.4 Case Study: Large Point Cloud Rendering in VR

Figure 4: 152M point cloud data set used for the evaluation.

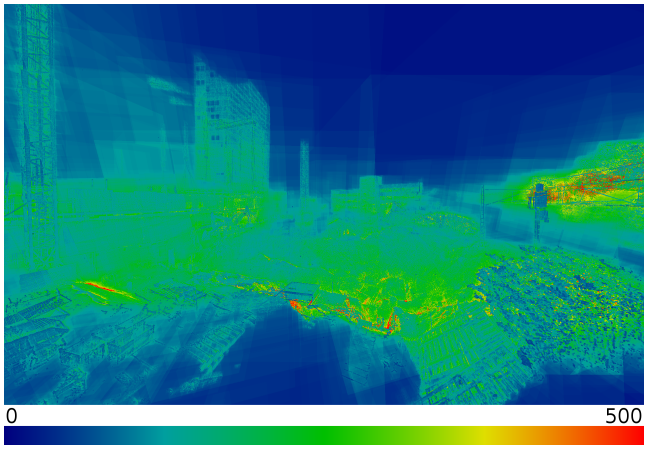


Figure 5: BVH traversal costs (96M nodes, 48M leaves). The color coding shows the number of ray / bounding box + ray / sphere intersections (the number of ray / sphere intersections is negligible compared to the number of ray / bounding box intersections).

The fourth application implements interactive rendering of scattered point data obtained from laser range scanning. Ray tracing is particularly well suited for large point cloud rendering, because the theoretical complexity for searching the acceleration data structure is logarithmic with respect to the geometric primitive count. We represent single points by a custom sphere type that also stores an RGB color containing the lighting information obtained from scanning. That way we can employ a most simple Visionaray kernel that only computes primary visibility and returns either the color of the sphere that was intersected, or the background color. Pseudo code for that can be found in Listing 5. We carefully design the custom sphere type to tightly fit into 16 bytes for efficient cache utilization. We therefore dedicate 12 bytes to the spheres' center, and one byte to the radius and each RGB color channel, respectively. For the radius and the color channels, we use the *visionaray::unorm<N>* template, which stores data with N bit fixed precision, and has a normalized floating point representation to seamlessly convert to clamped values between 0.0 and 1.0. We implement code paths using the multi threaded scheduler for CPUs (shown in Listing 5), and with the CUDA scheduler to run on NVIDIA GPUs. We evaluate the traversal costs on an NVIDIA Quadro K6000 GPU and with the data set that is depicted in Figure 4. The data set is comprised of 152,209,634 individual points. We build a BVH with Visionaray that consists of 96,059,881 nodes, where 48,029,941 nodes are leaves. The entire data set amounts to 5.7 GB memory. When rendering the viewport from Figure 4 with a resolution of 1920 $\times$ 1200 pixels, we obtain roughly 13 frames per second, which we found suitable for a back projection-based VR environment with head tracking. Figure 5 shows the traversal costs for BVH intersection. We obtain these by summing all intersections of rays with bounding boxes (which represent the major cost factor) and with spheres. The results show that our approach is suitable for interactive rendering of very large point cloud data sets without employing a level of detail approximation.

5 Discussion

While ray tracing libraries that are optimized for specific platforms are omnipresent, it is currently unclear what the most effective approach to implement a cross platform ray tracing library is.
DSL approaches are flexible and provide expressive language constructs that can be compiled into very efficient runtime executable code. On the downside, DSL programs do not seamlessly integrate with the toolchains that application programmers typically use, and require learning programming skills that are specific to the DSL. Furthermore, developing a DSL requires a team of experts with knowledge of *compiler construction*, which is typically not required when developing a mere C++ library.
An alternative approach is to implement a cross platform ray tracing library with OpenCL. The AMD Radeon Rays library e.g. targets GPU architectures with an OpenCL backend, and resorts to delegating ray traversal to the Intel Embree library on CPUs. The Radeon Rays developers opted to use OpenCL 1.2 as the lowest common denominator on GPU architectures. OpenCL 1.2's ANSI C99-based type system is less sophisticated than the C++ type system. OpenCL 1.2 only supports compile time meta programming through type unaware macros and does not support the single source paradigm where host and device code can be mixed across source files. This makes the OpenCL approach less flexible and more error-prone than programming with modern C++.
Using generic C++ to develop a ray tracing library is flexible because modalities are combined by the application programmer rather than by the library programmer. With generic C++ libraries, the compiler retains the full potential to optimize for runtime performance because the relevant code paths are assembled when compiling the ray traversal code in the context of the application. With inlining optimizations, memory efficient POD data structures, and complex language constructs such as virtual function calls not being allowed, a modern C++ compiler can compile very efficient code that performs on the same scale as hand optimized code does. The generic programming approach guarantees efficient instruction cache usage, which is crucial for good runtime performance on today's architectures. If the ray traversal code e.g. only foresees intersecting rays with quadrilaterals, it is guaranteed that no unnecessary instructions for ray / triangle intersection will end up in the instruction cache by accident, because that code path is never even compiled. This tendentially also results in smaller overall binary size. The flexibility of the generic program approach reaches so far that Visionaray can even interface with external ray tracing libraries. It is e.g. possible to write a Visionaray program that uses Embree for BVH traversal and the Visionaray algorithms and data structures for shading and texturing. The generic programming approach has however several downsides. Since templates are instan-

tiated by the application programmer, compile times often tend to shift so that more time is spent to compile the application than to compile the library. When custom data structures are supported, the responsibility of optimizing those for good cache utilization is placed on the shoulders of the application programmer as well (it is however worth mentioning that stateful libraries like Embree also support custom data structures). Another downside of the generic programming approach is cross platform availability of compilers and libraries. ANSI C99 and derivatives are in general more widely supported. With NVIDIA's commitment regarding the CUDA programming language, and AMD's newly released ROCm heterogeneous compute infrastructure, we however believe that C++ cross platform availability is in general acceptable even today, and is about to improve significantly in the near future.

6 Conclusions and Future Work

We presented the software architecture of the cross-platform C++ ray tracing library Visionaray. We proved that the generic programming approach is suitable for real-time rendering on modern HPC architectures.Visionaray's main advantage over competing C++ libraries is cross-platform code compatibility. We evaluated our approach with four case studies that are centered around real-time and interactive rendering in VR. Performance measurements show that our approach scales well on modern HPC architectures. Both Visionaray and the four case study applications are available for download under an open source license.

In the future we intend to improve Visionaray's interoperability with the OpenCOVER VR renderer, which organizes geometry using a scene graph. In order to ray trace scene graphs with transform nodes that may change over time, rays must be transformed into the local coordinate systems of the respective subgraphs. We further plan to more rigorously evaluate the performance overhead of using C++ variants for materials and geometric primitives to simulate polymorphism at compile time. Visionaray is currently compatible with Intel and ARM SIMD instruction sets, and with NVIDIA CUDA. Compatibility with AMD ROCm is experimental and part of our ongoing work.

Acknowledgements

The authors wish to thank Dorit Borrmann, Jan Elseberg, HamidReza Houshiar, and Andreas Nüchter from Jacobs University Bremen gGmbH, Germany, for publicly providing the construction site data set. The Fairy Forest scene is part of the Utah 3D Animation Repository by Ingo Wald. The COVISE CFD example was used with friendly permission by Uwe Wössner. The Conference, Sibenik Cathedral, and Sponza scenes are hosted online by Morgan McGuire [9]. The MRI data set is the T1-weighted MNI152 standard brain that ships with the FSL FMRI tools [25].

References

[1] Boost C++ libraries. `http://www.boost.org/`. Accessed: 2017-01-16.

[2] Radeon rays technology for developers. `http://developer.amd.com/tools-and-sdks/graphics-development/radeonpro/radeonrays-technology-developers/`. Accessed: 2017-01-17.

[3] Standard template library programmer's guide. `http://www.sgi.com/tech/stl/`. Accessed: 2017-01-16.

[4] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[5] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-m. W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 26, pages 359–373. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[6] P. Ganestam and M. Doggett. SAH Guided Spatial Split Partitioning for Fast BVH Construction. *Computer Graphics Forum*, 2016.

[7] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society.

[8] R. Leißa, K. Boesche, S. Hack, R. Membarth, and P. Slusallek. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 11–20. ACM, 10 2015. Best Paper Award.

[9] M. McGuire. Computer graphics archive, August 2011.

[10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[11] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[12] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13, May 2012.

[13] D. Rantzau, U. Lang, R. Lang, H. Nebel, A. Wierse, and R. Ruehle. *Collaborative and Interactive Visualization in a Distributed High Performance Software Environment*, pages 207–216. Springer London, London, 1996.

[14] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.

[15] P. Slusallek and I. Georgiev. Rtfact: Generic concepts for flexible and high performance ray tracing. In R. J. Trew, editor, *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008*, pages 115–122, RT08 Reception Warehouse Grill 4499 Admiralty Way Marina del Rey, CA 90292, 2008. IEEE Computer Society, Eurographics Association, IEEE.

[16] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA, 2009. ACM.

[17] B. Stroustrup. Software development for infrastructure. *Computer*, 45(1):47–58, 2011.

[18] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[19] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.*, pages 77–85, Oct 2003.

[20] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In S. J. Gortle and K. Myszkowski, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 2001.

[21] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

[22] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, July 2014.

[23] R. Wang and X. Qian. *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, 2010.

[24] N. P. Wilt. *Object-Oriented Ray Tracing in C++*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[25] M. W. Woolrich, S. Jbabdi, B. Patenaude, M. Chappell, S. Makni, T. Behrens, C. Beckmann, M. Jenkinson, and S. M. Smith. Bayesian analysis of neuroimaging data in FSL. *NeuroImage*, 45:S173–86, 2009 Mar 2009.

[26] S. Zellmann, M. Hoevels, and U. Lang. Ray traced volume clipping using multi-hit BVH traversal. In *Proceedings of Visualization and Data Analysis (VDA)*. IS&T, 2017.

APPENDIX

```
#include <visionaray/traversal.h>
#include <vector>

int pick_selection() {
    using namespace visionaray;
    // Build up vector with menu items.
    std::vector<quad> menus({menu1,menu2,...});
    // Assuming a directional VR pointing
    // device such as a magic wand.
    ray r = { device.org, device.dir };
    // Call the closest hit algorithm.
    auto hr = closest_hit(r, menus.begin(),
                             menus.end());
    // Return the ID of the closest menu.
    return hr.hit ? hr.prim_id : -1;
}
```

Listing 1: VR menu selection with a few lines of Visionaray code.

```
// 2-D RGBA texture in CUDA device memory.
cuda_texture<vec4, 2> tex;
// CUDA kernel processing textures.
__global__ void kern(cuda_texture_ref<vec4, 2> tr)
{
    vec2 tex_coord;
    vec4 rgba = tex2D(tr, tex_coord);
}
// The texture reference passed to the CUDA
// kernel stores only a device pointer.
kern<<<...>>>(cuda_texture_ref<vec4, 2>(tex));
kern<<<...>>>(cuda_texture_ref<vec4, 2>(tex));
kern<<<...>>>(cuda_texture_ref<vec4, 2>(tex));
```

Listing 2: Accessing persistent GPU data with reference objects.

```
// Type aliases defined by the application.
#if ARCH==SSE || ARCH==ARM_NEON
typedef simd::float4 Scalar;
#else
typedef float Scalar;
#endif
// In the SIMD case, this type stores a
// 4x structure of array (SoA) vector.
typedef vector<3, Scalar> Vec3;
// Portable cross platform function.
void func(Vec3 v1, Vec3 v2) {
    // Handle scalar and SIMD likewise.
    Vec3 v3 = v1 + v2;
    Vec3 v4 = cross(v2, v3);
    auto d = dot(v3, v4);
    basic_ray<Scalar> theRay;
    theRay.ori = v3
    theRay.dir = normalize(v4);
    // Sometimes we need to explicitly
    // handle SoA packets, though.
    if (any(dot(v3, v4) == .0f)) {
        /* however, this code is cross platform
         * compatible because in the scalar case,
         * any() trivially returns the boolean
         * result of the scalar comparison.*/ }
}
```

Listing 3: Example showing how type aliases can help to reduce cross platform application code complexity.

```
// Primitive, either triangle or sphere.
generic_primitive<basic_triangle<3, float>,
                  basic_sphere<float>> p;
// Intersect as triangle.
p = basic_triangle<3, float>(v1, e2, e3);
intersect(ray, p);
// Intersect as sphere.
p = basic_sphere<float>(center, radius);
intersect(ray, p);
// The template argument lists are
// "variadic", can take N types.
generic_primitive<PT1,PT2,PT3,PT4,PT5> p2;
// More examples:
generic_material<plastic<float>, matte<float>,
                 emissive<float>> m;
m.shade(rec);
generic_light<spot_light<float>,
              area_light<basic_sphere<float>>> l;
l.sample(dir);
```

Listing 4: Wrapper classes that implement Visionaray concepts and simulate polymorphism with generic constructs.

```
// Custom primitive type. We represent
// the point cloud with many spheres
// and pack the color into the sphere
// for better data locality.
struct color_sphere {
    vec3 center;        // 12 byte
    unorm<8> radius;    //  1 byte
    vector<unorm<8>, 3> //  3 byte
};

// Implement Primitive concept.
template <class T>
hit_record<T> intersect(basic_ray<T> r,
                        color_sphere) {
    /* ray/sphere intersection */ }

// get_bounds(), used during BVH construction.
aabb get_bounds(color_sphere) {
    /* compute bounding box */ }

// Fill a list with the point cloud.
std::vector<color_sphere> pts = ...;

// Construct the BVH.
auto b = build<index_bvh<color_sphere>>(pts.data(),
                                        pts.size());

// Multi-threaded CPU scheduler.
tiled_sched<basic_ray<Scalar>> sched(num_threads);
// Implement ray tracing kernel as C++11 lambda.
sched.frame(params, [&](basic_ray<Scalar> ray) {
    result_record<Scalar> result;
    // Determine primary visibility.
    auto hr = closest_hit(ray, &b, &b+1);
    // Store hit state in result record.
    result.hit = hr.hit;
    result.isect_pos = ray.ori + ray.dir * hr.t;
    // In case of SIMD, select() fills SIMD lanes
    // selectively based on the condition.
    // Else resorts to operator <c> ? <a> : <b>
    result.color = select(hr.hit, hr.color, bgclr);
    return result;
});
```

Listing 5: Point cloud ray tracing example. The points are represented by spheres and only primary visibility is considered. The example also shows how a custom primitive type is implemented that is compatible with Visionaray BVHs.