# RTfact
## Concepts for Generic Ray Tracing

**Iliyan Georgiev**

**Computer Graphics Group**
**Saarland University**
**66123 Saarbrücken, Germany**

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science at the
Faculty of Natural Sciences and Technology I,
Department of Computer Science at Saarland University

**Supervisor:**
Prof. Dr.-Ing. Philipp Slusallek
Saarland University, Saarbrücken, Germany
DFKI Saarbrücken, Germany

**Reviewers:**
Prof. Dr.-Ing. Philipp Slusallek
Saarland University, Saarbrücken, Germany
DFKI Saarbrücken, Germany

Jun.-Prof. Dr. Sebastian Hack
Saarland University, Saarbrücken, Germany

**Thesis submitted on:**
July 31, 2008

Revision 1, July 31st, 2008

# Abstract

For a long time now, interactive 3D graphics has been dominated by rasterization algorithms. However, thanks to more than a decade of research and the fast evolution of computer hardware, ray tracing has recently achieved real-time performance. Thus, it is likely that ray tracing will 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 algorithms and data structures.

In this thesis 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 seamless integration of new algorithms with maximum runtime performance, while leveraging as much of the existing code base as possible.

The SIMD abstraction layer of RTfact enables easy porting to new microprocessor architectures with wider SIMD instruction sets without the need of modifying existing code. The efficiency of C++ 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 existing optimized implementations.

# Acknowledgements

First of all, I would like to thank my thesis supervisor Prof. Dr. Philipp Slusallek for awakening my passion for computer graphics and for giving me the opportunity to work in a demanding and vibrant environment. Special thanks go to Johannes Günther for the many worthwhile discussions and Stefan Popov for motivating my work and giving me directions during the development of the project.

I would like to thank my family for their love and support, and especially my girlfriend for the many weekends we had to stay at home.

# Contents

# Chapter 1

# Introduction

Since the early development of computers there has been growing interest in interactive visualization of three-dimensional environments and simulation of natural lighting phenomena. In the last two decades interactive graphics technology has become a commodity – modern desktop microprocessors have enough computational power to enable real-time visualization of complex models. On the other hand, tremendous advances in photorealistic rendering have made it possible to synthesize images indistinguishable from photographs. As the demands for visualization have been increasing, research has focused in two areas – interactivity and realism.

Interactive visualization has been relying mostly on *rasterization* algorithms, where scenes are composed of triangles, which are rendered and shaded independently from each other. This model is suitable for pipeline-based implementations in hardware, and current dedicated graphics cards can rasterize many millions of triangles per second. Using this model, however, it is difficult to produce photo-realistic images. As each triangle is processed individually, only local information about geometry and illumination is available in one rendering pass. As a result, even the most basic optical effects come at a big price, requiring multiple rendering passes, while still compromising correctness. This is why rasterization has been mostly used in entertainment industry, where interactivity is more important than visual realism.

On the other hand, photorealistic rendering has been relying on algorithms based on *ray tracing*. Ray tracing determines the mutual visibility between two points and is a easy to use tool for physically correct simulations of light propagation in virtual environments. Ray tracing-based algorithms can capture illumination effects like soft shadows, reflection and refraction, as well as full global illumination.

However, ray tracing-based light simulation algorithms have been also famous for their high computational requirements – many ray visibility samples are needed for correct estimation of more complex effects like indirect

illumination, glossy reflections, and caustics. That is why ray tracing has been used mostly for offline rendering, where image quality is crucial and longer rendering times are more tolerable. Still, companies in movie industry have only recently started producing full-length movies and special effects using ray tracing based rendering techniques.

It was not until recently that ray tracing achieved real-time performance. Tremendous advances in hardware and algorithms made it possible to visualize complex scenes with correct shadows and secondary effects on a single commodity PC at real-time frame rates. Coherent ray tracing algorithms, making best use of hardware resources, have shown that interactive and physically correct lighting simulation is achievable on desktop computers. Companies in airplane and automotive industries are already using interactive ray tracing systems for lighting simulation and visualization.

Since the necessary computing power became available, interactive ray tracing research has focused on performance on desktop machines. While offline systems are usually designed to be flexible, some interactive systems had to compromise flexibility [46, 6], 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) [11, 5], but have relied on fixed rendering pipelines and strong assumptions, and additionally suffered from the run-time overhead of dynamic polymorphism.

In this thesis, we describe the architecture of RTfact – a generic, flexible, and high-performance library for interactive ray tracing. RTfact does not aim at delivering a self-contained rendering system, but at creating a flexible and extensible environment for testing and implementing custom ray tracing-based solutions.

Our solution is based on the observation that most of the flexibility required from a ray tracing framework is needed at design time and not necessarily at run-time. Thus, our main goal is to provide an highly flexible infrastructure which allows the user to pay a performance price only when the provided flexibility is really needed at run time. Generally, we try to combine the flexibility of an off-line rendering system with the performance of state-of-the-art ray tracing algorithms and take full advantage of the parallelism supported in modern hardware. We separate algorithms from data structures and ray tracing from rendering, and redefine the design of a ray tracing system in the 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. Certain applications, 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 picking). Thus, a modern ray tracing framework should be flexible enough to allow such freedom and to also deliver the best possible performance for each possible configuration of algorithms and data structures.

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. We do not try to give a one-size-fits-all solution, but take a more general design approach instead. We create 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 that handles ray packets of different size, nature (primary, shadow, secondary, etc.), and common origin 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 or other complex control flow. Thus, no unnecessary run-time overhead is imposed. The library focuses on flexibility and single-thread throughput, and its 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.

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.

## 1.2 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2 we will introduce the problem of light transport and the how two different algorithms try to solve it, in particular rasterization and ray tracing. We will discuss how to speed-up ray tracing with acceleration structures and will see how ray tracing can be used for global illumination

simulations. Finally, we will overview the state of the art algorithms for real-time ray tracing and interactive global illumination.

Chapter 3 introduces the basic infrastructure of a ray tracer. We will give an overview of existing offline and interactive ray tracing systems and discuss the advantages and disadvantages of each.

Based on the observations made in Chapter 3, we set the specific requirements for our real-time ray tracing library in Chapter 4. We then evaluate the possible approaches for designing the library, and finally we introduce the concepts of generic programming.

Chapter 5 presents the main contribution of this thesis – the design of our generic real-time ray tracing library. We will first present the basic software architecture and then we describe the individual components of the library.

In Chapter 6, we demonstrate how the components of RTfact can be composed together to build up custom ray tracing solutions, by giving examples for three different visualization tasks. Finally, we compare the achieved performance to other interactive ray tracing systems.

Chapter 7 summarizes the contributions of the thesis along with a final discussion and directions for future work.

# Chapter 2

# Background

In computer graphics, rendering is the process of producing a two-dimensional image of a virtual thee-dimensional scene from a camera perspective. To be able to do this, one must first understand the nature of light.

Light is measured in *radiance* – the energy per unit time which passes through a unit area orthogonal to the direction of the flow from a unit solid angle, or $\frac{Watt}{m^2 sr}$.

In the real world, light starts its journey from light sources, bounces at objects and continues traveling until absorption. If we can reproduce in a computer simulation the path followed from a light source to our eyes, we would be able to determine what our eye sees. How light scatters in the scene is described by the fundamental *rendering equation* [18]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(\omega_i, x, \omega_o) L_i(x, \omega_i) cos\theta_i \, d\omega_i. \qquad (2.1)$$

This equation models the physics of light and states that the radiance flowing from some surface point $x$ in direction $\omega_o$ equals the radiance emitted by this surface at that point and direction plus the radiance that comes from all other directions and is reflected in the direction of interest. The term $L_i(x, \omega_i)$ yields the incoming radiance from direction $\omega_i$ and is weighted by $f_r(\omega_i, x, \omega_o)$, which specifies the reflectance properties of the surface at point $x$. The term $cos\theta_i$ accounts for the orientation of the surface with respect to the direction of the incoming light.

In Equation 2.1, the incoming radiance $L_i(x, \omega_i)$ is actually equal to the outgoing radiance $L_o(y, -\omega_i)$ at another surface point. It can be obtained using the so-called *ray tracing operator* $h(x, \omega_i)$, which yields the first point visible from $x$ in direction $\omega_i$, i.e. $y = h(x, \omega_i)$. Thus, Equation 2.1 can be reformulated as:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(\omega_i, x, \omega_o) L_o(h(x, \omega_i), -\omega_i) cos\theta_i \, d\omega_i. \qquad (2.2)$$

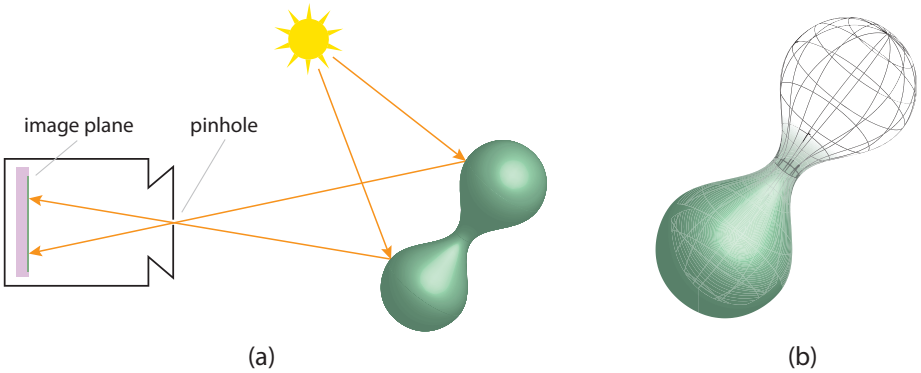


**Figure 2.1:** (a) The pinhole camera lets the light pass only though an infinitely small point. Thus, exactly one ray of light contributes to a single point on the image plane. (b) A three-dimensional solid object is usually represented by a mesh of geometric primitives or patches, which can be triangles, quads, or higher-order surfaces.

## 2.1 Basic Models

Simulating the light propagation in a synthetic scene and capturing an image requires modeling the basic components participating in the real-world process. Three basic models can be distinguished in a digital renderer – camera, geometry, and shading.

### 2.1.1 Camera

Recording an image in the real world requires a device that consists of tiny sensors which detect incoming radiance, or *irradiance*, such as cameras or the retina of the human eye. In a virtual environment, a virtual camera model is used whose sensors are the pixels of the image.

The most widely used camera model in computer graphics is the simple *pinhole camera* (see Figure 2.1.a)). The pinhole camera consists of box with an image plane on the back and a hole in the front, which lets light enter the box and fall onto the image plane. If the hole is infinitely small, a single light ray falls onto a single point on the image plane. In practice, an equivalent and more convenient camera model is used, where the image plane is in the front, and all incoming light concentrates in a single point on the back of the camera.

### 2.1.2 Geometry

Most of the real world scenes can be represented by the surfaces of their objects. A single object it usually given as a mesh of flat geometric primitives or higher-order surfaces (see Figure 2.1.b). Usually, surfaces are represented as triangle meshes, since the triangle is the most basic space filling primitive.

Volumetric data acquired by computer tomography scanners or natural phenomena simulations are usually given in the form of three-dimensional grids.

### 2.1.3 Shading

A very important aspect of a digital image is the appearance of the objects, which is given by the term $f_r(\omega_i, x, \omega_o)$ in Equation 2.2. Depending on the formulation of the equation, this term is called *bidirectional reflectance distribution function* (BRDF), or *bidirectional scattering distribution function* (BSDF), which accounts for the reflection and transmission of light that happens on the surfaces of objects. The BRDF and BSDF are probability distribution functions giving the probability that a light particle coming from direction $\omega_i$ to a surface point $x$ will be reflected or refracted in direction $\omega_o$. In general, light scattering is wavelength dependent, but a common approach in computer graphics is to distinguish between three main channels – red, green, and blue (RGB). The way of specifying the appearance of objects is called *reflectance model*, or *shading model*.

Many analytical shading models exist in computer graphics, most of which provide a number of parameters to set the desired properties of a surface. The most widely used is the Phong illumination model [29]. It encodes reflection properties as a combination of ambient light, diffuse reflection of rough materials, and specular reflection of shiny surfaces.

## 2.2 Rendering Algorithms

In order to generate an image with a computer, we place our camera somewhere in the scene and then we have to compute the radiance falling onto its sensors, or pixels. Discarding any participating media, this is usually done by finding the closest visible surface through each pixel and estimating its appearance as seen from the camera.

There are two general approaches of solving the rendering equation and performing visibility computations: *rasterization algorithms* and *ray tracing algorithms*. In order to find the closest visible surface from each pixel, rasterization algorithms perform *forward projection* of geometry onto the image plane, while ray tracing algorithms perform *backward projection* of rays onto the scene from the camera center through the pixels. The two approaches are somewhat dual to each other, however they have different properties. Rasterization is best known for its low-cost and efficient hardware implementations, but has difficulties in simulating most optical effects accurately. On the other hand, ray tracing is known for its ability to produce high quality images, but also for its poor performance.

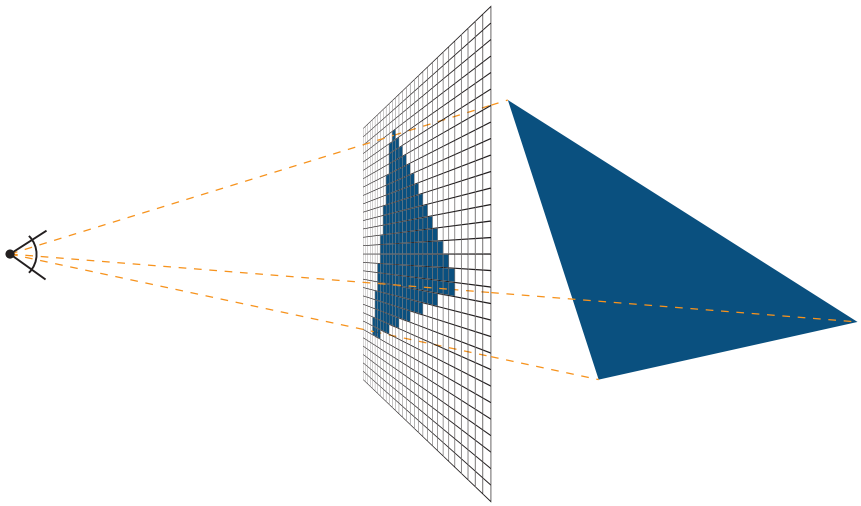

**Figure 2.2:** Triangle rasterization algorithms compute visibility by performing forward projections – each triangle is projected onto the image plane with respect to the view point. The covered pixels are shaded and their colors and corresponding depth values are stored in buffers. The color buffer of a pixel can be overwritten by a pixel with a smaller depth value.

### 2.2.1 Rasterization

The idea behind rasterization is to project geometric primitives independently onto the image plane and to shade the covered pixels (see Figure 2.2). For correct visibility determination the distance from the camera to the closest projected point for each pixel is kept in a depth buffer. A *depth test* ensures that after all triangles have been rasterized, each pixel will contain the color of the closest primitive. This scheme can be very fast but also inefficient, because newly rasterized primitives can overwrite previously computed pixels.

A single-pass rasterization algorithm can only solve a simplified version of the rendering equation:

$$L_o(x, \omega_o) = \sum_{l \in \mathcal{C}} f_r(\omega_i, x, \omega_o) L(x, l) cos\theta_l,$$

where in $l$ is a *point light source* in the set of all point light sources $\mathcal{C}$ in the scene, and the term $L(x, l)$ is the contribution of light $l$ to point $x$ discarding occlusion. This model works for simulating direct illumination from point light sources but because no global knowledge about the scene exists, soft shadows from real area light sources cannot be captured correctly, and even simple hard shadows from point light sources cannot be computed in a single pass. Programmable shaders have enabled multi-pass methods to be developed, which solve the hard shadows problem and try to simulate

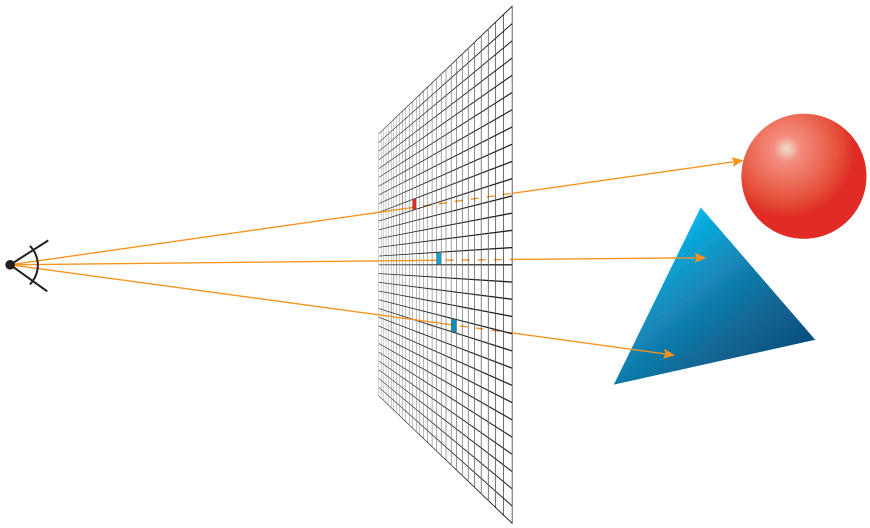

**Figure 2.3:** Ray tracing performs forward projections – rays are traced through each pixel on the image plane to find the closest visible object, and only the visible geometry is shaded.

secondary effects like reflection and refraction, but still compromise their correctness. Such methods also do not scale well, because by design in one rasterization pass the whole geometry is processed.

Nevertheless, triangle rasterization is the most widely used method for real-time rendering, because it can robustly handle dynamic geometry and can be efficiently implemented in highly parallel hardware pipelines on dedicated *graphics processing units* (GPUs). During the past two decades GPUs have been developing at high rates, with constantly increasing performance and programmability.

### 2.2.2 Ray Tracing

Ray tracing [2, 14], determines the closest visible surface through each pixel by shooting rays from the camera (see Figure 2.3). In contrast to rasterization, visibility computations are not performed with respect to a plane, thus ray tracing can be used to sample the whole light path space. This way, light transport can be *simulated* directly based on the rendering equation.

When talking about ray tracing, one usually refers to the classic recursive ray tracing algorithm, called *Whitted-style ray tracing* [52]. In Whitted ray tracing, a *primary ray* is first traced from the camera through each pixel. If a non-perfect specular object is hit, a *shadow ray* is shot toward each light source in the scene, in order to estimate its visibility and contribution to the hit point. If the BSDF of the material has perfect specular parts, *secondary rays* are recursively shot to account for light that is reflected or refracted

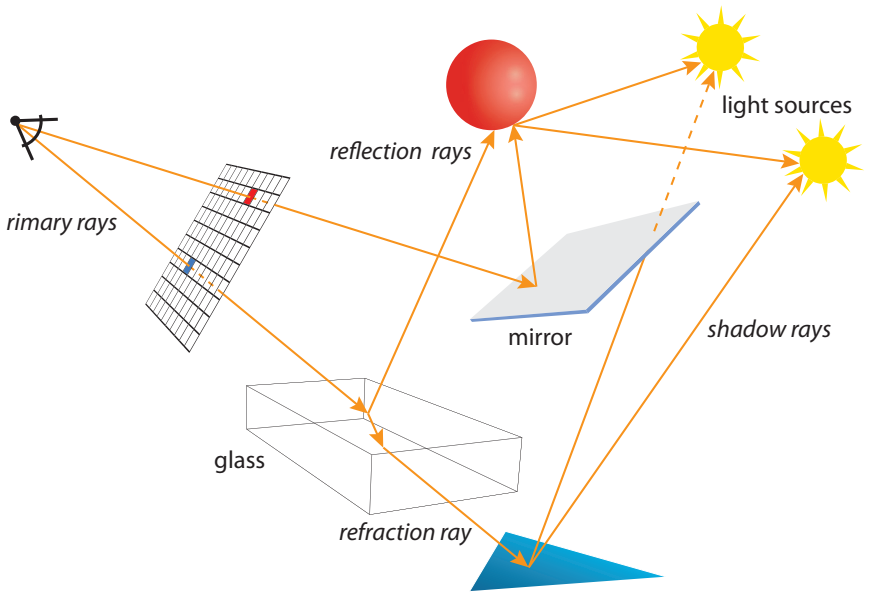


**Figure 2.4:** Whitted-style ray tracing. Reflection and refraction rays are recursively shot from specular surfaces, while direct illumination is estimated by tracing shadow rays toward light sources.

from the surface (see Figure 2.4). To correctly estimate light reflected back along the primary ray, the BSDF is evaluated for all shadow and secondary rays.

Ray tracing can handle different kinds of geometry, reflection, and camera models, whereas rasterization is mainly limited to triangles, local illumination and the pinhole camera model. Furthermore, in Whitted ray tracing invisible objects are never touched and pixel-correct shadows, reflection, and refraction are naturally captures. More complex effects like indirect illumination and caustics can be also captured in a similar way. In general, ray tracing can be used for stochastically solving the rendering equation, thereby computing the *full global illumination* in a scene.

### 2.2.3 Acceleration Structures for Ray Tracing

When recursively solving the rendering equation, the number of rays need to be cast grows exponentially with the number of reflective surfaces. A naïve ray casting algorithm would determine the closest intersection along a ray by simply testing it with all objects in the scene. This would result in a linear complexity with the number of primitives, which would be quite inefficient, except for very simple scenes.

The speed of ray casting is crucial for the performance of every physically based rendering algorithm. In order to reduce the complexity of ray

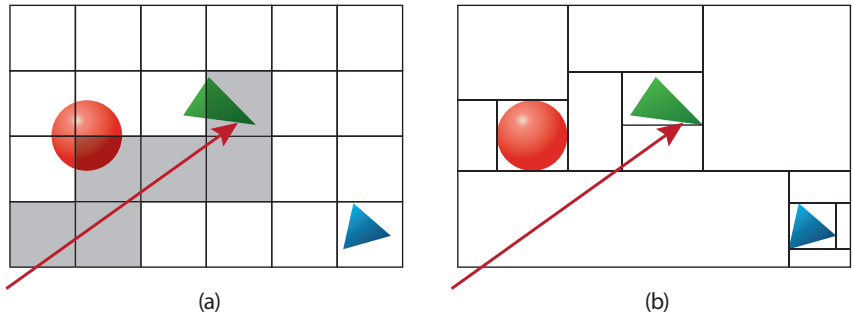


**Figure 2.5:** Acceleration structures for ray tracing. (a) Uniform grids divide space equally in each dimension. Rays traverse the cells sequentially and intersections are searched only in the touched cells. (b) Kd-trees can divide space according to the geometry distribution. Large empty spaces are quickly skipped and intersections are searched for only in the leaf nodes.

casting, fine-grained *spatial index structures* are usually used, which sort the space or the scene geometry into cells, which are then used to prevent intersecting geometry far away from the ray. These structures replace the costly primitive intersection with cheaper *traversal*, which usually has logarithmic asymptotic cost in the number of geometric primitives and can substantially increase performance. The cells in such structures contain references to the encompassed geometry and are traversed in front to back order with respect to the ray. Traversing essentially enumerates geometry along and close to the ray, which makes ray tracing *output sensitive*.

There are two major classes of acceleration structures for ray tracing – *space partitioning structures* and *bounding volume hierarchies*.

#### 2.2.3.1 Space Partitioning Structures

Space partitioning structures *divide space* into disjunctive subspaces. Since the cells of such structures do not overlap, the cell traversal order can be uniquely determined for each ray and thus as soon as an intersection is found, further traversal can be pruned, i.e. *early ray termination* can be performed. However, a side effect of dividing space is that geometry may overlap more than one cell and multiple references to the same primitives have to be kept. Furthermore, there may be cells containing no geometry, which can introduce redundant ray traversal steps.

##### Regular Grids

Regular grids [1] are the most basic form of space partitioning, as they split space uniformly in each dimension (see Figure 2.5.a). Such grids can be built in linear time with respect to the number of primitives, but do not

adapt well to non-uniformly distributed geometry. Grids are mainly used in volume ray tracing, where data samples are usually uniformly distributed, but have been also showed to perform well for surface ray tracing of dynamic scenes [49].

#### Octrees

Octrees [14] are hierarchical space partitioning structures, where each axis-aligned voxel has either 8 children (i.e. it is split in two along each axis), or no children if it is empty. Octrees can adapt to the distribution of geometry and thus can be more efficient for non-uniform distributions than regular grids. The most widely used are regular octrees, where cells are uniformly split, and can be traversed without a stack. However, octrees are more expensive to build than regular grids and cannot adapt to geometry as well as kd-trees or bounding volume hierarchies.

#### Kd-trees

Kd-trees are binary space partitioning (BSP) trees [17, 42], which recursively divide space in two subspaces using axis-aligned planes (see Figure 2.5.b). Internal tree nodes store information about the split axis, while leaves reference the encompassed geometry. Kd-trees can adapt well to the geometry distribution and, as being binary search trees, they have logarithmic traversal complexity. Kd-trees are the most widely used acceleration structures for ray tracing, because they provide a good trade-off between build time and traversal cost. A drawback is that an exact upper bound of the size of the structure cannot be predicted before building, because leaves can contain empty space.

### 2.2.3.2 Bounding Volume Hierarchies

The bounding volume hierarchies (BVHs) [35], in contrast to space partitioning structures, *partition geometry* into disjunctive subsets. These subsets are structured in a tree, whose internal nodes store the bounds of their two children, and whose leaves contain geometry. The bounds can be arbitrary, but usually axis-aligned boxes are used. Since geometry is bounded, and not space, BVHs are usually smaller and faster to build than kd-trees. Also, each primitive in the scene is contained in exactly one node. However, traversing a BVH with a single ray can be more costly than traversing a kd-tree, because cells can overlap and early ray termination is not always possible. However, a key advantage of BVH's is that the topology of the structure does not need to change when geometry changes, which makes them more flexible in handling dynamic environments.

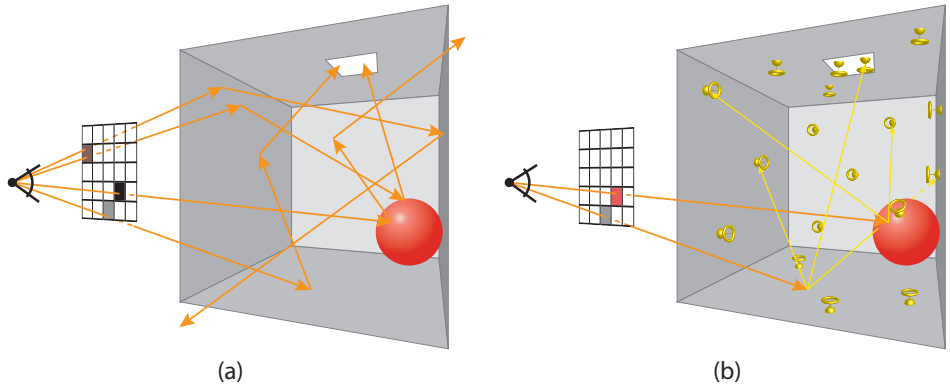


**Figure 2.6:** (a) The idea of Monte Carlo path tracing is to stochastically find paths which bring light to the camera. Images can be quite noisy unless many paths are traced per pixel. (b) In instant radiosity, as paths from light sources are traced and stored as virtual point light sources (VPLs). During rendering, these light sources are used to illuminate the surfaces visible from the camera. Images do not suffer from noise, but illumination artifacts may appear if a small numbers of VPLs is used to approximate indirect illumination.

The quality of the acceleration structure is crucial to the performance of ray tracing. Structures which tightly bound geometry allow efficient skipping of empty space and can drastically reduce the number of primitives to be tested for intersection. Algorithms for building hierarchical structures like kd-trees and BVHs use cost models, such as the surface area heuristic (SAH). SAH optimizes the quality of the structure by minimizing the expected cost for traversing a random ray through the structure.

## 2.3 Global Illumination with Ray Tracing

Producing realistic images requires physical simulation of light transport, which means capturing all possible lighting effects in the scene, such as reflections, refractions, indirect illumination, or volumetric scattering. Such algorithms, called *global illumination algorithms* try to solve Equation 2.2 in order simulate the global light transport between all mutually visible surfaces in a virtual environment.

Current state of the art global illumination algorithms rely on *Monte Carlo integration* methods [12], which estimate the values of complex integrals by evaluating the integrand at discrete sample locations and averaging the results. Pure Monte Carlo methods compute unbiased solutions and robustly handle discontinuities in the integrand, which is the case in the rendering equation.

In the context of ray tracing and the rendering equation, Monte Carlo

global illumination algorithms sample light paths and evaluate the radiance they bring to the camera. However, plausible approximations of the light field in the scene may require many paths to be evaluated. The main problem of Monte Carlo integration is the rapid drop of convergence with the increase of dimensionality, and undersampling can result in high variance or systematic error.

Path tracing [18] solves the rendering equation by stochastically tracing light paths from the camera. A path terminates when a light source is reached or when the path contribution reaches zero. A similar approach, called light tracing, estimates light transport starting from light sources. These algorithms produce unbiased solutions and can capture complex illumination effects but can be very noisy even with a large number of path samples. Because the probability of reaching a light source from every point in the scene is very low, only a small number of paths terminate by actually contributing some light to the camera (see Figure 2.6.a).

Bidirectional path tracing [21, 44] is a generalization of light tracing and path tracing, where paths are simultaneously traced from the camera and from the light sources. The vertices of camera and light source paths are connected and estimators for the resulting camera-light paths are built. This way, relevant paths which transport light from a light source to the camera are more easily found, resulting in less noisy solutions. However, variance is still a major problem, as no illumination coherence is exploited.

Modern Monte Carlo global illumination techniques exploit illumination coherence by performing the computations for one frame in two passes. First, particles from the light sources are traced and stored on surfaces, thereby approximating the light distribution in the scene. These particles are then used during rendering for estimating indirect illumination. The resulting images do not exhibit pixel noise, although undersampling results illumination artifacts.

Instant radiosity [20] relies on the assumption that most of the indirect illumination in a scene is caused by diffuse light scattering. Each stored particle represents the end of a path from the light source and is called a virtual point light source (VPL). During rendering, these VPLs are used to illuminate the surfaces visible from the camera, in order to estimate the diffuse indirect illumination (see Figure 2.6.b). This algorithm is a special case of bidirectional path tracing, in light paths are fixed in the pre-process phase and reused for all primary rays during rendering. The accuracy of the solution directly depends on the number of VPLs, although plausible approximations can already be achieved with a small number of VPLs (usually around 300) for moderately complex scenes. Instant radiosity works well with diffuse and not very specular objects, but cannot capture specular indirect illumination effects like caustics. Nevertheless, it has cheap preprocessing and is easy to parallelize.

Instant radiosity provides a robust way of handling diffuse indirect

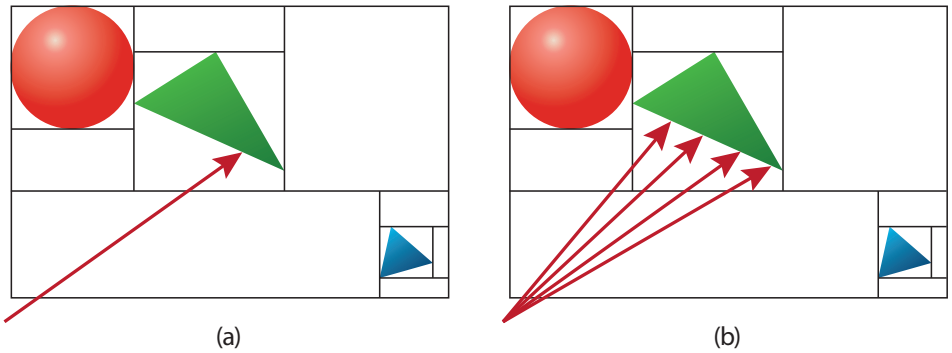


**Figure 2.7:** (a) Classic ray casting – each ray is traversed and intersected independently. (b) Packet ray casting – a bunch of rays are traced simultaneously, thus exploiting spatial coherence resulting in common traversal and intersection decisions.

illumination by converting it to direct illumination. However, high quality solutions need a large number of VPLs to approximate the light distribution without illumination artifacts. Furthermore, realistic direct illumination from area light sources and environment maps also requires many point light samples. Unfortunately, tracing shadow rays toward thousands of point light sources for each pixel can be prohibitively expensive.

Walter et al. proposed Lightcuts [51] as a scalable solution to the many lights problem. Lightcuts can accurately approximate illumination from many point lights by tracing shadow rays toward a small fraction of them. The algorithm relies on the fact that small changes in illumination are not percievable by humans and approximates a group of lights with a single representative light, while bounding the approximation error. Lights are stored in a binary tree, which is traversed during rendering to find an optimal clustering of lights whose error is below a perceptual threshold. Achieving logarithmic complexity in the number of point lights, Lightcuts can accurately approximate hundreds of thousands of light sources by shooting only a few hundred shadow rays. However, traversing the light tree during rendering remains relatively expensive, which drastically reduces the speedup for the small number of point lights that can be afforded in an interactive setup.

## 2.4 Interactive Ray Tracing

As discussed in the previous sections, ray tracing unleashes its full power when used for simulating physical light transport. The performance of this simulation depends heavily on the performance of ray casting itself. Because in the past ray casting has been extremely computationally expensive, it has been used mostly for offline tasks, where longer rendering times are tolerable

in favor of high image quality and correct solutions.

With the rapid the development of microprocessors, modern ray casting algorithms, and highly optimized implementations made it possible to visualize complex scenes with direct and full global illumination at interactive frame rates using ray tracing on multi-processor shared memory machines, small clusters of PCs, and even on a single commodity PC. Interactive visualization of massively complex models consisting of billions of triangles has been demonstrated, which is still impossible to do even with high-end rasterization hardware.

### 2.4.1 Packet Ray Tracing

One of the most prominent algorithmic and implementational improvements to the classic ray casting algorithm has been packet ray casting, proposed by Wald et al. [46] for kd-trees. Relying on the assumption that coherent rays intersect the same geometry, a bunch of rays is simultaneously traversed through the acceleration structure and tested for intersection against primitives (see Figure 2.7). A mask of the currently active rays is tracked and a kd-tree node is visited if any active ray intersects it. The benefit of such traversal is is big as long as the rays do not diverge, as the cost for loading and intersecting nodes and geometry is amortized over all rays in the packet.

Using a number of optimizations, Wald et al. achieved a speed-up of a factor of 10 to 30 compared to other implementations at that time. They traced packets of four rays, exploiting the Single Instruction Multiple Data (SIMD) instruction set implemented in the Streaming SIMD Extensions (SSE) [43] on the x86 microprocessors. The hardware parallel computations allow to amortize the traversal and intersection costs over the four rays in the packet. Ideally, all rays in a packet would make the same traversal and intersection decisions, resulting in four times performance speed-up over single ray casting. Other important optimizations Wald et al. made were optimizing and properly aligning kd-tree storage, which enabled better CPU cache performance, and optimizing ray-triangle intersection.

### 2.4.2 Frustum Culling for Packet Ray Tracing

The packet ray casting algorithm is trivially generalized to an arbitrary number of rays by storing ray data in a structure of SSE arrays layout. Larger packets allow coherence over more rays to be exploited. However, in practice the computational benefits do not increase, because parallel SIMD operations can be performed on chunks of 4 rays only. Excessive number of rays in a packet can also hurt performance, as ray coherence decreases and the working set gets larger.

Reshetov et al. [34] proposed to exploit coherence between rays in large packets on multiple levels. Based on the observation that as a packet

descends rays lose coherence down the hierarchy, their algorithm traverses a bounding frustum of the packet to quickly find a common entry point for all rays deep in the hierarchy by robustly culling nodes which the whole frustum skips. Since the frustum is represented by four corner rays, it is almost as cheap to traverse it as a standard four-wide ray packet. The frustum is eventually split into sub-frusta which are traced further to find deeper entry points for their corresponding sub-packets. Traversal of the actual rays then starts directly from the entry points found, thus avoiding redundant traversal in the higher levels of the tree. If the frustum terminates in an empty leaf, then all bounded rays do not intersect any geometry and can be safely discarded from further processing. If the entry point is a non-empty leaf node, then only geometry intersection needs to be performed with the actual rays. By further modifying the kd-tree data structure, Reshetov et al. achieved a speed-up of a factor of 10 over previous implementations. However, the traversal algorithm works well only for primary rays and cannot be easily applied for secondary rays.

Wald et al. [47] applied frustum culling of large ray packets to bounding volume hierarchies. Additionally, they keep track of the first active ray in the packet, which is used to quickly check whether the whole packet should descend down a BVH node. Their algorithm shows best performance when applied to large packets of size 8x8 or 16x16, which is justified by the constant complexity of the first active ray and frustum culling tests. Wald et al. also employ a conservative ordered (front to back) traversal of the nodes by storing extra information in the nodes during building. Exploiting the property of the BVH that its topology can be independent of geometry deformation, they were able to interactively render dynamic scenes on a single PC. The BVH traversal algorithm has been also successfully applied for shadow and secondary rays.

### 2.4.3 Interactive Global Illumination

As global illumination algorithms spend most of their time in tracing rays, the interactive ray tracing creates the potential for computing physically-correct light simulations in real time, which has been a long-standing goal for computer graphics research. However, computing global illumination is inherently more complex than classic Whitted ray tracing. In order to achieve interactive performance with a real-time ray tracer, a global illumination algorithm should be easily parallelized and should require as few rays as possible for a plausible solution. Furthermore, it should require minimum preprocessing in order to be able to quickly respond to changes in the scene, and should employ cheap surface shading which should not become a performance bottleneck.

Most of the current state of the art interactive global illumination algorithms are based on instant radiosity, which has cheap preprocessing and

only computes direct illumination from light sources during rendering. Wald at al. [50] and Benthin et al. [4] applied packet tracing algorithms for interactive rendering of complex scenes with diffuse global illumination on small clusters commodity of PCs. Relying on the fact that diffuse indirect illumination varies smoothly, they reduce its complexity using interleaved sampling of virtual point light sources and averaging VPL illumination across neighboring pixels by employing discontinuity buffer filtering. Using only a few coherently traced rays per pixel, their solution allows interaction with the scene and scales near linearly with the number of clients.

Recent research in interactive global illumination based on instant radiosity has focused mainly on efficient sampling of VPLs. Segovia et al. [37] sample VPL locations both from the camera and the light sources, which was later generalized to metropolis sampling [38], in order to provide a view-dependent solution in difficult visibility settings.

The recent advances in hardware have enabled interactive simulations of light transport, which have been not possible before. Modern processors provide more and more computational power as their design is shifting toward 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 GPUs utilize single instruction multiple thread (SIMT) data parallelism on hundreds of scalar cores [25]. Recently, interactive ray tracing has been also achieved on specialized hardware, such as the CELL processor [3] and GPUs [32, 8, 31, 15].

Massive parallelism and increased hardware programmability make it very likely that future rendering engines will be almost entirely implemented in software [22]. This implicates that flexible interactive systems which can deliver ray tracing to application developers and thus to end users will become of primary importance.

# Chapter 3

# Ray Tracing Systems

Ray tracing is a powerful and intuitive tool for generating realistic images. Rendering systems based on ray tracing are widely used for offline rendering tasks, but there is also an increasing interest in utilizing ray tracing for interactive rendering.

In this chapter, we will give an overview of some existing offline and interactive ray tracing systems and will evaluate their designs and performance.

## 3.1 Basic Infrastructure

In order to capture an image of a synthetic scene, we have to put a camera somewhere in the scene and compute the radiance falling onto its sensors, i.e. the pixels on the images plane. To do this, a ray tracer will usually shoot a ray through each pixel, traverse it though the acceleration structure, and shade the first hit surface point (assuming there are no participating media in the scene). Depending on the type of surface and the desired illumination effects, one or more rays will be eventually traced from the hit point – a shadow ray toward a light source, another ray in the reflection or refraction directions, or a ray in a random direction in the case of stochastic light integration. After the information about the incoming radiance has been gathered, the radiance reflected back along the camera ray is estimated by evaluating the BSDF of the surface, which is essentially solving Equation 2.2. The total radiance is finally written as a color to the framebuffer.

The stages in the process, illustrated in Figure 3.1, outline the basic functionality a ray tracer should support. In the following two sections we will discuss how existing ray tracing systems build their infrastructure for supporting various implementations and configurations of these stages.

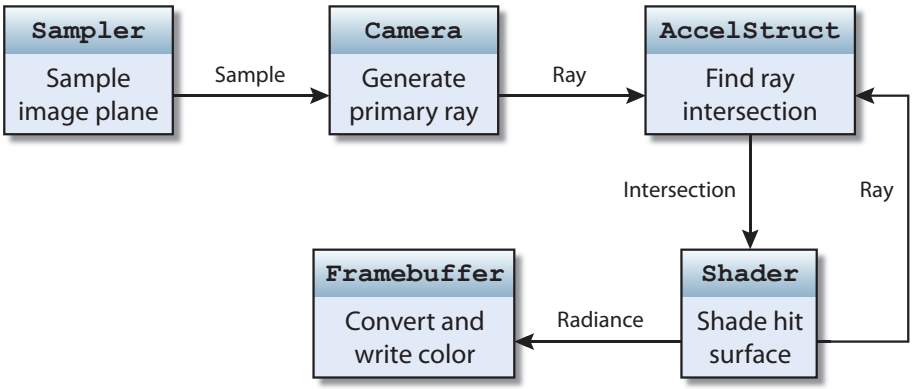


**Figure 3.1:** A ray tracer would usually first pick a sample on the image plane, then generate a primary ray from the camera and trace it. As soon as an intersection has been found, the hit point is shaded after eventually tracing and shading secondary rays recursively. Finally, the computed radiance is written to the framebuffer.

## 3.2 Offline Ray Tracing Systems

When the ultimate goal is photorealism, ray tracing is often the rendering method of choice. Companies in advertising, architecture, movie, and car industries use ray tracing systems to produce high quality images of their products, thus saving time and resources on taking pictures under natural conditions or even manufacturing these products. Such systems render complex geometry with sophisticated materials and illumination effects.

Offline high-quality ray tracing systems are usually required to provide great flexibility and fine control over the rendering process. This is achieved by carefully abstracting rendering stages and components in interfaces, which allow specific implementations with polymorphic behavior to be plugged in, depending on the specific application requirements. Such infrastructures are often implemented with C++ class hierarchies with fine granularity, which allow connecting loosely coupled components that communicate through predefined interfaces. This way the renderer, for example, can trace rays through an acceleration structure, invoke a shader at the hit point, which will sample a light source, all of which are unknown at design time and communicated through common abstract interfaces. Such infrastructure allows different implementations to coexist in a single environment and to be manages independently from each other.

In order to provide maximum extensibility and interoperability, some rendering systems expose binary APIs for specifying geometry and for implementing custom shaders [30, 28]. Some systems also provide specific shading languages [30, 26], which provide simple syntax for writing custom components, such as materials, light sources, and cameras.

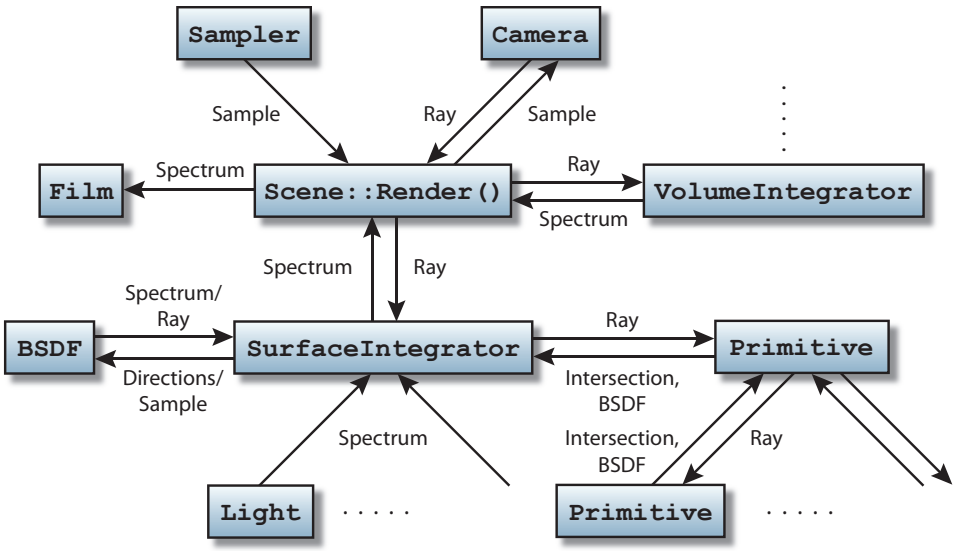


**Figure 3.2:** The main rendering loop and class relations of PBRT. The `Sampler` provides a sample which the `Camera` uses to generate a primary ray. This ray is passed to the integrators, which return the radiance along the ray, which is in turn given to the film to store it in image. The `SurfaceIntegrator` uses `Primitives` to find the first ray-surface intersection. The intersection structure extracts a `BSDF` for the surface, which is then used to determine the illumination from the `Lights` in the `Scene`.

### 3.2.1 PBRT

PBRT is a free and extensible physically-based ray tracing framework, released by the authors of the book "Physically Based Rendering" [28]. It is widely used in the academic field and in computer graphics courses. The renderer is written using a plug-in architecture – the core system only defines the main rendering loop with minimal control flow. The components in the pipeline are written in terms of fine-grained abstract base classes which define the interfaces to the plug-ins. Components are loaded at run time to provide scene-specific functionality.

#### System Overview

The rendering pipeline of PBRT, illustrated in Figure 3.2, starts by sampling the image plane and generating a camera ray using the sample position. The ray is given to volume and surface integrators, which compute the radiance flowing back along the ray toward the camera. Finally, the radiance is passed to the film, which stores it in the framebuffer.

All geometric primitives and acceleration structures implement the `Primitive` interface, which provides functions for ray intersection and querying surface reflection and emission properties. This makes the structure of

the scene geometry completely transparent to other components, as it can consist of a single primitive or a complex hierarchy of acceleration structures, each containing different kinds of primitives. Geometric primitives themselves are instantiated with transformations by their containers and perform intersection computations in object space.

The `Integrator` is a central component in PBRT. It is responsible for computing the radiance reaching the camera sensors and coordinates most of the rendering process. The two types of `Integrator`s – the `SurfaceIntegrator` and the `VolumeIntegrator`, process the camera ray one after the other, and the radiance values are finally blended based on the cumulative opacities along the ray. This allows for seamless integration of volumetric and geometric objects in one scene.

### Shading Infrastructure

In contrast to traditional rasterization-based and ray tracing renderers, PBRT employs a physically-based shading model. In PBRT, rays are traced by `Integrators`, while sampling and evaluating surface reflection is performed by `BSDF`s, which are aggregated by `Material`s. This fine-grained abstraction model allows a single integration algorithm to work with all kinds of materials and vice versa. `Material`s can in turn share common BSDF parts and are completely independent from the ray tracing functionality. Given a surface intersection structure, `Material`s query `Texture`s to get the color at the particular intersection point and return a `BSDF` object, which is handled by the integrator. Thus, all materials can be combined with any texture, which can return a constant color, a texel from an image, or generate the color procedurally.

Light sources work similarly to materials. The `Light` interface defines functions for sampling a position at the light source, computing the light contribution to a surface point, and for initializing a light ray for global illumination computations.

PBRT is a highly flexible system and provides a plug-in infrastructure, which facilitates extensibility. Emphasising on code reuse and well designed interfaces, the system is useful for a great variety of physically-based rendering tasks and is popular among researchers.

Unfortunately, as highly modular as the system is, its performance remains far from interactive. Even with optimized data structures and efficient light integration algorithms rendering a single frame within PBRT can take hours. Excessive run-time memory allocation and virtual function calls at fine granularity inhibits compiler optimizations and imposes significant run-time overhead. The single ray pipeline simplifies the overall design of the system, but also disables packet tracing algorithms which exploit ray coherence and improve cache behavior. As a batch ray tracer, PBRT was

also not designed to take advantage of the parallelism supported by modern CPU architectures.

PBRT has been designed as a self-contained application, which is only controllable via plug-in modules. While such architecture is convenient for educational purposes, it makes integration of the renderer with other applications much harder or even impossible.

## 3.3 Interactive Ray Tracing Systems

Ten years after first demonstrated on large supercomputers, interactive ray tracing has become available to commodity computers. With the rapid development of microprocessors and ray tracing algorithms, interactive ray tracing has become an interesting alternative to traditional rasterization-based rendering. Companies in airplane and automotive industries already use interactive ray tracing systems for physical light simulation and realistic rendering of complex models.

Current interactive ray tracing systems rely on software implementations and highly optimized algorithms and data structures. In order to enable compiler optimizations and achieve best performance, some interactive systems fix some functionality, such as the acceleration structures, the intersection algorithms, and the types of supported geometry, and expose only functionality for specifying geometry and writing shaders. Others provide coarse-grained abstractions for the components in their pipeline, balancing between flexibility and performance.

True interactivity means ability to perform modifications to the scene during rendering. Dynamic geometry has historically been a problem for ray tracing, as it relies on optimized pre-built spatial index structures to achieve interactive performance. While this is not a big issue in an offline setup, a real-time ray tracer cannot always afford to rebuild its data structures from scratch for each frame. Therefore, some interactive systems provide basic support for scene graphs, which track and localize scene changes, and build multiple levels of acceleration structures.

### 3.3.1 OpenRT

OpenRT was the first attempt to standardize an API for interactive ray tracing [10, 11]. As a developer-oriented framework, OpenRT provided an API close to OpenGL, in order to ease adoption and porting of existing applications. At the time OpenRT was developed, ray tracing was still not fast enough to deliver the needed interactive performance on a single machine, therefore the backend supported distributed rendering, which remained mostly transparent to the user. OpenRT actually specifies two different APIs – one for scene description and one for shading and extensibility.

### Core API

The core OpenRT API provides a set of C functions for specifying geometry, lights, textures, transformations and binding shaders to geometry. The API resembles the state stack and provides a sub-set of OpenGL's functionality. OpenRT supports only retained mode rendering, since immediate mode rendering does not make sense in a ray tracing environment, where visibility calculations only take place once the full scene geometry has been specified.

Geometric primitives in OpenRT are contained in objects which can be instantiated with transformation matrices. This allows geometry to be specified once and reused multiple times, thereby avoiding data replication and enabling visualization of massive models with low memory footprint. An acceleration structure is build for each object immediately after it has been specified and is kept unchanged until the destruction of the object. Individual instances can be modified by simply changing their transformation matrix. Each time an instance has been changed, a top-level acceleration structure is rebuilt over all instances, which is relatively small and thus cheap. This two-level scene graph allows for robust handling of dynamic scenes with rigid object animations.

### Shading API

The OpenRT shading API, called OpenRTS, provides a set of interfaces for extending the base system with new functionality and controlling the main rendering loop. It provides shader interfaces not only for surfaces, but also for the environment texturing, cameras, lights, and the main rendering loop.

The fixed function part of OpenRTS consists of C functions for managing and tracing rays. All ray and intersection data is stored in the `RTState` structure, and most of the shading infrastructure is organized around it.

The major part of the OpenRT shading API is formed by a C++ shader class framework. It defines a set of abstract interfaces for writing custom shaders, which can be loaded dynamically at run time. The class relations are illustrated in Figure 3.3.

The central component driving the rendering process in OpenRT is the `RTRenderingObject`. In a single threaded environment, it is responsible for sampling the image plane, generating primary rays using `RTCamera`, and writing the final color to the framebuffer. The rendering object invokes pixel (or sub-pixel) computations for each primary ray by calling the `rtsTrace()` function. In the case of client/server distributed environment, the main rendering object runs on the server, which subdivides the image plane into tiles, and delegates rendering to clients. After all tiles have been rendered, it combines the tiles into an image and writes it to the framebuffer.

The `rtsTrace()` function traces a given ray and invokes the shader of the first hit surface. In case of no intersection, an environment color is

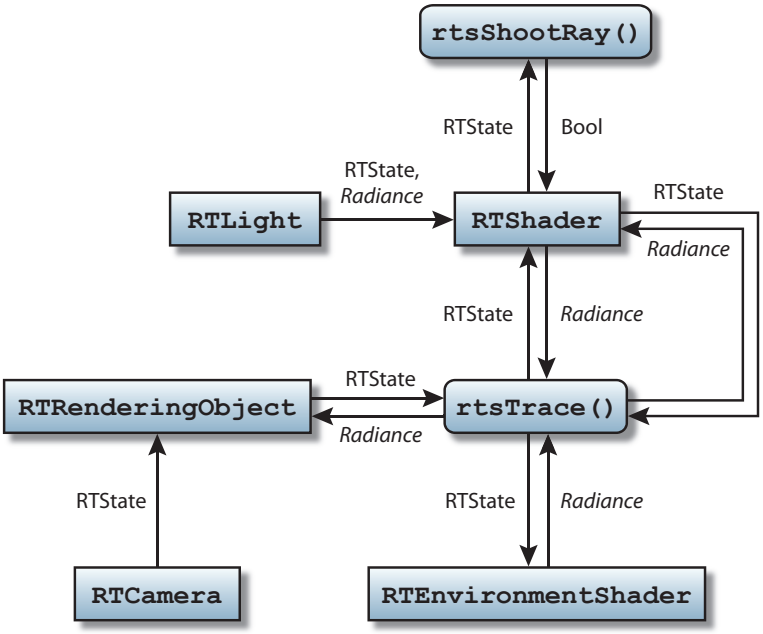


**Figure 3.3:** Class relations of the OpenRT shading API. The `RTRenderingObject` implements the main rendering loop and is responsible for generating camera rays and writing the final color values to the framebuffer. The `rtsTrace()` function traces a ray and invokes the shader of the first hit surface or `RTEnvironmentShader` in case of no intersection. The surface shader (`RTShader`) is responsible for computing the illumination at a surface point and can sample light sources and shoot occlusion rays through `rtsShootRay()` or secondary rays by calling `rtsTrace()` recursively.

computed by the global `RTEnvironmentShader`. The `RTShader` is responsible for computing the color at a given point and can sample lights using the `RTLight` interface and shoot occlusion rays by calling the `rtsShootRay()` function. The `rtsTrace()` function can be called recursively in `RTShader` in order to simulate secondary lighting effects, such as reflection or indirect illumination.

OpenRT has been successfully applied in industrial applications where interactive physical lighting simulations are needed [48]. Airplane companies use it for simulating lighting in passenger cabins, while automotive manufacturers use it for realistic visualization of car interior and exterior. Such companies install OpenRT in their visualization centers, where clusters of PCs and shared memory machines provide the sufficient horsepower for interactive visualization of their complex models.

Unfortunately, OpenRT has failed to adopt modern interactive ray tracing technology. The OpenRT shading API was designed to operate on single rays only, because it was too complicated for the end user to

write SIMD shaders. Thus, commercial implementations of OpenRT were restricted to tracing and shading single secondary rays, while only primary ray casting was hand-coded in SSE, which reduced the overall rendering performance. While at the time of designing the OpenRT infrastructure packet tracing was only about two times faster than single ray tracing, modern state of the art packet tracing algorithms can perform much faster, often being able to deliver on a single PC the same performance that OpenRT would provide on a small cluster of PCs.

OpenRT uses fixed internal acceleration structures and ray casting algorithms, which are entirely hidden behind the API. The API itself does not provide any control over how and when these structures are built, while rendering can be controlled only through plug-ins. As a result, extending the core ray tracing functionality of OpenRT is impossible, and tight integration into user applications is limited.

Another disadvantage of OpenRT is the surface shading model, which incorporates light integration, texturing, and BSDF evaluation in a single component – the `RTShader`. As discussed in the previous section, such shading model requires a particular reflectance model to be duplicated in multiple shaders for each light integration algorithm and vice versa. For example, one might end up having a simple `FlatPhongShader`, a more sophisticated `PathTracingPhongShader`, or a `TexturedPhongShader`, all having the same replicated piece of code.

### 3.3.2 Manta

The Manta interactive ray tracer [5] proposed a software model to leverage modern architectures, while delivering both interactivity and flexibility. The system is designed to take advantage of instruction- and thread-level parallelism that are available in commodity CPUs. Manta provides a set of virtual interfaces for plugging user extensions. These interfaces utilize wide ray packets in order to amortize the run-time overhead imposed by the interfaces. The system has a two piece model, focusing on scalability and maximum hardware utilization – a multi-threaded parallel pipeline and a collection of ray tracing components organized in a rendering stack.

#### Parallel Pipeline

One of the objectives of Manta is to achieve maximum scalability on parallel shared memory architectures. In order to fully leverage multi-core systems, different tasks should be well load balanced and synchronized as few times as possible. To address this, Manta employs a parallel rendering pipeline, which manages the execution of tasks and controls thread activity (see Figure 3.4).

The pipeline consists of several stages, in which tasks with similar load balancing characteristics are executed by each thread. Synchronization is

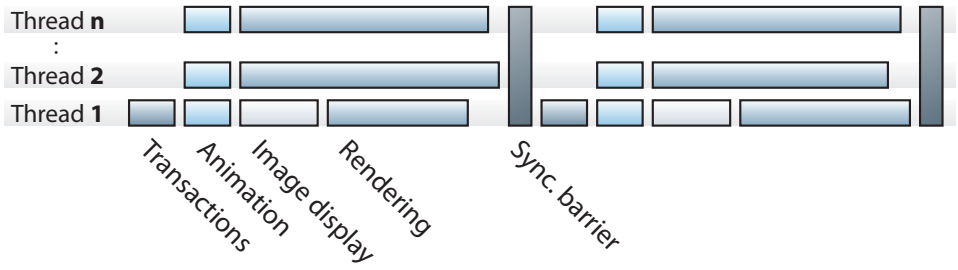


**Figure 3.4:** The Manta parallel rendering pipeline. Animation callbacks are first executed by all threads. After that thread 1 begins image display, while the rest start executing dynamically load balanced rendering jobs. After finishing, thread 1 joins rendering. In the mean time, external events are recorded as callbacks and added to the queue, which is safely is flushed when the next synchronization barrier is reached.

constrained to happen only at certain points, in order to efficiently scale with a large number of rendering threads. Inherently balanced tasks, like parallel animation callbacks, are executed first. Imbalanced tasks, like image display, are scheduled next, and finally dynamically balanced tasks are, like rendering, are executed. Image display is performed asynchronously with rendering to reduce the overhead of such batch tasks. This means that displaying one frame is delayed until the rendering of the next one starts. To achieve this, two framebuffers are used – one for displaying and one for rendering, which are swapped at the end of each frame.

A synchronization barrier at the end of the pipeline waits for rendering to finish. Only after the barrier has been reached, modifications to the scene can be safely performed. During rendering Manta receives events and pushes callbacks into a queue, which is flushed when the barrier is reached. This removes the need of double buffering the whole rendering state and reduces the amount of time threads spend on waiting for mutexes.

### Rendering Stack

The rendering pipeline of Manta is organized in a modular stack which is traversed asynchronously by each thread (see Figure 3.5). First, the image traverser divides the frame into regions and assigns these regions to threads. It tries to balance the workload so that all threads are kept as busy as possible and finish rendering at approximately the same time.

The image traverser component passes image fragments to the pixel sampler, which picks pixel sample locations and maps a ray packet to each tile. Ray packets are then passed to the renderer. The renderer is responsible for intersecting the rays with the scene and invoking surface shaders. After intersection, ray packets are split into sub-packets that hit the same shader. These sub-packets are not copies of the original data, but only store

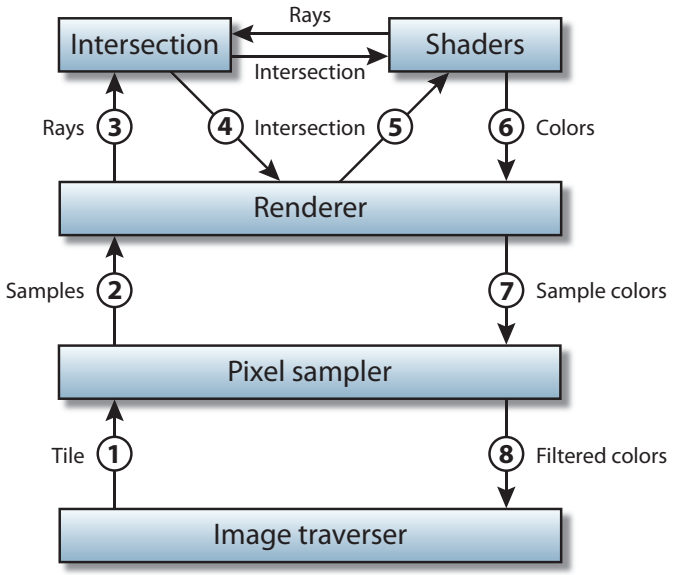


**Figure 3.5:** The Manta rendering stack is asynchronously traversed by the threads executing rendering tasks.

sub-ranges of active rays. After the rays have been shaded, they are passed back to the pixel sampler. The pixel sampler eventually performs filtering on the samples and gives them to the image traverser, which writes the final colors to the framebuffer.

Manta aims at delivering both flexibility and performance, while scaling to large shared memory machines. It achieves greater flexibility than OpenRT by providing interfaces for different components in the rendering pipeline and the rendering stack. The system has been successfully applied to a variety of rendering tasks, including massive model visualization, volume isosurface rendering, and has achieved near linear scalability to a large number of processors.

The system focuses on massive parallelism and its single thread performance is considerably lower than other optimized implementations. The interfaces in Manta have been designed to work on wide packets of rays, in order to amortize the cost of virtual functions over many rays. However, wide ray packets assume high coherence during all stages in the rendering stack, which is often true only for primary rays. As a result, packets are split at an early stage into incoherent sub-ranges, which are intersected and shaded independently, thus breaking the assumption. Furthermore, the size of the data allocated for a ray packet is globally fixed for the whole system at compile time, and sub-packets are represented either as sub-ranges or masks

on this data. This adds processing overhead and prohibits the co-existance of "real" packets of different sizes.

Ray packets in Manta are usually shaded and intersected in SIMD. However, separate scalar and SIMD code paths exist in the whole system, and SIMD code paths are written entirely using low-level intrinsics. This greatly reduces code maintenance and portability. Similarly to PBRT, Manta provides an application-centric rendering solution that can be controlled only through plug-ins, which makes it very hard to be integrated with other applications.

### 3.3.3 RTSL

Parker et al. [26] proposed RTSL as a domain-specific language (DSL) for extending ray tracing systems. It provides a simple and intuitive syntax for implementing custom camera, primitive, light, material, and texture shaders, which can be used in multiple rendering systems. The language includes some features found in general purpose languages, such as functions, classes, interfaces, and inheritance. Custom RTSL components should implement specific predefined interfaces. For example, the primitive interface defines functionality for intersection, and normal, bounds, texture, and derivative computation. Materials support both a physically-based BSDF evaluation and sampling interface, as well as an imperative shading function which computes the final color at a hit point.

A specialized compiler is used to produce scalar and SIMD code C++ from scalar RTSL code. This allows non-experienced programmers to write custom rendering components without having to deal with optimizations and parallelization on any level.

In order to retain a simple syntax, RTSL does not allow for controlling the rendering loop or writing acceleration structures, for which support from the underlying system is required. Also, in the case of physically-based rendering, the light integration algorithm has to be implemented outside RTSL, as RTSL materials only provide a BSDF interface. The strong barrier between the language and the underlying renderer requires all available functionality to be explicitly exported to the language, which is strongly renderer-dependent. RTSL balances between a least-common-denominator and a renderer-specific solution, which is a questionable trade-off. Furthermore, it is unclear how, for example, Quasi-Monte Carlo integration and interleaved sampling should be supported in the language, or how to take advantage of ray coherence in primitive intersection RTSL functions.

RTSL is a convenient and developer-friendly ray tracing shading language. However, it can only be added on top of an already working system, as it does not provide a full solution that is both flexible and efficient.

# Chapter 4

# Design Considerations

Software ray tracing has been historically slower than hardware rasterization, which dominates most interactive applications. The recent advances in hardware and research have made ray tracing competitive to rasterization for some applications, but complex indirect illumination effects still cannot be simulated in real time and bring complications to the architecture of a ray tracer. Therefore, ray tracing renderers have had to make a common design decision – the trade-off between flexibility and performance.

In this chapter we set the requirements for our real-time ray tracing framework, based on the observations we made in Chapter 3. We then evaluate some possible design approaches and choose the one that best meets our requirements.

## 4.1 Flexibility Requirements

A general problem of high performance ray tracing implementations is the tight integration of algorithms and data structures in the code. Such renderers also provide a fixed shading pipeline with a limited set of supported materials and geometry [6]. However, versatility is the most prominent advantage of ray tracing over rasterization-based visibility algorithms. For example, architectural objects are usually represented by planar triangles and quadrilaterals, while characters in movies are modeled as subdivision or higher order surfaces, all of which can be elegantly handled with ray tracing. Furthermore, there are many illumination models that can be handled accurately only with ray tracing. Thus, an absolute requirement for our framework is to support various ray casting and intersection algorithms, geometry, and illumination models, and to provide a flexible infrastructure for combining them.

Another requirement for our system is seamless support for ray packets. Modern high performance ray tracing algorithms operate on packets of ray data simultaneously. In addition, single ray SIMD tracing is also an

active area of research [9, 45]. Thus, it would be desirable for our framework to handle single rays and packets of rays in a unified way, i.e. without the need of two redundant code paths for the entire rendering pipeline.

All ray tracing systems, discussed in the previous section, have a common problem – integration with other applications. PBRT and Manta can be only extended and controlled to some degree using plugins, while OpenRT provides only a geometry specification API that can be integrated into user applications. To overcome this problem, we set the requirement for our framework to be a library which exposes all of its functionality via a set of unified interfaces. It would be also desirable to allow full control over the rendering process, i.e. our library should not rely on a fixed rendering pipeline.

## 4.2 Performance Requirements

The most pragmatic performance requirement for our ray tracing library is pretty natural – it should be able to deliver the same performance as single-purpose hand-optimized implementations. If it does not, people might still prefer to use other implementations.

The most important aspects of high performance computing on modern microprocessors are parallelism and memory utilization.

### 4.2.1 Parallelism

Ray tracing is often qualified as embarrassingly parallel, as computations for different pixels can be performed independently. The first interactive ray tracers ran on large supercomputers [24, 27]. They traced a single ray at a time and used screen-space parallelisation by assigning pixel tiles to different processors. Modern microprocessors accommodate multiple processing cores whose number is doubling almost every year, and systems like Manta and OpenRT target such machines to achieve high performance. Thus, our real-time ray tracer has to be thread-aware, in order to scale well to a large number of cores.

As discussed before, modern ray tracing algorithms operate on packets of data simultaneously, thereby employing hardware SIMD instruction sets. Follow the trend of hardware SIMD units getting wider in future, our library should take advantage of data parallel processing as much as possible and should provide an infrastructure for tracing and shading packets of rays. Wide SIMD units will also increase coherence requirements, thus new algorithms are likely to be developed for such architectures. Our infrastructure should be extensible enough to be able adapt to such future changes.

### 4.2.2 Memory Bandwidth and Cache Utilization

Rendering algorithms are usually easily parallelized, as they write only small amounts of data for each pixel to the framebuffer independently. However, the amounts of data being processed during ray casting and shading are usually large and accessed incoherently. Standard scenes consist of millions of geometric primitives for which acceleration structures are additionally built, and memory bandwidth can easily become a performance bottleneck. Thus, optimizing data storage and layout is essential for reducing bandwidth requirements and achieving maximum utilization of the multi-level caches of current CPUs. State of the art ray tracing implementations take special care of memory alignment and packing data for coherent memory access, e.g. minimizing and arranging kd-tree node data in a way that chunks of nodes fit exactly into a single cache line. They also minimize the working memory set at any point in time in order to avoid excessive spilling of CPU registers into memory. Our library should therefore be carefully designed to employ such optimizations.

## 4.3 Flexibility vs. Performance

There has always been a wide performance gap between flexible ray tracing systems and single-purpose hand-optimized implementations. This gap is caused to a large degree by the infrastructural overhead imposed by the support of various configurations of algorithms and data structures. However, our observation is that most of the flexibility provided by general purpose renderers is actually needed at compile time and not necessarily at run time. For example, more often than not, the choice of acceleration structure and intersection algorithm can be made before compiling the application, while for most cases the types of materials can be only determined at run time. In such cases, it seems reasonable to fix the ray intersection functionality at compile time, while providing run-time support for different surface materials.

One of our main goals will be to take advantage of the above observation and focus on maximum compile-time composability, while providing the opportunity to pay a performance price only when the provided flexibility is really needed at run time.

Some ray tracing systems provide too general abstractions for custom components. A classic example are the imperative surface shaders which are attached to geometry and are responsible for performing all the calculations for determining the color at a point. Such general abstractions enable highly optimized implementations, but can also lead to code replication, which reduces maintainability. Therefore, our library should be carefully designed, in order to avoid such code redundancy whenever possible.

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 a ray tracer, but imply unintuitive data layout and use of non-standard compiler intrinsics, which are thin wrappers around processor's instructions. Such low-level code will not be easily portable to future architectures which might have wider SIMD units. Thus, it would be desirable for our library to provide more intuitive abstractions for data-parallel computation, independent of the underlying instruction set and SIMD width. This would allow easy porting to future architectures without the need of modifying the source code of the entire library.

To talk it all together, the main objective of our real-time ray tracing library will be to provide a set of loosely coupled and fine-grained components and extensible interfaces, and all this without hurting performance. Well-designed interfaces help avoid many of the flexibility and extensibility problems we discussed above. The reason that most interactive ray tracing systems do not provide such flexibility has been that usually comes with considerable run-time overhead, especially at fine granularities. Therefore, we have take to take into account the fact that what is usually sufficient is compile-time flexibility, and we will optimize for that, while providing the opportunity for run-time flexibility where needed.

In the next sections we will discuss some design paradigms and will evaluate how they can help us meet our requirements.

## 4.4 Object-Oriented Design

All ray tracing systems, that we discussed and know of, have used object-oriented design with abstract virtual interfaces to achieve flexibility and polymorphic behavior of different component implementations, 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, dynamic polymorphism comes at a performance price. Late binding disables function inlining and inter-procedural 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 abstraction hierarchy can seriously degrade application's performance. That is why previous interactive systems that have employed object-oriented design have compromised both flexibility and performance, achieving neither the generality of offline rendering systems nor the speed of hand-tuned implementations.

Abstract interfaces also bring memory overhead – each virtual function increases the size of a type with the size of one pointer. This overhead gets larger if a specific alignment is required for a type, as adding the pointer causes padding of aligned data. Thus, the system might end up using as much as twice the amount of memory needed by the raw data, only because it allows polymorphic behavior of different components, even if this is not needed by the application. Thus, putting virtual interfaces at fine granularities will increase CPU register pressure and memory bandwidth requirements.

Another deficiency of object-oriented design is that it encourages coupling algorithms to data structures. Thus, ray tracing systems often implement building, traversal, and intersection algorithms within the acceleration structure and geometric primitive classes. As a result, different algorithms cannot be applied on the same data structures and vice versa, which reduces component reuse. The problem can be reduced by abstracting the acceleration structure implementations behind common sets of virtual interfaces. However, such separation is never actually used, as it would bring additional run-time overhead to functions which would need to be called millions of times each frame. A purely object-oriented design approach is therefore not an option for us, because it cannot deliver both flexibility and performance.

## 4.5 Domain-Specific Languages

One option for creating a flexible compile-time infrastructure would be to develop a domain-specific language (DSL). Such language would provide an intuitive and convenient syntax for writing custom components. At the same time, it would allow us to have more control over the low-level code generation via a specialized compiler.

A general problem of the DSL approach is that the development complexity directly depends on the level of functionality exposed to the language. Limited-scope problems tend to benefit from specialized languages, as their syntax remains simple and the compiler can be developed to fully understand the needs of a program from that domain and thus optimize the code well.

However, when we start increasing the scope of the language by allowing, for example, expressing ray tracing algorithms and data structures, we have to start considering efficiency, virtual functions, inlining, complex types, propagating properties, such as data alignment, to name a few. All this would bring considerable complication to the compiler, the supporting infrastructure, the language, for which it will become likely to converge to a general purpose language, which we have tried to avoid in the first place. For this reason, RTSL does not allow expressing complex algorithms and

data structures, but requires support from an underlying rendering system, for which the compiler also has to be modified accordingly.

Domain-specific languages are convenient because of their simple syntax, which makes them useful mostly 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, and object interaction. A context-dependent DSL would make such integration much harder. That is why we prefer to build a context-free library using a general purpose language and provide a flexible and convenient infrastructure for implementing reusable algorithms and data structures.

## 4.6 Generic Programming

The C++ programming language provides support for a third, often considered obscure, design paradigm – generic programming. Generic programming [23] 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.

While non-generic libraries use interfaces operating on predetermined data types, generic libraries provide algorithms that 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. The C++ Standard Template Library (STL) was the first widely used library to adopt these concepts.

### 4.6.1 Class and Function Templates

Generic algorithms can be expressed in C++ using class and function templates. Templates are definitions of classes and functions which can have class and integer compile-time parameters. Consider the following example:

```
template<class tElement, unsigned int taSize>
class Array
{
  Element values[taSize];
public:

  Element get(unsigned int aIndex)
  {
    assert(aIndex < taSize);
    return values[aIndex];
  }

  void set(unsigned int aIndex, Element aValue)
```

```
  {
    assert(aIndex < taSize);
    values[aIndex] = aValue;
  }
};

template<class tElement, unsigned int taSize>
void fillWithZero(Array<tElement, taSize>& aArray)
{
  for(unsigned int i = 0; i < taSize; ++i) {
    aArray.set(i, 0);
  }
}
```

The `Array` template class represents a simple static array. It can store elements of any type, providing safe accessors to the data, and can have arbitrary compile time defined size. The `fillWithZero` function can operate on `Array`s with any element type, as long as the element can be assigned the value `0`.

Templates can also have specializations for certain parameters:

```
template<unsigned int taSize>
class Array<bool, taSize>
{
  unsigned int values[taSize / 32 + taSize % 32 ? 1 : 0];
public:

  bool get(unsigned int aIndex)
  {
    assert(aIndex < taSize);
    return (values[aIndex / 32] >> (aIndex % 32)) & 1;
  }

  void set(unsigned int aIndex, bool aValue)
  {
    assert(aIndex < taSize);
    values[aIndex / 32] =
      (values[aIndex / 32] & ~(1 << (aIndex % 32))) |
      (aValue << (aIndex % 32));
  }
};
```

The above specialization provides a storage efficient implementation of the `Array` class for elements of type `bool`. The `fillWithZero` function will still work with this specialization of the class, at it provides the same interface as the generic version. 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 inter-procedural optimizations. This allows special case code to be resolved at compile time, without the need of separate code paths in the algorithms. For the above examples the compiler will *automatically* generate different core for each instantiation of the `Array` class and the `fillWithZero`

function, eventually optimizing each version in the best possible way.

### 4.6.2 Concepts and Models

A *concept* in generic programming is the set of requirements that a data type needs to meet in order to work correctly with a generic algorithm. Consider the following example:

```
template<class tType>
void order(tType& aElement1, tType& aElement2)
{
  if(aElement1 > aElement2)
  {
    tType tempElement = aElement2;
    aElement2 = aElement1;
    aElement1 = tempElement;
  }
}
```

The function `order` can order elements of any type that meets the requirements of the `Comparable` concept, i.e. which defines the **greater-than** operator. It is also said that such types *model* the concept.

A slight inconvenience in the above example is that requirements of the `Comparable` concept are concealed within the definition of the `order` function. Concepts can be explicitly described with *pseudosignatures*, which are pseudo class declarations specifying a set of requirements:

```
class Comparable
{
public:
  bool operator>(const Comparable& aOther);
}
```

Pseudosignatures can be extracted from actual implementations of a types that meet the specified requirements. Pseudosignatures have no semantic meaning – they play a purely documentation role. This means that the models of a concept do not need to explicitly inherit any specific classes or interfaces. On the other hand, the compiler cannot check whether a type models a certain concept, which might result in obscured error messages. Explicit support for concepts is scheduled for the next C++ standard release [41].

In some sense, concepts are 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 allow easier integration of different software libraries. Examples for successful generic libraries are Matrix Template Library [40], Boost [7], and Intel's Threading Building Blocks [33].

# Chapter 5

# Software Architecture

In this chapter we introduce the RTfact generic ray tracing library. We will first present the basic architecture of library and then we will describe the individual component categories and their interfaces.

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.

RTfact provides a set of generic packet data containers for convenient and efficient SIMD computation. Such low-level abstractions will become even more important, as future hardware will support 8-wide [13] and 16-wide [39] SIMD operations, which will become even more relevant for computationally demanding applications, such as ray tracing.

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. The algorithms can operate on packets of any size, and the packet size is simply a template parameter. As a result, algorithms can handle packets of different sizes simultaneously and can have manually specialized versions for certain 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. RTfact consists of five main groups of components: SIMD primitives (Section 5.1), ray tracing (Section 5.2), structure building, scene management (Section 5.3), and rendering (Section 5.4).

Figure 5.1 illustrates the basic structure of RTfact. The application has direct access to the scene management, acceleration structure building, and ray tracing components. It can use the ray tracing components for rendering or custom tasks, such as collision detection or object interaction.

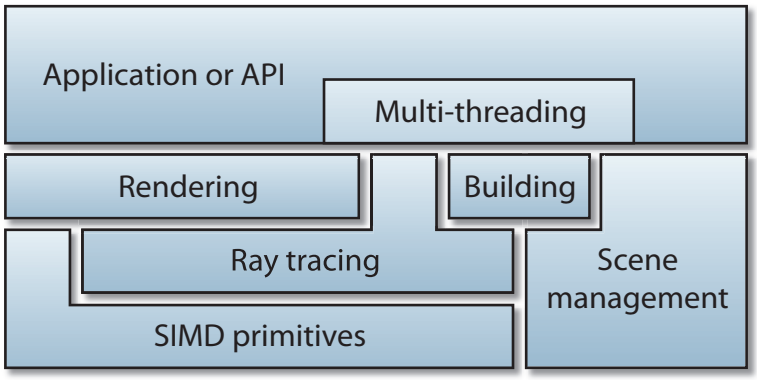


**Figure 5.1:** The multi-layer architecture of RTfact. The SIMD primitives form the basis for other generic components which are layered on top. Scene management and rendering are independent from each other and are connected through the ray tracing components. Thread management can be layered on top of the core components, or integrated into the application, which itself can be an API backend.

The components of the library are thread-aware but do not provide any thread management functionality, as this can be very application-specific.

## 5.1 SIMD Primitives

The basis of all algorithms and data structures in RTfact is formed by a collection of generic types for SIMD computation. These packets, as we will call them, are *compile-time fixed-sized* data containers 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.

### Basic Packet

```
template<unsigned int taSize, class tValue>
class Packet;
```

`Packet` is an ordered set of values. On SSE-compatible architectures, `tValue` can only be `float` or `int`.

### Three-component Vector

```
template<unsigned int taSize>
class Vec3f;
```

`Vec3f` is a three-component float vector packet with a structure-of-packets layout.

### Packet Mask

```
template<unsigned int taSize>
class PacketMask;
```

`PacketMask` is an ordered set of booleans. It stores the result of a comparison operation between two `Packet`s or `Vec3f`s, and defines a conditional `blend` operation, which blends two `Packet`s or `Vec3f`s according to the the stored mask.

### Bit Mask

```
template<unsigned int taSize>
class BitMask;
```

`BitMask` is an ordered set of bits.

RTfact's `Packet`, `Vec3f`, and `PacketMask` types resemble the standart scalar types used in computer graphics – `float`, `int`, `Vec3f`, and `bool`, but in a packet context. In this context, these scalar types are special cases of the packets, i.e. `Packet<1,float>` is a synonym for `float`. `BitMask` is a compact boolean value container, which serves mostly for efficient condition evaluation and code branching.

All four classes simultaneously model three concepts, which provide a unified interface independent of the packet size:

- 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 packet.

- `ValueContainer<Type>` defines indexing operations for accessing the individual values stored in the packet.

- `ContainerContainer<PacketType>` treats a packet as a container of sub-containers and defines indexing operators for accessing them. Sub-container are of type `PacketType` with a size depending on the size of the parent container and the SIMD width of the underlying architecture.

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

As packets of different sizes 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

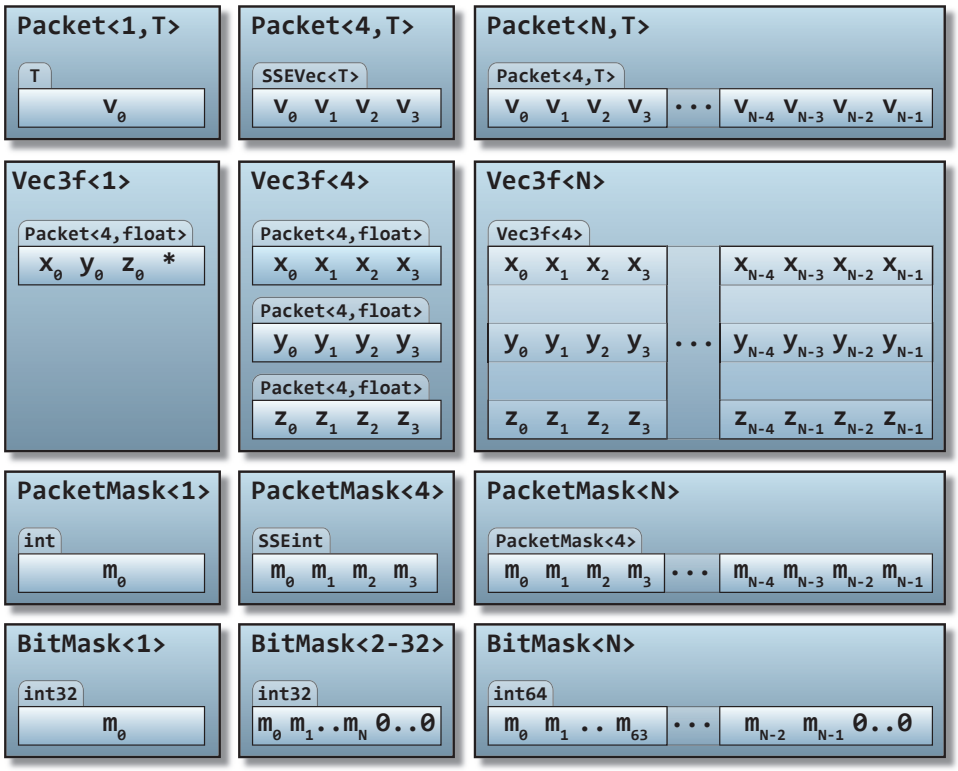


**Figure 5.2:** Data layout of packet primitives for SSE-compatible architectures. `Packet`, `Vec3f`, and `PacketMask` have specializations for sizes 1 and the architecture's SIMD width (in this case 4), for which the only sub-container is defined to be the packet itself. The generic implementations of the these packets can have size, which is a multiple of 4, and store 4-sized packets as subcontainers. `Vec3f<1>` internally uses an SSE vector for efficiency. `BitMask` implements a storage-efficient mask and uses built-in C++ types. `BitMasks` can have arbitrary size, and use specializations for sizes less than 32.

the three main concepts, but specialized versions of algorithms can take advantage of the additional functionality and internal layout of specialized packets. Figure 5.2 illustrates the data layout of the generic and specialized versions of the packet types.

All four packet classes model each of the three concepts, thus they have a three-fold nature. However, sequences of arithmetic operations may perform quite differently in each context, depending on the packet size and the amount of operations.

Consider the example illustrated in Listing 5.1, where the same sequence of operations is applied on the same data in three different ways. In the the scalar-like style of applying the operations, each arithmetic operation will be sequentially applied on the whole packets. Although the most

```
// specialized packets have convenient constructors
Vec3f<1>         v1(0.1, 0.2, 0.3);
Packet<4, float> p4(1, 2, 3, 4);

// shuffling (defined only for 4-sized packets)
// in this example p4s will have a value (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);
// inefficient for a sequence of operations on large packets
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();

// component-wise packet operations (ValueContainer concept);
// does not utilize SIMD units
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 concept);
// efficient for arithmetic operations on packets of all sizes
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 5.1:** SIMD computation with packets. Most binary operations (such as addition) can be performed in three different ways.

simple and intuitive, this is not the most efficient way of applying the operations in this particular case. Because the packets are large, they cannot fit into the CPU registers. Thus, each peace of data in a packet will be read from memory and written back many times, and will never be available in a register for the next operation. This will result in lower CPU utilization and a lot of memory bandwidth, and efficiency tends to decrease with larger packets.

The component-wise style of applying sequential operations can be more efficient for large packets than the scalar-like style, as it will keep the result from each operation in a CPU register, ready to be used for the next operation. However, it will not utilize the SIMD units at all, as it only performs scalar arithmetic. Therefore, such approach does not scale to larger SIMD units.

The most efficient way of applying sequential arithmetic operations on packets is the container-wise style, illustrated in Listing 5.1. It exhibits both lower memory bandwidth requirements and high SIMD utilization, as it executes all operations in parallel on a chunk of data sequentially. This is why most algorithms operate on the `ContainerContainer` concept.

As a rule of thumb, the `ContainerContainer` concept should be used whenever possible. It performs efficiently for all arithmetic operations, even for packets of size 1 and 4 – the `for` loops for such packets will have only one iteration, which is known at compile time and thus it will be optimized out by most compilers. The `ValueContainer` concept is useful for accessing individual values of packets incoherently. For example, iterating over all elements in a packet is currently the only way of simulating scatter and gather memory operations, as they are not supported by the SSE instruction set. The `Value` concept is still useful for performing single arithmetic operations, as they internally perform `ContainerContainer`-style iterations.

While our current implementation supports SSE only, the concepts themselves do not have assumptions or restrictions on the instruction set or the SIMD width. Thus, support for the Intel AVX [13] instruction set can be easily added. In fact, such abstractions over native SIMD operations can greatly improve portability – when adding support for wider SIMD operations, updates are needed only for code using horizontal operations currently defined only for 4-sized packets, such as shuffling. Most of the algorithms in RTfact, however, are written entirely in terms of the three basic concepts and will thus require no modifications at all.

### 5.1.1 Ray Packets

RTfact does not distinguish between individual rays and ray packets. Rays are simply packets aggregating other packets (see Listing 5.2). Thus, a single ray is simply a ray packet of size 1.

Template instantiation adds two desirable properties to ray packets.

```
template<unsigned int taSize>
struct RayPacket
{
  Vec3f<taSize>         org;  // ray origins
  Vec3f<taSize>         dir;  // ray directions
  Packet<taSize, float> tMin; // minimum clipping distances along the rays
  Packet<taSize, float> tMax; // maximum clipping distances along the rays
};
```

**Listing 5.2:** In RTfact, single rays and packets of rays are represented with the same generic data structure.

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 ray tracing systems. This allows us to efficiently trace the same or different acceleration structures with packets of different sizes.

Ray packets can be extended to carry information bout active rays, corner rays, and bounding planes, which enable efficient frustum-based acceleration structure traversal and primitive intersection. However, we do not store intersection data within ray packets, as intersection structure types are defined by intersectors (see Section 5.2.2).

## 5.2 Ray Tracing Components

Packets provide us with the basis on which we can 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.

### 5.2.1 Primitives and Acceleration Structures

In RTfact, all ray tracing structures model the very general `Primitive` concept, which only defines functionality for getting material ID. These include geometric primitives, compound primitives, and acceleration structures.

In contract to PBRT, `Primitive`s are simple data structures and do not directly provide intersection functionality, which is instead provided by `Intersector`s. For applications that 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 virtual function calls completely

or add them only when required and at the appropriate granularity. For example, we allow different polymorphic intersectors to be attached to different geometric primitives at run time. However, this can also be applied at a coarser granularity, e.g. assign different intersectors to acceleration structure instances, while fixing the intersectors for all triangles either inside each structure or globally. This allows us to have fine control over name binding and thus to the run-time infrastructural overhead.

We also set 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 the same or different types to be nested.

Listing 5.3 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.

```
template<class tElement>
class KdTree : public Primitive
{
public:

  class NodeIterator;
  class ElementIterator;

  // interface for building
  void createInnerNode(NodeIterator node,
                       int axis, float splitValue);
  template<class tIterator>
  void createLeaf(NodeIterator leaf, const BBox& bounds,
                  tIterator begin, tIterator end);

  // interface for traversal
  NodeIterator getRoot() const;
  NodeIterator getLeftChild(NodeIterator node) const;
  NodeIterator getRightChild(NodeIterator node) const;
  int gesplitAxis(NodeIterator node) const;
  float gesplitValue(NodeIterator node) const;
  std::pair<ElementIterator, ElementIterator>
    getElements(NodeIterator leaf) const;
};
```

**Listing 5.3:** A psudosignature of a generic kd-tree working with STL-like iterators. A BVH concept would be similar.

### 5.2.2 Intersectors

Given a ray packet and a primitive, `Intersectors` return an intersection structure. Similarly to PBRT, we design primitive and 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, i.e. the kd-tree traverser will also perform triangle intersection, as triangles may be generated on the fly. In order to facilitate component reuse, our design encourages but does not require separation of intersection algorithms.

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 5.4 illustrates such a routine for a kd-tree intersector.

```
template<class tElemIsect>                    // nested element intersector
template<int taIntersDataMask,                // intersection data needed
         bool taCommonOrg,                    // common ray origin?
         unsigned int taSize,                 // size of the ray packet
         class tKdTree>                       // models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
```

```
  RayPacket<taSize>& rayPacket,              // the actual rays
  tKdTree& tree,                             // the actual kd-tree
  tElemIsect::Intersection<taSize>& r)       // intersection defined
{                                            //  by the nested intersector
  typedef BitMask<taSize>              t_BitMask;
  typedef Packet<taSize, float>        t_Packet;
  typedef typename t_Packet::Container t_PContainer;
  /* omitted: initialize traversal */
  tKdTree::NodeIterator node = tree.getRoot();
  int splitDim; // split dimension (3 means leaf node)
  while(true) {
    while((splitDim = tree.gesplitAxis(node)) != 3) {
      t_PContainer splitValue =
        t_PContainer::replicate(tree.gesplitValue(node));
      t_BitMask nearChildMask, farChildMask;
      t_PContainer splitFactor;

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

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

        const t_PContainer split = splitFactor *
          rayPacket.invDir(i).get(splitDimension);
        nearChildMask.setCont(i,(split(i) > currentTMax(i)).getIntMask());
        farChildMask.setCont(i,(currentTMin(i) > split(i)).getIntMask());
      }
      /*omitted: get first child from masks and descend */
    }
    // a leaf node has been reached
    std::pair<tKdTree::ElementIterator, tKdTree::ElementIterator>
      elemIterators = aTree.getElements(currentIterator);
    if(elemIterators.first != elemIterators.second) {
      do { //invoke nested intersector for leaf elements
        mIntersector.intersect<taIntersDataMask, 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 5.4:** A generic kd-tree traversal routine. Only the kd-tree and ray data are passed at run-time. All other parameters are known at compile time.

## 5.3 Scene Management and Acceleration Structure Building

As with acceleration structures and intersection algorithms, 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. The `BasicScene` concept in addition defines geometry and light source lists, while `SceneGraph` provides functionality for managing scene hierarchies.

Acceleration structures are built by `Builder`s. 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.

As ray intersection algorithms do not directly operate on scenes and are separated from acceleration structure building, scene and acceleration structure management is entirely independent from ray tracing. This allows tighter coupling of scenes and acceleration structure building. For example, it has been shown that acceleration structures can be robustly built in linear time from scene hierarchies [16]. In such case, a builder can be specialized for a particular scene type and exploit its structured information to accelerate building.

## 5.4 Rendering Components

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

The clear separation of ray tracing and rendering functionality allows the application to use the same acceleration structures and intersection algorithms both for rendering and for other tasks too. For example, ray tracing can be used for collision detection in dynamic environments and acoustic simulation [36]. Object interaction can be also facilitated by ray tracing – when a user clicks on a pixel on the screen, a ray can be traced to detect which object the user wants to select. Thus, no special acceleration structures and algorithms need to be developed for such tasks.

### 5.4.1 Shading Model

Most rendering systems employ an *imperative* 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

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

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

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

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

**Listing 5.5:** A concept for physically-based materials. Materials represent surface reflection models and are independent from ray tracing.

simple to implement and gives freedom – shaders can shade surfaces in arbitrary ways.

Unfortunately, imperative 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 imperative shaders as well as a *declarative* physically-based shading model. Similarly to PBRT, we define surface reflection models as `Material`s and light integration algorithms as `Integrator`s. 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 5.5). Listing 5.6 illustrates the `Integrator` concept.

```
class Integrator {
public:
  template<int taSize> struct Result;
  template<int taSize, class tSample, class tScene,
           class tPrimitive, class tIntersector>
  Result<taSize> eval(
    tSample& sample               // image sample
    RayPacket<taSize>& rayPacket, // initial ray packet
    tPrimitive& primitive,        // top-level primitive
    tIntersector& intersector,    // top-level intersector
    tScene& scene);               // shading scene data
};
```

**Listing 5.6:** 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.

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. `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 material and light source types at run-time, we provide a *transparent* virtual function mechanism between the `Material` and `Light` concepts and their models. Custom materials have to implement the non-virtual template functions defined by the concepts, and include a special header file in their class definition. This header file automatically generates virtual functions for each possible combination of the template parameters of each function in the concept and includes them in the class body. When a template function of `Material` is called, it will divert the call to a virtual function which is bound to a corresponding auto-generated virtual function in the custom material. Finally, this function will call the generic version with the appropriate template parameters. Such infrastructure is necessary, because the C++ language does not currently support template virtual functions. Although this may seem like a big overhead, in practice all non-virtual functions are inlined by the compiler, resulting in no additional overhead except for the one introduced by virtual functions. Note also, that the number of generated virtual functions can grow exponentially with the possible combinations of template parameters. This will introduce memory overhead, but

```
class Texture
{
public:
    template<uint taSize>
    void sample(SurfaceIntersection<taSize>& aIntersection,
                implementation-defined& oResult);
};
```

**Listing 5.7:** `Texture` is an algorithm concept for querying material properties. The returned result is implementation-defined and is usually a float `Packet` or a `Vec3f`.

in practice it is negligible, as the number of materials in the scene is usually much smaller than the number of geometric primitives.

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.

### 5.4.2 Texturing

Similarly to PBRT, materials in RTfact always query textures for properties which may vary along surfaces. `Texture` is a basic algorithm concept for querying intersection-dependent data (see Listing 5.7). In contrast to PBRT, material classes are parameterized by the types of the textures they use. For example, `LambertianMaterial` is parameterized by the types of reflectance and emission textures:

```
template<class tReflectanceTexture, class tEmissionTexture>
class LambertianMaterial
{
  tReflectanceTexture mReflectanceTexture;
  tEmissionTexture    mEmissionTexture;
  ...
};
```

The most basic type of texture is the `Float3ConstantTexture`. It always returns the same value, independent of the intersection data. The `Float3Texture2D` concept represents a two-dimensional texture. Its models can either look up the return value in an image or generate it proceduraly. All texturing functionality is hidden behind the basic `Texture`, and is thus completely independent from materials.

Because the types of the textures are resolved at compile time, all code can be optimized by the compiler. Thus, `Float3ConstantTexture`, for

example, is as efficient during rendering as a hard-coded value in the material.

## 5.5 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 5.8 shows an example renderer, which defines a basic ray tracing pipeline, which works as follows. Samples are queried from the pixel sampler, and then are given to the camera to generate a primary ray packet from them. The ray packet is passed to the integrator, which returns the radiance flowing along the rays in the packet. Finally the original pixel sample and the result from the integrator is given back to the sampler, which writes the result to the framebuffer.

This renderer makes little assumptions about how different compo-

```
template<class tPixelSampler, class tIntegrator>
class RayTracingRenderer : public Renderer
{
  tPixelSampler mSampler;
  tIntegrator   mIntegrator;
public:
  template<int taSize, // the size of primary ray packets
           class tCamera, class tScene, class tPrimitive,
           class tIntersector, class tFramebuffer>
  void render(
    tScene& scene, tCamera& camera,
    tFramebuffer& framebuffer, ImageClipRegion& clip,
    tPrimitive& primitive, tIntersector& intersector)
  {
    tPixelSampler::Sample<taSize> sample;
    tPixelSampler::Iterator<taSize> it =
      mSampler.getIterator<taSize>(clip);

    while(it.getNextSample(sample))
    {
      RayPacket<taSize> rays = camera.genRay(sample);
      Integrator::Result<taSize> result = mIntegrator.eval(
        sample, rays, primitive, intersector, scene);
      mSampler.writeResult(sample, result, framebuffer);
    }
  }
  /* omitted: preprocess() function for pre-integration*/
};
```

**Listing 5.8:** 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.

nents 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 run-time 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.

# Chapter 6

# Applications

RTfact is a source code library and can be used in a similar way to other generic libraries, such as STL. Typically, the user includes the desired RTfact headers in his or her application source files and combines and instantiates the chosen algorithms and data structures in the most suitable way. The components of the library are designed to be highly configurable and extensible, so that the user can choose 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 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 acceleration structures and intersection algorithms, and instantiating the renderer with them.

In the following sections we will show how RTfact can be applied to the above mentioned visualization tasks. We will focus on the performance achieved in surface ray tracing, while demonstrating how the infrastructure can be applied for the other two tasks.

## 6.1 Surface Ray Tracing

RTfact was originally inspired by the recent advances in surface ray tracing. Modern intersection algorithms enable interactive visualization of complex geometry, and can achieve performance of over 10 million rays per second on a single CPU core on scenes consisting of several million polygons.

Unfortunately, there has been a gap between the performance reported

```
OpenGLFrameBuffer framebuffer;
ImageClipRegion clipRegion;
PinholeCamera camera;
BasicScene<Triangle> scene;
BVH<Triangle> bvh;
BVHIntersector<PlueckerTriangleIntersector> bvhIntersector;
SAHBVHBuilder builder;
RayTracingRenderer<PixelCenterSampler,
                   DirectIlluminationIntegrator> renderer;

// initialization omitted
...

builder.build(bvh, scene.primitives.begin(), scene.primitives.end());

renderer.render<256>(scene, camera, framebuffer, clipRegion,
                     bvh, bvhIntersector);
```

**Listing 6.1:** Using RTfact for surface ray tracing. Independent library components are instantiated and combined by the application according to its specific needs.

in research papers and the one achieved by more general purpose systems. Our goal is to achieve the performance of single-purpose hand-optimized implementations, while providing the same or even greater flexibility than other practical ray tracing systems.

Listing 6.1 shows an example usage of RTfact for surface ray tracing. The application usually creates its own framebuffer, depending on what subsystem is used for image display. After that it populates the scene and instantiates the desired ray tracing algorithms and builds the acceleration structures. Finally, the `render` method of the renderer is called to start the computations for the frame.

In the example shown in Listing 6.1, we have chosen to use a BVH, built using a SAH-based algorithm. Ray intersection is performed by the intersector resulting from the combination of a BVH traverser and a Pluecker triangle intersector. The renderer is configured to trace one ray per pixel and compute direct illumination. All structures and algorithms are passed to the `render` function with an additional parameter specifying the ray packet size.

At the point of invocation of the renderer's `render` method, the compiler will start resolving type dependencies and instantiating templates. The result will be an automatically generated specialized version of the `render` method for the given template parameters.

Specializations of algorithms will be also automatically resolved. For example, if the `BVHIntersector`'s `intersect` method has been specialized for 256-sized ray packets, it will be chosen by the compiler and embedded into the final code produced, resulting in no run-time overhead.

Other interactive ray tracers, such as Arauna [6] and Manta, have

| | 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. [47] 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 6.1:** 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 [47] 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 [46], while B denotes BVH packet traversal [47].

separate code paths for single rays or for SIMD processing. However, template instantiation allows special case code to be put at the appropriate granularity, without the need of separate code paths for the entire program.

Figure 6.1 shows some images rendered at a $1024 \times 1024$ resolution with configurations similar to the one in Listing 6.1. The Sponza scene has about 70,000 triangles and uses one point light source, textures, and interpolated shading normals. The Conference scene has roughly 280,000 polygons and uses two point light sources and has a mirror on one of the walls. The Soda Hall scene consists of 2.2 million triangles and is rendered with simple diffuse shading. All scenes are rendered interactively on a mobile Core 2 Duo 2.6Ghz processor.

### 6.1.1 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 have applied the same simplified conditions on all systems and measure the time for ray casting and simple shading only.

As it can be seen from Table 6.1, the performance of RTfact matches the one of Arauna [6]. 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.

**Figure 6.1: Top:** SPONZA with one light source (8.4 fps). **Left:** CONFERENCE with a mirror and two light sources (8 fps). **Right:** SODA HALL with simple shading (22 fps).

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 assembly output 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.

### 6.1.2 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 6.1 left) most of which are 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.

## 6.2 Point-Based Ray Tracing

We have also applied RTfact for interactive ray tracing of point clouds. We have implemented an optimized version of the algorithm proposed by Katov [19]. The algorithm directly visualizes points without hole filling, and employs a generic level-of-detail kd-tree, which stores primitives at multiple levels in the hierarchy.

Listing 6.2 illustrates an example how to setup a point-based ray tracing renderer. The structure of the code is mostly the same as for surface rat tracing, but uses a different acceleration structure, intersector, and builder. Also, the ray packet size used for rendering is set to 16. As discussed in the previous section, these two different rendering configurations can co-exist in the system and even render to the same framebuffer.

Figure 6.2 shows two images of ray traced point clouds. The DAVID model consists of approximately 3.7 million points, while the DRAGON model has 3.6 million points. On a mobile Core 2 Duo 2.6GHz processor at a $1024 \times 1024$ resolution, the scenes render at 6 fps and 7 fps, respectively.

## 6.3 Direct Volume Rendering

As a proof of concept, we have also applied RTfact for direct volume rendering of CT scanned datasets. The implemented algorithm is a brute-force single-ray regular grid traverser.

As can be seen from Listing 6.2, direct volume rendering also fits nicely into the rendering pipeline that we have used for surface and point-

```
OpenGLFrameBuffer framebuffer;
ImageClipRegion clipRegion;
PinholeCamera camera;
// point-based ray tracing setup
BasicScene<Point> scene;
LoDKdTree<Point> kdtree;
LoDKdTreeIntersector<PointIntersector> lodKdTreeIntersector;
LoDKdTreeBuilder builder;
LoDParams lodParams;
RayTracingRenderer<PixelCornerSampler, EyelightIntegrator> renderer;
// initialization omitted
...

builder.build(kdtree, lodParams,
              scene.primitives.begin(), scene.primitives.end());
renderer.render<16>(scene, camera, framebuffer, clipRegion,
                    tree, lodKdTreeIntersector);
// direct volume rendering setup
BasicScene<DensityPoint> scene;
Grid3D<DensityPoint> grid;
VolumeGridIntersector gridIntersector;
Grid3DBuilder builder;
RayTracingRenderer<PixelCenterSampler, VolumeIntegrator> renderer2;
// initialization omitted
...

renderer.render<1>(scene, camera, framebuffer, clipRegion,
                   grid, gridIntersector);
```

**Listing 6.2:** Using RTfact for level-of-detail point-based ray tracing. The structure of the application is the same and many of the components used for surface ray tracing can be reused.

based ray tracing. The major difference is that a different integrator is used, namely the `VolumeIntegrator`. This integrator is tightly coupled with the `Grid3Dintersector` and is called at each grid traversal step to compute the contribution of each grid cell a ray pierces. However, these details remain transparent to the `RayTracingRenderer`.

Figure 6.2 shows two images rendered with the volume renderer. The SKELETON dataset is visualized using a single-color transfer function, and the ENGINE dataset uses a two-color transfer function. Both images are rendered at a $1024 \times 1024$ resolution, although not interactively, as the grid traversal is not optimized and traces one ray at a time.

## 6.4 A Note on Parallelism

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

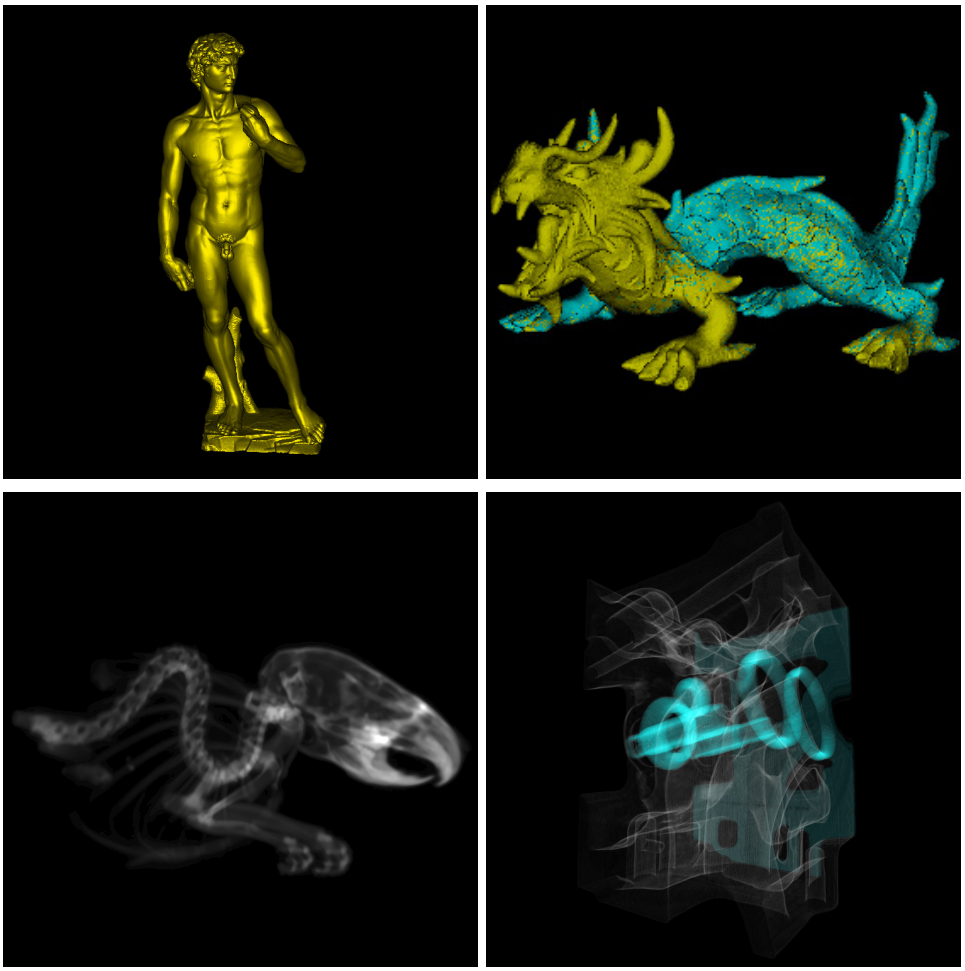

**Figure 6.2: Top row:** Point-based ray traced images with level of detail. **Left:** The DAVID model. **Right:** The ASIAN DRAGON model with level of detail visualized in false color. **Bottom row:** Volumetric datasets visualized using direct volume rendering. **Left:** The SKELETON dataset. **Right:** The ENGINE dataset with a two-color transfer function.

the image plane and invoke the renderer for each tile. We have run tests on up to 16 cores and achieved near-linear scalability. Note that RTfact optimizes for single-thread throughput, which is completely orthogonal to multi-threaded execution. Various parallelization schemes can be applied on top of the library for achieving maximum multi-core hardware utilization.

# Chapter 7

# Conclusions

In this thesis 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.

RTfact provides a SIMD abstraction layer, which constitutes the basis for generic packet-based allgorithms and enables seamless portability to architectures with wider SIMD instruction sets.

As graphics hardware is moving toward 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 [25]. We achieve high performance by putting the pressure on the compiler, thereby avoiding complex run-time control flow and overhead.

## 7.1 Limitations

Although we argue that the design paradigms used in RTfact provide both flexibility and performance, we have experienced some problems with this approach. These problems are mainly consequences of some limitations of the C++ language.

First of all, the concepts behind the library and its code can be hard to understand by inexperienced programmers. Object-oriented design has been well studied, supported elegantly in almost all modern programming

languages, and thus most people are familiar with it. The generic programming paradigms, however, are often considered too obscured. One reason for this is that the support for generic programming varies greatly among languages and compilers with respect to features, performance, and ease of use.

The C++ language provides the opportunity for achieving compile-time flexibility and run-time performance via templates. However, templates add complications to the compiler logic, and therefore the support for templates in the language itself has been limited in a number of ways. For example, one of the biggest problems in the template infrastructure of RTfact is the support for run-time polymorphism. The main reason for this is the lack of native support for template virtual functions. For evaluating materials and invoking shaders, we have to manually generate virtual functions from template versions in the base and derived classes. This, however, requires explicit knowledge of all possible template arguments and their combinations. Even worse, the number of virtual functions we have to generate grows exponentially with the number of template parameters. We end up generating a number of virtual functions, most of which are not called at all. Support by the compiler to automatically detect which combinations of template parameters are actually used and generate the appropriate virtual functions would solve this problem.

Another problem with templates are the often too long, confusing and unhelpful error messages in code that uses templates. Thus, developing template code can be difficult and time consuming to develop. This is mainly a consequence of the lack of explicit support for concepts. The C++0x revision of the C++ standard adds such support and we expect it to drastically reduce this problem.

There are many other peculiarities of the C++ language that make generic code hard to understand and write, such as obscure compiler hints through special keywords and inheritance and specialization rules.

## 7.2 Future Work

Although RTfact is still under development, it already shows great potential. In future, we will investigate the possibility of applying the generic design for building GPU ray tracers and building a unified framework for hybrid CPU-GPU rendering. In addition, we plan to integrate the library into a virtual reality system. In order to do this, we will continue extending the coverage of RTfact by providing for example support for collision detection for physics simulation.

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. In future, we plan

to add a continuation-based multi-tasking layer on top of the library, which will facilitate hybrid, multi-core, and distributed rendering.

We advocate generic software design as a key to flexibility and efficiency, especially for computationally intensive applications, such as realtime ray tracing. Therefore, as a long-term project, we plan to work together will compiler developers on languages that provide seamless support for generic programming.

# Bibliography


[1] John Amanatides and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: (1978), pp. 3–10.

[2] Arthur Appel. "Some Techniques for Shading Machine Rendering of Solids". In: *Spring Joint Computer Conference 32* (1968), pp. 37–49.

[3] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. "Ray Tracing on the CELL Processor". In: *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 2006.

[4] Carsten Benthin, Ingo Wald, and Philipp Slusallek. "A Scalable Approach to Interactive Global Illumination". In: 2003.

[5] James Bigler, Abe Stephens, and Steven G. Parker. "Design for Parallel Interactive Ray Tracing Systems". In: *IEEE Symposium on Interactive Ray Tracing* (2006).

[6] Jacco Bikker. "Real-time Ray Tracing through the Eyes of a Game Developer". In: *IEEE Symposium on Interactive Ray Tracing* (2007).

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

[8] Nathan A. Carr, Jesse D. Hall, and John C. Hart. "The ray engine". In: *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Saarbrucken, Germany: Eurographics Association, 2002, pp. 37–46.

[9] Holger Dammertz, Johannes Hanika, and Alexander Keller. "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays". In: *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*. 2008, pp. 1225–1234.

[10] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. *OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Tech. rep. Saarland University, 2002.

[11] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. "The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing". In: (2003), pp. 23–31.

[12] Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006. ISBN: 1568813074.

[13] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency.*

[14] Andrew S. Glassner. *An Introduction to Ray Tracing.* Morgan Kaufmann, 1989.

[15] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. "Realtime Ray Tracing on GPU with BVH-based Packet Traversal". In: *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing.* 2007.

[16] Warren Hunt, William R. Mark, and Don Fussell. "Fast and Lazy Build of Acceleration Structures from Scene Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing.* IEEE, Sept. 2007, pp. 47–54.

[17] Frederik W Jansen. "Data Structures for Ray Tracing". In: (57–73), p. 1986.

[18] Jim Kajiya. "The Rendering Equation". In: *SIGGRAPH '86 (Proceedings of the 13th annual conference on Computer graphics and interactive techniques)* (1986).

[19] Marin Katov. *Ray Tracing of Points.* 2007.

[20] Alexander Keller. "Instant Radiosity". In: *SIGGRAPH '97 (Proceedings of the 24th annual conference on Computer graphics and interactive techniques)* (1997).

[21] Eric Lafortune and Yves D. Willems. "Bi-directional Path Tracing". In: 1993.

[22] William Mark. "Future Graphics Architectures". In: *ACM Queue* 6.2 (2008).

[23] David Musser and Alexander Stepanov. "Generic Programming". In: *SSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation.* 1989.

[24] Michael J. Muuss. "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models". In: *Proceedings of BRL-CAD Symposium.* 1995.

[25] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable Parallel Programming with CUDA". In: *ACM Queue* 6.2 (2008). ISSN: 1542-7730.

[26] Steven Parker, Solomon Boulos, James Bigler, and Austin Robison. "RTSL: A Ray Tracing Shading Language". In: *IEEE Symposium on Interactive Ray Tracing* (2007).

[27] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen. "Interactive Ray Tracing". In: *Proceedings of Interactive 3D Graphics*. 1999.

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

[29] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (1975), pp. 311–317. ISSN: 0001-0782.

[30] Pixar Animation Studios. *The RenderMan Interface Specification*.

[31] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing". In: *Computer Graphics Forum*. Vol. 26. 3. 2007.

[32] Timothy Purcell. "Ray Tracing on a Stream Processor". Stanford University. PhD thesis. 2004.

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

[34] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. "Multi-level Ray Rracing Algorithm". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. Los Angeles, California: ACM Press, 2005, pp. 1176–1185.

[35] Steven M. Rubin and Turner Whitted. "A 3-dimensional Representation for Fast Rendering of Complex Scenes". In: (1980), pp. 110–116.

[36] Jörg Schmittler, Daniel Pohl, Tim Dahmen, Christian Vogelgesang, and Philipp Slusallek. "Ray Tracing for Current and Future Games". In: *Proceedings of 34. Jahrestagung der Gesellschaft für Informatik*. 2004.

[37] Benjamin Segovia, Jean-Claude Iehl, Richard Mitanchey, and Bernard Péroche. "Bidirectional Instant Radiosity". In: *Proceedings of the 17th Eurographics Workshop on Rendering, to appear*. 2006.

[38] Benjamin Segovia, Jean-Claude Iehl, and Bernard Péroche. "Metropolis Instant Radiosity". In: *Proceedings of Eurographics 2007, to appear*. 2007.

[39] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. "Larrabee: A Many-Core x86 Architecture for Visual Computing". In: *ACM SIGGRAPH 2008*. 2008.

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

[41] Bjarne Stroustrup. "Evolving a Language in and for the Real World: C++ 1991-2006". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. San Diego, California: ACM, 2007.

[42] Kalpathi R. Subramanian and Donald Fussel. *A Search Structure Based on K-D Tree Trees for Efficient Ray Tracing*. Tech. rep. University of Texas at Austin, 1990.

[43] Shreekant (Ticky) Thakkar and Tom Huff. "The Internet Streaming SIMD Extensions". In: *Intel Technology Journal* (1999).

[44] Eric Veach and Leonidas Guibas. "Bidirectional Estimators for Light Transport". In: 1994, pp. 147–162.

[45] Ingo Wald, Carsten Benthin, and Solomon Boulos. "Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs". In: (Aug. 2008).

[46] Ingo Wald, Carsten Benthin, Philipp Slusallek, and Michael Wagner. "Interactive Rendering with Coherent Ray Tracing". In: *Computer Graphics Forum* 20(3) (2001), pp. 153–164.

[47] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies". In: *ACM Trans. Graph.* 26.1 (2007), p. 6. ISSN: 0730-0301.

[48] Ingo Wald, Andreas Dietrich, Carsten Benthin, Alexander Efremov, Tim Dahmen, Johannes Guenther, Vlastimil Havran, Hans-Peter Seidel, and Philipp Slusallek. "Applying Ray Tracing for Virtual Reality and Industrial Design". In: *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, pp. 177–185.

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

[50] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. "Interactive Global Illumination Using Fast Ray Tracing". In: *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*. Pisa, Italy: Eurographics Association, 2002, pp. 15–24. ISBN: 1-58113-534-3.

[51] Bruce Walter, Sebastian Fernandez, Adam Arbee, Kavita Bala, Michael Donikian, and Donald Greenberg. "Lightcuts: A Scalable Approach to Illumination". In: *ACM SIGGRAPH Conference Proceedings* (2005).

[52] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Communications of the ACM* 23.6 (1980), pp. 343–349. ISSN: 0001-0782.