RTfact: Generic Concepts for Flexible and High Performance Ray Tracing

Iliyan Georgiev [*]
University of Saarland

Philipp Slusallek [†]
DFKI Saarbrücken
University of Saarland

Figure 1: A volume data set, a polygonal scene, and a point cloud rendered using our generic ray tracing library.

Abstract

Thanks to more than a decade of research and the fast evolution of computer hardware, ray tracing is likely to become a commodity choice for adding complex lighting effects to real-time rendering engines. Nonetheless, interactive ray tracing research has been mostly concentrated on few specific combinations of data structures and algorithms. In this paper we present RTfact – an attempt to bring the different aspects of ray tracing together in a component oriented, generic, and portable way, without sacrificing the performance benefits of hand-tuned single-purpose implementations. RTfact is a template library consisting of packet-centric components combined into an efficient ray tracing framework. Our generic design approach with loosely coupled algorithms and data structures allows for easy integration of new algorithms with maximum runtime performance, while leveraging as much of the existing code base as possible. The efficiency of templates allows us to achieve fine component granularity and to incorporate a flexible physically-based surface shading model, which enables exploitation of ray coherence. As a proof of concept we apply the library to a variety of rendering tasks and demonstrate its ability to deliver performance equal to currently existing optimized implementations.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 Introduction

Ray tracing is well known as a general and flexible rendering algorithm capable of simulating physically-based lighting, but it has also been famous for its high computational requirements. The recent developments of highly optimized packet ray casting algorithms and the advances in hardware have allowed ray tracing to achieve real-time performance. Since the necessary computing power became available, research has focused on raw ray tracing performance on desktop machines. In order to achieve optimal performance, some interactive systems had to compromise flexibility [24, 5], remaining fast only in specific configurations of algorithms and data structures. Others have provided certain degree of functional freedom exposed through custom application programming interfaces (APIs) [7, 4], but relied on fixed rendering pipelines and suffered from the run-time overhead of virtual polymorphism.

RTfact is inspired by the need of a modern multi-purpose real-time ray tracer prototyping library, which provides maximum performance on the latest generation of CPUs without compromising flexibility. Our goal is not to deliver a self-contained rendering system, but to create a flexible and extensible environment for testing and implementing custom ray tracing-based solutions. We aim at combining the flexibility of an off-line rendering system with the performance of modern ray tracing techniques. Furthermore, we separate ray tracing algorithms from data structures and redefine the design of ray tracing infrastructure in terms of generic programming. This allows us to simultaneously achieve higher reusability, composability, and efficiency.

1.1 Overview

The versatility of ray tracing as a visibility sampling technique in combination with modern hardware and coherent ray traversal and intersection algorithms impose certain challenges on the design of a real-time rendering system. These include the choice of supported functionality and the ease of adapting the rendering system to the particular needs of applications. A particular application, for example, might need to combine different acceleration structures, traversed with ray packets of different size, and use them in different contexts (e.g. for rendering, collision detection, object interaction). Thus, a modern ray tracing framework should be flexible enough to allow such freedom and to also deliver the best possible performance for various configurations of algorithms and data structures.

We do not try to give a one-size-fits-all solution, but take a more general design approach instead. We build a generic ray tracing library consisting of multiple levels of abstractions for both algorithms and data structures. Starting from basic data types for data parallel computation, we incrementally augment the library with functionality in the form of generic composable components. We do not fix a single pipeline, but instead provide the building blocks and a framework for combining them. Employing the full power of C++ templates, it is then possible within our framework, for example, to implement a single generic triangle intersection algorithm, handling ray packets of different size, nature, and common origin

[*]e-mail: georgiev@cs.uni-sb.de
[†]e-mail: slusallek@cs.uni-sb.de

properties. We let the compiler generate optimized code for each specific ray packet size and type used.

Compile-time dependency resolution and template instantiation not only enable low-level optimization by modern compilers, but also allow special case code to be directly embedded into a generic algorithm, without the need of virtual functions and other complex control flow. Thus, no unnecessary run-time overhead is imposed.

RTfact employs a physically-based shading model, which decouples surface shading from visibility computations and light integration. This separation facilitates code reuse and enables better exploitation of ray coherence.

In the remainder of this paper we describe the design of RTfact. Focusing on flexibility and performance, our thread-aware design is orthogonal to higher level parallelization schemes and APIs, which can be easily implemented on top as layers between the core library and user applications. We motivate the decisions made throughout designing and demonstrate a prototype implementation running on various operating systems for different visualization tasks.

2 Evolution of Interactive Ray Tracing

Accelerating ray tracing for use in interactive rendering applications has been a long standing goal for computer graphics research.

The first interactive ray tracing systems ran on large supercomputers [12, 15, 16]. They traced a single ray at a time and exploited the inherent parallelism of ray tracing by assigning pixel tiles to different processors. Wald et al. [24] first demonstrated interactive ray tracing on desktop PCs using SIMD packet ray tracing and clustering machines using commodity networks.

State-of-the-art ray casting algorithms exploit ray bundle properties to efficiently amortize acceleration structure traversal and primitive intersection costs among many rays [20, 23]. Such algorithms have enabled ray tracing to perform competitively to hardware rasterization in small game environments [5].

Modern processors provide more and more computational power as their design is shifting towards parallelism on two levels – multiple processing cores and explicit data parallelism. Current CPUs accommodate up to four separate cores, each implementing the SSE instruction set, while modern GPUs utilize single instruction multiple thread (SIMT) data parallelism on hundreds of scalar cores [13]. Recent ray tracing implementations on specialized hardware include the CELL processor [3] and GPUs [18, 9].

Massive parallelism and increased hardware programmability make it very likely that future rendering engines will be almost entirely implemented in software [10]. This implicates that flexible interactive systems which can deliver ray tracing to application developers and thus to end users will become of primary importance.

2.1 Related Work

Star-Ray [16] was the first flexible ray tracing system to achieve interactive performance. It provided an extendable object-oriented programming interface and was later adapted to cluster systems [6]. Star-Ray used brute-force single ray tracing algorithms, without employing ray packet algorithms which were developed later.

Dietrich et al. [7] proposed the OpenRT API which aimed at providing a complete interactive ray tracing solution on a cluster of PCs. The base API provided a scene description interface close to OpenGL, while an object-oriented shading API allowed for writing custom shaders for cameras, materials, lights, and the rendering loop. The shading API operated on single rays only because it was too complicated for the end user to write SIMD shaders. Thus, industrial implementations of OpenRT were restricted to shading and single secondary rays, while only the core ray tracer was hand-coded in SSE, which reduced the overall rendering performance.

The Manta interactive ray tracing system [4] aims at delivering both flexibility and performance, while scaling to a large number of processors. In this system, a configurable parallel pipeline takes care of task scheduling and synchronization, while rendering tasks asynchronously traverse a modular stack, which balances the workload of each thread. Flexibility is achieved by providing callbacks and abstract interfaces for different components in the pipeline and the rendering stack. The system focuses on massive parallelism and its single thread performance is lower than optimized implementations, because of the overhead of virtual interfaces and wide packet assumptions. Manta is also a complete rendering system which makes it hard to be integrated with other applications.

Recently, Parker et al. [14] proposed RTSL as a domain-specific language (DSL) for extending ray tracing systems. RTSL provides a simple and intuitive syntax for implementing custom camera, primitive, light, and material shaders, which can be used in multiple rendering systems. A specialized compiler is used to produce optimized SIMD code from scalar RTSL code. In order to retain a simple syntax, however, RTSL does not allow for controlling the rendering loop or writing acceleration structures, for which support from the underlying system is required.

Stoll et al. [22] proposed Razor as a software architecture for distribution ray tracing of dynamic scenes. The system employs coherent ray casting algorithms and decouples visibility computations from surface shading, in order to avoid redundant shading computations. Similarly to Manta, Razor aims at providing a complete rendering solution, but is mainly designed for subdivision surface rendering and is too slow compared to other systems.

The PBRT system [17] has proved to be very flexible and is widely used in academia. It provides fine-grained and modular interfaces for physically based rendering, employing a decoupled surface shading model. The system does not aim at performance, but on physical realism and flexibility. It traces single rays and its fine-grained infrastructure imposes high overhead, which keeps its performance far from interactive.

3 General Design Considerations

Prior ray tracing systems have always made design decisions trading between flexibility and performance. Extreme examples are the very flexible but slow PBRT and the very efficient but highly specialized Arauna [5]. Additionally, all systems have provided either full application-centric or rendering pipeline-specific solutions. Our goal is to build a generic library, useful in various application contexts and producing the fastest code for a specific problem.

In general, there are two approaches for achieving flexibility in a rendering system. The first and most commonly taken one is to use a general purpose language and object-oriented design. The second option is to design a domain-specific language (like RTSL) with a syntax suitable for our custom needs. It turns out there is another option, which we will explore in this paper.

3.1 Object-Oriented Design

All ray tracing systems, that we know of, have used object-oriented design with abstract virtual interfaces to achieve flexibility and polymorphic behavior of different components, e.g. for invoking shaders. This approach has the advantage of being well studied and allows software components to be connected at run-time, e.g. using plug-ins. This late binding facilitates decoupling in the sense that different components can be compiled more or less independently.

Unfortunately, the flexibility of dynamic polymorphism comes at a performance price. Late binding disables function inlining and interprocedural optimizations otherwise automatically performed by the compiler. Furthermore, each virtual function call imposes execution overhead, even if the provided flexibility is not required at run time. This implies that a fine-grained object hierarchy can seriously degrade the application's performance. That is why previous interactive systems have compromised both flexibility and performance, achieving neither the generality of offline rendering systems nor the speed of hand-tuned implementations.

Another deficiency of object-oriented design is that it encourages coupling algorithms to data structures. Thus, previous systems have provided acceleration structure classes together with traversal algorithms. This reduces component reuse, as different algorithms cannot be applied on the same data structures and vice versa. The problem can be reduced by finer grained abstractions, but which bring additional run-time overhead. A purely object-oriented design approach is therefore not an option for us, because it cannot deliver both flexibility and performance.

3.2 Domain-Specific Language

Another option for a flexible rendering system would be to provide a domain-specific language for writing custom components. The language would provide an intuitive and convenient syntax, while at the same time allowing us to have more control over the low-level code generation via a specialized compiler. However, if we allow expressing ray tracing algorithms and data structures, the language will become more complex, eventually converging to a general purpose language and losing its nice properties. Furthermore, we would have to build complex infrastructure around the compiler from scratch.

DSLs are convenient mostly because of their simple syntax, which makes them useful only for small and isolated problems. We believe that ray tracing is a tool which should also be used for applications other than rendering, such as collision detection. A context-dependent DSL would make such integration much harder. That is why we chose to build a context-free library entirely in a general purpose language and provide convenient abstractions for implementing reusable algorithms and data structures.

3.3 Generic Programming Approach

C++ provides the facilities for a third design paradigm – generic programming. Generic programming [11] is a software methodology for developing reusable and efficient software libraries. It advocates definition of algorithms at an abstract level, completely independent of the underlying data representation, in order to increase component composability and code reuse, while maintaining efficiency. The C++ Standard Template Library (STL) was the first widely used library to adopt these concepts.

While non-generic libraries use interfaces operating on predetermined data types, generic algorithms define the minimal requirements from the data types they are instantiated with. Thus, a generic algorithm can be used with *any* type meeting its requirements.

A *concept* in generic programming is namely the set of requirements of an algorithm. For example, a type is **comparable** if it defines a comparison operator, which can be used by a generic algorithm to order elements of this type. A type that meets these requirements is said to *model* the concept.

Generic algorithms can be expressed in C++ using class and function templates. Since templates are instantiated at compile time, type dependencies are resolved without the overhead of run-time support code, such as virtual function calls. Static binding also enables inline function expansion and interprocedural optimizations. Templates can also have specializations for specific data types. This allows special case code to be resolved at compile time, without the need of separate code paths for the whole program.

In C++ concepts can be described with *pseudosignatures*, which are pseudo class declarations specifying a set of requirements of an algorithm. They can be extracted from an actual implementation of a type that meets these requirements. Concepts are thus analogous to abstract interfaces in the object-oriented design, whereas models correspond to classes implementing these interfaces. However, the weaker requirements of concepts increase composability and allows easier integration of different software libraries.

Examples for successful generic libraries are Matrix Template Library [21], Boost [1], and Intel's Threading Building Blocks [19].

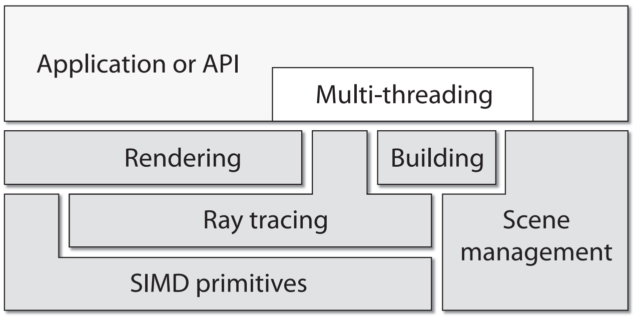

Figure 2: The multi-layer architecture of RTfact. The SIMD primitives form the basis for other generic components which are layered on top. Scene management and rendering are independent from each other and are connected through the ray tracing components. Thread management can be layered on top of the core components, or integrated into the application, which itself can be an API backend.

4 Software Architecture

The generic programming paradigms match the goals of RTfact well. C++ templates provide abstraction and composability while retaining the opportunity for optimal performance and compiler optimization. This implicates that we can achieve fine abstraction granularity, yet delivering the performance of hand-tuned implementations.

Performance on modern hardware is tightly coupled with parallelism. While asynchronous thread execution does have some impact on the low-level design of a ray tracing system, algorithms and data structures need to be updated or even developed from scratch in order to take full advantage of explicit data-level parallelism. Such low-level optimizations are crucial for the overall performance of modern ray tracing algorithms, but imply unintuitive data layout and programming model. Our framework provides a set of generic packet data containers for convenient and efficient SIMD computation. Such low-level abstractions will become even more relevant, as future hardware will support 8-wide [8] and even 16-wide (Intel's Larrabee [2], not yet officially announced) SIMD operations.

The algorithms in RTfact operate entirely on packet data, in order to take full advantage of modern packet ray tracing techniques. Packet concepts provide a common interface independent of the size and internal organization of the packet. Thus, the packet size is a parameter to every generic algorithm that operates on rays. As a result, algorithms can handle packets of different sizes simultaneously and can have manually specialized versions for specific sizes, which are *automatically* resolved and optimized by the compiler.

Our general design objectives are to decouple algorithms from data representation and to separate rendering from ray tracing and scene management. The library consists of five main groups of components: SIMD primitives (Section 4.1), ray tracing (Section 4.2), structure building, scene management (Section 4.3), and rendering (Section 4.4).

Figure 2 illustrates the architecture of RTfact. The application has direct access to the scene management and ray tracing components and can use them for rendering and custom tasks. The components of the library are thread-aware but do not provide any thread management functionality, as this can be very application-specific.

4.1 Generic SIMD Computation with Packets

The basis of our library is formed by a collection of generic types for SIMD computation. These packets, as we call them, are *fixed-sized* containers of values and form the basic arithmetic types of the library, along with build-in types like float, integer, etc. We define four basic packet types, all parameterized by size.

`Packet<size, type>` represents an ordered set of values. Currently, `type` can be `float` or `int`.

`Vec3f<size>` represents a three-component float vector packet with a structure-of-packets layout.

`PacketMask<size>` is an ordered set of booleans, storing the result of a comparison operation between two `Packets` or `Vec3fs`, and defines a conditional `blend` operation, which blends two `Packets` or `Vec3fs` according to the the stored mask.

`BitMask<size>` is an ordered set of bits.

All four classes simultaneously model three concepts – `Value`, `ValueContainer`, and `ContainerContainer`.

The `Value<type>` concept considers the packet a single entity and defines all arithmetic operations defined on `type` (addition, multiplication, bit-wise AND, etc.). These operations are applied component-wise to all values in the corresponding packet.

`ValueContainer<type>` defines indexing operators for accessing individual values stored in the packet.

`ContainerContainer<packet_type>` defines indexing operators for accessing packets' sub-containers. A sub-container is of type `packet_type` with a size depending on the size of the parent container and the SIMD width of the underlying architecture.

Packets are internally implemented as SIMD arrays. On SSE-compatible processors, the sub-containers of 1- and 4-sized packets are the packets themselves, whereas for larger packets this size is 4.

As all four packet classes model each of these concepts, they have a three-fold nature. However, most algorithms use the `ContainerContainer` context. This is because series of operations on a large packet are more cache efficient to be performed for each sub-container (SSE packet) sequentially, rather than performing the operations one by one on the entire packet.

While currently RTfact supports SSE only, the model itself does not have restrictions on the instruction set. Thus, support for the Intel AVX [8] can be easily added. In fact, abstractions over native SIMD operations improve portability – when adding support for wider operations, updates are needed only for code using horizontal operations defined only for 4-sized packets, such as shuffling.

As packets of different size are actually different C++ types, they are subject to specialization. Our implementation has specializations for packets of size 1 and 4. The most notable is the specialization for `Vec3f<1>`, which is internally implemented in SSE. These peculiarities are well hidden behind the three main concepts, but specialized versions of algorithms can take advantage of the additional functionality and internal layout of specialized packets. Listing 1 illustrates some common packet operations.

4.1.1 Ray Packets

RTfact does not distinguish between individual rays and ray packets. Rays are simply packet types aggregating other packets:

```
template<unsigned int size>
struct RayPacket {
  Vec3f<size>         org, dir;
  Packet<size, float> tMin, tMax;
};
```

Thus, a single ray is simply a ray packet of size 1.

Template instantiation adds two desirable properties to ray packets. First, memory for ray packets is allocated at compile time. This is relevant to performance, as many rays are created and destroyed each frame. Second, ray packets of different sizes can exist simultaneously within the system – a feature missing in other interactive ray tracing systems. This allows us to efficiently trace the same or different acceleration structures with packets of different sizes.

Our implementation uses extended ray packets with information about active rays, corner rays, and bounding planes, enabling flexible and efficient frustum-based traversal and intersection. However, we do not store intersection data inside ray packets, as intersection structure types are defined by intersectors (see Section 4.2.2).

```
//specialized packets have more convenient constructors
Vec3f<1>         v1(0.1, 0.2, 0.3);
Packet<4, float> p4(1, 2, 3, 4);
//shuffling (only for 4-sized packets), p4s == (1,1,2,4)
Packet<4, float> p4s = p4.shuffle<0,0,1,3>();
//replication of values (i.e. 1-sized packets)
Vec3f<4> v4 = Vec3f<4>::replicate(v1);
//sub-container replication (shown for 64-sized packets)
Vec3f<64> dir = Vec3f<64>::replicate(v4);
Vec3f<64> color, offset = ..., normal = ...;
Packet<64, float> dn;
PacketMask<64> mask;
BitMask<64> bitMask;
//scalar-like packet operations (Value concept)
dn = Dot(dir + offset, normal);
mask = (dn > Packet<64, float>::C_0());
color = mask.blend(Vec3f<64>::C_1_0_0(), //red constant
                   Vec3f<64>::C_0_0_0());//black constant
bitMask = mask.getBitMask();
//compoinent-wise packet operations (ValueContainer)
for(int i=0; i < 64; i++) {
  dn[i] = Dot(dir[i] + offset[i], normal[i]);
  mask[i] = (dn[i] > 0);
  color.set(i, mask[i] ? Vec3f<1>::C_1_0_0(),
                         Vec3f<1>::C_0_0_0());
  bitMask.set(i, mask[i]); }
//container-wise packet operations (ContainerContainer)
for(int i=0; i < Packet<64,float>::CONTAINER_COUNT;i++) {
  dn(i) = Dot(dir(i) + offset(i), normal(i));
  mask(i) = (dn(i) > Packet<64,float>::Container::C_0());
  color(i) = mask(i).blend(Vec3f<64>::Container::C_1_0_0(),
                           Vec3f<64>::Container::C_0_0_0());
  bitMask.setContainer(i, mask(i).getBitMask()); }
```

Listing 1: SIMD computation with packets. Most binary operations (such as addition) can be performed in three different ways.

4.2 Ray Tracing Components

Packets provide us with the basis on which we build generic ray tracing algorithms. RTfact makes a clear distinction between data structures and algorithms that operate on them. Building and traversing an acceleration structure are independent from its actual type and implementation, as long as it provides the necessary functionality for storing and accessing data. This allows us to apply different algorithms on the same or different acceleration structures.

4.2.1 Primitives and Acceleration Structures

In RTfact, all ray tracing structures model the very general `Primitive` concept. These include geometric primitives, compound primitives, and acceleration structures. However, they do not directly provide intersection functionality, which is instead provided by `Intersectors`. For applications which require simultaneous support for different primitives, an ID to a run-time polymorphic intersector can be stored with each structure. This allows us to avoid data-dependent indirect function calls completely or add them only when required and at the appropriate granularity.

We design acceleration structures to be independent of the types of objects they aggregate. Such types can be not only geometric primitives, but any other type (e.g. `Photon`). This also allows acceleration structures of same or different types to be nested.

Listing 2 shows a concept for a kd-tree. The concept only defines building and traversing functionality and completely hides the details about internal node representation, and provides an iterator interface similar to STL containers. It also does not define any creation and initialization functionality, as this is specific to the implementation of the structure and is controlled by the application.

4.2.2 Intersectors

Given a ray packet and a primitive, `Intersectors` return an intersection structure. Similarly to PBRT, we design primitive and

```
template<class Element>
class KdTree : public Primitive {
public:
  class NodeIterator;
  class ElementIterator;
  // interface for structure building
  void createInnerNode(NodeIterator node,
                       int axis, float splitValue);
  template<class Iterator>
  void createLeaf(NodeIterator leaf, const BBox& bounds,
                  Iterator begin, Iterator end);
  // interface for structure traversal
  NodeIterator getRoot() const;
  NodeIterator getLeftChild(NodeIterator node) const;
  NodeIterator getRightChild(NodeIterator node) const;
  int getSplitAxis(NodeIterator node) const;
  float getSplitValue(NodeIterator node) const;
  std::pair<ElementIterator, ElementIterator>
    getElements(NodeIterator leaf) const;
};
```

Listing 2: A psudosignature of a generic kd-tree working with STL-like iterators. A BVH concept would be similar.

acceleration structure intersectors to have a unified interface, which allows intersectors to be nested consistently with the acceleration structures. This gives us the ability to compose and traverse arbitrary deep acceleration structure hierarchies as easy as combining templates:

```
// data structure
BVH<KdTree<Triangle>> hierarchy;
// corresponding intersector
BVHIntersector<KdTreeIntersector<
  SimpleTriangleIntersector>> intersector;
```

In order to ensure correct nesting, every intersector defines the type of the returned intersection structure. Primitive intersectors return structures containing references to primitives and intersection data. Acceleration structure intersectors, such as the `KdTreeIntersector`, simply reuse the intersection type of the nested intersector. Intersection structures of instance intersectors would inherit the nested intersection structure and additionally store a reference to the intersected instance.

Some applications require tighter integration of acceleration structures and primitives. For example, a kd-tree for subdivision surfaces might incorporate the geometry representation. In such case, the acceleration structure and primitive intersectors can also be merged accordingly. In order to facilitate component reuse, our design only encourages but does not require separation.

Intersectors implement ray tracing algorithms and this is where generic programming with templates can show its greatest potential. We add two more template parameters to intersection routines (along with the size of the packet), which enable special case code. These parameters are flags for packets with common origin and what intersection data has to be computed, e.g. whether intersection normals or partial derivatives are needed. As a result, we can bring component reuse to an extreme level by allowing an intersector to have *a single generic implementation*. The intersection routine can be independent of the implementation of the acceleration structure, the type of objects it aggregates, the nature of the ray packet, its size and ray origin properties, and the shading data needed. Listing 3 illustrates such a routine for a kd-tree intersector.

4.3 Structure Building and Scene Management

As with algorithms and acceleration structures, the type and organization of the scene data can vary among applications. We define a `Scene` concept which provides basic functionality for querying materials and intersectors. A `BasicScene` concept in addition defines flat geometry and light source lists, while the others can define functionality for managing more complex scene graphs.

Acceleration structures are built by `Builders`. They operate on bounding boxes and can thus build structures over any object type that has bounds. During building, object bounds can be clipped to nodes' bounds using either a provided object type-specific clipper or a simple default box splitter. However, scene management and acceleration structure building are not the main focus in this article and due to space limitations we omit further details.

```
template<class ElemIsect>    //nested element intersector
template<int intersDataMask, //intersection data needed
         bool commonOrg,     //common ray origin?
         unsigned int size,  //size of the ray packet
         class KdTree>       //models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
  RayPacket<size>& rayPacket,//the actual rays
  KdTree& tree,              //the actual kd-tree
  ElemIsect::Intersection<size>& r)//intersection defined
{                            // by the nested intersector
  typedef BitMask<size>                t_BitMask;
  typedef Packet<size, float>          t_Packet;
  typedef typename t_Packet::Container t_PContainer;
  /* omitted: initialize traversal */
  KdTree::NodeIterator node = tree.getRoot();
  int splitDim; // split dimension (3 means leaf node)
  while(true) {
    while((splitDim = tree.getSplitAxis(node)) != 3) {
      t_PContainer splitValue =
        t_PContainer::replicate(tree.getSplitValue(node));
      t_BitMask nearChildMask, farChildMask;
      t_PContainer tSplitFactor;

      if(commonOrg)     // compile-time constant decision
        tSplitFactor = splitValue -
          rayPacket.org(0).get(splitDimension);

      for(int i=0;i<RayPacket<size>::CONTAINER_COUNT;++i){
        if(!commonOrg) // compile-time constant decision
          tSplitFactor = splitValue -
            rayPacket.org(i).get(splitDimension);

        const t_PContainer tSplit = tSplitFactor *
          rayPacket.invDir(i).get(splitDimension);
        nearChildMask.setContainer(
          i, (tSplit(i) > currentTMax(i)).getIntMask());
        farChildMask.setContainer(
          i, (currentTMin(i) > tSplit(i)).getIntMask());
      }
      /*omitted: get first child from masks and descend */
    }
    // a leaf node has been reached
    std::pair<KdTree::ElementIterator,
              KdTree::ElementIterator> elemIterators =
      aTree.getElements(currentIterator);
    if(elemIterators.first != elemIterators.second) {
      do { //invoke nested intersector for leaf elements
        m_inters.intersect<intersDataMask, commonOrg>(
          rayPacket, *(elemIterators.first++), r);
      } while(elemIterators.first != elemIterators.second);
      //check whether active rays found intersections
      terminationMask |= rayActiveMask &
        (result.dist <= currentTMax).getBitMask();
      if(terminationMask.isTrue()) return;
    }
    /* omitted: pop a node from the stack and mask rays*/
  }
}
```

Listing 3: A generic kd-tree traversal routine. Only the kd-tree data and the exact rays are passed at run-time. All other parameters are known at compile time, including a mask specifying what intersection data is needed.

```
class Material {
public:
  template<unsigned int size>    //packet size
  Vec3f<size> emittance(
    Vec3f<size>& w_o,            //outgoing direction
    ShadingData<size>& sh);      //hit point,normal,etc.

  template<unsigned int size,    //packet size
           unsigned int bsdfType>//BSDF parts to evaluate
  Vec3f<size> evaluate(
    Vec3f<size>& w_o,            //outgoing direction
    Vec3f<size>& w_i,            //incoming direction
    ShadingData<size>& sh);      //hit point,normal,etc.

  template<unsigned int size,    //packet size
           unsigned int bsdfType>//BSDF parts to sample
  Vec3f<size> sample(
    Vec3f<size>& w_o,            //outgoing direction
    Vec3f<size>& w_iOutput,      //incoming direction
    ShadingData<size>& sh,       //hit point,normal,etc.
    Packet<size,float>& pdfOutput);//sample probability

  /* omitted: evaluate() and sample() variants */
};
```

Listing 4: A concept for physically-based materials. Materials represent surface reflection models and are independent from ray tracing.

4.4 Rendering Components

In RTfact, rendering is layered on top of ray tracing and is independent from scene management. The application makes the connection between the scene data and rendering components by constructing acceleration structures and passing them along with ray tracing algorithms to a suitably configured rendering pipeline.

4.4.1 Shading Model

Most rendering systems employ an “all-in-one” surface shading model – shaders are attached to geometry and are executed whenever a ray hits a particular surface. They consist of imperative code that fully defines what computations are performed at a hit point. This approach has the benefit of being simple to implement and gives freedom – shaders can shade surfaces in arbitrary ways.

Unfortunately, “all-in-one” surface shading has two major drawbacks. First, it reduces flexibility – one needs to reimplement a particular reflectance model for each light simulation algorithm and vice versa. Furthermore, in a packet-based framework, this surface shading model *by design* reduces exploitation of coherence for secondary rays, as each shader independently handles rays hitting the surface it is attached to. As a consequence, coherent secondary rays emerging from different surfaces cannot be traced together.

RTfact supports both traditional shaders as well as a decoupled physically-based shading model. Similarly to PBRT, we define surface reflection models as `Materials` and light integration algorithms as `Integrators`. Materials represent BSDFs and can be evaluated, sampled, and its emission queried, while `Light` sources provide an interface for sampling illumination directions and light rays. Evaluation and sampling of materials can be performed on different BSDFs parts (transmission, reflection, specular, etc.) by optionally providing sample values and obtaining sampling probabilities (Listing 4). Listing 5 illustrates the `Integrator` concept.

In the context of generic programming, integrators are the algorithms that perform lighting simulation, while materials and light sources are the data structures that provide the appearance of the objects in a scene. An evident consequence of this separation is that all rays during rendering are shot at a central place, namely the light integration algorithm. This gives integrators the potential to shoot and regroup rays in arbitrary fashion. In our current implementation integrators mask out irrelevant rays when sampling materials and collect the sampled directions. A `WhittedIntegrator`, for example, traces separate packets for reflection, refraction, and shadow rays, which eventually emerge from different surfaces.

For applications that require simultaneous support for different materials and light sources at run-time, we provide a *transparent* virtual function mechanism between the `Material` and `Light` concepts and their models. While the decoupled shading model can require more virtual function calls than the traditional model for material evaluation and sampling, this overhead is overcompensated by the ability to trace coherent secondary rays together.

The traditional shading model is implemented by an integrator that shoots primary rays and calls shaders for the hit points. Individual shaders then perform the remaining computations themselves.

```
class Integrator {
public:
  template<int size> struct Color;
  template<int size, class Sample, class Scene,
           class Primitive, class Intersector>
  Color<size> eval(
    Sample& sample              //image sample
    RayPacket<size>& rayPacket,//initial ray packet
    Primitive& primitive,       //top-level primitive
    Intersector& intersector,   //top-level intersector
    Scene& scene);              //shading scene data
};
```

Listing 5: Given a ray packet, the integrator evaluates the radiance flowing back along the rays. The integrator is also responsible for shooting all rays. Specific implementations are similar to PBRT’s.

```
template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
  PixelSampler m_sampler;
  Integrator   m_integrator;
public:
  template<int size, // the size of primary ray packets
           class Camera, class Scene, class Primitive,
           class Intersector, class Framebuffer>
  void render(
    Scene& scene, Camera& camera,
    Framebuffer& framebuffer, ImageClipRegion& clip,
    Primitive& primitive, Intersector& intersector)
  {
    PixelSampler::Sample<size> sample;
    PixelSampler::Iterator<size> it =
      m_sampler.getIterator<size>(clip);
    while(it.getNextSample(sample))
    {
      RayPacket<size> rays = camera.genRay(sample);
      Integrator::Color<size> color = m_integrator.eval(
        sample, rays, primitive, intersector, scene);
      m_sampler.writeColor(sample, color, framebuffer);
    }
  }
  /* omitted: preprocess() function for pre-integration*/
};
```

Listing 6: A ray tracing renderer concept. The pipeline defines only basic control flow and is completely independent from the algorithms and data structures used for tracing rays and shading.

4.4.2 Rendering Pipelines

The `Renderer` is a top-level rendering concept which defines basic functionality for processing image tiles. It connects different components in a rendering pipeline.

Listing 6 shows an example renderer, which defines a basic ray tracing pipeline. This renderer makes little assumptions about how different components are functionally coupled and what types of data they communicate. For example, in addition to image samples, the pixel sampler can provide light integration samples to the integrator by defining its sample type accordingly. The integrator can in turn specialize for this specific type and can also return a radiance type that additionally contains depth and opacity values.

Integrators can be also specialized for specific acceleration structures and intersection algorithms. For example a volume integrator can be tightly coupled with a grid acceleration structure, where shading of rays is performed during grid traversal. This coupling, however, is completely transparent to the pipeline, as renderers operate on basic concepts only.

We should note at this point that our template infrastructure does not prohibit virtual polymorphism. For applications that require simultaneous support for different algorithms and data structures, components can have internal virtual mechanisms to enable runtime polymorphic behavior. For example, the `Framebuffer` concept, which defines functionality for storing radiance values, can be modeled by a virtual class that selects at run-time whether values are written to a network or a screen buffer. The same holds for intersectors supporting multiple geometric primitives. This flexibility allows us to only pay for the overhead when it is really needed.

5 Applications

RTfact can be used in a similar way to other generic libraries, such as STL. The user includes the desired RTfact headers in his or her application source files and instantiates the provided data structures and algorithms in the specific application. The components of the library are designed to be highly configurable and extensible, so that the user can chose the appropriate granularity and implement custom functionality where needed, or even adapt components from other libraries. For example, one might want to experiment with a new BVH traversal algorithm and a custom material. In this case, one would only implement two classes, modeling the corresponding concepts, and instantiate with them the desired data structures and algorithms the RTfact library already provides.

We have applied RTfact to several interactive visualization tasks, such as surface, point-based, and direct volume rendering. For all these applications we have used the `RayTracingRenderer` described in the previous section. Creating custom rendering configurations then boils down to combining different template classes:

```
PinholeCamera camera;
OpenGLFrameBuffer fb;
//surface rendering (Fig. 1 middle, 3)
BasicScene<Triangle> sc1; //initialization omitted
BVH<Triangle> bvh;
BVHBuilder builder;
builder.build(bvh, sc1.prim.begin(), sc1.prim.end());
BVHIntersector<PlueckerTriangleIntersector> bvhIsect;
RayTracingRenderer<PixelCenterSampler,
                   DirectIlluminationIntegrator> renderer1;
renderer1.render<64>(sc1, camera, fb, fb.getClipRegion(),
                     bvh, bvhIsect);

//direct volume rendering (Fig. 1 & 4 left)
BasicScene<DensityPoint> sc2; //initialization omitted
Grid3D<DensityPoint> grid;    //initialization omitted
VolumeGridIntersector gridIntersector;
RayTracingRenderer<SuperSampler<4>,
                   VolumeIntegrator> renderer2;
renderer2.render<1>(sc2, camera, fb, fb.getClipRegion(),
                    grid, gridIntersector);

//level-of-detail point cloud rendering (Fig. 1 & 4 right)
BasicScene<Point> sc3; //initialization omitted
LoDKdTree<Point> kdtree;
LoDKdTreeBuilder builder;
builder.build(kdtree, sc3.prim.begin(), sc3.prim.end());
LoDKdTreeIntersector<PointIntersector> lodKdTreeIsect;
RayTracingRenderer<PixelCenterSampler,
                   PointLoDIntegrator> renderer3;
renderer3.render<16>(sc3, camera, fb, fb.getClipRegion(),
                     tree, lodKdTreeIsect);
```

	Sponza	Conference	Soda Hall
OpenRT `K`	4.5	4.2	5.1
Manta `K`	4.7	4.2	5.4
RTfact `K`	6.8	6.4	6.5
Wald et al. [23] `B`	n/a	9.3	11.1
Manta `B`	4.5	4.8	5.6
Arauna `B`	13.2	11.3	n/a
RTfact `B`	13.1	11.6	11.4

Table 1: Ray tracing performance in frames per second for a 1024^2 image with simple shading of our implementation in comparison to other interactive systems. All results except for [23] were gathered on the single core of a mobile 2.6GHz Core 2 processor using the same or comparable data structures, intersection algorithms, and view points. `K` denotes kd-tree with packet traversal as proposed by [24], while `B` denotes BVH packet traversal [23].

The C++ template instantiation mechanism allows multiple rendering engines to exist simultaneously in the application, packets of different size to be traced though different acceleration structures, and data to be visualized using different rendering algorithms. Figures 1, 3, and 4 show some scenes rendered with these renderers.

6 Performance

We have measured the raw performance of RTfact and compared it to other interactive systems. Because a direct fair comparison is very difficult to achieve due to various differences in systems, we apply the same simplified conditions for all systems and measure the time for ray casting and simple shading only.

As it can be seen from Table 1, the performance of RTfact matches the one of Arauna [5]. Arauna is to our knowledge the fastest open source ray tracer currently available, and uses hard-coded acceleration structures and SSE algorithms that are manually tuned for optimal throughput. RTfact is also consistently faster under the same configuration of algorithms and structures than the other more versatile systems, even though it provides greater flexibility. Since components in our system are configured at compile time, the compiler can enable all optimizations and produce optimal code *for each* combination of algorithms and data structures.

We have tested RTfact on various operating systems (Windows, Linux, and MacOS) and compilers (MSVC, Intel, and GCC). Surprisingly, modern compilers have matured enough to optimize well all the templates in our code. We achieved best performance with the Intel C++ Compiler on all systems. Inspection of the produced assembly code showed that it managed to best optimize special case code and loops, making assembly code almost identical to the one produced from hand-tuned C++ code.

To run our test application on multi-core machines, we have used Intel's Threading Building Blocks (TBB) on top of the renderer to recursively split the image plane and invoke the it for each tile. We have run tests on up to 16 cores and achieved near-linear scalability. Note that our library optimizes for single-thread throughput, which is completely orthogonal to multi-threaded execution. Various parallelization schemes can be applied on top of the library to achieve maximum performance on multi-core hardware.

6.1 Shading Model Improvements

To compare the efficiency of the decoupled and the traditional surface shading models, we have counted the total number of traversed nodes and intersection tests for shadow rays over an entire frame using either models. We have run the tests on the Conference scene with 36 different materials visible (see Figure 1 middle) most of which hit by very few rays. Still, the traditional shading model resulted in traversing about 40% more kd-tree nodes and performing 25% more intersection tests for the shadow rays.

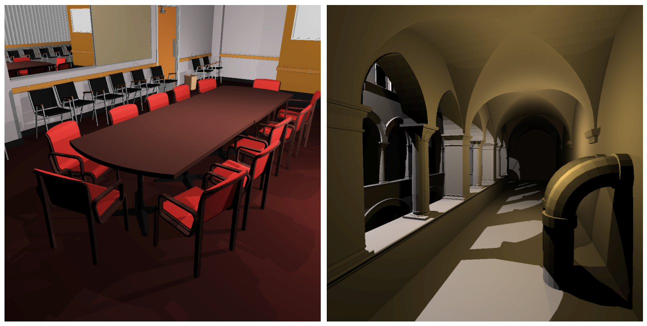

Figure 3: Scenes rendered at a 1024^2 resolution on a mobile Core 2 Duo processor. **Left:** CONFERENCE with a mirror and two light sources (8.2 fps). **Right:** SPONZA with three light sources (7.3 fps).

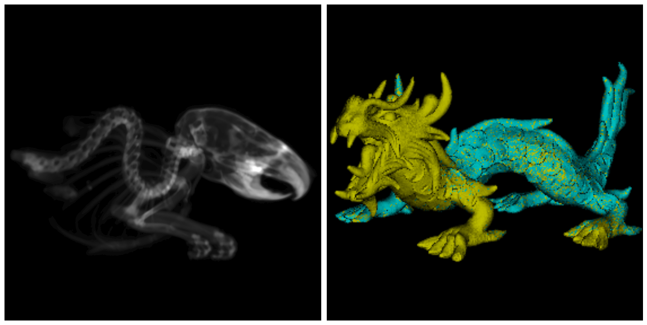

Figure 4: Direct volume and point-based rendering. Left: The Skeleton dataset. Right: The Asian Dragon model with level of detail.

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented RTfact – a design approach for building flexible and high-performance interactive ray tracing libraries. Using generic programming paradigms and standard C++ features, our implementation achieves high flexibility and fine component granularity, while maintaining the efficiency of hand-tuned code.

Instead of providing a stand-alone rendering system, RTfact follows context-free generic design concepts and provides the building blocks for creating custom ray tracing-based solutions. This allows components of the library at various granularities to be used for different tasks and to be easily integrated in custom applications. Separating ray tracing from rendering and algorithms from data structures allows us to achieve seamless component integration and composability not seen in prior interactive systems.

As graphics hardware is moving towards higher programmability and software implementations, we believe that the generic approach taken by RTfact will map very well to more restricted development platforms such as CUDA [13]. We achieve high performance by putting the pressure on the compiler, thereby avoiding complex runtime control flow and overhead.

Although RTfact is still in development, it already shows great potential. We plan to continue extending the coverage of its functionality by providing for example support for collision detection for physics simulation. In addition, we plan to integrate the library into a virtual reality system.

RTfact focuses on versatility, flexibility, and performance in its core components, and its design is orthogonal to binary APIs and parallelization schemes which could be layered on top of the library. We advocate generic design as a key to flexibility and efficiency, especially for computationally intensive applications, such as realtime ray tracing.

ACKNOWLEDGEMENTS

We would like to thank Stefan Popov and Dmitri Rubinstein for the fruitful discussions during the work on this project. We would also like to thank Yavor Kaloyanov and Rossen Dimov for implementing some of the algorithms in the library.

REFERENCES

[1] Boost C++ Libraries. http://www.boost.org.

[2] Ars Technica. Clearing up the Confusion over Intel's Larrabee.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

[4] J. Bigler, A. Stephens, and S. G. Parker. Design for Parallel Interactive Ray Tracing Systems. *IEEE Symposium on Interactive Ray Tracing*, 2006.

[5] J. Bikker. Real-time Ray Tracing through the Eyes of a Game Developer. *IEEE Symposium on Interactive Ray Tracing*, 2007.

[6] D. Demarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG*, 2003.

[7] A. Dietrich, I. Wald, C. Benthin, and P. Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. *Proceedings of the 2003 OpenSG Symposium*.

[8] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency.

[9] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2007.

[10] W. Mark. Future Graphics Architectures. *ACM Queue*, 6(2), 2008.

[11] D. Musser and A. Stepanov. Generic Programming. In *SSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1989.

[12] M. J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, 1995.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2), 2008.

[14] S. Parker, S. Boulos, J. Bigler, and A. Robison. RTSL: A Ray Tracing Shading Language. *IEEE Symposium on Interactive Ray Tracing*, 2007.

[15] S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, 1999.

[16] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Computer Graphics and Visualization*, 1999.

[17] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science & Technology Books, 2004.

[18] T. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, 2004. Stanford University.

[19] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[20] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3), 2005.

[21] J. G. Siek and A. Lumsdaine. A Modern Framework for Portable High Performance Numerical Linear Algebra. In *Modern Software Tools for Scientific Computing*. Birkhauser Boston Inc., 1997.

[22] G. Stoll, W. R. Mark, P. Djeu, R. Wang, and I. Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report TR-07-52, The University of Texas at Austin, 2006.

[23] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[24] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 2001.