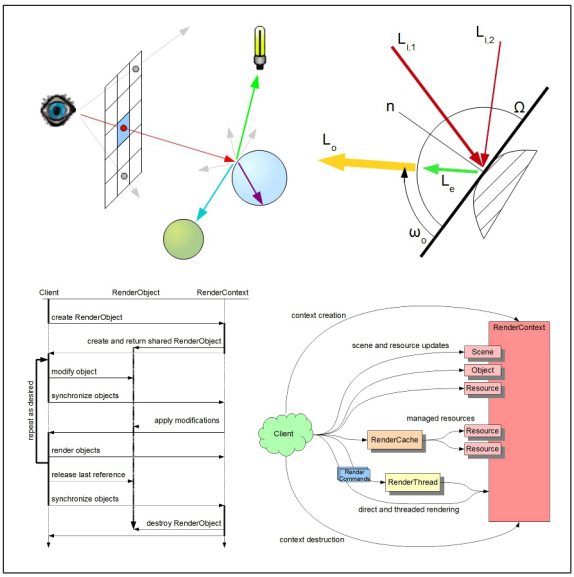
Integration of an Interactive Raytracing-Based Visualization Component into an Existing Simulation Kernel

Master Thesis



Tobias Werner

October 10, 2011

Angewandte Informatik V — Intelligent Graphics Group

Universität Bayreuth
95440 Bayreuth, Germany

Table of Contents

Table of Figures

1 Introduction

1.1 Abstract

Within this thesis, the development of an interactive raytracing component utilizing the NVIDIA OptiX API and the integration of said component into the Scala-based virtual reality kernel Simulator X are described. In particular, requirements for a general renderer interface with both support for an interactive raytracer back-end and integration into a Scala environment are derived. Thereafter, an appropriate C++ back-end implementation with OptiX hardware acceleration is created, and an associated Scala wrapping layer is developed. Finally, the resulting raytracing component is evaluated in comparison to a competing traditional rasterization component. This yields a conclusion on the current state of interactive raytracing.

Thus, the topic of this thesis: Integration of an Interactive Raytracing-Based Visualization Component into an Existing Simulation Kernel.

1.2 Overview

In the course of the following chapters, the entire development process of the interactive, hardware-accelerated raytracing component for the Simulator X platform is covered. This includes both basic concepts — such as Simulator X and OptiX API fundamentals — as well as interface design, implementation work, and an integration step into the simulator framework. In particular, each chapter covers a single aspect within the development of the raytracer:

- Chapter 2 gives an overview over basic rasterization and raytracing algorithms and explains their respective advantages and disadvantages. Thereafter, the history of computer graphics and graphics hardware is reviewed to explain why hardware accelerated raytracing became possible on modern graphics hardware.
- The current state-of-art, both for interactive raytracing and for renderer architecture, is investigated in chapter 3.
- Chapter 4 elaborates the software architecture of Simulator X. Fundamental concepts as well as their realization are discussed. Finally, the existing rasterization-based rendering component within the framework is reviewed.
- Covering of the actual raytracer development starts with chapter 5. In particular, NVIDIA's OptiX middleware platform for hardware accelerated raytracing is introduced as the groundwork for the raytracer. An example application is derived to further improve on the understanding of OptiX internals.
- Chapter 6 proposes a general renderer interface that meets requirements such as multithreading support and intuitive client-side use. The interface consists of certain subordinate components which are defined in regards to their functional behavior.

- Within chapter 7, the results from the previous two chapters are combined to implement the general renderer interface and its OptiX back-end in C++.
- The wrappers that connect the raytracer and the simulation kernel are presented in chapter 8. This concludes the development of the raytracing component. Thereafter, the OptiX raytracing component is integrated into a Simulator X example application for later testing.
- In chapter 9, the finished raytracing kernel is compared to the original rasterizer component within Simulator X. The results are evaluated to determine the fitness of the raytracer implementation.
- Finally, chapter 10 provides a summary on the results of this thesis, and draws a conclusion on the feasibility of raytracing on modern graphics hardware.

2 Rasterization and raytracing

This chapter defines basic rendering-related terms used throughout this thesis, and gives an introduction into both rasterization and raytracing. Similarities between both approaches are named, and their respective benefits are contrasted.

In-depth details on both rendering approaches may be obtained from a variety of academical and commercial sources, such as [AW00], [WPa] or [WPb]. In particular, the explanations and arguments within this chapter are based on [FA09].

2.1 Basic concepts

In the context of computer graphics, rendering in general refers to the process of deriving a 2D representation of a 3D scene by the use of a rendering algorithm.

Most commonly, the 3D scene is some in-memory, mathematical data set. For instance, a scene may contain research measurements, medical data from a MRT device, or the virtual environment of a computer generated movie. In many cases, the dataset is formed by continuous objects, such as lines, triangles, or mathematical surfaces.

The 2D representation usually is any raster image with discrete pixel elements. Common raster images are RGB bitmaps for display on a computer screen or similar device.

The rendering algorithm describes the mapping process that translates the 3D scene dataset to its 2D counterpart. Within the development of computer graphics, two fundamental algorithm families have evolved:

- **Rasterization** creates a 2D raster representation by projecting the 3D scene into an intermediate 2D form. The intermediate form is sampled in 2D to determine the result image.
- **Raytracing** creates the 2D representation by following rays directly within the 3D scene. Thus, the scene is sampled in 3D without any intermediate 2D form.

2.2 Rasterization

In traditional rasterization, graphic objects are tesselated into a series of 3D triangles. For curved surfaces, this means a discrete approximation has to be determined.

The input triangles are projected in-situ into a flat, normalized screen space that surrounds a virtual scene viewer. During this projection, the depth of triangles is mapped to perspective distortion to simulate a true 3D view. Mathematically, this involves a matrix multiplication operation that encapsulates the respective projection.

Finally, the resulting 2D triangles are sampled to the target raster image with a scanline algorithm. This algorithm iteratively determines the first and last pixel inside the triangle on each pixel row, and then fills all pixels in between.

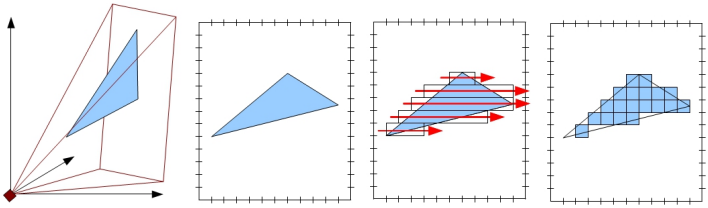


Figure 2.1: In this example rasterization process, a 3D triangle is first transformed into planar view space. Then each scanline of the triangle is processed and results in appropriate pixels in the final image.

For intuitive understanding, rasterization of a single triangle is presented in figure 2.1.

A vital problem that needs solving here is the rendering of multiple, overlapping objects: Consider triangle A that is positioned between the camera and another triangle B. This results in a depth conflict. The rasterizer must guarantee that B is correctly hidden behind A within the result image to meet viewer expectations.

Depending on the actual rasterizer setup, separate algorithms are used to solve any depth conflicts. In the original painter's algorithm, triangles are depth sorted and painted in back-to-front order so that closer objects overwrite any previously painted, farther ones. In the state-of-the-art z-buffer algorithm, the closest-most distance to the camera is stored for each pixel within an auxiliary, off-screen buffer. New objects are only written if their incoming depth indicates they are even closer to the camera, in which case the buffer is updated appropriately.

The depth conflict problem is symptomatic for the rasterization process. In particular, rasterization is a pure algorithmic construct rather than a physically correct approach. Thus additional workaround algorithms and hacks are required to fix up rasterization results to match with reality.

2.3 Raytracing

Raytracing follows a different, more physically oriented approach.

As motivation, consider light flow and the human vision system in the real world. In reality, light sources emit photons. Each photon travels through space until it hits some object. The photon then may be deflected into another direction, or it may be absorbed. At one point or another, the photon eventually reaches the human eye, where it triggers an appropriate sensory reaction. In its entirety, this is a probabilistic process. Each photon has its own starting direction and its own wavelength. Thus, photons hit different objects, are reflected differently, or absorbed by different objects. Overall, our vision system accumulates very many incoming photons for the final visual perception.

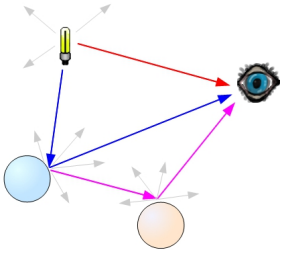


Figure 2.2: A photon tracer calculates the path that photons emitted from the light sources take through the scene. Any photons that hit the camera are accumulated, and determine the final pixel value.

These physical facts gave birth to the photon tracing algorithm. In particular, this algorithm follows discrete light rays within the 3D scene. Light rays randomly originate from light sources, and correspond to an entire bundle of photons. Light rays bounce around within the scene until they are completely absorbed. Whenever any light ray hits an object, it may also spawn additional light rays to simulate material behavior. As an example, a light ray that hits a glass surface spawns two secondary rays: A ray that penetrates the glass surface, and another ray that represents the reflected part of incoming light. The photon tracing algorithm is visualized in figure 2.2.

Disadvantages of this approach are obvious: Only those light rays that actually arrive at the virtual camera contribute to the final image. All other rays are calculated in vain. Additionally, since the approach is probabilistic, a metric has to be introduced that indicates when enough rays have accumulated within the image.

To counter these disadvantages, a slight modification to the photon tracing algorithm gives the actual raytracing algorithm: Instead of following light rays, a raytracer sends out a scanning ray through each pixel within the viewport. The collision of this ray with the first in-scene object is found to determine the pixel color. Again, secondary rays are introduced at the initial ray hit point to simulate mirroring, refraction, or translucency. Finally, secondary rays within a raytracer determine object shadowing as well: Any point is only lit by a light source if a ray from the point to the light source does not hit any other object. Compared to photon mapping, this yields a performance improvement at the expense of graphical quality. A schematic of raytracing is given in figure 2.3.

The above realization of raytracing under the use of secondary rays for shadow calculations, translucency and mirroring often is referred to as Whitted-style raytracing — named after its developer Turner Whitted.

In this context, one should also note that there are many different methods to adapt or combine various raytracing algorithms, and that there is no consistent terminology for these approaches.

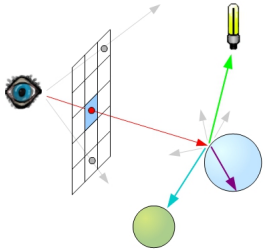


Figure 2.3: 3D raytracing sends a single ray through each pixel of the target image and into the scene. Secondary rays are spawned to determine occlusion of light sources, refraction effects, or translucency background.

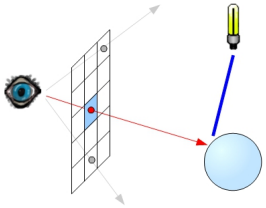


Figure 2.4: A basic raycaster traces a ray through each pixel of the target image into the scene. The first hit point of the ray with a scene object directly determines the pixel color.

As an example, secondary rays may be omitted entirely for performance optimization. The resulting algorithm came to be known as raycasting and played an important part in early raytracing applications. Raycasting is shown in figure 2.4.

Another approach — this time for increasing image quality — implies spawning additional, random light sampling rays from the initial ray hit point. These rays consider diffuse light interactions of nearby surfaces. For instance, light bouncing off a red ball creates a slightly red highlight on a very close wall. Such approaches often are termed diffuse raytracers or path-based raytracers.

Finally, Cook raytracing provides similar visual effects as diffuse raytracing, but avoids secondary sampling rays. Instead, multiple primary rays are generated per pixel of the input image and their results are combined to form the final pixel value. On scene intersection, any ray spawns at most a single additional ray with random behavior and direction. Thus, recursion is replaced by iteration at the cost of local coherence — each per-pixel ray potentially takes an entirely random path through the scene.

2.4 Comparison

As whenever there are two competing solution candidates for the same problem, the question arises which one is superior. Consequently, raytracing and rasterization are compared in the following, both in regards to performance and design elegance. Finally, an appropriate conclusion is drawn.

2.4.1 Performance

A typical measurement used in comparison of raytracer and rasterizer systems is the number of pixels touched throughout the rendering process [AW00].

Given a scene with t triangles, each of which maps to around n pixels on-screen, it is often stated that a traditional rasterizer requires $O(n * t)$ pixel operations. Due to depth complexity and overlapping objects, n may grow rather large, possibly much greater than the total number r of pixels in the target image.

In contrast, for the creation of an image with r pixels from t triangles, a raytracer is said to require around $O(r * \log t)$ intersection tests. The logarithm here indicates the use of a hierarchical optimization structure for the intersections within the raytracer kernel. The optimization structure allows to quickly discard triangle groups at once, thus not every scene triangle has to be tested against each ray.

At first, this performance comparison seems intuitive, and the naive conclusion is quickly drawn that raytracing asymptotically performs better than rasterization.

Sadly, there are flaws in this derivation. For instance, while raytracing uses a background optimization structure, no such structure is used in the rasterizer. Yet, there are many algorithms that allow for hierarchical occlusion culling of entire objects before these are even processed by the renderer. This on the one hand reduces the number of triangles drawn. On the other hand depth complexity is reduced, so that almost no on-screen pixel is overwritten. Thus, it is quite possible to achieve an average $O(r * \log t)$ complexity even for normal rasterizers.

Finally, asymptotic argumentation leaves out one practically relevant fact: Intersection tests, per pixel, are one or even more orders of magnitude slower than a respective pixel drawing operation within a scanline. Thus, in practice raytracers are known to be less efficient than traditional rasterizers. Obviously, the often cited asymptotic performance actually provides no usable measurement here.

2.4.2 Design elegance

A more important, though often overlooked argument in the comparison of raytracing and rasterization approaches is the actual rendering process.

Within current rasterizers, there are many different strategies in use depending on the rendering situation. For instance, portal culling and stencil shadows for indoor areas might be combined with an outdoor scenery based on a geo-mipmapped terrain and shadow mapping techniques. Environment mapping with a fake cubemap may give an impression of reflection on certain surfaces, and translucent objects must be depth-sorted before rasterization. Procedural surfaces and volumetric materials only

can be displayed after tesselation into triangles. All this puts a great strain on the rasterizer development, as the rasterization kernel needs many different code paths for these situations. Furthermore, clients such as shader writers or artists must be aware of the context any application asset is used in. For instance, a procedural material shader must output different values depending on its use either in normal rendering or in shadow mapping.

In contrast to the above, a raytracer provides an intuitive kernel, based on ray primitives, that is applied to every scene and to every use case. The same raycasting algorithm is used both for open and for closed scenes. Each shadow is created by casting rays, as are reflections, refractions and translucency. Refraction and translucency need not be faked, but are a direct result of the basic physical model. Objects within the scene only need to provide a single set of material properties, and then are compatible with the entire rendering kernel. Even procedural and volumetric objects intuitively are integrated into the respective raytracing kernel.

Thus, raytracers benefit greatly in elegance of the rendering kernel and in usability for client applications. At the same time, a raytracer needs less workarounds for a realistic rendering of physically motivated effects.

2.4.3 Conclusion

In a nutshell, neither of the rendering algorithms is better or worse than the other. Instead, each one has its respective advantages and disadvantages: While rasterization requires many workarounds to get realistic renderings, it also is quite fast even with naive implementations. For raytracing, believable renderings come naturally from the fundamental physical model, while efforts are necessary to achieve efficiency. A good wording for this conflict comes from David Luebke, engineer at NVIDIA: "Rasterization is fast, but needs cleverness to support complex visual effects. Raytracing supports complex visual effects, but needs cleverness to be fast" [FA09].

2.5 Historical development

As precedingly explained, rasterizers and raytracers are quite different rendering algorithms. However, it is only due to the popularity of rasterization based rendering that interactive raytracing is possible on consumer grade hardware these days. To gain further insight on this connection, it is important to examine the historical development of both algorithms — as well as the general development of graphics hardware.

2.5.1 Early years

Eventually, the first invention of the fundamental raytracing concept dates back to an age before computers and even electricity. In ancient Greece at around 400 B.C., philosopher Plato studied the human eye and came up with an explanation for human vision [IK07]: Plato was convinced that the eye sends out rays made from fire of reason. Once these hit any object, they merge with god-sent fire rays from the sun. This connection creates small particles on the object's surface, which in turn allow

an image of the object to enter the eye. There, the soul can reach out to the image and thus the object is perceived.

While the general understanding of light and the human vision apparatus have evolved since then, the idea of raytracing was taken on again at around 1960 for use in computer graphics. In 1963, the University of Maryland presented the first computer-generated, raycasted image on an oscilloscope screen. The first major publications on a formal raycasting algorithm were released five years later by Arthur Appel, Robert Goldstein and Roger Nagel [AA68] [RG71]. In 1979, Turner Whitted improved on the basic raycasting algorithm by introducing secondary rays for light occlusion, transparency, and mirror effects [TW79]. Thus, the fundamental raytracing algorithm had been developed. Eventually, the focus of later research shifted from basic algorithms to model improvements and performance optimization.

The first interactive raytracing system was introduced in 1987. Integrated into a CAD modeling system, this early raytracer distributed its workload over a network for parallel calculations. Thus, interactive framerates of several frames per second could be achieved for basic geometric shapes.

Scanline-based rasterization algorithms were developed in parallel to raytracing methods. The first publications on scanline rasterization appeared around 1970, such as in papers by Jack Bouknight [JB70] and Chris Wylie [CW67]. The original scanline implementations directly integrated the resolution of occlusion problems and depth conflicts into the pixel painting algorithm. However, with increasing triangle numbers, such calculations soon became infeasible, and alternative methods for depth sorting were required. As a potential tool for this task, Edwin Catmull invented the z-buffering algorithm in 1974 [EC74]. The fundamental combination of a scanline rasterization process and a z-buffer for depth handling quickly had been accepted, and even today remains the state-of-art interactive rendering algorithm.

2.5.2 The rendering equation

All previously named approaches to raytracing and rasterization did not care for physically realistic lighting. Instead, experimental solutions had been used to color the virtual scenes. This changed when James Kajiya developed a physical model of light and surface interaction in 1986 [JK68]. The mathematical implications of this model are represented by a compact integral equation [WPd]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(-\omega_i \cdot n) d\omega_i.$$

Due to its importance for all realistic rendering approaches this formula was termed the *rendering equation*.

Within the rendering equation, x is some point on the surface of an object, n denotes the surface normal at that point, and Ω is a hemisphere of direction vectors over x centered on n. The direction vectors for incoming and outgoing light energy respectively are expressed by ω_i and ω_o. Likewise, the functions L_i and L_o determine the amount of incident or outgoing light at x into direction ω_i or ω_o. The function L_e encapsulates any light directly emitted at x. Finally, f_r is a bidirectional reflectance

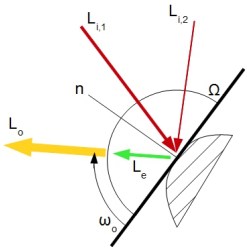


Figure 2.5: For a single observer direction ω_o, the outgoing light L_o consists of the light L_e emitted into that direction, and the sum of all incoming light L_i reflected into that direction.

distribution function, or BRDF for short. This function states how much incoming light from one direction ω_i is reflected into another direction ω_o.

Essentially, the rendering equation states that there always must be an equilibrium between the total outgoing light energy L_o and the sum of incoming light energy L_i and directly emitted light energy L_e at any point in space. Figure 2.5 presents an overview over these relations.

Further refinements exist to consider wavelength-specific, colored light or time-dynamic lights and objects.

Albeit the rendering equation does not consider all potential real-world lighting effects, it acts as the fundamental concept for most current lighting models. The rendering equation later on even is used to derive a lighting model for the raytracer implementation within this thesis.

2.5.3 Consumer adaption

In the late Eighties, basic research on both rendering approaches had been finished. Further publications improved on algorithmic details and on image realism, but did not introduce new concepts.

However, another development vitally influenced the further history of rendering: The market for home computers was steadily growing. In turn, computers became much more affordable, and provided ever increasing performance. Thus, 3D graphics were soon possible on consumer grade equipment.

Due to the low calculation performance and limited main memory of early consumer computers, initial 3D entertainment software centered around scanline-based rasterization. David Braben's space trading game Elite, developed in 1984 during his studies at the University of Cambridge, became an instant success. It internally used a scanline approach to represent a 3D rendition of the virtual space around the player's spaceship, including other spaceships, planets, and space stations. The algorithm's high performance allowed porting to a variety of then-popular consumer computers, such as the BBC Micro, Commodore 64, or Atari ST.

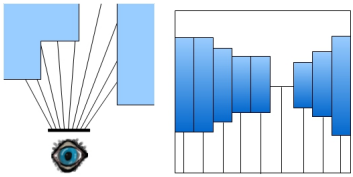


Figure 2.6: 2D raycasting sends but a single ray for each column of the resulting image. The ray is intersected with planar geometry. The distance to the intersection point sets the height and vertical offset of the respective column pixels.

Another important milestone in the development of rendering on consumer grade computers was a certain computer game released by ID software in 1992. From a technical point of view, the most noteworthy part of this game was the rendering algorithm. In particular, the 3D component was based on a simple raytracing algorithm instead of any prevalent rasterization approach. Many restrictions were enforced on the geometry of the virtual world to allow for but a single ray per column of pixels in the output image. The simplified calculations allowed the title to perform smoothly on the Intel 286 series CPUs that were current at that time — unlike many competing rasterizer based renderers that achieved the same graphics quality. An example scene suited for this basic form of raytracing is given in figure 2.6.

In the following years, graphics within entertainment software became a driving factor for sales in the home computer market. Faster and better graphics required new computers, which in turn allowed for a low-performance rendition of even more graphics enhancements. Thus, improved hardware was desired yet again, and the cycle continued.

2.5.4 Specialized graphics hardware

Apart from general performance improvements on consumer computers, the coming years saw the advent of two important developments:

On the one hand, interactive rasterizer-based approaches became more sophisticated, and outran interactive raytracing on home equipment both in performance and in image quality. In particular, rasterizer approaches were able to provide a full 3D environment at interactive framerates. In contrast, the performance-induced limitation on the number of rays and intersection geometry shapes in the competing interactive raytracers did not allow for this. An often-named milestone in this context is the 1996 release of another game by ID software that dropped the raytracing engine of its predecessors in favor of a pure triangle rasterizer.

On the other hand, 3D graphics became a vital selling point for specialized graphics hardware on the consumer market. Up to then, 3D graphics had been processed on the general-purpose CPU. The generated image was consequently sent on to the graphics board, which only displayed rendering results on the attached monitor. However,

hardware vendors such as S3 or 3DFX implemented additional hardware-side support for 3D rasterization in their struggle for market shares. At first, only few operations, such as texture mapping, were supported via vendor-specific programming APIs, and provided just little performance gain. Shortly thereafter, additional competing companies like ATI or NVIDIA entered the market. In this context, programming interfaces were standardized, and additional rendering functionality was moved from the CPU to the graphics processor. Soon, computer games and entertainment titles even required 3D support from the graphics hardware.

While the introduction of specialized 3D hardware improved performance, one limiting factor was the original restriction to a fixed function pipeline. In particular, only a certain set of default operations — rendering with per-vertex lighting, depth testing, alpha blending, and so on — was supported on initial 3D accelerator boards. This disallowed the use of more complicated techniques already possible on multipurpose CPUs. For example, per-pixel lighting could only be achieved by a workaround using static lightmaps. Due to competition for customers by means of improved graphics and special effects, hardware-based 3D graphics evolved on its own in this field. Starting with NVIDIA's register combiners in 1999, certain parts of the by then hard-coded pipeline were made customizable.

With the advent of pixel and vertex shaders in 2001, customization of the fixed-function pipeline was extended by user-programmable pipeline components. Albeit these components originally had been programmed in assembly, additional high-level languages were introduced later on to make development more intuitive.

2.5.5 General programming on graphics hardware

As a final step in development of graphics hardware, the once separate pixel and vertex shader units on a board were unified into a single processor type: A stream processor on a graphics board allows to process some kind of general input data stream to produce some general output data stream. In the context of graphics, these input and output streams correspond to the graphics data once moved through the fixed function pipeline. For instance, a stream processor might work on vertex input data to produce pixel output data.

The performance benefit for graphics is achieved by exploiting the independence of per-pixel calculations: In a traditional rasterizer, each pixel is treated independently from its neighbors. Thus, many pixels may be calculated in parallel without worrying about race conditions. A steam processor adapts to this by supporting the execution of many, lightweight parallel threads — ideally one thread per pixel in the output image.

Yet, stream processors are not limited to graphics data, but accept arbitrary input and output streams. As an example, a stream processor may be used to perform a calculation on a scientific data set. Unlike a CPU, a stream processor has certain restrictions that originate from the use for graphics processing. However, due to massively parallel processing of the input stream, a stream processor also performs much faster than a general-purpose CPU on suitable, data-intensive problems. This gave rise to the GPGPU programming paradigm: **G**eneral **P**rogramming on **G**raphics **P**rocessor **U**nits.

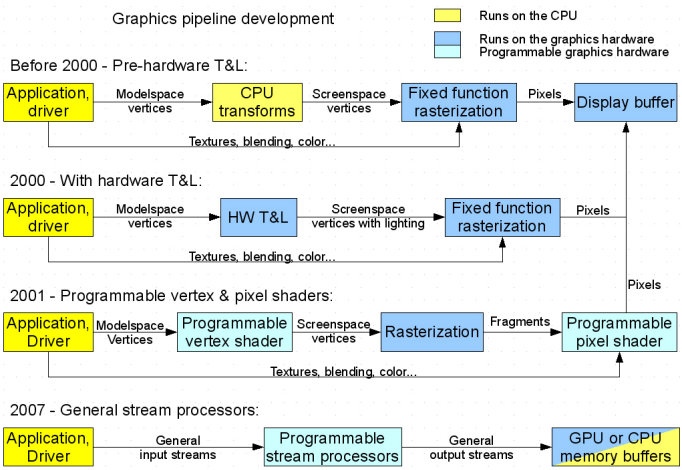


Figure 2.7: Development of the rasterization pipeline within graphics hardware over the past decades. The final, fully programmable pipeline is able to perform raytracing as well.

The shift from graphics-only processing to more general computations allowed the use of graphics boards for a variety of applications — from physics calculations to molecular folding. Coincidentally, this also allowed for the execution of raytracing algorithms on hardware that originally was developed for rasterization use only.

An overview over the described development of the programmable pipeline is presented in figure 2.7.

2.5.6 Use in modern applications

Today, the application fields of raytracing and conventional rasterization are clearly separated.

Raytracing is commonly used as an offline process, where quality matters more than interactivity. Application fields include computer generated stills and movies and product design as well as scientific data set visualization.

In contrast, rasterization mainly is used in interactive applications on consumer grade hardware, such as computer games, virtual reality software, or architectural walkthroughs. In these fields, the image need not be as exact or realistic as possible, but rather need only look believably correct and provide a set level of immersion.

However, with the ever steady increase in computing power, interactive raytracing slowly is becoming a viable rendering approach even for the latter use cases:

In 2007, the OpenRT project of the University of Saarland finished a programming API and specialized hardware for realtime raytracing. Albeit specialized hardware

was a novel approach to the raytracing problem, it also required a market shift and thus was accepted neither by consumers nor by commercial users.

The year thereafter, the Intel corporation demonstrated the first software-based, interactive raytracer integrated into a consumer-grade software title that runs on but a single PC system. In particular, the ID software title Enemy Territory: Quake Wars performed with around 20 frames per second on a high-performance, 16 core server rig.

Finally, the NVIDIA company augmented their product line for general computations on graphics hardware by the OptiX raytracing API. This API allows for out-of-core interactive raytracing on most current NVIDIA GPU boards, even consumer-grade entry level hardware. Eventually, OptiX also was used in the implementation work associated with this thesis.

The next chapter provides a more in-depth discussion of state-of-the-art interactive raytracing techniques from a scientific point of view.

3 State of the Art

This chapter serves three independent purposes: The outline of recent scientific research towards interactive raytracing, the selection of a middleware raytracing platform for integration into Simulator X, and the discussion of current state-of-art design for rendering components.

At first, current scientific approaches to interactive raytracing are reviewed. The focus within this review was to gain general insights on the feasibility and potential pitfalls of interactive raytracing within consumer-grade hardware.

Thereafter, middleware platforms for interactive raytracing are discussed, and the reasoning behind the choice for NVIDIA's OptiX API is elaborated.

Finally, the concepts behind select rendering engines — both traditional rasterizers and raytracers — are investigated. Particular attention in the design evaluation is put on the threading concept, as required for access from the multithreaded Simulator X kernel. Further points of interest are general client-side design, such as rendering process control, scene structures, and resource management.

The conclusions have been used as inspiration for the design of the actual raytracing component for Simulator X. A more in-depth investigation into rendering engines is available in [TW11a].

3.1 Interactive raytracing

In the following, recent academic results on interactive raytracing are presented. At first, a general publication on the overall state of interactive raytracing is reviewed. Thereafter, select algorithms specialized for various areas of interactive raytracing are discussed in greater detail. The review ends with a universal resume on the current state of affairs with interactive raytracing.

3.1.1 General review

One of the most recent and thorough surveys on interactive raytracing is provided in [IW07b]. In the following, a short review on this publication is given.

As motivation, the work states that it has become possible to perform basic raytracing in real-time due to algorithmic advances and increased computational power. However, with the advent of real-time raytracing, a new problem arises: While real-time raytracing by concept allows for dynamic scenes, the traditional optimization hierarchies used over the past decades have not been designed for moving contents. In particular, algorithms were optimized in regards to optimal raytracing performance, not in regards to optimal hierarchy build time. In turn the updates to scene structures required by dynamic objects often limit performance of the entire raytracing process.

Consequently, the work names recent interactive algorithms that propose improvements on raytracing and rebuild time.

Algorithms are classified depending on a series of characteristics:

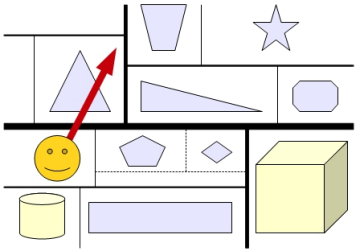


Figure 3.1: KD-trees provide a good fit even for irregularly distributed objects and primitives. Tree depth can adapt to provide resolution where needed. If any dynamic object crosses node planes (red arrow), the tree cannot easily be updated without any complex node restructuring. Thus, a complete rebuild is often faster.

- Type of animation: Static scenes, translations only, skeletal animation or random triangle movement.
- Ray coherency: Single ray traversal, ray packets, or volumetric rays.
- Data structure: Spatial structures or hierarchical structures.
- Bounding alignment: Axis-aligned or arbitrary positioning.
- Tesselation strategy: Adaptive or uniform data structure tesselation.
- Rebuild balance: Fast rebuild or optimal hierarchy.
- System architecture: Rasterizer-oriented or exposed scene graph.
- Rebuild variant: Update-based or full rebuild.

Thereafter, the work investigates few of the named algorithms in detail.

At first, construction algorithms for kd-trees are reviewed.

Kd-trees are the most popular structure for offline raytracing, as these in general provide the best ray traversal performance [VH00]. Yet, updates on kd-trees are known to be slow in practice. In turn, most interactive approaches try to rebuild kd-trees for dynamic scene objects from scratch on each frame.

Figure 3.1 illustrates both kd-tree characteristics.

One algorithm suited for kd-tree rebuild is the complex **S**urface **A**rea **H**euristic (or SAH for short).

The recursive SAH building process for a kd-tree recursively starts at a root node corresponding to the entire virtual space. To find a good splitting position for the introduction of children nodes, SAH considers the probability of a surface-ray intersection for all surfaces within either of two potential children nodes. For an uniform

ray distribution, this probability is proportional to the surface area of contained triangles. Intersection probabilities for two children nodes are weighted with heuristic costs for a ray traversal of the entire respective child. The outcomes still depend on the actual split position, but in sum represent the cost of a tree split at that position. Consequently, an optimal splitting point is determined by minimizing the sum of weighted intersection probabilities. This is possible with good performance as any such split point is guaranteed to coincide with a primitive starting or ending point. Thus only few points must to be considered. Finally, the optimal split is only performed if another heuristic indicates that the cost for both children of the split does not exceed the cost of the parent node.

Improvements on the SAH algorithm are named as well. For instance, one approach replaces the surface area as a measure for probability with distribution sampling at coarse tree levels. Other approaches perform the rebuild in parallel, and thus achieve even further performance benefits. With such adaptions in place, SAH implementations can handle the per-frame rebuild of smaller scenes with around 100.000 primitives at interactive framerates.

As an alternative to full SAH rebuilds on kd-trees, the work cites fuzzy kd-tree calculations. Unlike the SAH strategy, fuzzy kd-trees do not allow for arbitrary object animations. Instead, a set of predefined animations is used in an offline motion decomposition process to calculate a universally fitting kd-tree representation. Within the offline process, coherent parts of the animated object are recognized. For each such part, the corresponding animation is separated into an arbitrary affine transformation and a non-affine component. The former is used for the run-time transformation of rays into a local space for each animation frame and object component. The latter enlarges the boundings of each primitive for use in fuzzy kd-tree generation. Consequently, affinely-transformed rays are tested against an optimization structure that considers but non-affine movement of primitives. Thus, the resulting kd-tree can be used in an interactive raytracer without any online rebuild.

While predefined animations at first seem like a harsh restriction, there are adaptions that significantly loosen these. For instance, a fuzzy kd-tree can be calculated from a skeletal mesh and then allows for arbitrary online bone animation without rebuild.

As a rather new contender to kd-trees, the survey names **B**ounding **V**olume **H**ierarchies — BVH for short. These had originally been dismissed for offline raytracing due to their low intersection testing performance. However, BVHs are intuitively updated on dynamic changes, thus currently reinvestigated for their use in interactive raytracing.

Within bounding volume updates, leaf nodes of the bounding volume tree that contain modified content are found. Any appropriate leaf nodes are adapted to match their new primitive extensions. Bounding volumes are then updated from bottom to top within the tree so that all parent nodes tightly fit around respective children nodes. An early-out opportunity for an entire branch update is given once the bounding volume of any higher-level node did not change due to lower-level modifications. Consequently, most cases do not require a complete tree traversal.

Yet, even BVHs need a quick strategy for complete rebuilding. On the one hand, any update of course requires an initial, optimal tree to work with. On the other hand,

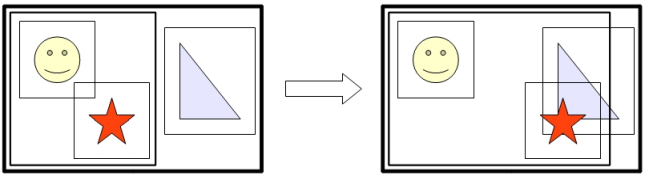


Figure 3.2: Even initially, ray traversal in bounding volume hierarchies may be slow due to overlapping bounding boxes. These require additional node traversals for rays within certain regions. An update from the left-hand side image to the right-hand side image is intuitively performed by moving object boundings and refitting parent nodes. However, there now is even more overlapping space. Thus, tree structure and traversal performance further degrade.

the ray intersection performance of an updated BVH deteriorates over animation time. Namely, while volumes are adapted, the structure of the tree is maintained. However, the structure might more and more diverge from the optimum hierarchy for the modified primitives.

Complete rebuilding strategies for bounding hierarchies match those for kd-trees with but little adoptions. Most notably, the SAH approach applies to BVHs as well.

Figure 3.2 presents an example update on a bounding volume hierarchy.

As the last major category of algorithms for interactive raytracing, the general review publication names grid-based approaches. Unlike previously investigated acceleration structures, grids are non-adaptive, spatial subdivison schemes. Consequently, grids only are suited for scenes with an even primitive density. Given any such scene, grids outperform both kd-trees and bounding based hierarchies: Grids require about the same time for complete rebuilds as updates for bounding volumes do, and incremental grid updates are even faster. At the same time, grids accelerate raytracing of evenly distributed primitives by about the same amount as kd-trees do for arbitrary geometry.

Advantages and disadvantages of grid acceleration structures are illustrated in figure 3.3.

Similar to both of the other approaches, grid-based algorithms allow for certain adoptions and improvements. For instance, grid rebuild can be performed by a rasterizer-like parallel kernel, and grid traversal is easily improved by Bresenham-type line drawing algorithms. Finally, the restriction on evenly distributed geometry can be lifted at the cost of update and raytracing performance by the introduction of multi-level adaptive grid hierarchies.

After recap of all three acceleration structures — kd-trees, BVHs, and grids — the review comes to the conclusion that there is not a sole, optimal algorithm for interactive raytracing. Instead, future raytracers will have to employ a variety of strategies, depending on the exact type of scene contents: kd-trees for non-animated, detailed

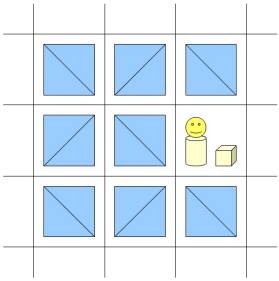


Figure 3.3: While the above grid structure holds a constant amount of evenly distributed primitives over most grid cells, it cannot adapt to the high-detail contents of one of the cells. Traversal performance thus is poor for this cell. On side of benefits, rebuild and update are easily performed by reinserting primitives into cells via fast lookup strategies. Actual traversal can be accelerated by rasterizer-like methods — such as Bresenham's line drawing algorithm.

geometry, grids for regularly spaced primitives, and bounding volume hierarchies for animated geometry and the higher-level object tree.

In the following, this thesis elaborates certain noteworthy subordinate algorithms referenced by the general review work in greater detail.

3.1.2 KD-trees

The kd-tree approach preferred by traditional offline raytracing is translated into an interactive, GPU-based variant in [TF05].

Due to graphics hardware restrictions, the originally recursive tree traversal had to be replaced by a stack-less iterative variant. This modification is possible since within the standard algorithm all recursive calls appear as tail recursions.

The suggested adoption of the algorithm iteratively tests a ray against the kd-tree in front-to-back order. Intersection tests are performed once the ray arrives at the first previously untested leaf node. If the node is not hit, the ray starting point is adapted to the point where the ray leaves the tested node, and traversal starts at the tree root again. For a scene with n leaf nodes and a balanced kd-tree representation, this variant implies a mean complexity of $O(n)$ with an upper bound of $O(n \log n)$ for rays that hit distant objects.

The worst case complexity is reduced by a further adoption to the kd-tree data structure. If parent node pointers are kept inside each node, restart from the root node after each miss can be replaced by tree ascend. The modified algorithm then achieves an upper bound of $O(\log n)$, which matches the worst-case behavior of the unmodified, traditional kd-tree traversal algorithm.

The GPU implementation associated with the publication is based on the little-known general GPU programming front-end Brook [BR10] under development by Stanford

University. Similar to more popular GPU programming languages, Brook employs the concept of device-side kernels: small GPU-executed program fragments that perform simple calculations on large-scale data with massive parallelism over all data element.

A series of Brook kernels is derived within the work, one for each step within the modified kd-tree traversal algorithm. For instance, there is an intersection kernel that iteratively calculates ray-primitive intersections within a single leaf node for many rays in parallel. To allow for per-ray parallelism, the data elements that the respective kernels work on are representations of each ray and the associated pixel outcome. For further performance optimization, branching instructions are replaced by masking operations.

Building of kd-trees is performed in an offline process on the CPU, dynamic scenes are not supported. With optimal tree generation, the investigated approach achieves almost interactive framerates on legacy graphics hardware with but small rendering resolutions. The main performance bottleneck is identified as memory bandwidth in combination with excess data-dependent and incoherent memory fetches. Compared to a CPU-only realization, the work concludes that raytracing is not fit for implementation on graphics boards yet.

The performance results of [TF05] are challenged in [DH07] by the development of a new kd-tree traversal algorithm that adapts to specific hardware features of more recent graphics hardware.

The reported performance gain is achieved by the move from multiple, CPU-controlled raytracing kernels to a single GPU kernel invocation. This required the by-then novel availability of branching and looping instructions on the GPU processor. Further improvements include the realization of ray packets and the implementation of a small fixed-size stack that is used instead of ascend or restart strategies for shallow trees. Finally, the publication proposes the application of normal rasterization strategies in place of primary camera rays. Raytracing is only applied to secondary rays for shadow rendering and refraction simulation.

A performance evaluation concludes the presentation of the improved GPU-side kd-tree raytracer. Compared to the original GPU implementation. framerate has been increased by an order of magnitude to at most 20 frames per second for scenes with up to 300.000 triangles. Thus, kd-trees are suited for static scenes within interactive applications.

3.1.3 Bounding volume hierarchies

In [IW06b], detailed build, traversal, and update strategies for BVHs in deformable scenes are elaborated. Deformable scenes — unlike completely dynamic scenes — do not allow for arbitrary insertion or deletion of primitives. Instead, only the shape of existing primitives is modified. This is sufficient for many applications, such as cloth simulation or skeletal animation.

As its starting point, the publication describes a SAH method for building an axis-aligned bounding box tree: The optimal split point for any node within the bounding volume tree is derived by minimizing the split-induced cost on later traversal over potential children sets S_1 and S_2.

For axis-aligned bounding boxes, the respective cost heuristic for splitting a parent node S into children S_1 and S_2 is cited as

$$T = 2T_{\text{AABB}} + \frac{A(S_1)}{A(S)} N(S_1) T_{\text{tri}} + \frac{A(S_2)}{A(S)} N(S_2) T_{\text{tri}},$$

where A is a surface area heuristic and N is the number of primitives in the respective node. T_{tri} and T_{AABB} express the relative estimated costs of per-triangle intersection detection and coarse ray-box tests. In more illustrative terms, T is a weighted average between the cost of the additional box test, and the cost from testing nodes in each of the potential children.

To avoid unnecessary splitting, the cost-minimal split is only performed if T does not exceed a cost estimate associated with the unsplit parent node.

The work names optimal selection of splitting sets S_1 and S_2 as a major difficulty in tree generation for BVHs. Within SAH algorithms on kd-trees, there only are $O(n)$ potential split candidates, where n is the number of primitives in the later parent node. In contrast, the number of potential split candidates within a bounding volume node is bounded by $O(2^n)$. Therefore, the publication suggests the selection of but few of these. While certain selection strategies have been discussed — such as evenly aligned axis splits or centroid based approaches — the conclusion is drawn that the actual candidate selection method does not significantly influence quality of the later tree.

After introduction of the SAH-based tree building algorithm, the publication elaborates the optimization of ray traversal within a BVH. Since the axis-aligned box intersection tests are rather expensive when compared to kd-tree plane intersection, the paper suggests several strategies to minimize the implied costs.

A ray packet scheme is explained where bundles of four rays pass through the tree at once. Low-level hardware SIMD instructions are applied to parallelize any necessary per-ray tests. Early hit and early miss optimization are performed for the entire ray group at once. Within the former, traversal descends into a node once the first ray hits the bounding box. In the latter, a conservative bounding of the ray group is tested against the bounding box, and recursion stops if the boundings do not overlap. Finally, an approximate approach to front-to-back ordered traversal within a BVH is detailed. Ordered descend allows for skipping certain expensive calculations on more distant tree nodes.

Thereafter, the above build and traversal strategies are applied to deformable, dynamic scenes by the integration of an intuitive update algorithm. A post-order hierarchy traversal is suggested where each node first updates both its children and then rebuilds its own bounding box appropriately.

While such calculations are relatively fast for small scenes, the approach names two potential problems: On the one hand, the update has $O(n)$ complexity with scenes of n primitives, and consequently is not suited for large scenes. On the other hand, the BVH can degrade for certain types of animation — namely, when the initial tree structure does not represent the new scene contents anymore.

The work concludes with a performance evaluation of the implemented strategies. Tree build times for scenes with up to 200.000 triangles are reported just below five

seconds. The resulting trees enable interactive, CPU-based raytracing of animated scenes with around ten frames per second. The update of tree structures contributes ten milliseconds, corresponding to around ten percent, of the total 100 milliseconds calculation time per frame.

Further publications by the same author enhance the above basic strategies: [IW07a] gives a strategy for complete, per-frame reconstruction of SAH hierarchies in the context of bounding volume representations. The split point heuristic is adapted to provide better performance at the cost of tree quality and later raytracing speed. [IW08] improves on the preceding algorithms by asynchronous processing within multiple parallel threads. The final SAH reconstruction strategy is applied to the Intel MIC (**M**any **I**ntegrated **C**ore architecture) in [IW10].

Albeit the original publication proposes the selection of a good keyframe from any given animation to build the initial bounding tree, there are more elaborate approaches. For instance, [DM06b] suggests a heuristic-triggered rebuild of subtrees within the hierarchy. The respective rebuild heuristic compares the current ratio of parent surface area to children surface area with the original ratio stored at tree build time. Once the difference between these exceeds a certain threshold, the parent node contains much empty space. Thus, it does not accurately represent its children anymore, and a full rebuild is enforced.

[MS09] elaborates a BVH that incorporates both spatial and surface area heuristics within tree building. This approach has been termed the SBVH algorithm — shorthand for **S**plit **B**ounding **V**olume **H**ierarchy. In particular, the work considers two split candidates for each bounding volume: On the one hand an optimal splitting candidate is determined as by the popular SAH strategy described above. On the other hand, a clip-based binning technique is applied to derive a spatially motivated split candidate.

The binning technique separates the bounding box of the parent node into regular volumes (i.e. bins) by a set number of equidistant planes. Thereafter, the bounding box of each primitive within the node is iteratively clipped into each of the bins. Another area-based heuristic which considers all clipped bounding boxes in a given bin is applied to determine the most cost-effective spatial split over all bin-delimiting planes. Finally, axis aligned bounding boxes over all clipped primitive fragments on either side of the optimal splitting plane are calculated as dimensions for potential children nodes.

Both the SAH and the bin splitting heuristic have been designed to be compatible. This allows to choose the candidate with smaller costs for the actual split. As previously detailed, the final split is only performed if the new children costs do not improve over the cost of the parent node.

On evaluation of the proposed approach, the work states a framerate improvement of an average 30 percent over pure SAH implementations, based on raytracing of large-scale scenes with millions of triangles. No absolute figures are given.

An especially noteworthy performance increment has been observed on a testing scene with greatly divergent level of detail: A densely tesselated, million-triangle statue is placed within an environment made from but few, yet very regular triangles. In this use case, pure SAH heuristics merge the environment and the statue within the top

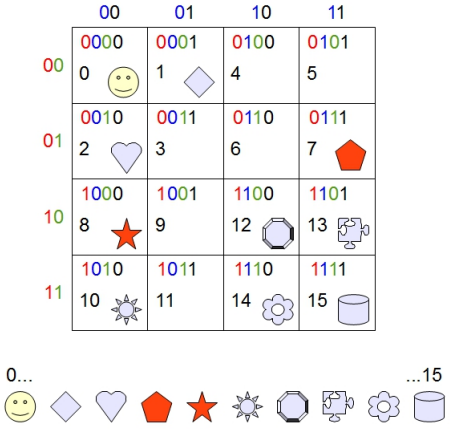


Figure 3.4: Morton codes are derived from bit-wise interleaving of integer-rounded primitive positions. Sorting primitives by their respective morton codes places neighboring primitives next to each other inside a linear array. Large gaps within this sorting appear only at higher-level bit flips — such as for the red primitives in the above example. After sorting, it is sufficient to build a tree over the linear array to bundle close-by primitives.

levels of the entire tree, resulting in expensive ray traversal. In contrast, the combined spatial and SAH-based approach clearly separates both geometries on the first tree levels.

3.1.4 GPU-based bounding volume rebuilds

[CL09] derives a fast, GPU-based rebuilding strategy for BVHs, specifically aimed at interactive raytracing of dynamic scenes. Unlike preceding approaches, animation within dynamic scenes is not handled by fast updates, but by a full rebuild of the hierarchy within each frame. This avoids degradation of the optimization structure over the course of animation.

The rebuilding strategy described by the publication relies on the concept of Morton codes. Given a point in 3D space, its Morton code is quickly constructed by bit-wise interleaving of its coordinate components into a single integer representation. Most notably, close-by points result in close-by Morton codes with but few exceptions.

Morton codes are applied to translate the problem of tree building to the problem of sorting a linear array of integers. In particular, each primitive is approximated by a single, integral point in 3D space, and the respective Morton code of this point is inserted into a linear array alongside the primitive ID. The array of Morton codes is sorted, consequently primitives with close-by positions end up within neighboring array elements. This process is illustrated within figure 3.4.

A tree structure over the entire sorted array is then intuitively built by recursively dividing the array at each bit of the Morton code, up to primitive level. Thereafter, bounding boxes are calculated in a bottom-up approach. Bounding boxes in general provide a good fit, with the exception of few large boxes caused by 3D discontinuities in between neighboring Morton codes.

Due to the linearization of tree construction, all similar strategies are categorized as LBVH algorithms (**L**inear **B**ounding **V**olume **H**ierarchy).

The parallel implementation of Morton code-based tree generation on the GPU is straightforward, and realized by but few device-side kernel calls. For all involved sorting operations, the publication suggests a fast, parallel radix sort technique.

While fast rebuild times are observed for pure LBVHs, the work also names slow traversal time as the most noteworthy disadvantage. Because Morton codes split primitives based on their volumetric median, scenes with spatially varying detail are handled especially poor. To counter these deficiencies, the publication proposes the combination of both traditional SAH metrics and Morton codes. Morton codes are used for higher levels of the tree, whereas surface area heuristics are applied on lower levels for but few primitives.

Performance evaluation yields interactive rebuild and raytracing framerates both for standalone Morton code, as well as for the hybrid Morton-SAH approach. Given testing scenes between 50.000 and 1.5 million triangles, the former peaks at 10 frames per second, while the later averages to 20 frames per second.

[JP10] improves on the hybrid Morton-SAH algorithm elaborated by the last publication. Both the GPU side kernel and the structure of the algorithm are optimized. Most notably, only the first few bits of the Morton code representation are considered for a first sorting pass. Once the high-level sorting pass has completed, a specialized odd-even sorting kernel is applied to sort amongst the remaining Morton bits. All modifications purportedly increase performance by a factor of four in comparison to the original approach.

3.1.5 Memory coherence algorithms

Effects of higher-level memory coherency for interactive raytracing are discussed in [DM06a]. As motivation, the work states that CPU performance alone is not sufficient for interactive raytracing of large-scale scenes. Instead, memory bandwidth and disk access performance are the limiting factors on current hardware. For instance, when a mesh with billions of triangles is raytraced, each pixel maps to many triangles at once. Thus, rays for neighboring pixels hit distant triangles which map to distant memory locations. In turn, the hardware memory cache is incoherently accessed, and slow caching operations regularly are required. Therefore, it is imperative to find an optimal management strategy for ray-scene traversal that ensures coherent memory access for intersection detection.

To achieve coherent memory access, the work proposes the extension of the standard kd-tree raytracing acceleration structure with implicit level-of-detail representations. At each kd-tree node, an additional plane is embedded that approximates the entire contained geometry. If a certain node-specific metric in regards to the on-screen

pixel error is met, incoming rays within tree traversal are then intersected with the per-node plane instead of subordinate nodes and geometry. Consequently, a single node potentially maps to multiple pixels and rays, and coherent memory access is established.

The actual error metric considers two weighted criteria: The number of on-screen pixels that a respective kd-tree node maps to, and the difference between the approximative plane and the primitives' surface. The metric evaluation itself is not explained in depth.

Offline construction of approximative planes is investigated as well. In particular, principal component analysis is used to derive a matching plane for subordinate primitives and node planes. These computations are applied hierarchically, thus each original primitive is accessed exactly once. A special layout scheme is used for resulting nodes to further improve cache coherency of the resulting data structure.

Finally, the work evaluates the presented solution: A performance improvement of at most two orders of magnitude is gained at the cost of a preprocessing step for building plane approximations. In respect to the performance benefits, the runtime penalty of metric evaluations — which contributes 30 percent of the total raytracing time — is considered acceptable. As a conclusion, two noteworthy disadvantages are named: On the one hand, there currently is no strategy to fix any visual artifacts caused by neighboring pixels from different level-of-detail representations. On the other hand, the approach does not support deformable geometry.

[GS06a] provides a different view on memory coherency. Instead of caching effects introduced by large-scale scenes, the work examines the caching effects introduced by ray scattering. Given a low-detail scene with but few triangles, the effects of cache misses are not observable on primary rays. Neighboring rays take similar paths through the optimization structure, and often hit the same or close-by triangles. Yet, even for coherent primary rays, secondary rays exhibit divergent behavior. For instance, refraction rays that are spawned by neighboring camera rays usually trace into vastly different directions.

Similar to [DM06a], the work suggests a level-of-detail approach to deal with the problems of incoherent memory access. Instead of direct integration into the optimization structure, level-of-detail representations are applied to entire objects at a time. Visual artifacts caused by incontinuities in the object surface are thus avoided. Artifacts are further reduced by run-time morphing between detail levels. Currently, only procedural, patch-type surfaces are supported by the corresponding implementation. The support of arbitrary, locally varying level-of-detail surfaces — such as required for large objects or scenes — and the integration of arbitrary objects are named as the major focus of future research.

3.1.6 BSP-based optimization structure

Within [TI08], BSP trees are applied to interactive raytracing as a novel optimization structure.

Unlike axis-aligned kd-trees, binary space partitioning trees allow for arbitrary placement of separating nodes. In practice, this allows for a very tight fit of the bounding

representation. Yet, BSP trees were commonly believed to be complicated to build, slow to traverse, and numerically instable. Consequently, these had been ignored in the development of raytracers for the past decades.

In contrast, the work shows that BSP trees are competitive with traditional kd-trees even in interactive raytracing. Two aspects of BSP usage for raytracing are discussed: Offline tree building and fast ray traversal.

The work achieves offline tree building by an approach similar to the SAH approach of kd-trees. For each node, costs for children node traversal are estimated as implied by certain splitting planes. An optimal splitting plane is found by minimization of children traversal cost. The associated split is only performed if costs for both children do not exceed estimated traversal costs for the current node itself.

All costs in the above process are based on weighted surface areas. Unlike the box-bounded kd-tree nodes within the SAH algorithm, BSP nodes form convex polytopes. Closed polytopes, instead of open half-spaces, are enforced by placing a bounding box around the entire virtual scene. The surface area of each polytope face is used instead of corresponding calculations in the SAH metric. Surface areas are iteratively refined during the BSP build process to avoid complete recalculations on an entire complex polytope.

In total, the expected splitting cost at any given split point is expressed by the metric

$$C_p = \frac{SA(v_l)}{SA(v_p)}\, c_l\, C_i \;+\; \frac{SA(v_r)}{SA(v_p)}\, c_r\, C_i \;+\; C_t.$$

Indices l and r are associated with the potential left-hand side and right-hand side children nodes. The index p indicates the parent node. The SA function provides the polytope-based surface area estimate. The variable c represents the count of primitives on either side of the split. Constants C_i and C_t encapsulate the relative costs of per-primitive intersection and node traversal.

Selection of splitting plane candidates for which to evaluate the above equation is more complex for BSP trees than for SAH-based kd-trees. Within kd-trees, each triangle primitive defines but six potential candidate for splitting a node. In contrast, the number of split candidates for a node within a BSP tree is bounded by $O(n^3)$, where n is the number of primitives contained in that node. To ensure practicability the work suggests a strategy to select but few of these: Each triangle plane itself is taken into account, as are the six candidates shared with kd-trees. Finally, three planes constructed from the triangle normal and either of the triangle's edges are considered.

The work implements the above offline build process alongside an auxiliary bounding value hierarchy for primitive counting in sub-quadratic complexity. The $O(n \log n)$ build time of SAH-based kd-tree rebuilds is not achieved, as arbitrary split planes do not allow for presorted triangles. Furthermore, improved build times have not been the focus of the work. Consequently, resulting BSP tree building times are slower than kd-tree building on the same scene by orders of magnitude.

The actual benefits are evident from the later interactive raytracing framerates. In particular, the work cites framerate improvements of at most an order of magnitude when compared to traditional kd-trees: On the one hand, the generated BSP

trees are better balanced than kd-trees, and less nodes need to be considered in any ray traversal. On the other hand, explicit ray-primitive intersection tests may be optimized with byproducts already generated during BSP traversal.

Finally, visual artifacts that arise due to numerical instabilities in between a triangle and its respective BSP node are limited by an epsilon factor that is used both at BSP build time and in later raytracing.

Thus the work relativized two of three common, negative claims on BSP trees.

3.1.7 Multi-frustum approach for soft shadows

[CB09] presents a fast, frustum-based algorithm for soft shadow calculation within an interactive raytracer. The driving factor for this work was the improvement of visual quality in competition to existing rasterization approaches. Namely, interactive raytracers usually are limited to single shadow rays and thus to hard-edged shadows. In contrast, rasterizers use various shadow mapping techniques to quickly approximate soft shadows.

Preceding raytracing approaches simulated soft shadows by casting multiple, costly rays per point and light. Packet-based techniques already improved on this by tracing entire ray packets at a time instead of single rays. The investigated work provides a further extension to packet-based approaches. In particular, packets — termed frusta in context of this work — are grouped into a higher-level multi-frustum beam.

Coarse traversal for the multi-frustum beam is performed on a BVH composed of axis-aligned bounding boxes. Culling is performed for the entire beam at once on each traversed node: Either the entire beam does not hit the node bounding, or all subordinate frusta and all contained rays have to be tested against subordinate nodes.

On primitive level, two different kind of optimizations are applied. On the one hand, certain tests on primitives — such as back-face culling or edge tests — are performed in parallel on all frusta. On the other hand, parallel, low-level hardware instructions within intersection tests are utilized for all rays.

A single flag is carried alongside each subordinate frustum to indicate a previous primitive intersection or any failed intersection tests. If raised, the flag disables further intersection calculations for the respective frustum. This allows for compatibility with certain low-level, massively parallel hardware designs. Within the work, this is an adaption to Intel's Larrabee architecture, but the same concept also applies to GPU characteristics — for instance when compared with CUDA / OptiX thread wraps in chapter 5.

On evaluation of the elaborated multi-frustum raytracing technique, the work states a performance benefit of around half an order of magnitude when compared to traditional ray packing approaches. This gain is attributed to a reduction of bounding box intersection tests to around five percent of those required by competing ray packing strategies.

The work concludes that multi-frustum tracing is not only suited for soft shadows, but any coherent raytracing task — including primary camera rays.

3.1.8 Conclusion

From scientific review, it is evident that interactive raytracing is becoming more and more feasible. Currently there is no one sole accepted interactive raytracing approach — and probably there never will be one. Yet research already has developed a toolkit of competing algorithms well suited for interactive raytracing of dynamic scenes.

However, even with a middleware platform that provides an appropriate algorithm suite, potential application pitfalls remain: An appropriate optimization structure must be chosen for various parts of the scene, restrictions in regards to dynamic scenes must be adhered to, and coherency of emitted rays must be observed. All of these aspects are considered in the later raytracer implementation within this thesis.

3.2 Middleware alternatives

The growing numbers of scientific publications indicate a general interest in interactive raytracing. NVIDIA is not the only GPU manufacturer out there, and CPUs are quite capable of raytracing on their own. Thus, there should be plenty of competing middleware platforms for interactive raytracing. Sadly, however, there currently are no alternatives to NVIDIA's OptiX API readily available.

One potential alternative in form of a plug-in hardware card was developed by the SaarCor project at the University of Saarland. As of this writing, development seems to have halted, and the most recent publication of the research group on the OpenRT project dates back to 2007 [WPe] .

These days, the computer graphics group at the University of Saarland still works on interactive raytracing, but the concept of specific hardware has been replaced by a GPU-oriented approach. A new, modular raytracing framework named GPURT is currently under development. While official feature listings appear promising — purported features match those of OptiX — there is no version available to the public yet.

ATI and AMD, traditional market contenders of NVIDIA, do currently not offer any similar API to OptiX. Their only contribution to interactive GPU raytracing are several feasibility studies and videos that work on OpenCl-based, specialized raytracing kernels.

IBM designed an interactive raytracing component by the name of alphaWorks. Yet, this component is meant more for product visualization than for general raytracing purposes. Most noteworthy, alphaWorks is restricted in use to proprietary IBM cell processors — such as in the Playstation 3 or IBM's Blade-type server systems. Furthermore, there is but a standalone Linux scene viewer executable available to the public. No source code is included, neither are animation features. Thus, integration into the Simulator X environment is not possible.

Finally, Intel corporation, more a CPU than a GPU developer, integrated interactive raytracing components into few ID software computer games. This aimed at the demonstration of various interactive raytracing techniques. For instance one implementation realized on-core interactive raytracing on a modern, many-core server

CPU [DP09]. Another implementation achieved the same results by distributed raytracing within a server cloud on a thin notebook client [DP10]. The corresponding implementations are not available to the public, though.

As there are no middleware platform alternatives available, the only option for interactive raytracing on graphics hardware had been the development of an alternative raytracing API from scratch. Yet, the common ground for GPU computing over all GPU vendors is formed by the recently specified and still buggy [MR11] OpenCl GPU programming language. The OpenCl approach consequently has been dismissed for this thesis. A native, NVIDIA-specific CUDA implementation has been considered as well, but was rejected for two reasons: For one, under consideration of the sheer amount of recent scientific publications on interactive raytracing, this was deemed more a research than an implementation task. Second, NVIDIA already provides a proprietary, optimized raytracing kernel within the OptiX API, and there is no need to reinvent the wheel.

In the overall context, the reason behind the focus on OptiX as the only solution within this thesis becomes evident. However, even with OptiX, all is not well. Most importantly, as OptiX is a proprietary solution, the resulting applications are fixed on hardware from a single manufacturer. Furthermore, OptiX strives to be a general-use raytracer instead of a raytracing-based renderer, which in turn leads to a rather clumsy interface when used for rendering only. To counter these problems, an appropriate suggestion for upcoming implementation and research work in regards to API alternatives is found in chapter 10.

3.3 Renderer architecture

Even though OptiX has been chosen as the implementing middleware API for the raytracer component within Simulator X, it still needs several wrapping layers. Most importantly, OptiX must be encapsulated in a standardized, OptiX-independent interface front-end. This allows for the integration of multithreading support on the one hand, and for later integration of alternative rendering platforms on the other hand. Supported rendering platforms could both include rasterization and raytracing based approaches.

In other words, a new, general renderer interface has to be designed. In this section, several existing rendering systems are investigated as inspiration and motivation for the new interface — both in terms of general architecture, as well as in terms of multithreaded behavior.

3.3.1 Blender

Perhaps the most relevant case study for the implementation of a new rendering kernel with both raytracer and rasterization support are conventional modeling and rendering packages. In particular, these packages already provide a rasterizer for the interactive WYSIWYG editor, and a raytracer for generation of high-quality final images. Albeit the raytracer in general is not interactive, both rendering components are fueled by the same background data structures. Thus there must be some sort of concept for data sharing among them. Finally, most rendering packages allow for

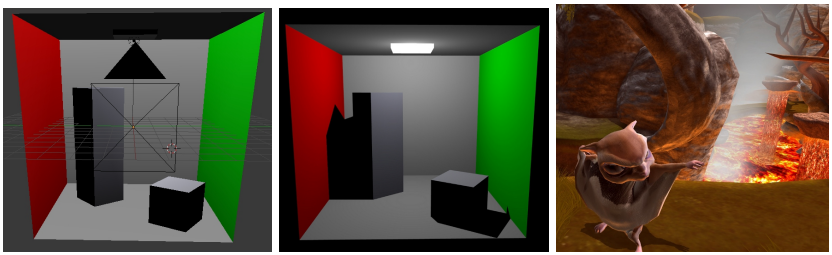


Figure 3.5: The Blender-internal rasterizer produces the left-hand side WYSIWYG preview of the middle raytraced image. The right-hand image [YF11] was created by the interactive Blender game engine.

multithreaded rendering, and consequently need to consider threading guidelines in their interface design.

Albeit most 3D rendering suites are commercial and thus do not provide public documentation for their internal design, there is one popular freeware package: The Blender software suite [BL11a].

An aspect that makes Blender even more relevant is its integrated interactive game engine, driven by the same rendering component as the WYSIWYG editor. Figure 3.5 illustrates the raytracing, rasterization, and game components within blender.

In terms of overall system architecture, the Blender system is centered around a general-purpose scene graph. The structure of this scene graph closely follows the needs of the main modeling component — apart from typical instancing or material nodes, there also are nodes that store user preferences or tool settings. On a lower level, each graph node is represented by a reference-counted data block. Data blocks are arbitrarily linked within the final scene graph.

Both raytracing and traditional rasterization within Blender's interactive rendering component work on the modeling-oriented scene graph. This has particular implications for the game component. On the one hand, no rasterization-typical optimization structures (portals or occluders) have been implemented. Instead, a CPU-based raytracing process generates a low-resolution depth map that is used in a later rough occlusion culling phase. On the other hand, the Python-scripted interface of the Blender game engine maintains a rather hard relationship with the internal scene graph — consequently, undesired coupling between unrelated components results. In turn, development of the game component often lags behind that of other Blender functions.

Separate from its general architecture, another relevant aspect in Blender is its multithreading functionality. Offline raytracing in blender is not particularly optimized for performance and uses a basic tile-based parallelization scheme [BL11b]. While no official statements exist on multithreading in the remaining codebase, including the WYSIWYG rasterizer, experiments indicate that no extra OS-side worker threads are spawned for complicated operations. The user interface even becomes unresponsive

Figure 3.6: The above image was generated by the Ogre3D rendering engine. Image courtesy of [OE11]

when working with many triangles, or hangs on resource loading. Thus, no further insights are to be gained here.

3.3.2 Ogre3D

Unlike the raytracer-focused Blender system, the popular open source engine Ogre3D [OE11] is specialized on pure rasterization-based rendering.

Most notable features include wide cross-platform support, a concise and easy to use client-side interface, and a hard-coded lighting strategy with an integrated, customizable shader system. Figure 3.6 provides an illustrative example of an Ogre3D-based rendering.

One of the focus points within the development of the Ogre3D engine was its general design and rendering API abstraction layer.

Within the Ogre3D engine, all major classes derive from a single root type, simulating a Java-like class hierarchy. Base features — for example garbage collection or container types — are implemented atop this hierarchy root.

On client-side, all of the engine's functionality is accessible by abstract interface types. For instance, an abstract `RenderSystem` class collaborates together with abstract `Texture` and `Model` classes.

All abstract types are grouped into corresponding functionality modules such as scene management or resource management. Implementations for the functionality modules are connected by a plug-in-based strategy. To continue on the previous example, the rendering system can easily be extended to support additional APIs by the integration of a plug-in. Currently, there are back-ends for both the OpenGl platform and the DirectX platform available.

In terms of general scene management, the Ogre3D engine employs a hard-coded, unified scene graph system. This scene graph is directly exposed to client applications. In turn, another application layer is required to abstract the rendering scene graph from high-level application logics. In contrast to the rigorous client-side front-end,

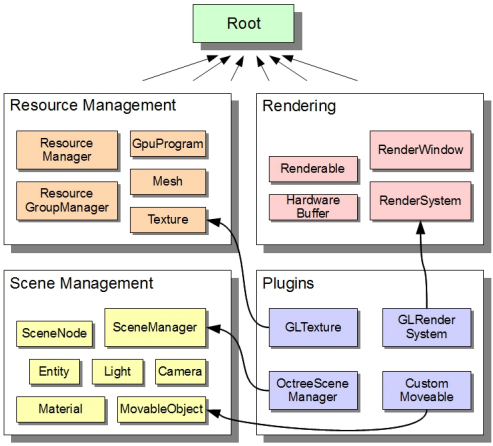


Figure 3.7: An overview over the general Ogre3D architecture. All classes derive from a base root object. Client-side abstract interfaces are extended by plug-in component implementations. Figure courtesy of [OE11].

the back-end of the scene graph module again is exchangeable. For example, there are both octree and portal based modules available.

Figure 3.7 provides an illustrative example of certain functionality modules and plug-ins within the Ogre3D engine.

The general feasibility and extendability of the abstract rendering API within Ogre3D is demonstrated in [BS06a]. As proof of concept, this work develops an interactive, GPU-based raytracing prototype for the Ogre3D engine from scratch. An early BVH-based approach is integrated into a custom Ogre3D scene management plug-in. With support for but affine transformations of entire objects, the BVH is entirely prebuilt in an offline process. A corresponding GPU-side traversal algorithm that utilizes NVIDIA's legacy Cg programming language is integrated as a rendering module plug-in. In total, the Ogre3D engine and the raytracer plug-in achieve almost interactive framerates of up to ten frames per second at rather low resolutions.

In contrast to general extensibility concepts, the Ogre3D engine does not support multithreading — neither by an internal rendering thread nor by multiple client-side scene update threads. The only exception to this is an undocumented multithreaded background resource loader that can be enabled by a compile time switch. Yet its use is discouraged for stability reasons. The developer statement on multithreading support hints at the complexities involved with retrofitting of thread safety into an existing system as extensive as a 3D engine. The suggested approach for decoupling client-side logics from rendering involves a separate rendering thread that communicates with multiple client threads via a message-passing system. This strategy bears similarities to the jVR renderer currently implemented within Simulator X. Further

Figure 3.8: Even with but traditional rasterization, the Unreal Engine achieves rich visual effects — such as reflections or realistic human skin and hair. However, this comes at the cost of a complicated, shader-based integration. Images courtesy of [UE11].

discussion is provided in chapter 4.

3.3.3 Unreal Engine

In terms of released software titles and general popularity, the commercial Unreal Engine [UE11] is the counterpart to the open-source Ogre3D engine.

Like the Ogre3D engine, the Unreal Engine intrinsically supports but rasterization. Yet its proposed feature set is much more advanced in comparison to its open-source competitor. Multiple dynamic and static lighting strategies are supported, as well as a custom, graph-based shader editor. This is complemented by a wide array of predefined rendering techniques, such as forward and deferred shading. Finally, cross-platform compatibility for the Unreal Engine is not limited to personal computers, but also includes handheld devices, entertainment consoles, and high-end mobile phones.

Figure 3.8 shows two example screenshots that utilize some of the above features.

As the Unreal Engine is a commercial product, little public documentation about its internal structure is available. Apart from official feature lists, there only is a freely available development kit that allows partial insights into the engine's concepts.

Most of the application side logics within the Unreal Engine are written in a custom scripting language. UnrealScript — as the language is called — merges features from both C and Java to provide a high-level interface for virtual world control. The language combines the concept of standalone entities with an intrinsic state system, dynamic load balancing, and a multithreaded back-end interpreter. Thread safety is not always maintained, certain race conditions and inconsistencies between objects are allowed. Instead of a more intricate scene graph system, entities within UnrealScript are managed in a flat object pool. All application logics are decoupled from the rendering core by device-independent rendering representations that can be attached to each entity.

The above architecture is further illustrated in figure 3.9.

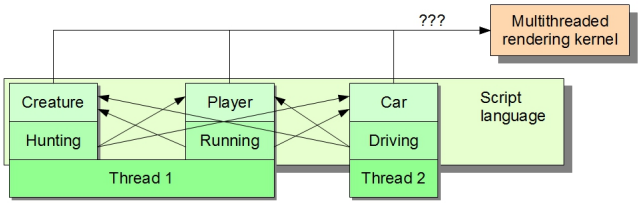


Figure 3.9: Within the Unreal Engine, multiple scripted entities are processed in parallel with automated load distribution. State management is provided as an intrinsic feature of the scripting language. There is no detailed public documentation available for the rendering kernel itself.

As the actual rendering core is separated from the client-side script-able logics, no definitive statement can be made about its multithreading capabilities. However, the current online presentation by the developing company advertises the core as inherently multithreaded in regards to various aspects of the entire rendering process — such as occlusion culling or animation.

3.3.4 Conclusion

There are two conclusions that can be drawn from the investigation of existing systems:

First, there currently is not even one rendering kernel that was developed with support for both raytracing and rendering in mind. Existing systems favor but one of the rendering algorithms. If both algorithms are to be supported, a general, API-independent client-side interface is required. This interface must not expose implementation details — such as a raytracing tree structure or a rasterizer pipeline.

Second, freely available rendering engines in general do not provide any sort of sophisticated multithreading capabilities. This has multiple reasons. For instance, all investigated software has grown over the years, starting at a time where multithreading was simulated on a single core and thus gave no performance benefit. Therefore, threading support was not included in the original design and cannot easily be integrated into any complicated, existing system. Another reason might be the difficulty in managing development of complex multithreaded systems — in particular for many contributors in an open source environment. Consequently, a rendering system with support for multithreading must consider thread safety in its initial design. Appropriate safety rules should be lightweight and must be formulated explicitly and concisely.

Both of these points will be addressed in development of the Simulator X rendering component: The component will provide a unified renderer interface for both rasterizer and raytracer back-ends. Furthermore, the general renderer interface will implement multithreading support, both on client-side and in terms of potential internal threads — with but few, short rules on thread safety.

4 Simulation kernel

This chapter provides an overview over the Simulator X framework.

Simulator X was developed in a joint venture by the University of Bayreuth and the Beuth University for Applied Sciences Berlin. It acts as a testing environment and middleware platform for various architectural approaches to an intelligent, realtime interactive system. Such systems are of particular relevance for the development of virtual reality applications like computer games or scientific simulations.

The overview starts with the presentation of fundamental concepts within Simulator X. The design metrics of coupling and cohesion are introduced, and a basic data and object model for Simulator X is derived. Consequent sections discuss functional layers within the simulator architecture. This includes the concept of state variables, the message-based, event-driven communication scheme and the global world interface. Finally, the existing rasterizer-based display component within Simulator X is studied in depth.

Most of the descriptions herein are closely based on the research releases covering Simulator X development, mainly [ML10], [ML11], and [DW10].

4.1 Fundamental concepts

The most fundamental goal of the Simulator X project is to provide a modern middleware platform for any kind of virtual reality software. This requires satisfying a variety of requirements. The most important of these requirements is the implementation of an abstract, modular architecture. For one, a modular, well-defined architecture encourages code reuse and extends the general life-time of any piece of software. On the other hand, modularization eases the realization of additional requirements - such as effortless, fine-grained parallelization, intuitive extendability or painless integration of new components.

4.1.1 Coupling and cohesion

In Simulator X, a modular design is achieved both by minimizing coupling and by maximizing cohesion within the system architecture. In this context, [ML10] defines

- **coupling** "as the measure of the independence of relations between functional units", and
- **cohesion** "as the measure of the semantic nature of relations between components of a functional unit".

In other words, minimized coupling refers to functional units that interface with but few other units via small and unique interfaces. At the same time, maximized cohesion indicates that a functional unit should contain a maximum number of components that are closely related.

As an example for these concepts, consider three traditional approaches to the simulation of logics and object relations within a virtual world: Scene graphs, event systems, and entity models.

4.1.2 Scene graphs

The popular scene graph is a good example for a less than optimal system architecture that violates the coupling and cohesion rules. In particular, a scene graph combines all virtual objects into a single, abstract graph structure.

While a scene graph primarily contains hierarchical or spatial information, it also is traversed for rendering, collision detection, sound playback, and simulation control logics. Thus any graph traverser must pay attention to the types of objects it encounters. For example, the rendering traverser must ignore any objects that only contribute sounds to the virtual environment. At the same time, objects within the scene graph potentially are required provide a wide array of glue functionality, even for functions that are not directly related to the actual object.

Consequently, the scene graph system generally tends to become a mess [TF10]. This is an expression of its coupling and cohesion attributes: Scene graph coupling is high, as it exposes all its separate functional components to all clients. At the same time, cohesion is minimal, as each functional component (i.e. an object in the graph) may be used for a wide area of distinct operations.

Cohesion and coupling within a scene graph is visualized in figure 4.1.

4.1.3 Event systems

From an architectural point of view, event systems can provide a vast improvement on scene graphs. In an event system, objects do not directly collaborate by function calls. Instead, notifications are sent to arbitrary receiver objects on certain occasions.

There is a wide range of implementations for such event systems. Simple systems provide hard-coded event processing based on user-registered callback functions. In contrast, there also are very flexible systems that allow for arbitrary customization of the event dispatch mechanism — such as message passing schemes.

Depending on the actual implementation, event systems tend to improve functional cohesion: Any object now contains all logics to handle various events that work with the object's state, and the state remains opaque to other objects. However, one must also consider that this improvement comes at the price of reduced cohesion in regards to overall functionality. For instance, the rendering code within an event-based system might be implemented in a frame drawing event handler of each object type — instead of a single shared rendering module.

Apart from improved cohesion, event systems also provide an opportunity to reduce coupling. Once more, the gain here is vastly implementation dependent: A hard-coded dispatch scheme even increases coupling due to the hard links and order requirements between separate objects. Yet, an event system that allows for flexible event routing establishes only mandatory object connections, and thus decreases coupling.

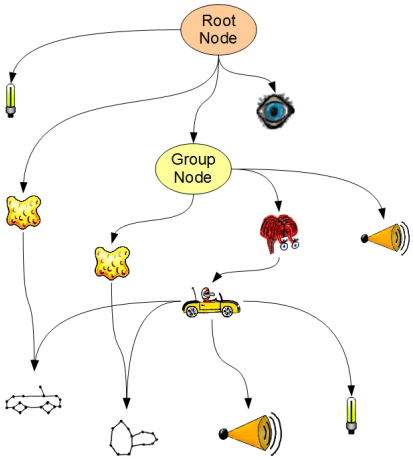


Figure 4.1: This example scene graph unites rendering, AI, audio, and logics functionalities in a single hierarchical structure. Any functionality-specific traverser within this graph potentially has to be aware of all object types. For instance, a rendering traverser must be able to navigate past the AI brain controller and the car logics representation to get to the actual geometry.

Figure 4.2 shows the coupling and cohesion implications of an event system architecture.

4.1.4 Entity models

Entity models provide even better coupling and cohesion attributes than event systems.

Within an entity model, an entity represents the smallest logical object. However, simulation functionality — such as rendering, collision detection, or AI — and the associated simulation states are not directly integrated into the entity object. Instead, all simulation functions and simulation states are outsourced to specific functionality modules. The entity then links to various states within various modules. Communication with the modules and contained states triggers module functionality and provides state change notifications. The functionality modules, in turn, are free to choose an arbitrary interface for encapsulated states, and thus hide the function implementation from the entity.

In other words, an entity within the entity model is a general, abstract logics controller that remotely controls functionality-specific representations of itself that separately are held within functionality modules.

In this context, note that multiple states may be shared by each entity, and there may be arbitrary links in between both entities and representations within each subsystem.

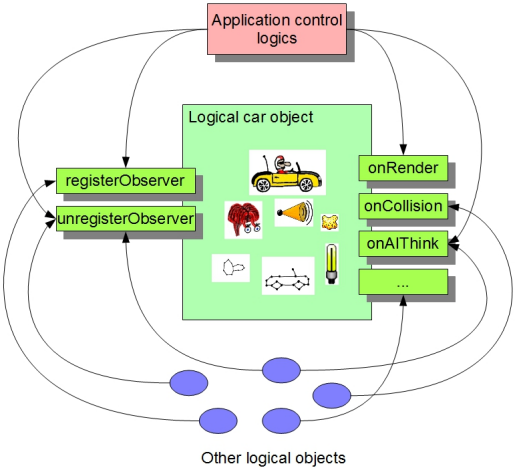


Figure 4.2: In this rather hard-coded event system, each object is a composition of all required states from various functionality types — rendering, AI, and audio. Thus, general cohesion is better than for pure scene graphs, while functional cohesion still is not achieved. Event handlers are registered on each object, and remotely triggered on adapt occasions. If used correctly, event handlers only induce dependencies where these are required for functionality, and thus decrease coupling.

Only relations between state representations from different functionality subsystems are forbidden to avoid coupling.

Entity models offer a similar abstraction of object relations as event systems. However, the additional gain in cohesion comes from the outsourcing of state and functionality into function-specific modules. Coupling is reduced by the intermediate entity instances that replace any direct communication in between modules.

For example, within a computer racing game, there might be a car entity that represents an abstract, logical car concept. For rendering, the car entity holds a reference to a car geometry object within the rendering module. For audio output, the entity links to an engine sound object within the audio module. Finally, collision detection is delegated to a special box-shaped collider object within the physics module. This keeps the actual entity free from non-logics related functionality. At the same time all related functions and states over the car entity and any additional entities are grouped within each functionality module. This example for an entity model architecture is visualized in figure 4.3.

Even though entity systems have been compared to event-based architectures, these are not exclusive strategies. In contrast, it even is possible to merge both approaches into a single unified data and object model: Entities reference separate functional

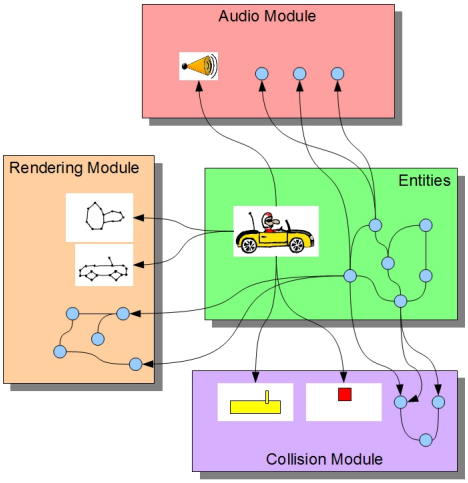


Figure 4.3: Within the above entity model architecture, the car entity references functionality-specific state representation within rendering (geometry mesh), collision detection (reduced bounding boxes), and audio (engine sound instance). Thus, maximal function-based cohesion is achieved within each functionality module. At the same time, minimum coupling is realized by the intermediate entity objects. Colored dots visualize any additional entities, state representations, and their relations.

states, and any communication in-between entities and states is managed by an event-driven message passing realization.

This allows for exploitation of coupling and cohesion benefits from both system architectures. Consequentially, a combined entity model and event system approach was chosen for the Simulator X platform.

4.2 Architecture overview

As previously derived, Simulator X is based on a fundamental entity model with an integrated event handling system to achieve maximized cohesion with minimal coupling. The following sections discuss the integration of both strategies within Simulator X and their embedding in the large-scale, layer-based system architecture.

In general, the architecture of Simulator X consists of separate functionality layers: A high-level world interface layer acts as a client-side entry point into the Simulator X platform. Functional components communicate within the world interface by the event system layer. Relations between those components are represented by a symbolic binding layer. The Simulator X actor system is implemented on a lower level than preceding functionality modules. Entities are connected to the remaining

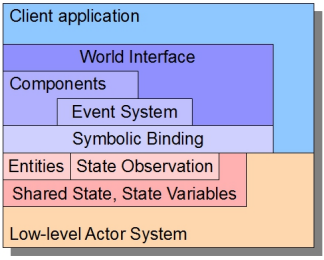


Figure 4.4: An overview over the architectural layers within Simulator X. Image courtesy of [ML11].

platform by symbolic binding. Their state contents are further made accessible by a messaging-based state observation system.

Figure 4.4 depicts the above architectural layers and their dependencies within Simulator X.

The remaining section follows a bottom-up discussion of Simulator X layers in two steps: In the first step, the realization of an actor model and an entity model in terms of the Simulator X environment is discussed. In the later second step, the higher-level world interface and component structure are reviewed.

4.2.1 Actors, entities, and state variables

The root concept for distributed processing within Simulator X is founded on the actor paradigm. In the actor paradigm, program execution is distributed over a series of independent actor components. Within the Simulator X realization, each actor typically represents a single thread of execution. There is no globally available, shared application state representation — such as a global scene graph. Instead, each actor stores a local state copy of its own. This allows for intuitive, conflict-free synchronization between multiple actors. Synchronization of states and general communication in-between actors is implemented by asynchronous message transfer. In other words, actor communication is realized by an adaption of the precedingly described event systems. Finally, Simulator X allows for lightweight actor components: Actors may be created or destroyed at any time without harsh performance breakdowns.

The local application states within each Simulator X actor implement the entity model. The lowest-level building blocks within the entity model are formed by state variables. A state variable in general stores an arbitrary value. However, this value physically resides in but one actor. If other actors require access to a variable, they insert a respective variable reference into their local state. In turn, the message-based synchronization scheme automatically retrieves the value of a reference-typed state variable from its respective owning actor. Consequentially, the Simulator X entity model hides the internal details of message passing from client applications. This view on state variables is represented in figure 4.5.

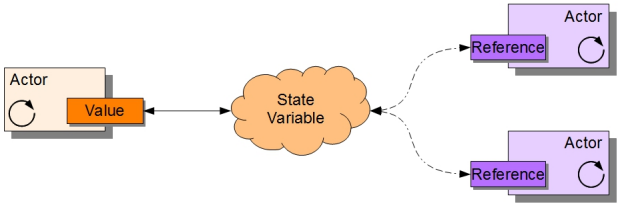


Figure 4.5: A single, global state variable is accessed by multiple independent, threaded actors. One of the actors owns the actual value instance of the state variable. All others work but on transparent references. Image courtesy of [ML11].

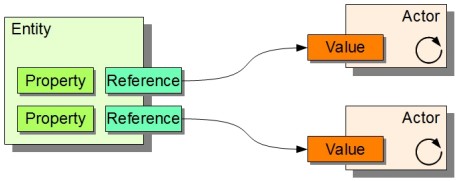


Figure 4.6: An entity combines multiple state variables that are housed within different actors. Each state variable within an entity is associated with a property to bind further semantic information. Image courtesy of [ML11].

Entities provide a further abstraction layer atop the state variable concept. Each entity bundles a series of logically related state variables into a single collection. Still, each of the state variables within an entity in this context may be owned by another actor — respective a functional component in terms of the conceptual entity model introduction. Further functionality-based grouping collects state variables within an entity that are owned by the same actor or functionality component. The resulting groups have been termed aspects, and completely describe a single functional facet of an entity.

Apart from bundling and grouping state variables, an entity also associates each of its encapsuled variables with a descriptive property. Each property in turn defines an abstract semantic symbol for the respective state variable. Semantic symbols specify the semantics and use of a single state variable within global application context.

Figure 4.6 provides an example for an entity and contained state variables and properties.

In regards to more formal architectural criteria, the concept of standalone, independent actors with distributed state variables and grouping entities emphasizes functional cohesion and data decoupling: On the one hand, each actor realizes an exactly specified functionality and owns but state relevant for its function — consequentially, cohesion is maximized. On the other hand, each actor has an unique representation

of each entity and each variable. In turn, no direct access to data from other actors is required, hence coupling is reduced.

4.2.2 World interface, events and components

For intuitive use in client applications and for the connection of components, the world interface defines a configurable, event-based abstraction layer over entities and state variables.

In particular, the world interface offers four major operation types: At first, a configuration operation specifies a set of relevant general events and allowed actions. Thereafter, the world interface performs notification operations to signal relevant events. Likewise, execution operations are invoked to process any allowed actions on state variables. Finally, the world interface provides an operation that explicitly queries the value of any state variable.

Within the above, the concept of custom event selection has been introduced. Event selection allows an application or component programmer to define a certain subset of interesting state data that is required in higher-level application logics. Client-side configurable events are a rather novel concept. In comparison, existing systems only propagate fixed event types in between functionality subsystems and the main application.

Within Simulator X, events further decompose into two separate groups: Generic events and value-change events.

Generic events convey general information that need not necessarily be coupled with a modification to any state variable. Here, the world interface acts as a mediator between potential event senders and receivers: Each sender and receiver registers its respective event types with the world interface. Matching sender and receiver pairs are then automatically connected by the world interface, and consequently switch to direct peer-to-peer communication.

Unlike generic events, value-change events directly signal changes in state variables. Such messages are generated by the world interface itself, and contain a reference to the modified state variable alongside a copy of the current variable value. This gives non-owning actors the opportunity to update local state appropriately.

As the last relevant facet of the Simulator X architecture, high-level components define functional application modules that encapsulate a series of related actors. Components are characterized by adding a single aspect type — a single group of state variables — to certain entities. At the time of this writing, there are components that handle rendering, audio output, physics simulation, AI, and various input devices.

4.3 Existing rasterization module

In this section, the existing jVR rasterization component is investigated. Its general Java-side architecture is discussed in greater depth. Thereafter, particular attention is placed upon the interface that the rendering component establishes in communication with the remaining Simulator X framework.

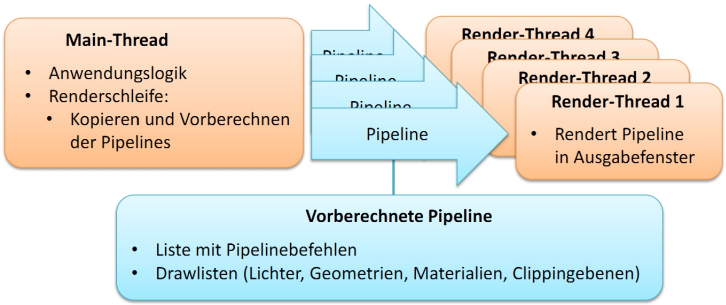


Figure 4.7: A high-level overview over the jVR rendering and threading concept. Figure courtesy of [MR10].

4.3.1 Java-side architecture

The Java-based renderer jVR originally has been designed in terms of a Master thesis [MR10] as an intuitive alternative to existing C++ engines for use within study courses at the Beuth Hochschule fuer Technik Berlin.

The basic Java-side architecture of jVR is centered around two concepts: Scene graphs and rendering pipelines. The former define the virtual scene — materials, light sources or geometries. The latter contain a series of rendering commands for later processing. This allows for the application of various standard rasterization techniques. For example, deferred shading and forward shading can both be realized with the same scene graph and rendering back-end by exchanging the pipeline setup.

The architecture further allots for a single-threaded client-side main loop that triggers the accumulation of a rendering pipeline. Thus, the main rendering loop is restricted to but a single scene graph user.

Yet, there still are separate back-end rendering threads. In particular, one such thread is spawned for each output window. The main thread passes any complete rendering pipeline to a corresponding rendering thread for final processing. This decouples any rendering calculations from main logics. In turn, interruptions in application logics due to excessive frame rendering times are avoided.

Figure 4.7 provides an illustrative overview over the entire jVR client-side rendering and threading concept.

4.3.2 Simulator integration

Within the Simulator X project, the Java-based jVR renderer is integrated in terms of two separate actors: `JVRRenderActor` and `JVRConnector`

The JVRRenderActor corresponds to a single jVR main thread and several attached target windows. It is responsible for translation of Simulator X entities and state

variables to corresponding jVR scene graph representations. To allow for various application scenarios, a rendering pipeline can be constructed by an external, pluggable pipeline provider. If no explicit pipeline is given, a default pipeline setup is applied.

The JVRConnector abstracts over multiple JVRRenderActors, and provides a unified client-side interface. In particular, incoming state variable and entity notifications, as well as general rendering control messages are passed on to all referenced rendering actors. The JVRConnector also provides the main entry point into client-side rendering: A special configuration message creates a series of ready-to-use rendering actors, and defines initial display parameters.

In a system-wide context, both actors are controlled by means of two main Simulator X concepts, the event system and the entity model.

On the one hand, there are few general graphics-related events — such as a message for rendering a new frame. These are triggered by client applications whenever necessary, and thereafter are processed by the rendering actors.

On the other hand, any scene management is performed by the concept of actor-local entities. To be specific, a new graphics-based aspect is attached to each renderable entity in the simulator. State variables that control the client-relevant part of the object's visual representation are encapsulated within the graphics aspect. Each entity in turn is sent on to the rendering actors by Simulator X mechanisms. These create entity copies with state variable references inside their local application state. Thereafter, the rendering actors map their entity copies to jVR-internal scene graph components and initialize these from state variables. Finally, the resulting scene graph is processed on the next incoming rendering event.

Apart from general event and entity facilities, the current architecture does not further abstract over the use of the jVR rendering actors.

Most notably, the graphics aspect defined in each entity still is quite specific to the jVR rendering actors. For instance, the following Scala excerpt defines the graphics aspect from a fireball-typed entity within a framework example application:

```
private def description(pos : Vec3f, name : String) =
    new EntityDescription(
    EntityAspect(Symbols.graphics, new TypedCreateParamSet(
        SemanticSymbols.aspects.shapeFromFile,
        JVR.geometryFile <= "fireball-model.dae",
        JVR.initialPosition <=
            ConstMat4f(Mat3x4f.translate(pos)),
        JVR.scale <= ConstMat4f(Mat3x4f.scale(1f )),
        JVR.shaderProgram <=
            ("AMBIENT", "phong.vs", "fireball.fs")),
        Ontology.transform isRequired),
        /* ... */
```

Here, even general attributes like geometry or transforms are defined in a JVR-specific representation.

Furthermore, certain parts of client-specific code — such as management of a certain type of on-screen user interface — are directly realized within the general rendering component.

While the basic state variable and message-passing architecture achieves reduced functional coupling, the direct communication with a specific rendering implementation again induces coupling on semantic level. Semantic coupling against a functionality module has several disadvantages, such as in this case disallowing for the intuitive exchange of the rendering component. Consequently, this design inconsistency will be revisited in chapter 8 alongside the integration of the new raytracer component into the Simulator X platform.

5 OptiX platform

The following chapter gives a tour over the API of the NVIDIA OptiX raytracing engine and the underlying CUDA platform.

As mentioned in the introduction, OptiX is a raytracing wrapper built atop CUDA, NVIDIA's graphics hardware programming platform. As such, at least a basic understanding of CUDA is required to comprehend the later OptiX code.

Consequently, the section starts off with a short CUDA introduction. The custom CUDA language for programming of the graphics hardware as well as the associated high level language bindings are detailed. Finally, the required compilation process is presented.

Then, the actual OptiX rendering process is investigated. This includes information on API functionality as well as an overview of the programmable components within OptiX and the background optimization hierarchy used for ray intersection tests.

The chapter ends with a small example raytracing application binds together the previously described steps and operations on the CUDA and OptiX platforms.

In respect to bibliographic references, most of this chapter is based on the OptiX programming guide [NV11a] and the accompanying quickstart guide [NV10]. An in-depth CUDA programming guide is found in the respective manual [NV11c]. A more compact editing of the CUDA manual that is particularly suited for beginners is available in [TW08]. Further references are cited where appropriate.

5.1 CUDA overview

As described in the introductory chapter, ever increasing performance of graphics hardware motivated its use for general computations.

In the absence of any programming interface suited for general computations, first experiments abused pixel and vertex shaders alongside the respective programming languages to perform the actual calculations. Input data had to be wrapped into textures, and output data was generated within the framebuffer of the graphics hardware. As such, it was for instance possible to perform a fast multiplication of two large matrices by combining two textured quads in the framebuffer by means of a pixel shader [AM03].

NVIDIA tried to accommodate the GPGPU trend by developing a language and a high-level API specifically tailored to the requirements of general calculations on graphics hardware. The resulting platform was called **C**ompute **U**nified **D**evice **A**rchitecture, or CUDA for short.

In analogy to the original graphics APIs, the CUDA platform differentiates between the host (i.e. the CPU and main memory) and the device (i.e. the GPU and graphics on-board memory).

An overview over the entire GPU-augmented calculation process using CUDA is as follows:

1. A CUDA program for execution on graphics hardware — termed compute kernel or kernel for short — is written in the CUDA language.
2. The kernel is compiled with the NVIDIA-specific compiler nvcc.
3. A host program written in a high-level language such as C++ copies input data from host memory to device memory using a CUDA API call.
4. Execution of the CUDA program is triggered on the device from within host application code, again using a CUDA API call, or alternatively a CUDA-specific language extensions.
5. After kernel execution has finished, the results must be copied back from device memory to main memory before further use, once more using a CUDA API call within the host application.

5.1.1 Parallelism on graphics hardware

The actual performance improvement over CPU computations comes from the high amount of parallelism supported on graphics hardware. Each CUDA kernel execution spawns a high number of worker threads on the graphics hardware. Current graphics hardware designed for general computation, such as NVIDIA's Tesla series GPUs, sports as many as 1000 independent thread processors that work in parallel.

Of course, an intuitive concept is required to allow for effortless management of that many threads. In CUDA, thread management follows several separate guidelines:

First, all threads within a CUDA program start at the same device-side entry point function. Program execution does not end before all threads have finished execution of that function. As a short preview on the next chapter, the entire OptiX raytracing process is encapsulated into a single such entry point.

Then, the user usually does not have direct control over thread count or thread scheduling. In particular, the user does not specify the actual number of threads to spawn. Instead, the user defines but an abstract thread grid that typically corresponds to the problem dimension. CUDA internally manages its thread pool and distributes threads on this workload as required. During processing, it is only guaranteed that a single GPU thread runs through the main entry point function for each problem grid cell. For instance, when working on a very large bitmap, the problem grid holds a single thread for each pixel. This gives several orders of magnitude more logical threads than natively available on the GPU. However, the hardware only processes as many of these at once as possible. Figure 5.1 gives an overview over thread grids.

Next, all threads in the problem grid are grouped into warps. A warp contains but a small number of consecutive threads, typically around 16 threads. Due to hardware restrictions, all threads within a warp must carry out the same processor instruction in each processing cycle. The only workaround here is that output of certain warp threads can be deactivated momentarily. This indicates a typical bottleneck within CUDA device code: If only part of a warp enters a conditional if-else clause, both code paths need to be run through with all threads of the warp in sequential order. This

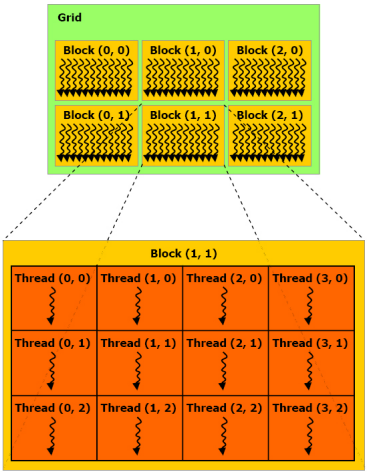


Figure 5.1: Distribution of threads into a problem grid. Threads are grouped into blocks, and blocks are grouped to form the final grid. Figure courtesy of NVIDIA, [NV11c].

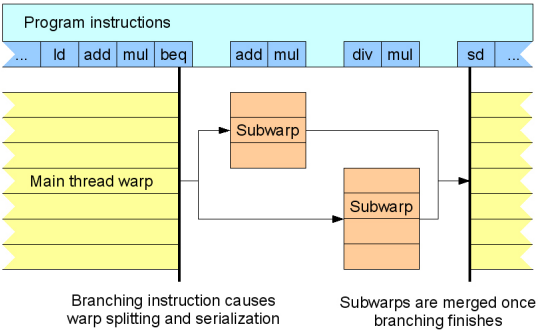


Figure 5.2: Albeit only half of the provided threads within a warp enter each path of the branch_if_equal *beq* instruction, the entire warp needs to process both branches sequentially.

problem is visualized in figure 5.2. Thus, it is important to optimize all conditional clauses for local coherence.

Finally, it is the user's responsibility to avoid any write conflicts caused by concurrent threads. Within CUDA, this is supported by certain atomic operations. However, to allow for maximum throughput, all these operations are rather simple. For instance, it

```
__global__ void vectorAddDevice
    (const float* va, const float* vb, float* out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    out[index] = va[index] + vb[index];
}
```

Figure 5.3: An example, GPU-executed CUDA function that performs per-component parallel vector addition on `va` and `vb` and stores the result in `out`. The `__global__` keyword indicates an entry point into a GPU kernel. The per-thread global variables `blockIdx` and `blockDim` provide thread and block indices within the thread grid. The method `vectorAddDevice` is automatically called once for each thread in the logical thread grid. The dimensions of the grid in turn corresponds to the size of the input vector.

is only possible to synchronize all threads at once via a device code barrier. As thread scheduling is entirely managed on GPU-side, no other synchronization is available.

5.1.2 Language and API

Unlike custom graphics shader languages, the CUDA language for GPU programming was originally aimed at scientists in numerics. As C to this day remains one of the most-used and most-supported language for numerics, CUDA was based on C to allow for intuitive user adoption. For instance, most standard C libraries (such as `math.h`) are available, and certain functions (e.g. sin() or log()) are optimized by GPU-side intrinsics. For improved usability, a select few C++ concepts — such as classes or namespaces — also found their way into the CUDA language.

Yet, certain restrictions and features for graphics hardware control have been added to CUDA that exceed both C and C++. For example, there are global variables with the index of the current thread within the owning thread block. These allow GPU code to identify the workload that the current thread should process. Additional keywords have been introduced to designate entry point methods or register variables.

A small example program for per-component parallel vector addition written in the CUDA language is given in figure 5.3.

For triggering CUDA processing and graphics hardware communication from within a high-level language, CUDA contains a library of C-style functions. As an example, the C function cudaMalloc allocates memory on the graphics device, much like standard C malloc allocates memory on the CPU host.

Additionally, extensions to the C and C++ languages that simplify certain tasks are provided via a preprocessor front-end in nvcc. In particular, a series of boilerplate C statements are required to execute a GPU kernel: The thread grid must be initialized, the kernel entry point has to be located, and the host application must wait for kernel success. However, CUDA also provides a one-line kernel starting point via a

```
__host__ void vectorAddHost
    (const float* va, const float* vb, float* out, int dim)
{
    float *vadevice, *vbdevice, *outdevice;
    int size = sizeof(float) * dim;

    // Allocate on-device memory
    cudaMalloc((void**)&vadevice, size);
    cudaMalloc((void**)&vbdevice, size);
    cudaMalloc((void**)&outdevice, size);

    // Copy input vectors to device
    cudaMemcpy(vadevice, va, size, cudaMemcpyHostToDevice);
    cudaMemcpy(vbdevice, vb, size, cudaMemcpyHostToDevice);

    // Execute our kernel and wait for success
    dim3 dimblock(128);
    dim3 dimgrid(dim / dimblock.x);
    vectorAddDevice<<<dimgrid, dimblock>>>
        (vadevice, vbdevice, outdevice);

    // Copy result back to host memory
    cudaMemcpy(out, outdevice, size, cudaMemcpyDeviceToHost);

    // Clean up device-side memory
    cudaFree(vadevice);
    cudaFree(vbdevice);
    cudaFree(outdevice);
}
```

Figure 5.4: Before calling the CUDA vector addition from figure 5.3, the vectors first need to be copied from host memory onto the GPU device via C functions provided by the CUDA API. The GPU kernel then is executed by a CUDA language extension (triple brackets), and the result is copied back to main memory.

C language extension: Triple pointy braces are used to indicate a device kernel call, appropriate grid extents are intuitively passed in as template-like parameters.

Both the CUDA C API and language extensions are shown in listing 5.4.

5.1.3 Compilation

A custom compiler is required to translate the CUDA language itself and any CUDA extensions in high-level code into an executable binary. NVIDIA provides two alternatives to approach this task, both based on the nvcc compiler toolchain shown in figure 5.5.

On the one hand, nvcc is able to completely handle an entire application. NVIDIA

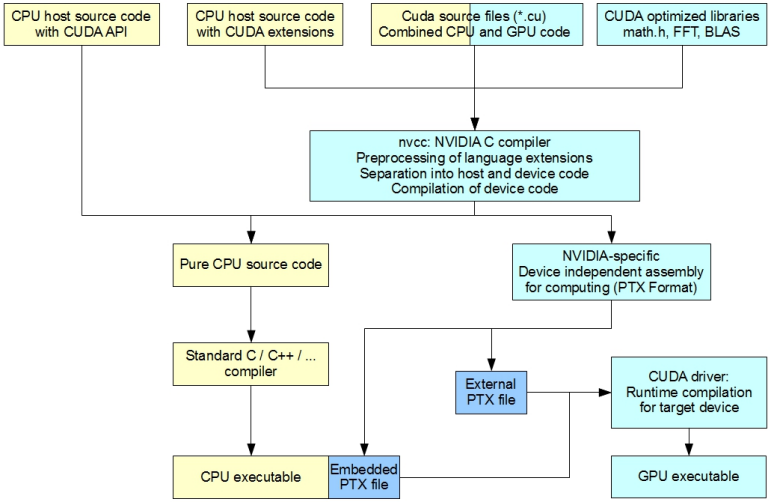


Figure 5.5: CUDA compilation process: NVCC and a host compiler cooperate to process extended high-level sources. CUDA-specific sources are compiled to either standalone or embedded PTX assembly files.

extensions in high language sources are replaced by appropriate CUDA API calls, and the resulting standard-compliant sources are automatically sent to some other compiler for the host machine. For instance, nvcc collaborates with the GNU Compiler Suite under Linux, or with Visual Studio under Windows. In the same process, nvcc compiles CUDA files and CUDA language sections within other sources into device-independent assembly code. The resulting assembly is stored in human-readable text files, also termed PTX files. PTX here is shorthand for **P**arallel **T**hread **E**xecution assembly language. The PTX files themselves in turn are embedded into the application binary. At application startup, the PTX files then automatically are extracted from the executable and recompiled for the current graphics hardware by the graphics driver software.

On the other hand, nvcc directly can compile any CUDA language file into a device independent assembler listing in the PTX text format. Applications can manually load such files at runtime, then manually recompile them for the current GPU device, and finally execute the resulting hardware kernel. In this context, one should especially take note that certain comfort functions within the CUDA API are only available if used alongside the language extensions.

The latter use case of runtime-compiled CUDA code is rather atypical and sparsely documented. Yet, this approach is required to extend the OptiX raytracer with programmable components (e.g. procedural materials) at runtime without an entire host program recompilation.

This concludes the overview over the fundamental CUDA platform.

5.2 OptiX overview

The OptiX raytracing engine provides another convenience layer atop the previously detailed CUDA platform: A series of exchangeable, CUDA-programmable components works together with GPU control logics pre-implemented within OptiX to perform raytracing operations. An opaque OptiX-internal optimization structure is used alongside a client-side scene hierarchy to further increase raytracing performance.

In the following, an overview over the high-level code flow required for raytracing with the OptiX platform is presented, both on CPU and on GPU side. Consequent sections explain the programmable components within the raytracing process and scene management facilities in depth. Finally, the OptiX API itself and its native multithreading capabilities are discussed from an implementation point of view.

5.2.1 High-level code flow

Raytracing with the OptiX API generally requires a series of high-level steps: At first, programmable components need to be created and compiled, separate from the remaining raytracing process. Within the actual application, the first task is the creation of an OptiX program context. Thereafter, OptiX must appropriately be initialized by loading programmable components and by sending scene data and global variables to the hardware. Only then does GPU-side raytracing begin.

The above host-side raytracing process is detailed in the following:

1. **Create and pre-compile programmable components**

   In a standalone step, code for programmable components of the raytracing process is written in the CUDA language and compiled to device-independent PTX text files with the nvcc compiler. Programmable components include ray generation programs or ray-triangle intersection handlers, and are described in depth in 5.2.2.

2. **Create OptiX context**

   Within actual application code, an OptiX context is created via a call to the OptiX API. This context provides the high-level entry point into raytracing, and acts as a handle for all further operations.

3. **Load and recompile programmable components**

   Previously generated programmable components are read from PTX files into an in-memory text representation, and compiled to device-specific GPU code with either the CUDA or the OptiX API.

4. **Initialize scene**

   The client application encapsulates all scene objects into an OptiX-internal, GPU-side scene hierarchy to include these into intersection tests.

Small-scale, object-specific input data is attached to scene objects in form of device-side variables. Any associated large-scale data is copied into device-side memory buffers. Data in this context refers either to traditional vertex and triangle indices, camera and light positions, or to any other custom data.

Certain programmable components, for example material programs or intersection tests for custom object data, are directly connected to the corresponding OptiX scene objects as well.

Finally, the scene objects hold the OptiX-internal background optimization structure required for efficient raytracing.

All OptiX-internal scene facilities are investigated in detail in 5.2.4. Large-scale object data management is reviewed in 5.2.6, and 5.2.7 provides in-depth information about global variables.

5. **Initialize global parameters**

   Global programmable components, such as the ray generation program, are attached to the OptiX context. Likewise, certain global input variables for the later raytracing operation — e.g. the current window size or the starting node within the client-side scene hierarchy — are directly set on the OptiX context.

6. **Execute GPU raytracing**

   Once initialization has finished, the application starts the OptiX raytracing process. Exactly as with a CUDA kernel execution, the client specifies a grid of threads to determine parallelism here. Typically, there is one thread per pixel of the output image. Then, the GPU-side raytracing process starts with the initial ray generation program for each such thread. The GPU process is described in detail just below.

7. **Show resulting image**

   Currently, OptiX cannot directly render into any output window. Thus, raytracing results must be rendered into any device-memory buffer. Results then are shown using an alternative API such as OpenGl or SDL.

8. **Repeat as required**

   Once raytracing has finished, interactive raytracing applications iteratively modify the OptiX-side scene and re-render to display any modifications on screen.

One should note the similarities to the fundamental CUDA calculation process here: In CUDA, input data is uploaded to the hardware, processed there by a GPU kernel, and results are retrieved again. Likewise, an OptiX input scene is wrapped up into a hardware-side hierarchy, processed by a kernel there, and the results are transferred back from a general device-memory buffer into some framebuffer.

For an in-depth understanding of the programmable functionality within the OptiX raytracing operation, it is also vital to comprehend the actual GPU-side process flow for the raytracing process. To be specific, GPU raytracing within the above step 6 encapsulates a series of subordinate steps: First, the optimization hierarchy must be

rebuilt to reflect any scene updates. Thereafter, an initial set of rays is generated and traced through the scene. Intersections with scene geometry must be handled appropriately. Recursion is applied to determine the outcome of any secondary rays. Finally, raytracing results are stored in a device-memory output buffer.

The preceding GPU-side raytracing steps are investigated in-depth in the following:

1. **Rebuild the optimization hierarchy**

   On initialization and after certain client-side updates, the optimization hierarchy that is attached to the OptiX scene representation becomes outdated and must be updated or rebuilt. OptiX triggers such operations automatically before raytracing once a relevant change is detected.

   Just as the actual raytracing, the rebuild is performed by a GPU-side, massively parallel kernel and does not require any CPU round-trip for scene data. Due to the support for custom primitive types within OptiX, part of the rebuild is customizable by programmable components.

   The actual type of acceleration structure can be chosen per object, and is detailed in 5.2.5.

2. **Generate rays**

   For each position in the initial thread grid, an unique thread starts at the ray generation program entry point. The user-written ray generation program creates one or more rays, and an OptiX-internal helper method is invoked to trace these rays through the scene. The client must indicate the starting object for intersection calculations within the OptiX scene hierarchy, typically by passing a suitable root object onto the GPU in a global device-side variable.

   Rays themselves carry a custom ray type to enable specialized behavior on scene intersection. Additionally, each ray contains a user-defined data payload that provides further customization options, such as input or return data.

   For example, a common raytracer might use separate types for camera rays and shadow test rays. While camera rays use their data payload to return the pixel color for the output image, shadow rays hold a single boolean that indicates detection of an occluding object. The initial ray generation program spawns but a single camera ray for each thread, or respective for each pixel in the input image.

3. **Trace rays through scene**

   All of the rays sent on to the OptiX raytracing call by the ray generation program are followed through the OptiX scene in parallel.

   At first, rays are automatically sent through the optimization structure. On object primitive level (traditionally single triangles), a programmable intersection test is invoked to determine if an intersection occurred. This test typically evaluates the large-scale object data stored within device-memory buffers to retrieve primitive coordinates. Whenever a ray-primitive intersection is detected, a user-programmable hit handler is invoked to provide an appropriate reaction.

Hit handlers are assigned to each object via their associated material. Furthermore, special handlers can be specified per material for each potential type of incoming ray, and both for an arbitrary intersection or the intersection closest-most to the ray origin. If a material defines no handler for a certain ray type and intersection event combination, the respective intersection with the object is ignored.

To expand on the example from the previous step, object material handlers react to camera rays but on the closest hit. The corresponding handlers run custom program code to set the color of the output pixel within camera ray data. In contrast, objects react to any secondary shadow ray hit, and appropriate handlers set the hit indicator within shadow ray data to influence later lighting calculations.

4. **Recursion as required**

   As a reaction to any intersection incidents, a client-provided hit handler is allowed to spawn arbitrary secondary rays. These are recursively traced through the scene just as the original rays. One should note that recursive rays can carry a type different from the original ray, and contain their own data payload.

   Within the example scenario, recursive shadow rays are spawned within the per-object closest-intersection handler program for the camera rays.

5. **Store output data**

   Once tracing of the primary rays has finished, code flow transfers back to the ray generation program. The raytracing results are returned within the associated per-ray data of each ray. The ray generation program typically concludes the GPU-side raytracing process by copying the return data of primary rays into an output buffer.

   In the preceding example, the ray generation program takes the pixel color from the ray data of the camera rays and places it into the result image.

A more compact summary of the entire collaboration between OptiX objects and the client application is presented in figure 5.6.

5.2.2 Programmable components

As seen from the above overview, the OptiX API uses the CUDA language to allow for custom programming of certain plug-able components within the raytracing process. Not only does this support the implementation of different raytracing strategies for rendering, but it also enables the use of OptiX for more general raytracing purposes. For instance, OptiX can be applied to accelerate sweep-and-prune based collision detection as in [SG07].

Currently OptiX supports the following programmable components:

- **Ray generation**

  This program defines the entry point for a single, parallel execution unit. It is responsible for spawning one or more rays for intersection detection. Each ray carries its own ray type and data payload.

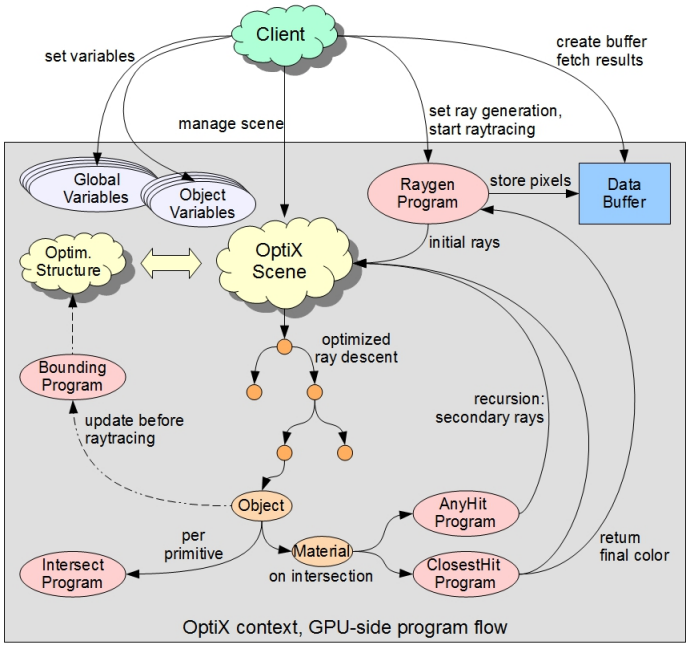


Figure 5.6: Overview over collaboration between application logics and the OptiX platform.

In a raytracing renderer, the ray generation program simulates the virtual camera by creating a primary ray for each pixel in the viewport.

In CUDA context, the OptiX-internal CUDA entry point distributes the generated rays over the thread grid, and executes corresponding parallelized hit tests on the acceleration structure within the OptiX scene.

Ray generation programs directly are assigned to the main OptiX context. While there may be multiple such programs assigned to a single context, only one of these drives any given raytracing process.

- **Exception**

  The exception program is called whenever any error occurs during intersection calculations. This includes errors such as the overflow of the static stack for recursive, secondary rays, or invalid out-of-bounds access to geometry elements. As exception programs drastically decrease performance, their use is encouraged for debugging purposes only. Similar to ray generation programs, exception programs are assigned to the overall OptiX context.

- **Bounding box**

OptiX uses hard-coded bounding box tests in its core optimization hierarchy. On rebuild of the optimization hierarchy, the per-object bounding box program is responsible for calculating the bounding box of some scene primitive — either a traditional triangle, or a custom primitive type.

- **Visit**

  This program is executed on a special condition-type object (see selector node, 5.2.4) before the actual intersection tests.

  In particular, such objects house multiple subordinate representations, and the outcome of the visit program determines which of these to intersect with each incoming ray. Within a renderer, this allows to use multiple levels of detail in an object, depending on its distance to the camera.

- **Intersection**

  The intersection program is invoked to calculate the intersection between a ray and any general primitive object.

  In a typical raytracer, a low-level ray-triangle intersection program is combined with a higher-level ray-bounding box intersection algorithm for optimization. However, via programmable intersection code, OptiX also allows for more general shaped objects, such as procedurally generated surfaces.

- **Closest hit**

  This program is run whenever OptiX determines the closest-most hit-point between a ray and geometry. Closest hit programs are assigned to object materials according to the type of the incoming ray.

  Within a renderer, the closest hit program typically calculates the object color at the intersection point via texture lookup and recursive, secondary rays. The result then is stored in the frame buffer.

- **Any hit**

  An any hit program executes whenever a first-chance intersection between a ray and the environment is detected. This allows early-out optimizations for occlusion or line of sight tests such as on rays used for shadow calculations. As with closest hit programs, any hit programs are assigned to object materials in respect to the incoming ray type.

- **Miss**

  OptiX runs the miss program whenever a ray intersects with no scene object. A renderer can use this program to draw a scene background image. For each ray type, a single miss program can be defined on the OptiX context.

On GPU side, each programmable component is realized by a single CUDA function. To allow for multiple programmable components within the same CUDA compilation unit, all components are referenced from within host code by the name of their associated implementing function.

5.2.3 Building programmable components

Before use from within the OptiX API, any CUDA source files with programmable components first must be compiled to the intermediate PTX assembly text format via NVIDIA's nvcc compiler.

The corresponding compilation process is sparsely documented and difficult to comprehend. The OptiX documentation proposes the adaption of the CMake-based build process of the OptiX SDK sample applications for new projects. Yet, this workflow is not suited for the integration of shader-like, run-time programmable material components as desired for the raytracer implementation of this thesis.

The alternative approach of direct nvcc compilation suffers from a serious problem: If no explicit machine architecture for the resulting GPU-specific PTX code is requested on the command line, a seemingly unrelated compilation error occurs due to several undefined intrinsic functions. In other words, it is not sufficient to invoke the nvcc compiler with the ptx generation argument, but the arguments m32 or m64 are required as well.

Once compiled to PTX assembly sources, programmable components intuitively are recompiled into device-specific binaries from within host code. There are several competing recompilation options: The traditional CUDA API can be utilized, the OptiX API provides a convenience wrapper for PTX file recompilation, and there is a similar wrapper for in-memory PTX string representations. The last option will be employed by the later raytracer implementation to decouple resource loading from the actual file system.

5.2.4 Scene hierarchy

To achieve best performance, a raytracer kernel must utilize some sort of hierarchical optimization structure in the back-end for ray-triangle intersection tests.

In NVIDIA's OptiX API, the corresponding structure is not directly exposed on triangle basis. Instead, the API client must encapsulate all relevant objects into a logical scene hierarchy, similar to a traditional scene graph. This hierarchy needs to be aggregated from a set of fundamental node types that the OptiX API provides. Finally, opaque OptiX-internal optimization structures are attached to certain nodes within the hierarchy.

At the time of this writing, OptiX supports the following fundamental node types:

- **Geometry**

  A geometry node houses multiple geometry primitives of the same type — triangles in most applications.

  Primitive data, such as triangle vertices and indices, is not stored directly within the geometry node, but within a separate device-memory buffer. A device-side variable within the geometry node links to an appropriate data buffer.

  Apart from wrapping primitive data, a geometry node associates an intersection program and a bounding box program with the entire primitive set. The former

is invoked to test for ray-intersection with each primitive, while the latter is executed on each primitive to rebuild the OptiX-internal optimization structure whenever necessary.

Geometry nodes cannot be rendered on their own, but must be encapsulated into geometry instance nodes.

- **Material**

  A material node defines a series of closest-hit and any-hit programs for use during raytracing. For optimization, specific intersection programs can be assigned for each type of incoming ray. Materials are connected to one or more geometry instance nodes.

- **Geometry instance**

  A geometry instance encapsulates a single geometry node for placement in the virtual scene and binds a material node for use on this geometry. Multiple geometry instances may refer to the same geometry and material nodes to reuse primitive data. On a high-level point of view, an OptiX geometry instance node resembles a discrete object in a traditional scene graph.

- **Geometry group**

  A geometry group holds one or more geometry instances, and binds them to an OptiX-internal bounding box-based optimization structure.

- **Transform**

  A transform node applies a general 4x4 matrix transformation on a single subordinate node. This allows to place the same geometry or group at multiple places inside the scene. At the same time, transform nodes offer simple scene graph functionality, such as physical object relations and constraints. One should note that a transformation is optimized in that incoming rays are transformed, instead of contained geometry. Thus transformations may be modified without enforcing a rebuild of the optimization structure.

- **Group**

  General group nodes form the highest-level element in the bounding volume hierarchy, and contain arbitrary group, geometry group, transform, or selector nodes. Group nodes allow for complex instancing and object relation schemes. Intersection tests within any such complex scheme manually may be optimized by attaching higher-level optimization structures to certain group nodes.

- **Selector**

  Just as a group node, a selector node binds together a series of arbitrary other nodes. However, unlike the group node, a selector calculates intersections only on one of the contained objects. The object to use for a given incoming ray is determined by a visit program (see 5.2.2) attached to the respective selector instance.

- **Program**

  Certain programmable components — such as hit or intersection programs – must be attached to scene nodes, and thus become part of the node hierarchy. Note that any program may be shared and reused by multiple scene nodes.

- **Acceleration structure**

  Acceleration structure nodes are attached to either group or geometry group nodes. Each acceleration node holds a complete, opaque optimization data structure for all primitives contained in the entire associated subtree within the scene hierarchy. To accommodate for various scene compositions, acceleration structures may be customized in terms of their optimization algorithm. The next section provides in-depth information about acceleration nodes and their content.

Host-side management of scene objects is performed by a series of C helper functions within the OptiX API. There are, for instance, functions for object creation, object destruction, and the binding of programmable components.

The scene hierarchy compositions shown in figures 5.7, 5.8, and 5.9 provide examples for data sharing and optimization structure integration.

5.2.5 Acceleration structures

At certain points throughout the preceding elaborations, the OptiX-internal acceleration structures have already been named. It has been explained that such structures are attached as nodes within the OptiX-internal scene hierarchy.

The process that derives an acceleration structure from primitives within subordinate nodes has been mentioned already as well. For review, a bounding box program calculates axis-aligned bounding boxes for each primitive within a geometry instance on certain scene updates. The bounding boxes in turn automatically are inserted into higher-level acceleration structures alongside a reference to the encapsulated primitive. This improves performance for intersection tests. The acceleration structure is responsible for a mean $O(\log t)$ raytracing complexity, as opposed to $O(t)$ for a naive intersection algorithm (t is the number of triangles).

In the following, the acceleration structures themselves are now investigated.

The most noteworthy facet about the OptiX-internal acceleration structure is that it provides a choice of various algorithms suited for tree building and traversal. While this choice does not support the same level of customizability as the general programmable components, it still allows for adaption to different kind of scene primitives and scene animations. One must note the reference to the scientific state-of-art review, where the conclusion was drawn that multiple acceleration algorithms are required depending on the type of moving content.

On a conceptual level, OptiX separates between tree building and tree traversal algorithms. Both builder and traverser are separately assigned to each acceleration structure — with certain restrictions due to algorithm compatibility.

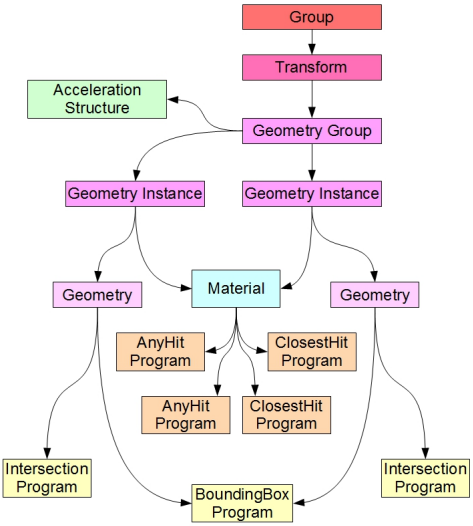


Figure 5.7: This OptiX hierarchy holds two geometric primitives that share their material and the bounding box program. The material itself holds a series of hit handling programs, depending on the actual type of incoming ray. A single acceleration structure is used for both geometry primitives. The primitives are grouped by a transformation node for efficient movement within the scene.

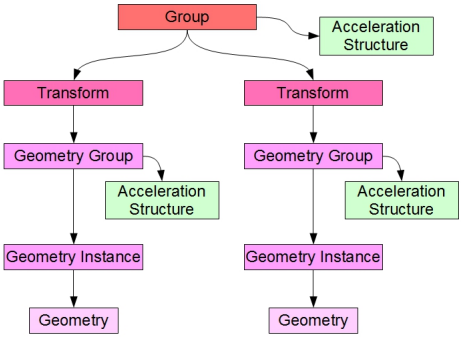


Figure 5.8: An example OptiX node hierarchy that demonstrates multiple concurrent acceleration structures at various levels of the hierarchy. Note that it is not possible to attach the acceleration directly to either geometry or geometry instance nodes — thus, a geometry group is required even for a single geometry. Materials and programs have been omitted for clarity.

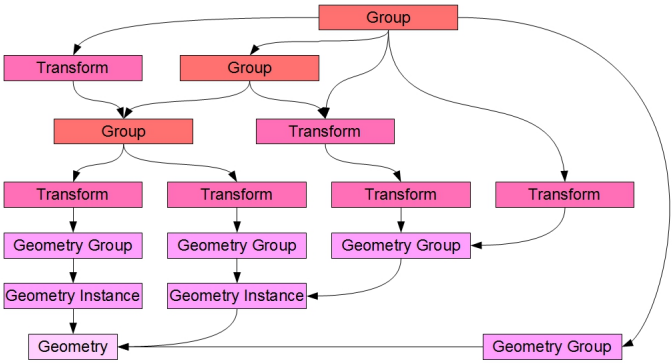


Figure 5.9: The OptiX scene system allows for the realization of complex instancing graphs by stacking transformation and group nodes atop of geometry instances and geometry objects.

Currently, OptiX comes with six different building algorithms, and three corresponding traversers.

The most noteworthy supported tree builders are the SBVH and LBVH algorithms. These already have been elaborated in their function and design within chapter 3. For review, the SBVH algorithm is suited for the generation of a static, high-quality bounding box tree in an offline process. Later on, the raytracer implementation will use this algorithm for all non-animated models. In contrast, the LBVH algorithm was designed for fast construction of a new bounding hierarchy from scratch. It is suited for dynamic geometry, but does not provide the same raytracing performance as the more thorough SBVH approach. The later raytracer component applies the LBVH algorithm to bone-animated meshes.

Apart from both bounding volume approaches, three traditional kd-tree representations and a dummy empty structure are available as well.

Traversers exhibit less variety: There currently is a traverser for kd-trees, another traverser for both of NVIDIA's proprietary tree solutions (SBVH, LBVH), and a final traverser that directly tests geometric primitives without any optimization.

This concludes the overview over the OptiX-internal scene management and raytracing functionality.

5.2.6 Data buffers

In the preceding, all structural input data for the raytracing process has been investigated. Yet structural data contains but high-level scene objects and programmable components. This brings up the question of how OptiX actually manages large-scale data such as primitive coordinates, input textures, or raytracing results.

As a solution, the OptiX platform offers typed, device-side data buffers layered atop CUDA's untyped device memory allocations. Data buffers allow for arbitrary input and output communications in between programmable components and the host system.

In detail, host-side code creates a new data buffer by an OptiX API function call. Thereafter, the size and element type — e.g. integer numbers or floating-point vector tuples — of the buffer are defined, and the buffer is ready for use.

Data buffers reside primarily on device memory. Thus, any host-side data access involves a mapping operation that creates a temporary copy of the buffer within host memory.

In contrast, device-side reading or writing on an existing data buffer does not require any explicit mapping operation. Yet, before access from a programmable component, the buffer must be bound to a device-side variable name from within host-side code. This process is described in the next section.

In respect to the original question, buffers are used for all three operations: Buffers store primitives, hold texture data, and simulate an output framebuffer.

The latter use case requires some further elaboration: As the OptiX platform has been developed for general raytracing operations, it does not provide implied access to any sort of default framebuffer. A naive approach thus creates an appropriate OptiX data buffer for storing the raytraced image. Once raytracing finishes, the completed image is copied back to host memory and blitted to screen from there.

Yet there is still room for improvement. The above approach involves a GPU-CPU-GPU round trip time that easily is avoided: Buffer objects from certain other GPU programming languages (CUDA, OpenCL) and APIs (DirectX, OpenGl) can be integrated into OptiX using a host-level function call. For instance, it is possible to bind a DirectX surface into OptiX and to use that surface as raytracing target. Then, the surface can be transferred to the actual window framebuffer by a hardware-optimized blit operation. Consequently, the round-trip is avoided.

The same data sharing system allows for many more potential use cases: Interoperability with an OpenGl rasterizer for shadow rendering using shared geometry resources, collaboration with texture units that have been initialized in DirectX, the raytracing of CUDA-calculated scientific data sets, ...

Sadly, the buffer sharing scheme is not fully implemented within OptiX yet. Apart from several buffer-related, non-reproducible crashes during the raytracer development, buffering currently does not support all buffer features from respective graphics APIs. It is, for example, not possible to use the current OpenGl framebuffer object. Instead, the legacy pixel buffer extension (which already has numerous issues on its own [BS06b]) has to be applied. These problems hopefully will be solved in future OptiX and graphics driver releases.

5.2.7 Device variables

As stated above, data buffers need to be bound to device-side variable names before use. Yet, the concept of bind-able device-side variables is not restricted to buffers,

but used for all small-scale communication between the host application and the device-side raytracing process.

In general terms, the host application specifies some variable value alongside the name of an associated device-side variable. On later GPU raytracing, existing and specially marked variables that match the indicated variable name are located within PTX device code — respectively within programmable components. Any such variables are initialized to the host-specified value.

The exact binding scheme is more sophisticated. To be specific, complicated binding and scoping rules are adopted by GPU variable lookup: On host-side, variables can be assigned to arbitrary scene nodes or to the root context. Once OptiX tries to resolve existing device variables against host-made bindings, only bindings on certain objects are considered. Further scoping rules indicate in which order objects are investigated in search for bindings. The lookup itself is dynamic, and depends on the current raytracing process. For example, even a single programmable component may have different variable mappings, depending on which parent object currently is under processing.

This scheme is different from most other GPU programming languages — typically, the host application queries a list of available device-side variables instead of independent variable specification.

Compared to the traditional approach, variable binding at first seems unreasonably difficult: The host code must explicitly set any desired variables before raytracing execution. Host-side access to variable content is performed through unchecked by-name declarations, and no variable-related errors are reported back to the client application.

However, there is some benefit to the OptiX binding scheme for device variables as well. In particular, the scoping rules allow to bind different variable settings on different objects. For instance, the global camera position and orientation may be bound to respective CUDA-side variables on the root context. These then are valid from within each programmable component. In contrast, the binding of a single texture buffer is reassigned on each geometry instance. Thus, the same closest-hit program looks up the output color in different textures depending on the current geometry object.

The advantages of variable binding are exploited in the later raytracer implementation to allow for a reasonable amount of data sharing within a dynamic scene structure.

5.2.8 Programming interface

Up to now, only general concepts within the OptiX platform have been discussed. On a more practically oriented point of view, the following section names and describes certain relevant functions from the client-side C API — respective the `optix.h` header.

In this context, one must consider that an in-depth tour over the entire API is outside the scope of this thesis. Instead, only certain noteworthy functions for controlling OptiX have been selected for review to give an impression of the API. The OptiX programming guide [NV11a] should be consulted for any further in-depth information.

The most important function within the host-side OptiX API probably is the actual invocation of the raytracing process:

```
RTresult rtContextLaunch2D
    (RTcontext context, unsigned int entry_point_index,
     RTsize image_width, RTsize image_height);
```

Parameters specify the OptiX context to work on, the initial ray generation program, and the extends of the CUDA thread grid — respective the size of the output image.

An appropriate OptiX context is created and destroyed by the next two functions:

```
RTresult rtContextCreate(RTcontext* context);
RTresult rtContextDestroy(RTcontext context);
```

With these two calls, an emerging concept within the OptiX C language API becomes evident: The object oriented, internal OptiX architecture has completely been mapped to C-style functions. As such, most functions are simulated member methods that carry the object type within function name, and expect an object instance within their first parameter. Return values are used to report back some of the errors that occurred during execution of the relevant function.

As further examples, the following calls set material-specific hit programs, create geometry instances, and declare global variables:

```
RTresult rtMaterialSetAnyHitProgram
    (RTmaterial material, unsigned int ray_type_index,
     RTprogram program);

RTresult rtGeometryInstanceCreate
    (RTcontext context, RTgeometryinstance* geometryinstance);

RTresult rtContextDeclareVariable
    (RTcontext context, const char* name, RTvariable* v);
```

Similar functions are used for control of the entire OptiX raytracing process — from acceleration structure selection, over buffer creation, to scene management.

Because of the repeated type specifiers within function names, the OptiX C API results in rather long and messy code.

In contrast to the C API, there is a much improved C++ API available as well. However, the C++ API has not been stable when work on this thesis began, and thus was not considered.

5.2.9 Multithreading capabilities

As mentioned in the state-of-art review in chapter 3, the later rendering component must be able to handle access from multiple independent threads without any harsh performance break-in. Since the rendering component eventually must call the C functions provided by the OptiX API, their multithreading behavior is relevant in this context as well.

Yet while OptiX makes many statements on threaded processing within the GPU, it sadly does not provide any explicit guarantees or safety for host-side threaded access. While experiments have shown the OptiX API to provide at least basic thread-safety, this cannot be used as groundwork for designing a stable client application without official confirmation.

Thus the later design of the rendering component explicitly needs to lock on the OptiX context and any attached OptiX objects. The global lock becomes especially relevant in consideration of performance factors: Any GPU-side scene update that is triggered by application logics only can be performed quickly when no rendering is currently underway. Thus, some intermediate decoupling mechanism has to be found to ensure fluid client logics. The respective decoupling strategy has been tightly integrated into a general interface for a multithreaded renderer, and is described in depth in chapter 6.

5.2.10 CUDA extensions

Apart from the C language API that is used to control the OptiX raytracing process from within host code, the OptiX platform integrates few new extensions and restrictions into the CUDA GPU programming language as well.

As with the OptiX C API, CUDA programs that use any OptiX functions must include the header `optix.h`. When used within CUDA, this header specifies all functions, defines, and types for use in OptiX programmable components.

The most important function within the CUDA-side OptiX API is the `rtTrace` call, which traces a ray through the scene hierarchy, starting at an arbitrary user-provided scene object.

Noteworthy defines include `RT_PROGRAM` and `rtDeclareVariable`. The former precedes a C function definition and marks the respective function as an entry point into one of the user-programmable OptiX components. The latter declares a global CUDA variable that is valid over an entire source code file.

On declaration, global variables within OptiX are associated with various semantics that define their program-wide behavior. For instance, a specific semantic is used to mark data for automatic binding to host-side variable settings — already discussed in 5.2.7.

Other semantics indicate that a single variable is present once for each parallel thread. Per-thread variables include the per-ray structures that carry ray input and output data. An often confusing specialty of global per-ray variables requires their redefinition within the scope of but certain calling functions. If this requirement is not strictly adhered to, undefined behavior occurs in form of rare bugs.

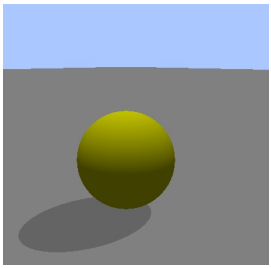


Figure 5.10: The image produced by the OptiX example application.

Types provided by the OptiX header mainly consist of opaque OptiX object handles and helper structures, like matrices or vectors. Yet, there also is the type `Ray`, which defines the core ray used in all raytracing operations.

In contrast to OptiX-specific CUDA extensions, the use of certain standard CUDA functions is prohibited within any OptiX program. For instance, CUDA thread synchronization with the `syncthreads` barrier conflicts with OptiX-internal ray scheduling logics and stalls the entire system.

5.3 Sample application

Now that all components of the OptiX platform have been described, the collaboration of those components is further detailed by an intuitively understood example application. The goal of the example application is to generate a raytraced, standalone image of a sphere-based geometry primitive. The resulting image is shown in figure 5.10.

One must note that — while modified at various locations — the example application follows both the OptiX tutorial [NV10] and the introductory chapter within the OptiX programming guide [NV11a]. However, the consequent steps are presented in a more appropriate order, their relationship is explained in more detail, and feature usage has been reduced for clarity.

At first, the CUDA-side implementation of the example application will be discussed. The global variables used both for communication from host to device and for communication in-between programmable components are introduced. Then, both global and per-object programmable components are described. Finally, the main application — including context initialization, scene management, and raytracing — is elaborated.

5.3.1 Device-side variables

At the beginning of the main CUDA file, all global variables for communication between host and device code are declared: The pixel currently under processing, the camera position and orientation, the scene root object, and the output pixel buffer.

```
rtDeclareVariable(uint2, pixel_index, rtLaunchIndex, );

rtDeclareVariable(float3, camera, , );
rtDeclareVariable(float3, looking_dir, , );

rtDeclareVariable(rtObject, scene_root, , );

rtBuffer<uchar4, 2> output;
```

Variable declaration is usually performed with the `rtDeclareVariable` define. Here, the first parameter indicates the variable type. Both `uint2` and `float3` are vector types that are intrinsically defined by the CUDA language. `rtObject` is an OptiX-specific handle to any scene object. The second parameter sets the variable name that is used from within both device code and host code. An optional third parameter — only `rtLaunchIndex` is present in this example — defines the variable semantic.

In the case of rtLaunchIndex, the per-thread variable `pixel_index` is automatically initialized to the respective thread index within the thread grid. In the example application, the thread index directly corresponds to the coordinates of the associated on-screen pixel.

The fourth parameter can be assigned to per-variable string annotations, and remains unused within the entire thesis.

On a final note, the atypical syntax of missing parameters within rtDeclareVariable attributes to the fact that rtDeclareVariable is a define, and not a function.

Unlike normal variables, buffer objects are not declared as opaque handles, but as template-like container instances with specified encapsulated element and tuple sizes. Still, the output buffer must be bound to an appropriate data buffer instance by name from within host code just like the remaining globals.

Similar to the preceding segment, the following code excerpt defines another set of global variables:

```
rtDeclareVariable(optix::Ray, ray, rtCurrentRay, );

struct camera_ray_data_type { uchar4 color; };
rtDeclareVariable
    (camera_ray_data_type, camera_ray_data, rtPayload, );

struct shadow_ray_data_type { unsigned char hit; };
rtDeclareVariable
    (shadow_ray_data_type, shadow_ray_data, rtPayload, );

rtDeclareVariable(float3, hit_point, attribute hit_point, );
rtDeclareVariable(float3, hit_normal, attribute hit_normal, );
```

Yet, variables this time are used only for GPU-internal communication. The ray variable with the `rtCurrentRay` semantic stores the per-thread ray that currently is under processing, including its starting point and direction. It is initially set by the ray generation program for primary rays or by a closest-hit program for recursive secondary rays. Thereafter, intersection programs access the stored ray data for testing against primitives and acceleration structures.

All variables with the `rtPayload` semantics hold per-ray data that is returned by later hit programs: Camera rays store an output color, while shadow rays contain a hit-testing result.

Note that data payload variables are defined globally for both camera rays and shadow rays, independent of the actual program that currently is under compilation. This is a restriction enforced by the underlying graphics hardware, where per-thread globals must be declared within each compilation unit.

Finally, the `hit_normal` variable is utilized for communication between the intersection program and the hit program: The normal calculated by the intersection program is returned to the hit program for reuse in shading. This behavior is defined by the semantic type `attribute variable_name`, a legacy of the underlying GPU architecture.

5.3.2 Global programmable components

The ray generation program marks the entry point into the programmable CUDA pipeline. Namely, the raygen function is executed in parallel once per pixel of the image and stores a respective output color.

```
RT_PROGRAM void raygen()
{
    float sx(output.size().x),
          sy(output.size().y);
    float aspect(sy / sx);

    float rx(  (float)pixel_index.x / sx * 2 - 1),
          ry(-((float)pixel_index.y / sy * 2 - 1) * aspect);

    float3 front(normalize(looking_dir)),
           right(cross(make_float3(0, 1, 0), front)),
           up(cross(front, right));

    float3 ray_dir
        (normalize(right * rx + up * ry + front));

    Ray ray(camera, ray_dir, 0, 0.0001f, RT_DEFAULT_MAX);
    camera_ray_data_type camera_ray_data;
    rtTrace(scene_root, ray, camera_ray_data);

    output[pixel_index] = camera_ray_data.color;
}
```

The finer-grained procedure is intuitively understood: At first, a ray through the output pixel of the current thread is constructed. A ray data structure specifically suited for primary camera rays is initialized, and the rtTrace function is invoked to process the appropriate ray against the global scene root.

On return, ray processing has filled the color field of the ray data structure with the color of any hit object or with the background color. Consequently, the respective color is stored in the output image.

The most noteworthy aspects of this function are safety-related: An integral ray type is specified independently of the associated data type within the third construction parameter of the ray object. In the example application, ray type 0 has been defined as a camera ray, and type 1 corresponds to shadow rays. The integral ray type in turn is used to bind corresponding closest-hit and any-hit handler programs from within host code. In the example, the closest-hit program — which works with camera ray data — is bound to ray type 0. The any-hit program — which expects shadow ray data — is bound to ray type 1. However, there is neither a compile time nor a runtime check whether a matching data structure type for the current ray type has been passed into rtTrace. Such mistypings lead to hard-to-find bugs, including random operating system freezes. There currently is no workaround except for the use of defines or constants instead of integer ray types so that respective errors become more obvious.

Further care is required to ensure that the names of local variables for ray data and the ray itself match with those initially assigned by rtDeclareVariable. Otherwise, access to these structures from within other programmable components yields unspecified behavior.

Similar to the global ray generation program, the miss programs work independent of any scene primitive. Camera rays that miss all scene objects return the color of the background to the ray generation program. Likewise, shadow testing rays indicate on miss that no occluding object was found.

```
RT_PROGRAM void camera_miss()
    { camera_ray_data.color =
            make_uchar4(170, 200, 255, 255); }

RT_PROGRAM void shadow_miss()
    { shadow_ray_data.hit = 0; }
```

Particular attention is directed to the fact that the ray and ray data structures are used within these and all later programmable components without any declaration or definition. Ray structures must only be defined globally and once more within the scope surrounding the associated rtTrace call.

5.3.3 Geometry-based programs

The bounding box of the square-shaped floor object is realized by another custom component:

```
RT_PROGRAM void floor_bounding(int index, float result[6])
{
    result[0] = result[2] = -100.f;
    result[3] = result[4] = 100.f;

    result[1] = -1.01f;
    result[4] = -0.99f;
}
```

The signature of bounding box methods is rigidly defined by OptiX: The index input parameter references the current primitive within any geometry for which to rebuild the boundings. The result output parameter is filled with the bounding box coordinates of the respective primitive.

More complicated geometry objects potentially use the index to look up data for an appropriate primitive within an OptiX data buffer. As there is but a single floor in the example application, this lookup is omitted in favor of a constant bounding box.

Once the floor bounding box generated by the above program is hit by a ray within ray tracing, the intersection program is invoked:

```
RT_PROGRAM void floor_intersection(int index)
{
    float3 n = make_float3(0, 1, 0);
    float d = 1;

    float a = -dot(ray.direction, n);
    if (fabsf(a) < 0.0001f) return;

    float do = dot(n, ray.origin) + d;
    float t = do / a;

    float3 p = ray.origin + t * ray.direction;

    if (p.x > -100.f && p.x < 100.f &&
        p.z > -100.f && p.z < 100.f)
    {
        if (rtPotentialIntersection(t))
        {
            hit_point = p;
            hit_normal = n;
            rtReportIntersection(0);
        }
    }
}
```

The intersection program for the floor primitive calculates the intersection point between the ray and the floor square by few linear algebra formulas.

Intersection testing again takes an index parameter that identifies the respective primitive, but the example continues to ignore this parameter.

Intersections are reported back to OptiX by two different calls from the CUDA-side API: `rtPotentialIntersection` and `rtReportIntersection`.

The former queries whether the calculated intersection point is closer to the ray's origin than any previously found intersection. The latter is executed to trigger the actual any-hit program.

In between both calls, the intersection program is allowed to appropriately adjust communication attributes that are passed to the later any-hit and closest-hit handlers. In this context, note that the closest-hit handler is not directly invoked by the intersection program, but by OptiX on completion of the entire tracing process for a single ray.

As the floor object does not drop a shadow within the scene, it does not have an associated any-hit handler. However, its closest-hit handler — responsible for setting the color returned by camera rays — is presented in the following code fragment:

```
RT_PROGRAM void floor_closest_camera_hit()
{
    Ray ray(hit_point, normalize(make_float3(1, 1, 0)),
                1, 0.0001f, RT_DEFAULT_MAX);
    shadow_ray_data_type shadow_ray_data;
    rtTrace(scene_root, ray, shadow_ray_data);

    if (shadow_ray_data.hit)
        camera_ray_data.color = make_uchar4(100, 100, 100, 255);
    else
        camera_ray_data.color = make_uchar4(128, 128, 128, 255);
}
```

While most of the above code is intuitively understood, particular attention is directed to the recursive raytracing process that generates the sphere's shadow: Another ray primitive is instantiated, starting at the hit point calculated by the intersection program.

The ray primitive must go by the variable name of the global ray. Yet, the ray type is set to 1, corresponding to a shadow ray, and an appropriate data structure is passed into the rtTrace invocation.

On return from shadow recursion, the result data is used to attenuate the floor color within shaded regions.

Both sphere bounding box and sphere intersection programs are quite similar to respective programs for the floor object, and thus omitted for brevity here.

The CUDA-side code review ends with the sphere's closest-hit and any-hit handlers:

```
RT_PROGRAM void sphere_closest_camera_hit()
{
    float b = (hit_normal.y > 0) ? (hit_normal.y * 128 + 64) : 64;
    camera_ray_data.color = make_uchar4(b, b, 0, 255);
}

RT_PROGRAM void sphere_any_shadow_hit()
{
    shadow_ray_data.hit = 1;
    rtTerminateRay();
}
```

The former uses the normal already calculated by the sphere's intersection program to shade the sphere's surface. The latter reports back an occluding object to the shadow testing ray sent by the floor's closest-hit program. The closing call to `rtTerminateRay` indicates that further collisions of a shadowing ray may be disregarded. This is a vital optimization employed in the later, main raytracer implementation.

Finally, the complete CUDA source containing all preceding code excerpts must be compiled with nvcc. The building process has already been described in 5.2.3, and is not repeated here.

5.3.4 Main application

The main application starts with the creation of the OptiX raytracing context:

```
RTcontext context;
rtContextCreate(&context);
```

Once an OptiX context has been created, global programmable components are compiled from PTX files and applied to the raytracer setup:

```
RTprogram raygen;
rtProgramCreateFromPTXFile
        (context, "OptixExa.ptx", "raygen", &raygen);
rtContextSetEntryPointCount(context, 1);
rtContextSetRayGenerationProgram(context, 0, raygen);
rtContextSetRayTypeCount(context, 2);

RTprogram camera_miss;
rtProgramCreateFromPTXFile
        (context, "OptixExa.ptx", "camera_miss", &camera_miss);
rtContextSetMissProgram(context, 0, camera_miss);

RTprogram shadow_miss;
```

```
rtProgramCreateFromPTXFile
        (context, "OptixExa.ptx", "shadow_miss", &shadow_miss);
rtContextSetMissProgram(context, 1, shadow_miss);
```

Most of the above code is intuitively understood. At first, the ray generation program is retrieved from disk, recompiled, and defined as entry point 0 for the raytracing process. Thereafter, miss programs are installed depending on their integral ray type identifier.

Integral ray type identifier have already been introduced and criticized within the GPU code elaborations. Yet, even within host code, these identifiers are error-prone: If the total number of ray types has not been set correctly before ray-specific programs are bound, the respective binding statements are silently ignored.

This has a vital implication on the later full raytracer implementation, as the runtime extension of ray types requires a remapping of any ray-specific program bindings over all OptiX objects.

Apart from binding the global programs, the host-side example application also creates the output buffer and applies initial values to the global GPU-side variables:

```
RTvariable camera_var;
rtContextDeclareVariable(context, "camera", &camera_var);
rtVariableSet3f(camera_var, -5, 2, -5);

RTvariable looking_dir_var /* ... same as above ... */

RTbuffer output;
rtBufferCreate(context, RT_BUFFER_OUTPUT, &output);
rtBufferSetFormat(output, RT_FORMAT_UNSIGNED_BYTE4);
rtBufferSetElementSize(output, 4);
rtBufferSetSize2D(output, sizex, sizey);

RTvariable output_var;
rtContextDeclareVariable(context, "output", &output_var);
rtVariableSetObject(output_var, output);
```

The code for buffer creation and variable access seems mostly self-explanatory. Attention should only be directed to the variable binding process: Variable objects are matched to GPU-side variables by their string-based name. For instance, the string identifier for the camera variable declared on host-side must match up with the respective CUDA variable name. While similar to by-name matching of programmable component functions, mismatches in variable naming are not reported back to the client application. On the contrary, GPU-side globals then simply remain uninitialized.

5.3.5 Scene management

After the example application initialized all global programs and variables, code flow continues with management of the example scene.

In the following code excerpt, the OptiX geometry, material, and geometry instance objects for the sphere representation are created:

```
RTprogram sphere_bounding /* ... load from PTX ... */;
RTprogram sphere_intersection /* ... likewise ... */;
RTprogram sphere_closest_camera_hit /* ... */;
RTprogram sphere_any_shadow_hit /* ... */;

RTgeometry sphere_geometry;
rtGeometryCreate(context, &sphere_geometry);
rtGeometrySetBoundingBoxProgram
    (sphere_geometry, sphere_bounding);
rtGeometrySetIntersectionProgram
    (sphere_geometry, sphere_intersection);
rtGeometrySetPrimitiveCount(sphere_geometry, 1);

RTmaterial sphere_material;
rtMaterialCreate(context, &sphere_material);
rtMaterialSetClosestHitProgram
    (sphere_material, 0, sphere_closest_camera_hit);
rtMaterialSetAnyHitProgram
    (sphere_material, 1, sphere_any_shadow_hit);

RTgeometryinstance sphere_instance;
rtGeometryInstanceCreate(context, &sphere_instance);
rtGeometryInstanceSetMaterialCount(sphere_instance, 1);
rtGeometryInstanceSetMaterial
    (sphere_instance, 0, sphere_material);
rtGeometryInstanceSetGeometry
    (sphere_instance, sphere_geometry);
```

Once more, one should take note of the by-index binding of closest-hit and any-hit programs to associated ray types within the sphere material.

Corresponding code for the floor object is mostly identical, and has been omitted for brevity.

The final part of scene management handles the construction of a root grouping object and the assignment of an optimization structure. As a last step before the actual raytracing process, the GPU-side scene root variable is bound to the previously generated root object.

```
RTacceleration scene_accel;
rtAccelerationCreate(context, &scene_accel);
```

```
rtAccelerationSetBuilder(scene_accel, "Lbvh");
rtAccelerationSetTraverser(scene_accel, "Bvh");

RTgeometrygroup scene_root;
rtGeometryGroupCreate(context, &scene_root);
rtGeometryGroupSetChildCount(scene_root, 2);
rtGeometryGroupSetChild(scene_root, 0, floor_instance);
rtGeometryGroupSetChild(scene_root, 1, sphere_instance);
rtGeometryGroupSetAcceleration(scene_root, scene_accel);

RTvariable scene_root_var;
rtContextDeclareVariable(context, "scene_root", &scene_root_var);
rtVariableSetObject(scene_root_var, scene_root);
```

With the exception of the by-name selection of acceleration structure builder and traverser, the above source fragment is straightforward, and remains without further explanation.

5.3.6 Raytracing

Once all programs, all parameters, and the virtual scene have been initialized, the GPU-accelerated raytracing process is triggered:

```
rtContextLaunch2D(context, 0, sizex, sizey);
```

The committed dimensions indicate the number of threads spawned — and thus the number of times that the ray generation program is executed. Additionally, the context-wide specifier 0 for the initially registered ray generation program must be provided.

Once `rtContextLaunch` returns, the output buffer has been filled with pixel data for the entire example scene. As this buffer resides on device memory, it must be mapped to a host-side pointer before access from within the example application:

```
void *output_mapped;
rtBufferMap(output, &output_mapped);
```

Buffer mapping potentially involves an internal device-to-host CUDA copy operation. Benchmarking has indicated this to be a major performance bottleneck for interactive applications, and thus the later Simulator X raytracer component chose a different approach.

Yet the resulting, mapped pointer can be arbitrarily used by the client application. For instance, the application might show the resulting image in a window or save all data to an image file.

After the example application has finished access to the rendered image, the buffer is unmapped from host memory, and the OptiX raytracing context is destroyed:

```
rtBufferUnmap(output);
rtContextDestroy(context);
```

Context destruction automatically cleans up any associated OptiX objects and programs, as well as any device-side memory allocations. Thus, no further per-object destruction is required.

As a final caveat, it should be noted that context destruction regularly failed with an exception due to access violation on several test machines with various graphics boards and NVIDIA driver versions. Thus, context destruction has not been compiled into the example binary on the delivery disc. Albeit the operating system should clean any remaining OptiX resources on program termination, small memory leaks on the graphics hardware due to omitted destruction have been observed.

This concludes the tour over the example application.

5.3.7 Conclusion

As seen from the example application, the OptiX platform offers more of a raytracer construction kit than an actual renderer implementation. This allows maximum flexibility and intuitive extensibility for potential clients. For instance, the example application required but few additional lines for the integration of shadow detection.

Yet, flexibility comes at a certain cost. In relation to traditional rendering APIs, the OptiX API requires more complicated boiler-plate code both on CPU and GPU side even for simple scenes. This is further aggravated by the inconsistent GPU-side syntax and semantics, which necessitates both knowledge about the CUDA language and the original shader influences. The latter statement is specifically aimed at the mixed use of global, local, and parameter variables, and at the dual by-name and by-index binding schemes. Finally, error checking and overall stability issues still persist — although OptiX arrived at an official version 2 release half-way through development of this thesis.

Within this thesis, most of the above problems are dealt with alongside the raytracer component development: All OptiX functionality will be wrapped up into a more accessible, less error-prone, and pure rendering-oriented interface in the next chapters.

6 Renderer interface

This chapter elaborates the interface of the raytracing component that was developed over the course of this thesis.

At first, certain requirements are reviewed that result from preceding state-of-the art studies, Simulator X specialties, and traits of the OptiX API. These form the foundations of the later interface design.

The remaining chapter follows a top-down route in explaining the interface architecture. Initially, an overview over the entire rendering process is presented. The following sections diverge into subordinate modules of the renderer. The resource management strategy of the renderer is detailed, and scenes are composed from resource instances. Scenes in turn are enqueued into a command buffer to control the rendering process. Thereafter, guidelines on the implemented multithreading requirements and blocking behavior are defined in the context of application timing. Finally, design alternatives that have been considered and dismissed during the initial design phases are reviewed.

The chapter ends with a short recap of the entire interface design.

6.1 Requirements

The first relevant design requirement concerns the level of abstraction within the overall renderer architecture: Instead of a hard-coded, client-exposed integration with the OptiX platform, it was decided to require but a general-purpose rendering interface. Motivation for this requirement was the concise, API-independent interface design of the Ogre3D engine.

There are several benefits to a general layer of abstraction that mediates between the client and an actual API-specific back-end. For instance, a general interface must be reduced to most noteworthy, shared aspects of the rendering process. This encourages the development of slim and intuitively used rendering components. Furthermore, a general rendering interface promises a seamless switch of the implementation from a conventional rasterizer to a raytracing algorithm, both for performance comparison and future extensibility. Finally, once in place, a standardized rendering interface abstracts even from the current jVR-specific component within Simulator X. In turn, semantic coupling between client components and the graphics system can be reduced.

Another design requirement concerns multithreading functionality. In particular, the rendering component was required to support both client-side and internal worker threads of various types. Such support is hard to retrofit into existing systems, and comes at almost no cost if done correctly from the start. Even in the context of a Simulator X rendering component, where there typically is but a single thread within each actor, there are advantages to a multithreaded rendering kernel — for instance, in terms of a subordinate resource loader or particle system actor.

Finally, certain performance factors have been a requirement for the interface design as well. While early optimization in popular belief is despised as the "root of all evil", this does not apply to general design strategies. In contrast, a performance

bottleneck induced by the fundamental architecture of a system may even require a total rewrite on deferred optimization.

On review of all three design requirements, one must note that these interact with each other and formulate partially exclusive conditions. For example, multithreading support must not be exposed on the client-side interface via unintuitive, mandatory locking before certain operations. At the same time, multithreading support must not be designed as to infer too much of a performance burden. Likewise, performance optimization must not violate intuitive usability of the general interface.

For the resolution of potential conflicts, the above introduction order of requirements has also been used in architecture priorities: The utmost goal was an universal and intuitive client-side interface, followed by multithreading support, and performance considerations on third place.

6.2 Rendering process overview

The preceding section named an universal, slim, and intuitive client-side interface as the most important requirement of the renderer design. To provide this interface, the core renderer architecture allots for several small functionality modules that transparently collaborate with but few other modules — a contribution to the cohesion and coupling concepts within Simulator X, albeit at a lower level.

In a typical scenario, the client applications create, use, and destroy functionality modules in a well-defined order. At first, a client application needs to create the actual renderer component. Within this thesis, the renderer component is also referred to as a rendering context, realized by the `RenderContext` interface and later implementing classes.

Once an appropriate RenderContext has been created, the client mostly communicates with the RenderContext via the creation, update, and destruction of render-side objects. Render-side objects represent data that is shared between the client and the renderer, but is held in a format dependent on the implementation of the RenderContext. Example render-side objects are textures, meshes, cameras, or scenes. Further utility modules, such as a resource cache module, help the client with the management of render objects.

To show the current state of one or more virtual scenes after any updates, the client must trigger the actual rendering process. For this, a set of rendering commands is collected within client logics. Commands contain viewer information, scene selection, and additional rendering configuration. The command set is passed to the RenderContext for display. Again, certain helper modules, like a separate rendering thread instance, support this stage of the rendering process.

Finally, the client needs to shut down the application at one point or another. Thus, all shared render objects need to be released on client-side. Only then should the RenderContext itself be destroyed.

Figure 6.1 provides a more abstract overview over the relations between the client application and various components of the renderer interface.

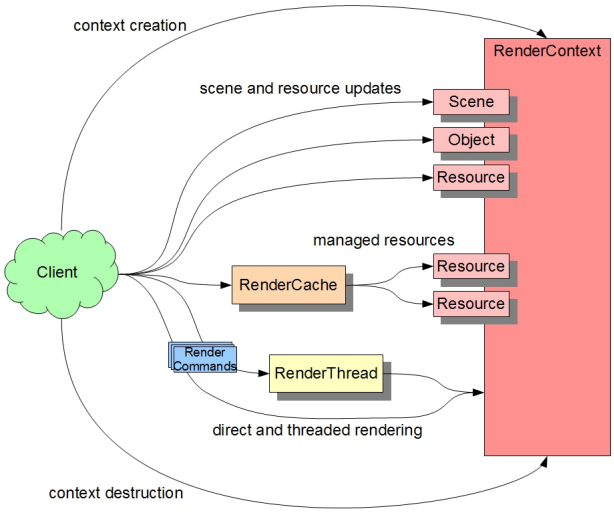


Figure 6.1: Abstract overview over collaboration between a client application and the renderer interface.

This concludes the overview over the entire rendering process. In the following, certain submodules of the above process are re-investigated in detail. In particular, this refers to various render object types, to the collection of commands that controls the actual rendering, and to the standalone auxiliary modules.

6.3 Render-side objects

In the preceding summary of the rendering process, render-side shared objects have been introduced as the most important communication channel between client and renderer: Render objects such as textures or geometry instances are used to compose the virtual scene that is later on shown by the application. The following text provides more in-depth information about render-side objects, their life-cycle and their use within client applications. The RenderObject interface is derived from the life-cycle, and the two major functional varieties of RenderObjects are introduced.

6.3.1 Object life-cycle

From the point of view of the client application, render-side objects exhibit the following life-cycle:

- **Creation**

Creation of render objects follows a run-of-the-mill factory pattern, where the RenderContext acts as the factory instance for a concrete type of render object. A shared reference to the new object instance is returned to the client.

Shared references are used to control the remaining object life-cycle to ensure correct object release. At the same time, shared references enable sharing of a single object even over multiple users within both the client application and the rendering subsystem — even over users within different threads.

- **Life**

  After object creation, the client is responsible for holding onto at least a single copy of the returned reference to the render-side object as long as the object is required.

  In a standard use case, a client now regularly modifies render objects to update and animate the virtual scene. At one point or another, the client might query object states as well.

  As a preview on later multithreading guidelines, any state-changing or state-querying operations on render objects require particular design care to achieve concurrency-free behavior and high performance within threaded applications.

  The respective solution mostly involves intermediate buffering of incoming state changes. Buffered changes are only applied to the render object contents on certain synchronization events that are triggered by either the RenderContext or client logics. Due to its importance, buffering must already be considered in the initial render object design.

- **Release**

  Once the last client-side shared reference to a render-side object (including any client-accessible reference to said object within any other object) has been released, the RenderContext is free to destroy the object at any later time.

  The delay between client-side reference release and actual destruction of a render-side object is required to accommodate for later multithreading. In particular, an object still may be in use from within a separate rendering thread even though no client-side references remain. Destruction of any orphaned render-side objects may be enforced by certain RenderContext operations such as purge or forced synchronization. These are detailed at a later point.

  Finally, one should note that cyclic, shared object dependencies are to be avoided to ensure correct destruction. This implies that render-side objects do not carry an owning reference back to the RenderContext to avoid shutdown problems because of leaked object references. Yet this also means that the RenderObject interface must provide a detach function to tell any object that its associated RenderContext has been destroyed prematurely. This allows the object to release any actual render-side data and fall back to a safe default state.

With synchronization and rendering, the lifetime and usage of a single render-side object might proceed as depicted in the sequence diagram of figure 6.2.

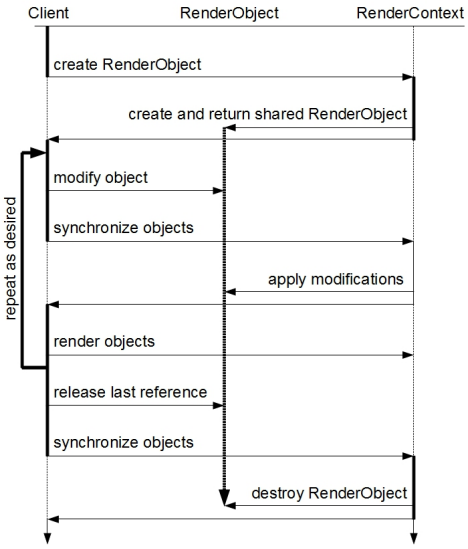


Figure 6.2: Sequence diagram for RenderObject usage from creation over update and rendering cycles to destruction.

6.3.2 RenderObject interface

The above life-cycle induces a basic `RenderObject` interface and associated functions:

Function	Description
RenderObject—	
create	Default-initialize object.
modify	Enqueue object state change.
query	Query object state.
synch	Apply any pending state changes.
release	Release a single client-side shared reference, objects are not destroyed immediately.
detach	Detach leaked object from destroyed context.
destroy	Destroy object during context-wide synch or purge.

In this context, any operations that involve the object's state — buffering on modification, query, and synchronization — exhibit a rather free functional specification: The actual buffering scheme is not restricted. Such refinements are provided by functional overrides but within derived interface types.

6.3.3 RenderObject varieties

Concrete render-side objects come in two major interface varieties which derive from the basic RenderObject interface:

- **RenderResource**

  On the one hand, render-side resources capsule large-scale binary data. BLOBs — or binary large objects — is a term often used here. Example resources are GPU-stored textures or triangle meshes. Resources are represented by the subordinate class hierarchy rooting at the `RenderResource` interface.

- **RenderPuppet**

  On the other hand, render-side puppet objects represent remote controlled objects in the virtual, interactive scene. Typically, puppet objects combine one or more resource objects into a single physical instance. For instance, a character puppet might link to the appropriate triangle mesh resource and the associated texture resource. However, there also are puppets with more abstract semantics, such as scene cameras or light sources. Even the virtual scene itself is considered a render-side puppet to allow for recursive scene compositions. The base class for all puppets is the `RenderPuppet` interface.

Both RenderResources and RenderPuppets are detailed alongside their interfaces and functional specialties in the following sections.

6.4 Resource management

This section explains the interface devised for renderer-side resource management. For review, render-side resources are large-scale binary objects (i.e. images, geometry data, shaders) that are managed in an implementation-specific storage format within the rendering component for performance reasons.

At first, the RenderResource interface itself and its functional characteristics are discussed in detail. Alongside this discussion, all concrete resource types derived from the RenderResource interface are presented.

Thereafter, the management of resources is discussed. Intuitive and safe resource management is a difficult topic, mainly because there are many different requirements dependent on the later client application. Thus, a wide spectrum of use cases is defined that the later resource management should comply to. Finally, the actual management strategy is chosen.

6.4.1 RenderResource interface

As stated in context of the RenderObject interface, the actual buffering behavior of state-changing operations (state modification, state queries, and synchronization) is defined but by lower-level interfaces. Thus, RenderResource characteristics are now analyzed to derive an appropriate interface refinement.

The most important common ground of all resource operations concerns the amount of data involved. As render-side resources hold large-scale data, appropriate operations typically move large amounts of data as well. Thus, a simple two-state double-buffering scheme — one state for a single, current renderer and another state shared by all client threads — becomes infeasible.

Instead, a move semantic has been integrated as the resource buffering strategy. Any resource operation that attempts to modify the resource state buffers appropriate results within the resource. On state synchronization, the buffer is copied into the state of the resource. Thereafter, the buffer is not required anymore, and thus is discarded to conserve memory.

For example, if a texture resource is retrieved from disk, the resource is not changed immediately. Instead, bitmap contents are read and stored in intermediate memory within the texture object. Once state synchronization occurs, the in-memory image is copied to the graphics hardware and used in consequent rendering. Finally, the in-memory image is released again to conserve host memory.

As incoming, buffered resource modifications are discarded on synchronization, queries cannot rely on the presence of any CPU-side resource representation. Thus, it has been decided to always fetch resource data from the actual resource state on any incoming queries.

The following table recapitulates the functional overrides within the RenderResource interface:

Function	Description
RenderResource—	
modify	Create new buffer with incoming data.
query	Retrieve render-side data directly.
synch	Apply modification to render-side state, then drop client-side state to conserve memory.

6.4.2 Resource types

All actual resource types derive from the RenderResource interface. Within the current concept, the following resources have been included:

- **Texture**

  A Texture resource corresponds to a render-side stored bitmap image.

- **Shader**

  The Shader interface represents some programmable component of an object's surface. This includes procedural textures and special material effects.

- **Material**

  A Material resource binds together a Shader and a Texture resource, and describes the look of a scene object.

- **Model**

  Model resources store a potentially animated triangle mesh, associated bones, skeletal poses, texture coordinates, vertex normals, and other auxiliary data. Additionally, they link their contents to a series of corresponding Material resources.

Note that certain resources contain owning references to other resources to allow for reuse of shared assets. However, there is no cycle in the resource dependency graph.

Each resource type is implemented anew alongside each RenderContext implementation. Reimplementing in this respect still does not necessarily imply any code redundancy, as each context implementation has its own internal storage format that is optimal for the context's rendering API. Certain components that are shared between resources or implementations later on can be decoupled by aggregation or composition patterns to maximize code reuse and minimize redundancy.

On client-side, the actual resource implementation is hidden via the RenderContext factory methods and the base resource interfaces.

While each resource type has an abstract interface of its own, a complete enlistment of these exceeds the scope of this thesis. In particular, each resource type only diversifies on the state change, synchronization and query concept for basic resources — new concepts are not introduced. As a sole example, the following table provides concrete functions for texture-typed resources:

Function	Description
Texture—	
set	Copy in-memory image into state buffer.
load	Copy on-disk image into state buffer.
query	Copy render-side image into client representation.
synch	Apply state buffer contents to render-side image, release state buffer thereafter.

Corresponding explanations for other resource types may be found within the auto-generated Doxygen documentation that accompanies the source code delivery.

6.4.3 Resource management requirements

As stated above, there are many concurrent requirements that may be impeded on a resource management module for a rendering component, depending on the actual client application. The following three example use cases present a broad spectrum of potential client applications:

- **Use case 1**

  For instance, a basic rendering client application desires but a slim resource management interface. There are few resources, all of which are loaded into

main or graphics memory on program startup. Each of the resources comes from a single on-disk file that is intuitively specified on resource creation. At fixed points throughout code — such as due to an environment change — all resources are dropped, and new resources are retrieved from disk again. This use case matches up with the proof-of-concept applications available for the Simulator X platform.

- **Use case 2**

  On the other extreme, consider a large-scale world simulation, such as provided by MMORPGs like the popular World of Warcraft. Here, the entire world does not fit into memory at once. Intermediate loading pauses are not acceptable, as these disrupt both the simulation flow for the local client, and the feeling of immersion into a boundless virtual world. Thus, an in-situ resource caching scheme has to be devised. To ensure steady logics processing, this caching scheme furthermore needs to be outsourced to an extra thread.

- **Use case 3**

  Finally, a scientific visualization application may not use file-based resources at all. Instead, all render-side resources are generated from an in-memory representation of a large problem working set.

The current resource management design has been chosen as to allow the maximum flexibility, without restrictions on either of these client-side use cases.

6.4.4 Resource management strategy

Resource management requirements are vastly different among the above use cases. While one application requires near to no resource management, another relies on heavy management logics with constant resource loading and unloading.

This conflict of interest cannot easily be solved by integrating resource management into the renderer component. Each client would be affected by the chosen resource management strategy, and undesired coupling would result.

Instead, it is advisable to create an optional, plug-able resource management component. In this design, the core RenderContext interface — available even without any resource manager instance — just is responsible for creation and destruction of plain resources. The actual resource lifetime management is handled either directly by client-side shared pointers, or by the plug-able resource manager. Use case 3 already is satisfied by these options.

Through different resource managers, in turn, different resource management strategies can transparently be realized. Within the current design, one example resource manager component is implemented in the form of the `RenderCache` interface. This component allows for many of the most typical resource operations, and thus benefits both use case 1 and use case 2.

The most noteworthy feature here is the binding of shared resource references to resource names: Resources themselves have no concept of any associated file name, mainly since there are types of resources — procedurally generated textures, or

runtime-built materials — that do not have an associated file. Instead, file-based resources may be requested at a RenderCache by their file name. The resource cache in turn looks for the resource within a filename lookup map. If the cache contains no resource with the provided filename, it creates a new resource at the RenderContext factory. Then, the resource cache triggers resource loading from the client-provided file, and on success attaches the completed resource to its internal lookup structure. Finally, the RenderCache returns the shared resource reference to the client.

This setup even allows for the optional load-time resolution of shared resource dependencies. For instance, two material resources might reference the same texture file. Yet there should only be a single, shared texture instance within the application to conserve memory and improve load times. This is achieved by integrating the RenderCache into resource loading code: Whenever a RenderCache is provided alongside a resource loading operation, any dependencies are recursively resolved by the use of the resource cache.

On the downside, by-name dependency resolution obviously couples a dependency on the RenderCache into any resource-specific loading code. Albeit this is generally undesired, the appropriate integration is not forcibly exposed to the client. Furthermore, functional coupling cannot be avoided for this case study.

Suitability of the RenderCache for use case 2 is substantially improved by the provided slim client-side interface: A slim interface can provide intuitive threading behavior for the use from within an external resource loading thread.

Another facet that requires attention is the realization of the cache-internal lookup map. Storing shared resource references within a RenderCache interferes with client-side resource release strategies. In particular, a client then had to keep track of resources within the cache, and had to release these alongside its own resource references. Thus, a better design stores non-owning references within the resource cache. On resource requests, any invalid non-owning references (i.e. references whose resources have been destroyed already) are not considered. In this case, normal resource creation and loading commences.

On constant resource loading and unloading, or after any large resource unloading operation, it is advisable to clean any weak references to deleted resources from the resource cache. This is especially relevant to avoid memory pollution within use case 2. Thus, a corresponding client-side purge function must be devised.

Finally, certain applications (consider use case 1) require a full cache-clearing operation that removes all by-name bindings from the cache.

In total, this leads to the following RenderCache interface:

Function	Description
RenderCache—	
request	Lookup resource in cache by file name. Create, load, and store in cache if not found. Any dependencies are resolved within this cache.
purge	Remove orphaned resource references.
clear	Remove all resource references.

6.5 Scene and puppet management

Similar to RenderResources, RenderPuppets also host render-side specific data. However, unlike resources, puppets have but small-scale data (e.g. position, direction, or resource references) that is regularly updated.

This immediately brings up the question whether any such render-side representation is required at all. In fact, within a traditional rasterization setup, such as in [OS11], no separate scene representation is required within the rendering component. Instead, scene data often is merged into the scene graph or the application logics graph. Rendering then is performed by a visitor pattern, where traversing a node sends the appropriate object on to the rendering component. In this case, the optimization structure is not dictated by the rendering kernel, but by application logics. For instance, both a PVS-based rasterizer and a portal engine can use the same rendering back-end, but provide a different traversal logic.

Yet, this approach does not work out for a raytracing back-end: For the optimal asymptotic performance of $O(\log t)$, a raytracer needs to maintain a hierarchical scene structure. The layout of this structure heavily depends on the raytracer back-end. As detailed in the previous chapter, OptiX uses a lose bounding box tree over all objects. This tree is not regenerated in-situ on each frame, but only adapted on object movement to maintain time coherence for improved performance. Thus, client-side encoding of the optimization tree via a visitor pattern is not possible. Furthermore, directly merging the tree into application logics creates undesired hard coupling between application logics and the rendering subsystem implementation.

Consequently, RenderPuppets have been integrated into the RenderObject hierarchy as a flat, interface-only representation of any render-side scene structure.

6.5.1 RenderPuppet interface

Just as RenderResources, RenderPuppets have certain traits that affect their basic RenderObject functionality.

Contrary to RenderResources, the introduction of a temporary buffer for any incoming state change is not desirable, though — in particular, due to the high update frequency and small-scale involved data.

Instead, a constant double-buffering scheme is used. Within this scheme, each RenderPuppet separates up into two internal states: A client-side state on the one hand,

and a render-side state on the other. The former is used concurrently by any client threads for state modifications and query operations. The latter is restricted for use by any rendering thread, and holds the scene data that actually is displayed. On any synchronization operation, the client state is temporarily locked and copied over to the render-side state. Thus, pending modifications are accepted.

This elaboration yields the following puppet-specific interface:

Function	Description
RenderPuppet—	
modify	Change client-side state.
query	Query client-side state.
synch	Copy client-side to render-side state.

6.5.2 Puppet types

Rendering puppets come in various types, all derived from the above RenderPuppet interface. Each puppet type targets a very specific use case. Currently, the following puppet types have been implemented:

- **RenderScene**

  A RenderScene holds an assorted pool of RenderPuppets that can be rendered in one go. A RenderScene itself also is a RenderPuppet to allow for stacking of RenderScenes.

- **LightPuppet**

  A puppet that represents a single light source within the scene. For compatibility with the fixed function pipeline of the original jVR rendering component, lights provide a basic, fixed-function inspired parameter set. However, parameter interpretation depends entirely on the RenderContext implementation. For instance, a later advanced raytracer may introduce soft shadows or HDR rendering.

- **ViewerPuppet**

  A scene camera is encapsuled into the ViewerPuppet. Cameras are defined via their position and orientation in space, as well as the associated projection mode.

- **SpherePuppet**

  This puppet provides a traditional raytracer testing sphere, with origin, radius and a Material reference.

- **RiggedModelPuppet**

  A rigged model puppet allows to position a skeleton-animated Model resource into the scene. Additionally, the bone setup for the model is provided to select

an appropriate pose. In this context, note that the raytracer back-end provides basic animation capabilities. However, skeletal animation is currently not integrated within the Simulator X platform and thus unsupported.

- **StaticModelPuppet**

  This puppet places a non-animated Model resource into the RenderScene. In contrast to a RiggedModelPuppet, this allows for certain renderer internal optimizations, as no bones are used for display.

As with RenderResources, the implementation for each puppet interface type is hidden by the opaque base interface and the opaque RenderContext factory methods.

Once more, a full overview over functionality in each puppet type exceeds the scope of this thesis. However, the following RenderScene and StaticModelPuppet interfaces are representative for all other puppet interfaces:

Function	Description
RenderScene—	
add	Add a puppet in client-side state.
remove	Remove a puppet in client-side state.
get	Return any puppet within client state by index.
synch	Synch render-side and client-side puppet list.
StaticModelPuppet—	
setPosition	Apply client-side transformation.
getPosition	Return client-side transformation.
setOrientation	Apply client-side orientation.
getOrientation	Return client-side orientation.
setModel	Apply associated client-side model resource.
getModel	Return associated client-side model resource .
synch	Synch render-side and client-side model data.

This concludes the overview over the RenderObject hierarchy. Both RenderResources and RenderPuppets have been described and their interfaces have been presented.

An example overview over the entire RenderObject hierarchy including potential implementing classes is presented in figure 6.9.

The next section explains how RenderScenes actually are selected for rendering, and describes how the client initiates the rendering process.

6.6 Process control

In this section, the mechanism that is used to control the rendering process is elaborated.

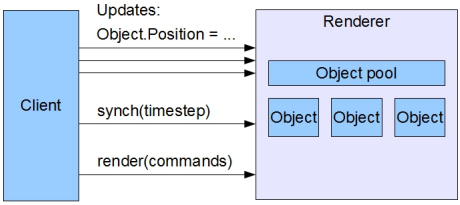


Figure 6.3: With push semantics, a client notifies the renderer whenever a new frame should be processed.

Most importantly, the RenderContext must be notified whenever rendering of a new frame shall commence. For each frame, the RenderContext additionally must know which of the scenes to display, and which parameters to use for rendering. Parameters include the camera position, the viewport within the application window, and an implementation-specific rendering technique.

All such information is sent to the renderer by a command-driven interface. Rendering commands and their use are discussed in the following. With the discussion of the rendering control mechanism, all essential renderer functionality has been introduced. Thus, the section concludes with a final definition of the main RenderContext interface.

6.6.1 Commands and command buffers

In the initial discussion on the command architecture, only general rendering parameters are considered — for instance, the source scene and the target camera.

All parameters that are required for single rendering pass on any scene are bundled within the RenderCommand type. To continue on the above example, a RenderCommand holds shared references to a single RenderScene and an appropriate Camera puppet.

Rendering commands, in turn, are aggregated in a RenderCommandBuffer object. Command aggregation allows to render various scenes and viewports in a single rendering run.

It is the responsibility of the client to accumulate suitable commands for desired rendering operations within a command buffer for each frame.

Finally, the client passes a complete RenderCommandBuffer to the RenderContext for processing. This triggers the rendering of a single frame: The RenderContext runs through all commands within the buffer in FIFO order, and executes appropriate rendering operations for all objects within each scene via the implementing back-end.

The command-based strategy associates push semantics with renderer control. These are illustrated in figure 6.3.

The advantage of push-semantics over other approaches — such as event-driven schemes or pull-semantics — is the intuitive client-side interface. In particular,

thread-safety guidelines are intuitively implemented and documented for a push-based architecture.

6.6.2 Rendering techniques

Unlike implementation-independent rendering parameters, concrete rendering techniques and their options cannot be included into the general RenderCommand type: As seen in chapter 2, there are lots of different approaches to rendering, and each of these requires few parameters of its own. For instance, a raytracing-based renderer may expose whether to use secondary rays for lighting calculations. A rasterizer, on the other hand, might provide a technique switch for choosing between forward and deferred light calculations.

Consequently, rendering techniques can only be accessed on implementation level. All technique parameter sets, however, must derive from a shared, function-less RenderTechnique interface. A single object with the RenderTechnique interface in turn can be attached to each RenderCommand to specify a technique for rendering of a single scene.

This setup allows maximum flexibility while maintaining a common code path for standard rendering operations. The setup also implies that any technique-specific application code has to be written against an implementation class. This can't be avoided though, and usually any such code is small and well-placed.

6.6.3 RenderContext interface

With all subordinate functionality modules in place, the RenderContext interface itself easily is assembled.

The first group of functions deals with high-level management of RenderObjects:

- For each RenderObject type (Material, RenderScene, Model, StaticModelPuppet, ...) an unique factory function is available.
- The entry point for global object synchronization is contained within the RenderContext. On synchronization, buffered state changes within all RenderResources are applied, and the client-side state of any RenderPuppet is copied to the render-side state.
- Finally, the context offers a global purge function. Purging enforces the actual deletion of any RenderObjects that have completely been released by the client.

Apart from the management of render-side objects, the RenderContext interface contains the main rendering entry point as well. Here, the client provides the accumulated command buffer, and thus triggers the rendering process and the display of a new frame.

In the current design, rendering has two side-effects:

On the one hand, the RenderContext automatically issues synchronization on each RenderObject and RenderResource touched during rendering. This partially conflicts

with the manual synchronization function, but in practice, both functions have been useful for one client or another.

On the other hand, the RenderContext cleans up an arbitrary amount of orphaned RenderObjects to keep a low memory profile. Again, this overlaps with interface functionality — this time, the manual purge. However, for large virtual environments, the manual purge is inefficient and only used offline, for example at load-time. In contrast, the renderer-internal purge always works online, and without further difficulties for the client. This is a much larger benefit than the more clean design achieved by exclusively manual purges.

The finished RenderContext interface is summarized in the below table:

Function	Description
RenderContext—	
create*Object*	Create new, default-initialized RenderObject.
synch	Accept all buffered changes for any RenderObject.
render	Display render-side state of scenes in command buffer, and remove some unreferenced, shared RenderObjects.
purge	Remove all unreferenced, shared RenderObjects.

6.7 Window management

Albeit functionality for the actual rendering process is complete now, another vital part of the renderer still is missing: The current architecture elaboration does not consider any way to manage the output window.

This attributes to the fact that all windowing functionality has been outsourced into a separate `WindowContext` interface. In turn, unnecessary coupling between two mostly unrelated modules — namely the renderer and the window manager — is avoided. In particular, a single window manager can be attached to several other subsystems (audio, input), and can be reused over various different renderer implementations.

The WindowContext interface itself is implemented intuitively for support of different OS-side window management systems, such as the Win32 API or Linux's X Window System. A window context provides both client-side and system internal functionality. Available operations include window size and position queries as well as handlers for certain UI element events (e.g. window closing, maximization, minimization). However, the full interface and implementation of the WindowContext are outside the scope of this thesis. Respective details can be found in the code documentation that accompanies the thesis source code.

On construction of a RenderContext implementation, an appropriate WindowContext instance is bound to it by non-owning reference. Thereafter, the RenderContext consults the bound WindowContext for any window-related tasks. For instance, the renderer might query the current window size to appropriately scale a pixel-sized font image.

As a matter of facts, one must note that certain relations between WindowContext and RenderContext are possible on implementation level only. In particular, several run-time and compile-time switches are required on renderer construction to adapt to various kind of windowing systems. For example, the OpenGl rendering API requires different calls for initialization on the Microsoft Windows and Linux platforms. Such functionality cannot be provided by general RenderContext and WindowContext interfaces. Yet, the affected code sections are small, and clean code is maintained — thus this remains a rather theoretical design problem.

This concludes the description of the overall rendering process. In the following section, various aspects of the renderer interfaces will be revisited, and refined in regards to their behavior under multithreaded use.

6.8 Blocking and threading behavior

Modern CPUs provide a series of independent cores. For optimal performance, calculations need to be distributed over these cores. Thus, an ideal modern rendering component allows access for multiple client threads. For instance, there may be a thread that reloads resources in the background, while another thread performs the actual rendering operations, and yet another two threads update scene objects in parallel.

However, the design of threaded components with soft realtime attributes is intuitively understood to be difficult: If an improper locking scheme is used for concurrent access, application threads may spend more time waiting for critical sections than a standalone thread would require to perform the entire workload. Thus, a well-considered multithreading and timing semantic has to be associated with the precedingly defined interface functions.

This section first discusses application timing and its implications on the renderer and client applications. Then, certain relevant interfaces within the rendering component are reviewed in terms of multithreaded access. For each interface, additional multithreading and blocking behavior is derived.

6.8.1 Timing

A base factor to consider when deciding on timing-related multithreading requirements is the application timestep mechanism.

As computers are only capable of discrete calculations, any continuous simulation timeflow needs to be broken down into discrete steps. The timestep mechanism describes how these steps are organized. In the following, three different time-stepping schemes are presented. All of these are supported by the renderer and multithreading design.

The most basic time-stepping scheme is a variable timestep. The simulation measures the time Δt an entire step takes for calculation. In the next simulation step, the simulation advances all its objects by Δt. For instance, all moving objects calculate a new position based on their old position and their velocity by integration over Δt.

Rendering then typically is integrated into variable time-stepping by displaying a single frame with the current simulation state after each simulation step.

Albeit variable time-stepping is intuitively understood, it has certain caveats. For instance, physics simulations tend to become unstable if the step interval becomes too small or too large. Large simulation steps may for instance be caused by a complicated rendering image, or by a renderer resource loading operation within logics. If an increase in the simulation step interval Δt introduces additional complexities into the simulation, the application may even run into a vicious cycle and come to halt.

The traditional workaround to stabilize simulations and to achieve more fluid simulations is a fixed timestep system. The corresponding assumption is that the target machine with minimum set requirements always is capable of running a single simulation frame in a certain, fixed time Δt_{const}. The application main loop then regularly checks if at least Δt_{const} time has passed since the last simulation frame. If so, another simulation frame is calculated and rendered.

While the resulting fixed-timestep simulation is more stable than a variable time-stepping approach, it does not handle peaks in simulation or rendering very well. Additionally, most applications require different simulation and rendering rates. Eventually, most current games try to maintain a constant simulation rate of around 100hz for responsive game logics, while rendering rates of 30hz are considered satisfying. The fixed timestep system does not support such constellations.

Finally, a time accumulator system represents a compromise between fixed and variable timestep systems. In particular, the simulation main loop in this system adds up all time passed while idle or during calculations into an internal time accumulator. Each complete, fixed time Δt_{const} that is available in the accumulator is extracted, and a corresponding fixed-time simulation step is executed. A single rendering step only is performed once all potential simulation steps have been extracted (i.e. accumulated time $< \Delta t_{\text{const}}$). Thus, the simulation performs multiple steps to catch up with reality if a simulation or rendering frame takes more than Δt_{const} time.

To avoid a vicious cycle that results in an ever-filling time accumulator, the maximum number of time to accumulate has a constant upper bound. On systems that are too slow for the actual simulation, this limit is reached rather quickly, and the simulation then slows down in relation to realtime.

In all three described time-stepping systems, the influence of the frame rendering time on simulation logics may further be reduced by multithreading. In particular, the actual rendering operation may be carried out in another thread. Thus, the actual time required within simulation logics is reduced to a mere function call, instead of an entire graphics frame. Consequently, logics steps are evenly distributed over time, instead of batch processing. In turn, the simulation runs more fluidly and feels more responsive.

Yet, it then is imperative to keep the remaining simulation logics code free from conflicts with the renderer. For instance, a naive implementation might prohibit access to any renderer object while rendering to avoid inconsistent states. The resulting synchronization again serializes rendering and logics, and the advantages of multithreaded rendering are lost.

6.8.2 Blocking behavior

As seen from the introduction on application timing, concurrency-related waiting times are to be avoided for a fluid simulation.

However, a completely block-less multithreaded rendering kernel cannot be realized. For instance, access to most structures must be protected to avoid inconsistencies or hard program crashes. At the same time, too fine-grained locking is not desirable either. In particular, such locking strategies result in a general performance bottleneck, and ease the introduction of cyclic, hard-to-find deadlock situations. Thus, multithreading and blocking behavior for all functionality within the renderer must carefully be balanced.

In this context, the less common term blocking behavior refers to the expected waiting time that occurs on concurrent access to a shared object. This must not be confused with the time the actual operation takes — an operation with a large potential waiting time still may perform quickly once access to the shared state is granted.

The relevant concept within the renderer architecture is to separate between two different types of blocking behavior:

- Behavior for functions that potentially wait for a long time, mostly until an entire frame has been rendered.

  Such functions typically require a lock on the entire RenderContext, and access the global state of the entire renderer. This behavior is often paraphrased as "block on the RenderContext" or "block on the renderer" within the remaining thesis.

- Behavior for operations that block for a short time at most — typically orders of magnitude less than blocking on the entire RenderContext.

  Generally, such functions do not work with renderer-internal state. The remaining thesis at times terms this "non-blocking behavior".

The ideal blocking behavior for a function depends mainly on its usage frequency: Functions that are frequently used, such as very often per frame, must never block on the RenderContext. Operations that are invoked less frequently are better suited for the less intricate blocking on the entire RenderContext.

Finally, certain operations might be better off without any complete thread-safety at all. For instance, a function might be restricted to invocation by another function which already guarantees thread-safety.

6.8.3 Blocking on RenderContext

At first, the above concept of blocking is applied to the functions within the main RenderContext interface: Blocking behavior is defined for RenderObject creation, synchronization, rendering, and purging of unreferenced objects.

The creation of any sort of render-side object has unsteady, but potentially frequent access characteristics. In detail, most frames do not involve any object instantiation

at all. Yet, any spawning operation within a frame potentially creates many objects in one go. For instance, once a new user enters a virtual world, her avatar puppet has to be created, and corresponding textures and meshes have to be loaded. If each of these operations blocks on the RenderContext, the respective waiting times cause a perceivable on-load stutter in the simulation. To avoid this, object creation must never block on the entire RenderContext.

This non-blocking requirement has certain implications on the implementation of render-side objects that must be considered from an architecture point of view. Since most rendering back-ends do not provide blocking-less multithreaded access, objects must be initialized to some safe default state that does not contain a render-side, implementation specific representation. For illustrative purposes, consider an OptiX based puppet representation. On creation, no OptiX scene node can be created and assigned to the render-side puppet state, as OptiX is not threadsafe and blocking were required.

Consequently, a zombie state for default-initialized objects is introduced. The zombie representation is replaced by actual render-side data structures but on the first synchronization. Most design guidelines despise of zombie-like default states [MC10], as a second code path is introduced for all object-related code that must handle objects in this state. However, there is no other way around blocking behavior here. Furthermore, the zombie state also is required by the detach functionality that has been included within any RenderPuppet.

Unlike object creation, object synchronization in general is required once per frame. Obviously, multiple scene synchronizations within a frame have no effect and are easily avoided. The only exception to this rule are background resource loaders that want pending resource modifications applied immediately to conserve memory. However, blocking behavior on the entire renderer is tolerable in both cases.

Object purges occur even less often than synchronization. In particular, the renderer-internal online purge that is performed automatically during rendering operations suits most applications already. Offline purges are executed only whenever a large-scale loading operation has left too many orphaned objects for efficient online purging. Thus, purges may block on the entire RenderContext as well.

In the context of object creation and object purges, it is reasonable to reconsider client-side object release and object destruction as well. In the initial life-cycle review, the delayed destruction scheme for RenderObjects has already been introduced. For review, objects are not immediately deleted once the last client-side accessible pointer is released. Instead, the RenderContext still maintains at least another reference to each object, and releases that one on purging. This scheme was devised exactly because of blocking behavior consequences. On the one hand, object destruction requires mutual access to the RenderContext and thus must always exhibit blocking behavior. On the other hand, client-side pointer release is hard to control, and could again interfere with simulation timing. Therefore, the purge mechanism delays any destruction operations to a point where blocking does not hurt. This elaboration also holds for the intermediate purging performed by on each actual frame rendering.

Finally, blocking behavior for rendering itself has to be decided. The choice was made to enforce another block on the entire context during rendering. Albeit this choice

seems intuitive at first, it has certain less obvious implications: The invocation of the render function blocks on the RenderContext, as rendering directly processes all commands within the buffer and immediately displays the corresponding frame to the user. Thus, there is no internal thread hidden within the RenderContext — but the calling thread becomes the rendering thread. This constellation is further discussed in the next section, where a helper class with an encapsulated, separate rendering thread is introduced.

The below table provides a summary over the preceding blocking discussion:

Function	Threadsafety	Blocking on	Usage
RenderContext—			
create*Object*	safe	none	frequent
synch	safe	RenderContext	per frame
render	safe	RenderContext	per frame
purge	safe	RenderContext	rare

6.8.4 Decoupling via rendering thread

As mentioned in the initial elaboration of timestep mechanisms, a major benefit in smooth timing is achieved by outsourcing the rendering from the main application loop into a separate rendering thread. In other words, the blocking behavior of the original RenderContext interface is avoided by an intermediate thread.

For client-side convenience, the interface architecture within this thesis allots a separate `RenderThread` class that hosts an internal rendering thread. For non-blocking rendering, a single client thread may first test if the rendering thread is idle. In this case, the client thread accumulates a command buffer, and provides this buffer to the RenderThread. The buffer then is copied and sent on to the RenderContext by the internal thread. In the meantime, program flow on client-side thread directly returns without blocking, and the client is free to do other things.

Both of the above RenderThread functions are combined in the below interface:

Function	Description
RenderThread—	
isIdle	Test if RenderThread is not rendering anything.
render	Render provided command buffer, blocks only if not idle currently.

The above RenderThread interface functionality overlaps with the following threading behaviour:

Function	Threadsafety	Blocking on	Usage
RenderThread—			
isIdle	safe	none	frequent
render	safe	RenderThread and RenderContext	per frame

Finally, some thoughts are contributed to the fact that a separate rendering thread was preferred over a solution directly integrated within the RenderContext. For one, decoupling the RenderThread from the RenderContext interface allows to reuse a single rendering thread over all potential RenderContext implementations. Next, the client application is free to either use standard, blocking rendering, or the non-blocking, threaded approach within the current design. Last but not least, the separate rendering thread provides an option to avoid blocking against the framerate for certain operations: Any potentially blocking operation may be performed depending on the RenderThread's isIdle test. For instance, an explicit synch operation does not block against the framerate if the RenderThread indicates idle state, as no concurrent frame rendering occurs at that time.

6.8.5 Blocking on puppets

Within a scenario involving a multithreaded client application, concurrent access to rendering puppets must be allowed as well. Threaded updates to the rendering scene even represents one of the most important use case for client-side parallelization.

Creation and destruction of render-side objects and thus of rendering puppets has already been discussed alongside the blocking behavior of the main RenderContext interface. In the following, the remaining puppet operations — state access and state synchronization — are investigated.

In a common use case, multiple client threads perform frequent asynchronous updates and queries on rendering puppets. Due to the update frequency, access to puppet state must consider performance as the major design goal.

At the same time, there may be a background rendering thread that displays the scene at the same time. Designs that involve locking a single, shared puppet state both within renderer and client side access are error prone and lead to undesired, long blocking. For instance, a RenderScene then had to be blocked for almost the entire rendering pass.

This observation led to the integration of the double-buffering scheme that already has been introduced when the puppet interface was elaborated. For review, there is a client-side state that is accessed by clients, and a render-side state that is used for display by any rendering thread. Both states are synchronized regularly by copying over all client-side data to the render-side state.

With this double-buffering scheme in place, an efficient blocking mechanism for puppets is intuitively found:

On the one hand, any client-side access to a puppet only blocks against any concurrent access on the puppet's client state. With but the little amount of data that is associated with a puppet, a single access takes little time, even with locking in place. Consequently, the system remains responsive even for many client threads.

On the other hand, access to the render-side state is only allowed for a sole rendering thread. The rendering thread in this case already holds a lock on the RenderContext itself, and thus the access is concurrency-free.

Synchronization fits seamlessly into the above picture if a small restriction is applied: Synchronization may only be performed by the global synchronization function that is contained within the RenderContext interface. Client-side per-puppet synchronization is not allowed. Then, within the per-puppet synch function, it is sufficient to lock but the affected puppet. As synchronization again only copies the small-scale puppet data, the lock is fine-grained enough not to disrupt application flow for normal client threads.

The restriction on renderer-triggered state synchronization seems arbitrary at first. However, the alternative had been the introduction of two locks per puppet, one for each state, which in turn created potential for cyclic deadlocks and misuse.

In this context, note that the synchronization function within the RenderPuppet interface is not threadsafe on its own: The render-side state is not explicitly protected. Thread-safety, however, is later on guaranteed by making the function only accessible from within any RenderContext implementation.

In total, rendering puppets exhibit the following blocking and threading behavior:

Function	Threadsafety	Blocking on	Usage
RenderPuppet—			
modify	safe	single puppet	frequent
query	safe	single puppet	frequent
synch	unsafe	none	frequent

On the implementation side, the above blocking behavior is easily realized by a single mutex within each puppet object. The mutex then is acquired on synchronization and on any client-side access to puppet state.

6.8.6 Blocking on resources

Thread-safety and blocking is a relevant topic for render-side resource operations as well. Similar to puppets, resources offer five different operation types over their lifetime: Creation at a RenderContext, resource update and readout operations, synchronization, and release.

Resource creation, release, and synchronization operations match those of RenderPuppets in their relevant access characteristics. Consequently, the same derivation of blocking behavior and thread safety applies for resources as well. For review, creation

and client-side release of resources are performed frequently, and thus must not block on the RenderContext. This is achieved by a zombie-like, render-side default state on resource creation, and by delayed destruction after client-side release by means of the global purge function. Synchronization is only called internally by the RenderContext as a reaction to a client-triggered global synch, and thus not threadsafe on its own.

Just like creation and release, resource updates exhibit non-blocking behavior within the current architecture. As already introduced alongside the RenderResource interface, this is guaranteed by two separate states within each resource: one temporary client-side state that holds any pending modifications, and one render-side state for use by any rendering thread. On synchronization, pending modifications are applied to the render-side state. Thereafter, the client-side state is discarded.

One might argue that most resources are updated with data only once on construction, and that duplicate states thus only increase memory consumption. However, non-blocking resource updates guarantee that a background resource loading thread is able to run without disturbing the framerate of any concurrent rendering client (compare with use case 2 of resource management).

In contrast to the above resource operations, resource queries are rather rare and well-defined. In most interactive applications, the only data ever retrieved from a render-side resource is viewport contents for use as a screenshot. Here, blocking behavior is acceptable. If client-side states are dropped to conserve memory, there anyways is a large code and time overhead for retrieving the resource from graphics hardware. Finally, if desired, blocking on resource queries even can be averted by a separate code path that only runs if the RenderContext is known to be idle — such as after a call to the isIdle functionality of a RenderThread.

The overall safety and blocking behavior on the RenderResource interface is compiled in the consequent table:

Function	Threadsafety	Blocking on	Usage
RenderResource—			
modify	safe	single resource	frequent
query	safe	RenderContext	rare
synch	unsafe	none	frequent

6.9 Interpolation

The blocking behavior specified up to now already avoids most framerate inconsistencies. Yet, another small improvement in rendering smoothness may intuitively be achieved by interpolation in between logics frames.

In particular, frame rendering up to now only considers a single, completed simulation state for display. Depending on the actual time-stepping scheme, a single rendered frame may however correspond to a fractional number of simulation frames.

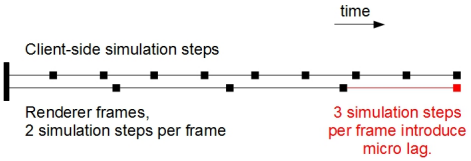


Figure 6.4: Rendering exact simulation frames results in micro lag, as one rendered frame corresponds to an uneven amount of logics frame.

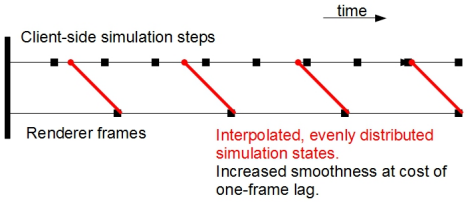


Figure 6.5: Timing for interpolated logics.

If this fractional part is not considered, rendering skips an inconsistent number of simulation frames on each rendering frame. This results in a rough, micro-lagging framerate at certain rendering and simulation rates. A corresponding situation on a time accumulation scheme is shown in figure 6.4.

Interpolation works around this problem by keeping two states for certain objects within the renderer. One of these holds the current simulation state, while the other holds the state from the preceding simulation frame. On rendering, an interpolation of these is shown, depending on the fractional part of a timestep that has passed when rendering begins. Figure 6.5 provides an example for rendering with interpolated logics frames.

Albeit this approach reduces timing-related micro lags, it comes at the cost of a constant lag of at most one simulation frame between the current simulation state and frame rendering. While this lag adds to the application's irresponsiveness, it is generally less perceiveable than the micro-lag introduced by small framerate variations, in particular for the typically high simulation rates.

In this context, one must also note that client applications are not bound to use interpolation. In particular, an application can also always pass an interpolant of 1 to enforce display of the most current simulation state.

Finally, interpolation is only supported on objects and object states that allow for reasonable and effortless interpolation. As an example, transformations of scene object are lightweight data that is intuitively interpolated. On the contrary, texture data is not interpolated for many reasons, such as large additional memory consumption and rather few actual state changes.

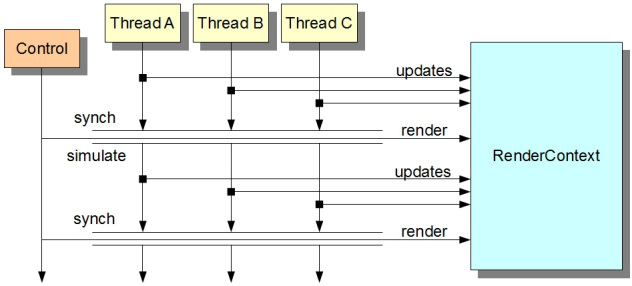


Figure 6.6: Multiple client threads update render-side objects (scenes, puppets, resources) in parallel while processing a single simulation frame. All client threads need to finish their simulation step before the corresponding, synchronized state can be rendered. Then client threads restart work on the next frame. A separate control thread manages the root application loop in this example. Note that this figure depicts a simple, fixed timestep. Alternative timestep mechanisms work similarly.

6.10 Multithreaded client applications

The preceding elaborations on timing and blocking behavior already considered access from multiple client threads at the level of single functions within each interface. In the following, the interaction of multiple client threads with the renderer is investigated from a much higher point of view.

In particular, calculations for each simulation frame may be performed by an arbitrary number of parallel threads. For instance, a single thread may be used to simulate each object within the virtual scene. The specified threading and blocking behavior then ensures that any renderer access from within client code need not consider any further concurrent threads. Yet, to achieve a synchronized final simulation state, a barrier over all client threads is required once each thread has performed its per-frame workload. Only then — and only if an optional rendering thread is idle — can another frame be rendered.

A potential multithreaded client setup is visualized in figure 6.6.

6.11 Extendability

Another vital point that up to now has not been considered in the architecture review is the extendability of the base design with new features.

However, upgrades of the object system are rather intuitive. If any use case emerges that has no associated object type, there are two possible design choices: First, one tries to simulate the new object type by combining one or more existing types. If this is not possible, or yields suboptimal performance, a new object type interface

deriving from either RenderPuppet or RenderResource needs to be defined. Finally, the new interface must be implemented once for each RenderContext type.

For instance, a large-scale height-field terrain with texture splatting and geo-mipmapping is not included in the core design. On the one hand, this terrain can be simulated by using a series of StaticModelPuppets alongside Model resources, one for each LOD level and terrain cell. On the other hand, a new resource type might be implemented that handles caching and LOD rebuilds automatically for improved performance. A new puppet type would then allow to place this new terrain resource within the virtual world.

RenderContext implementations for the support of additional rendering back-ends are easily integrated into an existing system as well: Each predefined interface has to be reimplemented for context specific functionality. On client-side, the interface design does not expose any implementation details, except for a single instantiation of a RenderContext implementing object. Yet, context creation corresponds to at most a single line of central code, and thus is easily exchanged. An application may even consider script-able logics that allow for runtime selection of a rendering component.

6.12 Design alternatives

During the development of the renderer interfaces and their respective functionality, several alternative design ideas have been tested, and discarded for various reasons. For future reference, the following text lists some alternative design choices, their advantages, and the respective reason for rejection.

- **Owning back-reference from RenderObject to RenderContext**

  The first design alternative relates to RenderObject handling. In particular, the current design allots for a non-owning back-link from each RenderObject to the corresponding RenderContext instance. This enforces an inelegant workaround for order-of-destruction problems (i.e. detach functionality) and violates object-oriented design semantics. Yet, with owning objects, the client application needs to ensure that all objects are released before the actual RenderContext can be destroyed. This is undesirable, as a single, accidentally leaked object keeps the entire application from clean shutdown. In particular, such leaks regularly occur within the Java integration due to the destruction semantics in the Java garbage collector.

- **Internal rendering thread in RenderContext**

  In the original design, the external rendering thread has been devised directly into the RenderContext interface. The blocking behavior of the render call was slightly modified, as to allow for an independent isIdle query directly on the RenderContext. The advantage here is a more compact, less cluttered system interface. Any context implementation would then, however, have had to reimplement a thread of its own, and to provide appropriate methods for remote control. Furthermore, resource upload operations could then not be clearly separated from any ongoing rendering process from a client-side perspective.

- **RenderThread-internal RenderObject updates**

  A rather intuitive first approach places the update of RenderObjects into the RenderThread functionality. In particular, to ensure non-blocking behavior, the client only updates objects if the RenderThread is currently idle. This simplifies the blocking definitions required for all interfaces, and avoids updates to RenderObjects in unrendered simulation states. However, this approach also yields a second code path through the entire application logics graph. Each relevant logics object needs to provide a separate, logics internal state, and copies that one over to the corresponding RenderObject on a special trigger event. Furthermore, the client application must be wary not to accidentally perform any blocking operation outside the separate code path for RenderObject updates.

- **Message-based RenderObject updates**

  Another way to achieve the previously described non-blocking interface design works with a general message queue setup. Within this setup, the client does not update RenderObjects by method calls. Instead, the client accumulates all modifications as messages inside some sort of client-side queue. Once all updates for a synchronized simulation state have been enqueued, the message queue is applied to the RenderContext by a single blocking call. This setup provides a less cluttered interface, as there are fewer blocking guidelines for the client to consider. It also is possible to unify both resource and puppet updates with a single message system. On the downside, however, the messages themselves add another, heavy-weight interface layer atop the actual RenderObjects. Additionally, special provisions need to be provided to encapsulate state queries into messages. Finally, the Simulator X application already implements message queues for communication with the rendering system, thus there is no need to replicate this functionality in the actual rendering kernel. A rendering core that is centered around multithreaded messaging support is currently under development in [TW11b].

- **Rendering with pull semantics**

  Currently, the rendering process control is implemented with push semantics. The client explicitly tells the renderer when to render. Yet, an alternative design involves a pull semantics, as shown in figure 6.7. Here, the renderer notifies an arbitrary client via a callback method or similar interface that a new frame should be rendered. In reaction to this notification, the client performs an appropriate scene update. This mechanism allows for more logics inside the actual RenderContext, which in turn simplifies certain client tasks. For instance, the entire interpolation code could be replaced by a timestep that the renderer sends back to the client application. Yet, in the context of the Scala-written Simulator X application, the simulation logics already run as standalone components. Thus, a pull semantics renderer is not easily integrated. However, the concept of a master rendering system with pull semantics again is under current investigation in [TW11b].

- **Extrapolated rendering**

  Apart from the currently supported interpolated rendering mechanism, support for extrapolation intuitively is possible as well. In particular, instead of interpo-

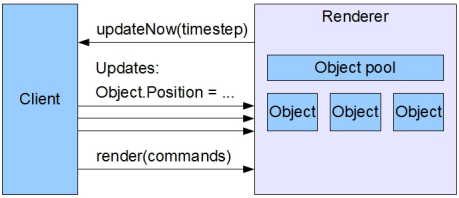


Figure 6.7: With pull semantics, client-side updates to scene objects are triggered by the renderer once a new frame can be processed.

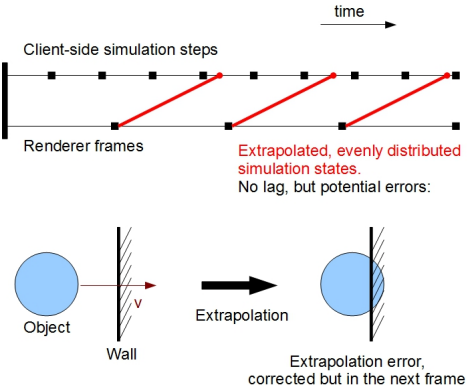


Figure 6.8: Timing for extrapolated logics.

lation in between the last two simulation frames, the current ratio of a timestep may also be used to extrapolate from these frames. Extrapolation removes the one-frame lag of interpolation, while still providing the smoothness of interpolated rendering. Yet, extrapolation has a severe side effect: Extrapolation may induce errors if the extrapolated state does not match up with the result of the next simulation step. One such case is shown in figure 6.8. Subjectively, the resulting graphics glitches have been considered worse as the small rendering lag, thus interpolation was chosen for implementation. Finally, one should note that extrapolation behavior can be simulated with the current interpolation setup by adding a full timestep to the interpolation factor on rendering.

6.13 Interface summary

In the preceding sections, a general rendering interface has been designed based on a series of initial requirements.

For review, the requirement of a slim and intuitive interface has been realized by

small, standalone functionality modules and interface types that collaborate with client applications by few transparent method invocations. Multithreading and safety rules have been specified per-function. However, at no point need any client thread know about any concurrent access by competing threads. The entire thread safety realization has been hidden away in the actual renderer implementation. In a similar context, most frequently used functions do not block against the framerate. This both accounts for more intuitive client-side use, and for general design-based performance optimization. Finally, blocking behavior has been accepted for few and rare calls on terms of memory conservation and implementation simplicity.

As a conclusion to the interface development within this chapter, the following figures and tables provide a summary over all interfaces within the rendering component.

The next chapter translates the below interfaces to the C++ language, and provides an implementation based on the OptiX API.

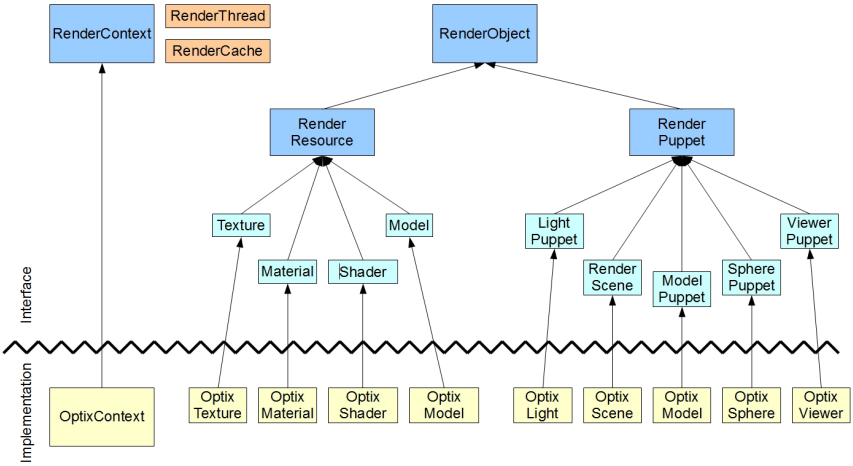


Figure 6.9: A potential class hierarchy based on the devised RenderContext interface and the OptiX Api.

Function	Description
RenderContext—	
create*Object*	Create new, default-initialized RenderObject.
synch	Accept all buffered changes for any RenderObject.
render	Display render-side state of scenes in command buffer, and remove some unreferenced, shared RenderObjects.
purge	Remove all unreferenced, shared RenderObjects.
RenderThread—	
isIdle	Test if RenderThread is not rendering anything.
render	Render provided command buffer, blocks only if not idle currently.
RenderCache—	
request	Lookup resource in cache by file name. Create, load, and store in cache if not found. Any dependencies are resolved within this cache.
purge	Remove orphaned resource references.
clear	Remove all resource references.
RenderObject—	
create	Default-initialize object.
modify	Enqueue object state change.
query	Query object state.
synch	Apply any pending state changes.
release	Release a single client-side shared reference, objects are not destroyed immediately.
detach	Detach leaked object from destroyed context.
destroy	Destroy object during context-wide synch or purge.
RenderPuppet—	
modify	Change client-side state.
query	Query client-side state.
synch	Copy client-side to render-side state.
RenderResource—	
modify	Create new buffer with incoming data.
query	Retrieve render-side data directly.
synch	Apply modification to render-side state, then drop client-side state to conserve memory.

Figure 6.10: Function overview for renderer interfaces.

Function	Threadsafety	Blocking on	Usage
RenderContext—			
create*Object*	safe	none	frequent
synch	safe	RenderContext	per frame
render	safe	RenderContext	per frame
purge	safe	RenderContext	rare
RenderThread—			
isIdle	safe	none	frequent
render	safe	RenderThread and RenderContext	per frame
RenderCache—			
request	safe	single cache	frequent
purge	safe	single cache	rare
clear	safe	single cache	rare
RenderObject—			
create	unsafe	none	frequent
modify	varies for puppets and resources		
query	varies for puppets and resources		
synch	unsafe	none	frequent
release	safe	single share	frequent
detach	unsafe	none	rare
destroy	unsafe	none	frequent
RenderPuppet—			
modify	safe	single puppet	frequent
query	safe	single puppet	frequent
synch	unsafe	none	frequent
RenderResource—			
modify	safe	single resource	frequent
query	safe	RenderContext	rare
synch	unsafe	none	frequent

Figure 6.11: Threading and blocking behaviour for renderer functions, as well the estimated usage frequency.

7 Raytracer implementation

After the base component interface has been introduced, this chapter describes the actual OptiX raytracer implementation.

First, the basic choice of C++ as an implementation language is discussed. Then, certain C++ adaptions to the interfaces from the preceding chapter are developed and corresponding C++ interface classes are provided. This also includes full implementations for some of the auxiliary modules. Finally, the actual OptiX implementation of the C++ interfaces is detailed, and the core raytracing process within the raytracer component is explained. An example client application for the raytracer component and an introduction into the conducted unit testing close this chapter.

7.1 Native and Java approaches

The primary implementation question that needs to be answered is what language and level to implement the raytracing kernel in.

Since the Simulator X application already is written in a high level language, the first option that comes to mind is a direct Java or Scala implementation. This yields many advantages, such as existing high-level, OS-independent helper components and automatic object management.

However, the OptiX SDK as well as most other rendering APIs are not directly available within Java. Instead, a custom native wrapper has to be written for optimal performance. Direct JNA calls lead past this problem, but at an intolerable performance penalty for the transfer of large-scale resource data.

Additional difficulties arise when combining C-style resource management with Java-style automatic garbage collection and exception safety practices. Generally, the only practicable choice is to use the Java finalize() method, which carries a heap of synchronization and order-of-destruction problems.

Thus, in the course of this thesis, it was decided to use C++ as an implementation language for the rendering kernel. This allows to both directly access the C-style OptiX API, as well as to exploit high-level object-orientated language features. The resulting slim C++ interface can then be integrated into Java via a low-bandwidth wrapper that generally does not require large-scale data transfers. The resource management problem, however, remains unsolved even by this choice, and is discussed separately in chapter 8.

7.2 Interface implementation

Using native C++ as an implementation language, the first implementation step involves generating C++ interface classes from the original, abstract interface definitions. However, the translation from abstract interfaces to C++ code involves certain modifications to the initial design, driven by language limits. These are detailed in the following. Only then are some of the actual C++ interface classes presented.

7.2.1 C++ specifics in the RenderContext interface

C++ is a rather old high-level language and thus by default does not have many popular features known from more recent languages — such as Java's garbage collection, pure interfaces and intrinsic thread safety. While many of these are currently under consideration for a new C++ standard, C++ standard releases are rare, and acceptance by compiler vendors takes some time. Finally, there are excellent extension libraries that provide various missing features. Yet, the inclusion of auxiliary third-party libraries has been kept to a minimum to maintain a slim implementation package.

Consequently, it was decided to use but basic C++ for the implementation work within this thesis. Therefore, the following relevant implications on the interface implementation have been recognized:

- C++ does not support distinct interface types. Thus, all interfaces are realized by abstract classes.
- Up to the most current standard, C++ did not come with a standardized way to deal with shared references. Consequently, all shared references to RenderObjects within the base interface design have been wrapped up into a custom SharedPointer implementation. This implementation is compatible with boost::shared_ptr, and thus with the C++0X std::shared_ptr — yet works even on systems without a recent C++ compiler.
- Certain non-owning references (such as the reference from RenderContext to WindowContext) within the interface descriptions are realized by plain C++ references or pointers. Life-time management for bindings between major components has to be performed by the calling application. Bindings within a functionality module are managed automatically.
- Exceptions are used to signal any errors during program execution — e.g. failure to retrieve a resource file or out-of-memory conditions. Any client applications must be aware of exceptions and must ensure exception safety. Due to missing garbage collection and finally constructs, maintaining exception safety in C++ is not a trivial task.

7.2.2 Interface classes

Under consideration of the above guidelines, the C++ interface classes are intuitively created.

As interfaces already have been defined from a functional point of view, a full elaboration of their C++ realization has no further benefit. Thus, the following listings just present select few interfaces for illustrative purposes. Remaining interfaces can be found within the source code delivery.

The main RenderContext interface is realized in the below listing. One should particularly note the separation of the logical create*Object* functionality into separate factory methods for each object type.

```
class RenderContext :
    private NonCopyable
{

    public:

        RenderContext();
        virtual ~RenderContext() NOTHROW;

        virtual void synch() = 0;

        virtual void render
            (const RenderCommandBuffer &buffer) = 0;

        virtual SharedPointer<RenderScene>
                                        createRenderScene() = 0;
        virtual SharedPointer<LightPuppet>
                                        createLightPuppet() = 0;
        virtual SharedPointer<ViewerPuppet>
                                        createViewerPuppet() = 0;
        /* ... */

        virtual SharedPointer<Texture> createTexture() = 0;
        virtual SharedPointer<Material> createMaterial() = 0;
        virtual SharedPointer<Model> createModel() = 0;
        /* ... */

        virtual void purge() = 0;
};
```

Further relevance is attributed to the fact that interface classes from the entire RenderObject hierarchy already provide certain shared implementation aspects. From a pure design point of view, this practice is questionable. However, there are just few ways to achieve a functionality-induced implementation. Thus other solutions involving aggregation or multiple inheritance have less benefits than the chosen direct approach.

As an example, consider the back-reference to the RenderContext and the per-object mutex in the below RenderObject interface:

```
class RenderObject :
    private NonCopyable
{

    public:

        virtual ~RenderObject() NOTHROW { };
```

```
    protected:

        RenderObject(RenderContext* rendercontextbinding);

        virtual void synchStates() = 0;
        virtual void unbindContext() NOTHROW = 0;

        RenderContext* RenderContextBinding;

        mutable DefaultMutex ObjectMutex;
};
```

A more intricate example of shared implementation details within an interface class is given by the RenderScene interface:

```
class RenderScene : public RenderPuppet
{
    public:

        void insert
            (const SharedPointer<RenderPuppet>& puppet);
        void remove
            (const SharedPointer<RenderPuppet>& puppet) NOTHROW;
        void clear() NOTHROW;

    protected:

        RenderScene(RenderContext* rendercontextbinding);

        virtual void synchStates();

        typedef TreeSet<SharedPointer<RenderPuppet> > PuppetSet;

        struct InternalState
        {
            PuppetSet Puppets;
        };

        InternalState ClientState;
        InternalState RenderState;
};
```

In particular, the entire list of puppets is already integrated into the RenderScene class. This includes puppet management and synchronization between client-side and render-side state. Renderer-specific implementing classes only have to add their own background structures and must appropriately update these on state synchronization.

Similar default functionality is integrated in all other RenderPuppet interfaces. For instance, StaticModelPuppets hold a client state and a render state with instance transformations and a pointer to a RenderModel object. State synchronization code is readily available within StaticModelPuppet as well.

RenderResource interfaces have less default functionality than their RenderPuppet counterparts: Only the blocking and buffering behavior for incoming client-side modifications as well as default loading routines have been implemented. The actual processing and clearing of any pending resource data is left to the context-specific implementation.

Finally, certain of the previously discussed functional interfaces directly translate to C++ classes. For instance, the RenderCommand and RenderCommandBuffer types that are used to control the RenderContext are a rigid part of the client-side interface. Thus a context-specific implementation makes but little sense. Instead, there are hard-coded, hand-optimized command and command buffer classes.

Likewise, there is no gain in allowing for a customizable RenderThread or RenderCache: Both auxiliary modules provide rather inflexible functionality, which maps to a fixed C++ implementation. However, the use of a rendering thread and a resource manager is optional. Therefore, clients are free to use custom modules with different functionality if the original modules do not fit their requirements.

For reference, the below C++ class both represents and implements the original RenderThread interface:

```
class RenderThread
{

    public:

        RenderThread(const String& renderthreadname,
                     RenderContext& rendercontextbinding);
        ~RenderThread() NOTHROW;

        bool isIdle();
        void render(const RenderCommandBuffer& buffer);

        const String RenderThreadName;

    private:

        void renderThread();
        static void renderThreadStatic(void *startupparam);

        RenderContext& RenderContextBinding;

        DefaultMutex ClientMutex;
        DefaultThread InternalThread;
        DefaultSignal HasBufferSignal;
```

```
        RenderCommandBuffer InternalBuffer;
};
```

This concludes the presentation of noteworthy C++ interface classes.

7.3 OptixContext implementation

In the preceding section, the client-side C++ interface classes have been discussed. In the following, their OptiX-specific implementation is examined.

At first, the general implementation-side class layout is presented. Thereafter, the realization of classes within the context of the OptiX platform is explained in more detail, and the collaboration between raytracer components and the OptiX API is elaborated. The section concludes with a tour over the RenderContext implementation from construction over the rendering entry point and predefined programmable components to context destruction.

7.3.1 Mirror hierarchy

RenderObjects carry most of the communication data between a client application and the raytracer implementation. Thus, the respective implementing classes for RenderObjects, RenderResources, and RenderPuppets form the foundations of the later main RenderContext implementation.

To allow for intuitive management of RenderObjects, the OptiX implementation introduces another class hierarchy, in parallel to the class hierarchy already defined by the client-side interfaces. The new hierarchy starts with a separate `OptixObject` interface that corresponds to the RenderObject interface. In turn, interfaces `OptixPuppet` and `OptixResource` inherit from OptixObject. All final implementing classes derive both from the original client-side interfaces defined in the preceding chapter and from either OptixPuppet or OptixResource. For instance, `OptixModel` implements both OptixResource and RenderResource. Figure 7.1 further illustrates the implementation-side class hierarchy.

Arguably, multiple inheritance is often despised — even more so when data-carrying parents like the raytracer's C++ interfaces classes are involved. However, the above constellation is a corner case. Most notably, the resulting hierarchy is an extension on the well-known Facade pattern [EG04], where the facade reference has been replaced by inheritance. The original Facade pattern was considered, but required additional implementation and management effort. In particular, further measures would have been necessary to coordinate life-times of the client-side facade object and the actual render-side implementation. In contrast, multiple inheritance allows for convenient life-time management on both client-side interface and implementation-side data by means of a single shared pointer.

Each of the final implementing classes within the implementation-side hierarchy aggregates objects from the OptiX API to realize respective functionality. For instance,

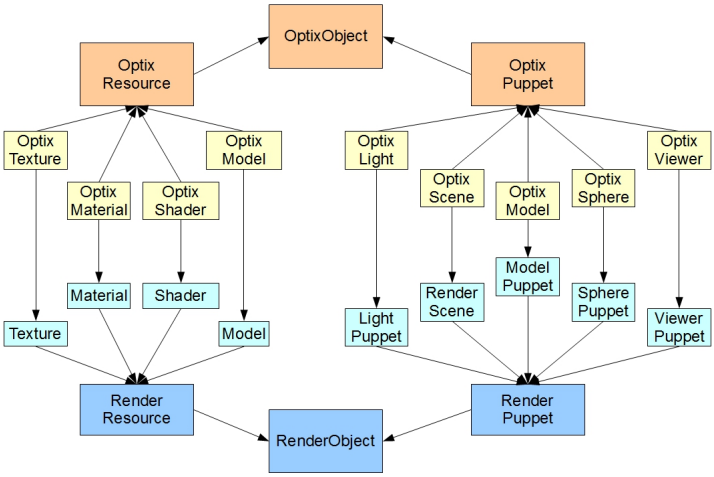


Figure 7.1: The final, implementation-side RenderObject hierarchy.

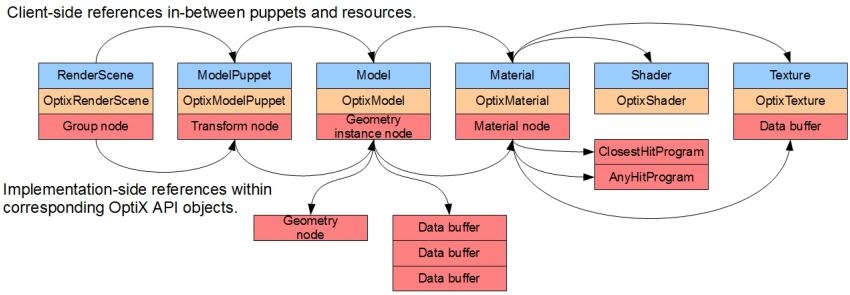


Figure 7.2: The above graph shows the translation of references in-between client interfaces to respective objects from the OptiX API via intermediate implementing classes. Any links to OptiX data buffer objects are not hard references, but instead realized by device-side variables.

each OptixRenderScene instance is attached to an OptiX group node that encapsulates all further scene objects. Certain device-side variables are mapped to implementing classes as well. General relations between client-side interfaces and OptiX API objects are further illustrated in figure 7.2.

In the following, the internals of the OptiX API integration are investigated in depth for all RenderPuppet and RenderResource implementations.

7.3.2 Resource implementation

The below list elaborates the contents and behavior of each OptixResource implementation class:

- **OptixTexture**

  An OptixTexture implements the Texture interface and houses the OptiX representation of a single texture. All texture data is stored in form of a device-side data buffer.

  In terms of texture data updates, the client application first sends an image to the parent Texture instance by an appropriate client-side interface invocation. The Texture instance buffers the respective image directly within the client-side interface class. On synchronization, the OptixTexture implementation detects any such incoming image within its parent's data. The image is then copied into the device buffer, and the host-side image is cleared thereafter.

  The convenience texture loading function provided by the parent Texture interface works similarly: It loads a texture image (BMP, TGA, or JPG format) from disk, and buffers the resulting host-memory representation for processing by the OptixTexture implementation.

  In the context of texturing, two vital restrictions in the OptiX API must be noted:

  On the one hand, OptiX currently supports but four-component, floating point RGBA images. Any incoming textures are automatically converted into this format at the expense of device-side memory. Any other image formats result in graphics artifacts or system freezes. This stability problem is reviewed in greater detail in chapter 9.

  On the other hand, one must note that the OptiX API for now does not support mipmapping. Thus, a single data buffer is sufficient at the moment. Yet, code for automated mipmap generation has already been included in the implementation, and can be enabled by uncommenting once OptiX mipmapping support is released.

- **OptixShader**

  As an implementation of the basic Shader interface, an OptixShader houses entry points for both programmable components that are directly involved in the calculation of ray results: Closest-hit and any-hit programs.

  Initialization and updates on an OptixShader are rather intuitive: Alongside the respective loading method invocation, the client passes a general name for a shader file. This name is buffered as pending data within the basic Shader class. On synchronization of an OptixShader, any incoming file name is retrieved from the parent Shader. An intermediate virtual file system (which is outside the scope of this thesis — see appendix A for details) translates the incoming file name into an actual on-disk file.

  The file extension is used to figure out file contents:

  On the one hand, clients can directly pass in a `.ptx` shader file. In this case, the assembly is retrieved into memory by the virtual file system and re-compiled for

the current GPU device. Finally, closest-hit and any-hit programs are located by compulsory function names. These are described in depth alongside the overview over client-side programmability in 7.3.4.

On the other hand, custom XML shader definition files are supported. These are recognized by a `.shd` extension, and contain a map of context type to context-specific shader files. Within this map, the OptiX implementation finds the file name of a suitable PTX shader file. The PTX shader file in turn is applied regularly.

The latter approach is particularly relevant because it allows to specify shaders within client code in an implementation-agnostic way. In turn, the RenderContext implementation can be exchanged without touching any client-side resource names.

- **OptixMaterial**

  An OptixMaterial serves three purposes: It integrates an OptiX material object into the Material interface, connects closest-hit and any-hit programs from an OptixShader, and binds the image data buffer from an OptixTexture.

  Material loading follows a scheme quite similar to the loading process for shader resources. The base Material class buffers an incoming material file name, which is applied by the OptixMaterial class but on synchronization.

  Usually, the client specifies the name of a general XML description file, extension `.mat`. This file contains a tuple of shader file name and texture file name. The OptixMaterial retrieves both file names, and either creates corresponding OptixTexture and OptixShader objects, or fetches these from an optional RenderCache. Thereafter, shared references to the connected shader and texture resources are stored within the parent Material interface for life-time management and client accessibility.

  Instead of a material XML file, the client can directly indicate the name of either a Shader or Texture resource. These are recognized by appropriate file extensions. In turn, only a single corresponding resource is created or retrieved from the RenderCache. Instead of the omitted texture or shader, a default OptixShader or OptixTexture instance is referenced.

  Once shader and texture resources have been linked, the OptixMaterial creates an OptiX material node. Consequently, the any-hit and closest-hit programs from the OptixShader are attached to the new node. Finally, a device-side variable for the texture buffer within the OptixTexture is created on the material node and assigned appropriately.

  On a side-note, a default texture is required even for purely procedural surfaces. Otherwise, the device-side texture buffer variable remains unset, and OptiX raytracing does not commence.

- **OptixModel**

  The OptixModel class implements the Model interface based on the OptiX API.

  The parent Model class already provides most of the model loading functionality. Namely, various types of model files (Collada DAE, Wavefront OBJ, Autodesk

FBX, ...) can be retrieved into a general in-memory representation by the intermediate virtual file system. The host memory representation in turn is stored as pending data, and fetched by the OptixModel implementation on synchronization.

The OptixModel thereafter translates all host-memory geometry data into respective OptiX API data buffers and corresponding OptiX scene nodes.

During the translation to OptiX structures, the original geometry is split into mesh groups with unique materials. Each mesh group is represented by an OptiX geometry instance and a subordinate OptiX geometry object.

The actual material of each group is either created directly or requested from a RenderCache instance. The respective material file name is typically stored within the original mesh format or provided by an external XML mapping file. Once an OptixMaterial has been obtained, its internal OptiX API material object is bound to the geometry instance node. Finally, shared pointers to all referenced Material interfaces are exposed to clients by the base Model interface.

After all materials have been handled, all geometry data of each mesh group — vertex coordinates, normals, and texture coordinates — is moved into device-side data buffers. Each buffer is bound to a device-side variables on the geometry instance node with a predefined name. This enables a single intersection or bounding program to work with various meshes through use of the same code-side variables.

In this context, the OptixModel also binds global triangle intersection and triangle bounding programs to each geometry instance object. These default programs are loaded on OptixContext creation, and described in 7.3.8.

Finally, the OptixModel groups all geometry instance objects within a geometry group and an associated acceleration structure. The resulting geometry group can now be instanced into any OptiX scene without further effort.

7.3.3 Puppet implementation

The next list describes the features of OptixPuppet implementations:

- **OptixRenderScene**

  The OptixRenderScene class translates the puppet list within the render-side state of the base RenderScene interface into an OptiX representation. A new OptiX API group node is created on the first synchronization. Any children nodes that correspond to puppets within the scene are dynamically attached below this group object. Most notably, scenes can recursively be combined to realize complex instancing schemes by the use of OptiX group hierarchies. Further potential children nodes are discussed in the following.

- **OptixStaticModelPuppet**

  The OptixStaticModelPuppet implementation class provides OptiX-specific functionality for the non-animated StaticModelPuppet interface. In particular, the base interface already holds a shared pointer to the bound Model

resource within the render-side state. The corresponding resource implementation class OptixModel contains an OptiX geometry instance node that bundles together all object geometry and material parameters. The OptixStaticModelPuppet only encapsulates the existing geometry instance node into an OptiX transformation node. The transformation node finally is made available to OptixRenderScene.

- **OptixRiggedModelPuppet**

  An implementation for the skeletally animated RiggedModelPuppet client-side interface. Unlike static models, rigged model puppets do not instantiate any geometry. Instead, the OptiX API contents of any linked model resource are copied into another, duplicate representation within the rigged model implementation. Then, the CPU updates respective vertex position buffers according to the current animation pose on each state synchronization. This is quite inefficient and complicated, but attributes to the fact that OptiX intrinsically does not support any animated geometries. An alternative implementation based on loose bounding volumes is left for future work.

- **OptixSpherePuppet**

  The procedural sphere class differs from the previous model classes in that it supports but a single type of geometry with hard-coded OptiX nodes, programs, and materials. However, one must understand that this object was designed as a diagnostic for general renderer capabilities, and not for actual use within client applications.

- **OptixViewerPuppet**

  This class implements the client-side ViewerPuppet interface. All respective functionality has been realized in ViewerPuppet already, as there is no corresponding camera object within the OptiX scene structure. Instead, all camera parameters are associated to context-wide global variables which are managed by the OptixContext class. Further details are given in 7.3.9.

- **OptixLightPuppet**

  OptixLightPuppet implements the general LightPuppet interface. Similar to the OptixViewerPuppet, lights do not directly have any corresponding node within the OptiX scene hierarchy. Any active lights rather are copied into a specialized OptiX data buffer at the start of each new frame. This buffer in turn is made available to any programmable components for lighting calculations by means of a global device variable. The global light buffer is directly managed by the OptixContext and detailed in 7.3.9.

7.3.4 Programmable component interface

As mentioned in the preceding implementation elaboration, certain predefined PTX variables are used for communication with both custom client programs and default programmable components. Further global variables are introduced by the OptixContext. The following table provides an overview over all these device-side variable names and associated usage guidelines:

Type	Name	Usage
rtBuffer<float4, 2>	OutputBuffer	Floating-point output image.
uint2	PixelIndex	Current pixel within thread grid.
float3	CameraPosition CameraUp CameraRight CameraDepth	Camera transforms.
rtObject	RootObject	Geometry group of current scene.
rtBuffer<Light>	GlobalLights	Active lights, defined in 7.3.7.
optix::Ray	CurrentRay	Global ray for current thread.
CameraRDT	CameraRD	Ray data of camera-type rays,
ShadowRDT	ShadowRD	and shadow-type rays.
rtBuffer<float3>	ModelVertices ModelTexCoords ModelNormals	Model data of geometry instance currently under processing. Only defined for hit, intersection
rtBuffer<int3>	ModelIndices	and bounding box programs.
rtTextureSampler <float4, 2>	TexSampler	Current texture data buffer.
float3	HitPoint	Interpolated hit point, normal, and
float3	Normal	texture coordinates sent from
float3	TexCoord	intersection to hit program.

Similar to device-side variable names, device-side function names are relevant for custom programmable components. In particular, functions with predefined names are extracted from PTX files within the OptixShader implementation.

Currently, only closest-hit and any-hit functions are customizable per shader. The raytracer employs a total of two different ray types — the already introduced camera rays and shadow rays. Consequentially, there are four different functions that can be defined by the client: **CameraClosestHit**, **CameraAnyHit**, **ShadowClosestHit** and **ShadowAnyHit**. The respective function names are explicit enough that no further explanation is required.

All four device functions must be specified inside each custom PTX file. However, corresponding standard implementations can be pasted into the CUDA sources by inclusion of certain CUDA headers provided within the source code delivery.

7.3.5 General OptixContext functionality

With the introduction of mandatory device-side variables and functions, most OptiX functionality has already been described. Only the OptixContext remains for elabo-

ration in the following sections. This section concentrates on general functionality — construction, resource management, and synchronization. Consequent sections investigate the lighting implementation, default programs, the actual rendering process, and OptiX context destruction.

From the perspective of a potential client application, the creation of an OptixContext marks the entry point into raytracing.

On construction, the OptixContext is bound to a respective WindowContext implementation via a constant reference. The context uses an OS-specific switch to build a GL output context on the target window. For review, the GL output context is required to display the OptiX rendering result because OptiX has no display functionality of its own.

After creation of the target window, the main OptiX API context is created. A GL pixel buffer object is initialized, and mapped to an OptiX data buffer. In turn, default OptiX programs, variables, and nodes not suited for client-side customization are created. These are investigated in depth in the next section. Finally, the OptixContext is ready for use.

At this point, the client typically generates a series of resources and puppets for later rendering. On implementation side, it is not sufficient to return default-initialized OptixObject instances. Instead, these must additionally be inserted into a renderer-wide, shared pointer set to meet later purging and destruction specifications. This is one of many cases where the need for a separate mirror hierarchy becomes evident: Storing OptixObject shared pointers within this set instead of general RenderObjects avoids many instances of dynamic casting.

Insertion of shared resource pointers into a global set of course must maintain both thread-safety and blocking behavior rules. This is achieved by a separate creation mutex that protects a separate set of recently created OptixObjects. On synchronization, this set is merged with the global set of OptixObjects — in other words, creation of objects is double-buffered within the Optix implementation.

The actual synchronization functionality requires but little effort in implementation: As frame-blocking behavior is acceptable, a global lock on the entire OptixContext is acquired, and state synchronization is invoked on every registered OptixObject. Due to the node- and reference-based design of the OptiX platform, synchronization must ensure that referenced objects are already up to date once the referrer is updated. Thus, objects are synchronized in order of potential dependencies: OptixTexture and OptixShader first, then OptixMaterial, OptixModel, thereafter all non-scene puppets, and finally any OptixScenes.

Similar to synchronization, RenderObject purging also locks the entire OptixContext. Thereafter, all shared pointers within the global set of OptixObjects are checked for uniqueness. Any unique shared pointer is not used by the client anymore. Thus, the pointer is released and the associated render-side object is destroyed. This process iteratively continues until no more OptixObjects could be deleted.

7.3.6 Lighting algorithm

Before any default programs of the raytracer can be presented, it is important to review the fundamental lighting algorithm that has been implemented within the raytracer kernel.

As described within the initial raytracing introduction, any realistic rendering process aims at finding a solution for the rendering equation

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(-\omega_i \cdot n) d\omega_i.$$

Solving the entire equation over a discretization of all points x within the scene typically is not possible due to the involved numeric complexity. Instead, most renderers solve but an approximation of the rendering equation. Within the approximation, a trade-off must be found between performance and quality of rendering.

Over the years, several lighting models have evolved that consider but certain parts of the above equation. The most popular of these models is the Blinn-Phong model [WPf].

Within Blinn-Phong shading, only discrete parallel or point light sources are considered in terms of generated light energy. The entire light distribution integral is further reduced to but three separately modeled effects: Ambient, diffuse, and specular lighting. Ambient lighting models transfer of general, direction-independent light. Diffuse lighting additionally considers the angle in-between incoming light and reflecting surface, and models light transfer on a completely dull surface. Finally, specular highlighting models transfer of incoming light into a single direction — such as characteristic for most shiny or metal surfaces. The final equation for light output at any given point then is composed by adding the effects of all three models.

This yields an approximative rendering equation

$$L_o = \sum S_a \, L_{a,i} + S_d \, L_{d,i} \, (D_i \cdot N) \; + \; S_s \, L_{s,i} \, (H \cdot N)^{\alpha}{}_{i=1,\ldots,n}.$$

Here, S_a, S_d and S_s are constants that describe the weight between ambient, diffuse, and specular behavior of the surface at the current point x (omitted above for brevity). Likewise, $L_{a,i}$, $L_{d,i}$ and $L_{s,i}$ represent respective components within the incoming light from a given light source i. In practice, these depend on the distance between x and the light source in accordance to some attenuation rule. The total number of light sources is indicated by n. D_i defines a vector from point x to the position of light i, and H denotes the reflection of D_i along the surface normal N at x. Finally, the exponent α controls the sharpness of the specular highlight.

Image 7.3 provides a more visual explanation of the Blinn-Phong shading model.

The most relevant benefits of Blinn-Phong shading are simplicity and performance. For instance, Blinn-Phong shading has been in use within the fixed-function pipeline of both DirectX and OpenGl standards. Consequentially, the Blinn-Phong model has also been chosen for implementation within the default lighting calculations within this thesis.

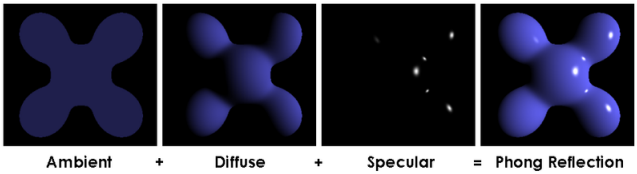


Figure 7.3: In the Blinn-Phong shading model, the final lighting is composed from the effects of ambient, diffuse and specular components. Image courtesy of [WPf].

7.3.7 Device-side light buffer

In the raytracer implementation, lights for use with the Blinn-Phong model are stored inside a custom-typed OptiX data buffer. The data buffer holds tightly packed structs of the following Light type:

```
struct Light
{
    float3 Position;
    float DummyPaddingA;
    float4 ColorAmbient;
    float4 ColorDiffuse;
    float4 ColorSpecular;
    float SpecularExponent;
    int Parallel;
    int HasShadows;
    int Enabled;
    int DummyPaddingB;
};
```

Most of the fields are self-explanatory or correspond to respective variables from the above Blinn-Phong model. Thus, only few special variables are discussed in the following:

The **Parallel** flag denotes the type of light: Parallel lights (such as the sun) are not attenuated with distance, and their **Position** component directly identifies the uniform direction of incoming light. In contrast, point lights are attenuated by distance, and the light direction D_i within the Blinn-Phong model is recalculated for each point x.

Specific lights within the data buffer can be enabled or disabled with the **Enabled** flag. This works around a problem within the OptiX API that would freeze the system once a buffer has been resized too often. Hence, the light buffer is statically allocated but once on context construction. Then, only a maximum number of

(most prominent) lights are selected within each scene and transferred into the buffer. Consequentially, unused lights within the statically-sized buffer simply are disabled.

The dummy padding fields require some further elaboration:

In C++ terms, the above **Light** structure is a POD (**P**lain **O**ld **Data**) type: It has but data members, no parent class, and no member functions. For POD types, the C++ standard makes certain guarantees on memory layout. A shared, standard memory layout is required for binary compatibility of light buffer data in-between host and device code.

Yet, while NVCC officially should match the memory layout of the host compiler, this does not seem to work in practice. Instead, differences in size on host and device side have been detected by the OptiX API for a naive Light struct. Consequentially, the rendering process was aborted.

Thus, the above padding fields have been inserted to ensure that the Light structure has the same internal memory layout both in device and in host code. Yet, this is not a preferred solution, and padding will have to be adapted when porting to new compilers or operating systems.

7.3.8 Default programs

As already mentioned before, default programs are used both for non-customizable parts of the raytracer implementation, and for materials that do not come with a user-defined programmable shader. In the following, all default programs are reviewed.

The first program invoked by the OptiX raytracing process is a hard-coded bounding box program that updates the acceleration structure of all Model-typed resources. The device-side variables **ModelVertices** and **ModelIndices** have already been set depending on the current object. Consequentially, the data for each triangle primitive within the current object can be retrieved and an appropriate bounding box is calculated and returned.

In the next step of the raytracing process, a predefined ray generation program is executed. The implementation of the default ray generation program within the OptixContext closely matches the ray generator of the example application in 5.3 and is not repeated here.

All resulting rays are sent through the geometry, where triangle-ray intersections are detected by a fixed ray-triangle intersection program. The original implementation used a naive, angle-based approach in this phase. Even though the naive approach successfully had been applied to other intersection detection problems, it exhibited numerical stability issues within raytracing. Thus, an optimized implementation based on barycentric coordinates had been adopted thereafter. In the final version, the hand-written intersection code was replaced by an even more optimized, more robust, but undocumented intersection helper routine from the OptiX API libraries.

Apart from pure intersection detection, the default intersection handler program determines the exact ray endpoint on each triangle, and calculates interpolated normals and texture coordinates. The respective device-side attribute variables are set for later use in shading.

Ray-triangle intersections trigger default closest-hit and any-hit handlers, unless these have been redefined by a custom shader attached to the respective model.

Behavior of default hit handlers depends on the type of incoming ray:

Primary camera rays do not react to any-hit events, the respective program directly returns. Once the closest intersection has been determined, the closest-hit program evaluates the Blinn-Phong lighting equation at the provided hit-point. In particular, all lights within the global light data buffer are considered. For each light with enabled shadowing attributes, an appropriate shadow ray is generated. If a light is not occluded, its contribution to specular, diffuse, and ambient lighting at the surface point is calculated. For point lights, contribution factors further are attenuated based on light distance. Thereafter, light contribution is merged with surface color effects. In terms of surface colors, the current implementation only considers a single 2D texture for ambient and diffuse effects, and a hard-coded white specular highlight. Results of lighting calculations are accumulated over all lights, and ultimately returned within the camera ray data.

In contrast to the more sophisticated camera ray behavior, both hit programs for shadow-typed rays match those of the example application: Once any intersection has been detected, the ray data returns an occlusion flag to the main lighting calculations.

Finally, default miss handler programs for camera and shadow ray types exhibit the expected behavior: Camera rays return a default black background color, while shadow rays indicate that the light source is not occluded.

7.3.9 Rendering process

Given the preceding OptiX integration, the realization of the actual rendering process — i.e. the implementation of the render function — is straight-forward.

At first, blocking behavior and thread safety requirements are maintained by a global lock on the entire OptixContext.

Thereafter, any command in the incoming command buffer must be processed. The RenderScene associated with each command is translated into an OptixRenderScene by dynamic casting. The corresponding OptiX group node of the OptixRenderScene is transmitted to the GPU by means of a global device variable. The OptiX buffer that contains light data is initialized with the most prominent lights contained within the scene. Other global parameters — such as the camera position and orientation — are initialized as well. The ray generation program loaded at construction is executed and calls both construction-loaded and runtime-loaded OptiX components for raytracing. In turn, the pixel buffer shared between OpenGl and OptiX is filled with raytracing results.

Once all commands have been processed, the GL pixel buffer with the final rendering is copied to the main application window by invocation of the OpenGl drawPixels method.

7.3.10 Renderer destruction

Whenever the application shuts down, the OptiX renderer must be destroyed again. This implies several steps:

1. The OptixContext releases all of its default programs, variables, and resources.
2. The OptixContext iteratively purges OptixObjects until no more objects could be deleted.
3. Consequently, any remaining puppets and resources are still referenced by the client and need to be detached to maintain safety requirements. This follows the interface suggestion of releasing the associated renderer-side objects (e.g. OptiX nodes and programs), and resetting the RenderObject internal RenderContext back-reference. Any access to OptiX API internals via the client-side RenderObject pointers then results in an appropriate C++ exception.

   This step is especially relevant for the later Java and Scala integration, as order-of-destruction rules within the VM regularly leak resources.
4. Finally, the internal Optix API handle is destroyed, and the GL context binding is released. Thus the entire OptixContext shut down cleanly.

   As with the OptiX example application, one must note that the final context release currently is not built into the delivered application due to OptiX-internal instability issues.

The destruction of the OptiX renderer concludes the implementation of the raytracing component. The next section recaps the entire implementation within a small, illustrative client application.

7.4 Example client application

In the following, a small C++ client application is presented as an example for raytracer usage. The application opens up a new window for rendering, a mesh and associated textures are loaded, and a basic scene is composed. Thereafter, code flow loops through rendering until the window is closed. Rendering is outsourced to a separate thread, and the use of a RenderCache resource manager is demonstrated. One should note that the client does only access renderer internals where necessary, such as on creation and technique management. Management of render-side objects is performed completely independent of any implementation via few, abstract calls.

```
DefaultWindow window("Caption::ExampleApplication");

OptixContext   optix     (window);
OptixTechnique technique (optix);
RenderThread   thread    (optix);
RenderCache    cache     (optix);
```

```
RenderCommandBuffer commands;

SharedPointer<StaticModel> castle_model
                        (cache.requestModel("castle.dae"));

SharedPointer<RenderScene>  scene  (optix.createRenderScene());

SharedPointer<ViewerPuppet> viewer (optix.createViewerPuppet());
scene.insert(viewer);

SharedPointer<LightPuppet> light (optix.createLightPuppet());
scene.insert(light);

SharedPointer<StaticModelPuppet> castle_puppet
                    (optix.createStaticModelPuppet());
castle_puppet.setStaticModel(castle_model);
scene.insert(castle_puppet);

while (!window.getExitFlag())
{
    viewer->move(Vector(1, 0, 0));

    if (thread.isIdle())
    {
        optix.synch();
        optix.purge();

        commands.add(RenderCommand(scene, viewer, technique));
        thread.render(commands);
        commands.clear();
    }
}
```

Figure 7.4 shows the results of raytracing within the example application.

7.5 Testing suite

To ensure correctness of the raytracer kernel and the remaining framework, a C++ testing suite has been designed in the proceedings of this thesis. This suite is delivered as a standalone executable, and performs both component and integration tests. Tests consider basic functionality, such as hash maps or shared pointers, and higher-level classes, like system counters, render-side resource, or the actual rendering function.

In this context, one must note that validating the correctness of a rendering process is not an easy task. For instance, it is not possible to pre-render a single reference screenshot of some complex scene, and then regenerate that screenshot within automated testing. In particular, even the same executable on the same operating

Figure 7.4: A demonstration scene rendered by the raytracer implementation. Particular attention is directed to the hard-edged shadows that naturally arise from raytracing.

system may generate slightly different rendering results depending on inaccuracies and performance optimizations within graphics hardware from various vendors.

Thus, raytracer testing applies an approximate comparison between desired rendering results and the actual testing render. This only aims at finding harsh incompatibilities or bugs. For instance, one such bug in the CUDA compiler generated completely white surfaces on certain texture lookups during development of this thesis.

8 System Integration

The preceding chapters introduced the interface of the raytracer kernel and provided its C++ implementation. This chapter in turn integrates the finished raytracer component into the Simulator X framework.

In particular, as Scala is only a Java wrapper itself, two separate steps are required: First, the C++ raytracer kernel is encapsuled into Java classes and methods. Then, resulting Java calls must be integrated within the Scala code of Simulator X. Both steps are discussed in order. Finally, this chapter explains certain noteworthy caveats that were discovered during the wrapping process.

8.1 Java wrapper

The first step of the raytracer integration within Simulator X involves the translation of C++ interfaces and classes to corresponding Java representations.

In this section, the translation step is discussed in detail: Initially, both alternatives for C++ to Java wrapping — the JNI and JNA platforms — are reviewed in regards to respective benefits. Consequentially, the choice for the JNI route within this thesis is motivated. Thereafter, the concepts behind the translation of C++ specialties to Java code are given, and few example interfaces are detailed both on native and Java sides.

Thus, at the end of this section, the entire raytracer is ready for use in Scala and the Simulator X application.

8.1.1 JNI and JNA solutions

As mentioned before, there are two state-of-the art approaches for translation of C++ code to Java wrappers: both the JNI and the JNA platform allow for the binding of native language level code. As these are two competing platforms, the respective advantages and disadvantages are to be reviewed before settling on either solution.

JNI (**J**ava **N**ative **I**nterface) was the original method of native code use within the Java language [CA99]. JNI is based upon automated binding to a native C or C++ library, where the access points into the library are generated at compile-time by a Java-side preprocessor.

Using JNI, one first creates a Java class that encapsulates the desired native functionality. Within this class, certain methods — both static and object-bound — are tagged with the **native** keyword. The implementation for native functions is omitted within the Java source. Instead, a sole static initializer block that loads the later native library is inserted into the class.

Once the Java class is completed, the class is compiled and the resulting VM byte code is fed through the **javah** preprocessor. The preprocessor, in turn, generates a C header which provides method signatures for implementation in the native library. In particular, the native header gives direct access to the internals of Java objects, arrays, and variables. Thus, no performance is lost on intermediate data conversions.

This is the major advantage of the Java native interface — but comes at the price of a convoluted and error prone implementation workflow.

JNA (Java native access) was developed as a less effort-intensive alternative to the original JNI interface [JF98].

JNA does not require any additional boiler-plate, native language code for Java collaboration. Instead, any methods exported by name from a native library can be invoked automatically. The only requirement here is that a respective native library for the current operating system has to be loaded explicitly via a runtime code switch. Then, Java code may call all exported methods from the native library. JNA automatically wraps any outgoing parameters into native representation, and likewise, any incoming return values are translated into Java VM variables and objects.

Yet, while the JNA approach is much easier to handle, it has several important shortcomings for the integration of a rendering module. Most importantly, the wrapping process for shared pointers is even more problematic with JNA, as another layer of indirection is required between the actual C++ pointer object and its Java side representation. Additionally, both the translation of data in between Java and native formats and the per-call by-name function lookup take their toll on application performance. For instance, any Java-side array may be encapsulated into an intermediate copy within a C-style pointer array before it is passed on to native code.

After both alternatives have been considered, JNI was chosen for the actual integration work for several reasons: For one, the rendering core should be as efficient as possible. Especially for the transfer of application-generated resource data to the renderer the performance impact for data duplication is relevant. Next, the workload and complexity of the rendering component itself and of its Scala integration already outweighs the integration complexity of the JNI native-side library header. JNA in this context seems more ideal for the integration of one-shot system calls than for the development of an entire system glue layer. Finally, once in place, further changes to the wrapping layer only are required on rare occasions. Thus the more dynamic binding allowed by JNA is of no use for the raytracer integration.

8.1.2 JNI realization

After the choice of JNI has been elaborated, the JNI binding layer between the Java language and the native raytracer application can now be discussed.

In general, each of the language-independent interfaces defined in chapter 6 is represented in the Java wrapper as well. For instance, there are Java classes `RenderContext`, `RenderCommandBuffer`, `RenderObject`, and `StaticModelPuppet`.

Internally, each of the Java classes only houses a reference to a corresponding native C++ class instance.

Any Java method signatures closely match those of the C++ interfaces given in chapter 7. To continue the preceding example, the Java RenderContext has a public `render` method which accepts a Java RenderCommandBuffer instance as parameter. There are but two relevant exceptions within method parameters: Strings go by the Java `String` class, and C++ shared pointers are replaced by Java object references.

Most Java methods within the Java-side interfaces are tagged with the JNI native keyword. Consequentially, these call native code implementations. The native implementation for Java classes first converts certain parameters — e.g. Strings or Java object references — from their Java representation to a representation compatible with the C++ interfaces. In turn, the instance-internal native object reference is retrieved. Finally, the actual C++ implementation from chapter 7 is invoked on the native C++ object.

While this process seems intuitive at first, the attachment of C++ instances to Java objects requires some further elaboration.

On native code side, object lifetime management is performed by shared pointers. These cannot be included directly into a Java object due to binary incompatibilities. Instead, a more intricate scheme is required. On binding a C++ SharedPointer to a Java object, a new copy of the input SharedPointer instance must be created on the heap. The respective native address (i.e. a pointer to the SharedPointer copy) can be converted into a Java `jlong` typed intrinsic variable on all current 32 and 64 bit architectures by bit-based casting. Finally, the resulting integer can be stored inside a Java member field.

Access to the stored pointer in turn is performed intuitively by reading out the pointer's address from the respective Java object field again.

Destruction of the encapsulated native pointer — corresponding to an object release — is more problematic, though. Two different strategies have been applied within this thesis: mandatory manual release, and automated destruction on finalization.

The former case defines a separate `release` method within each Java object instance. Releasing any Java wrapper destroys the internal shared pointer, and thus releases a single native reference. The Java object remains in an unusable but safe state. Any further invocations on the object throw a runtime error. The client application must call the release method on any Java-side object before it becomes inaccessible.

Within the latter case, the release functionality is automatically triggered by the `finalize` method of the Java object. This ensures that native bindings are deleted even for unreachable Java objects. Thus, any memory leaks are avoided. However, Java finalizers incur both order-of-destruction and stability issues. Therefore, a corresponding error message about leaked resources is still printed into the application log.

Albeit the above lifetime management strategy was the best of but few possible implementations, it still exhibits certain problems. These are discussed in 8.3.

8.1.3 JNI examples

For illustrative purposes, the following three code excerpts present certain aspects of the JNI wrapper realization. As with previous sections on pure implementation work, a complete code coverage is outside the scope of this thesis. Further details can be found within the code documentation and the program sources on the accompanying delivery medium.

As a trivial case, loading a new material from within the Java `Material` class

```
public class Material extends RenderResource
{
    /** Load material from indicated path. */
    public native void load(String path);

    /** Destroy internal material representation. */
    public native void destroy();

    /** Address of SharedPointer<Material> */
    private long InternalPointer;
}
```

wraps to the below native implementation:

```
JNIEXPORT void JNICALL Java_de_ubt_optixwrap_Material_load
  (JNIEnv *env, jobject obj, jstring path)
{
    /** Get C++ object pointer within Java object. */
    jclass cls = env->GetObjectClass(obj);
    jfieldID fid = env->GetFieldID(cls, "InternalPointer", "J");
    jlong intp = env->GetLongField(obj, fid);

    VoidApi::SharedPointer<Material> *mat =
        reinterpret_cast<VoidApi::SharedPointer<Material>*>(intp);

    /** Get C string for path, and load on C++ object. */
    const char *cstr =
        env->GetStringUTFChars(path, 0);
    mat->load(cstr);
    env->ReleaseStringUTFChars(path, cstr);
}
```

Any error handling has been omitted for brevity. The rather large native function name has been auto-generated by the JNI wrapping mechanism, and considers package, type, and method name of any natively bound method.

Material creation within a RenderContext and native-code destruction use similar access schemes, but instead set or reset the internal pointer address. Consequentially, these are not presented here.

Unlike the rather simple Model, the mapping of RenderPuppets — such as StaticModelPuppet — requires a more sophisticated Java integration. Namely, the Java class must hold two different pointer addresses, `InternalPointer` and `InternalPuppetPointer`:

```
public class ModelPuppet extends RenderPuppet
{
    /** .... */

    /** Address of SharedPointer<ModelPuppet> */
    private long InternalPointer;

    /** Address of SharedPointer<RenderPuppet> */
    private long InternalPuppetPointer;
}
```

Within the implementation, these are required for insertion of puppets into a RenderScene — an unapparent imperative that is discussed in the following.

Peculiarly, a C++-side RenderScene expects a shared pointer to RenderPuppet. Usually, this pointer is created by copy-construction from the original, type-specific pointer. In this context, note that shared pointers for parent and derived classes do not have any inheriting relationship — for instance, SharedPointer<RenderPuppet> is not the parent class of SharedPointer<ViewerPuppet>.

Yet, only the base RenderPuppet type is known for any incoming puppet on scene insertion. The actual, most-derived type is not available. Consequentially, it is not possible to retrieve a correctly-typed, shared pointer to the incoming puppet from the internal pointer address. Because pointers do not remodel derivation relationships, it neither is sufficient to simply cast the type-specific InternalPointer address of any RenderPuppet to a SharedPointer<RenderPuppet> instance. Instead, the second RenderPuppet-typed pointer must be used.

8.2 Scala integration

The preceding section detailed the realization of the fundamental Java wrapper around the C++ rendering core. This wrapper intuitively is included into the Scala-written Simulator X: As Scala is binary compatible to the Java language, the wrapper JAR archive can directly be imported into any Simulator X component.

This leaves but two tasks for this section: For one, a Simulator X actor and a new rendering component are designed around the JAR wrapper of the OptiX raytracer. This concludes the actual development process within this thesis. At last, the new Simulator X rendering component is integrated into the logics of a select existing application alongside the current jVR rasterizer to allow for later performance evaluations.

8.2.1 Raytracing actor

The development of a new actor component for use within the Simulator X framework is a rather intuitive task. To be specific, Scala itself already offers actor types and general message passing facilities by means of an actor library. Simulator X extends

on Scala internal features by the integration of another, higher-level convenience layer. Hence it suffices for a component developer to derive from a standard base class and to implement but few event handlers.

In terms of this thesis, a new raytracing actor `OptixRenderActor` has been developed. In its current implementation, the raytracing actor implements five different handler types, two direct and three indirect ones.

The direct handlers concern renderer configuration and frame rendering. Renderer configuration creates the main OptixContext, initializes its bound WindowContext, and sets a provided display resolution. Once an OptixContext has been created and configured, the main application actor must trigger the frame-rendering handler by sending a single `RenderNextFrame` message to the OptixRenderActor. This message type is defined universally within the Simulator X platform, and is not specific to the jVR rendering actor. Thereafter, rendering on the Optix raytracing actor runs continuously by self-triggering via an appropriate message. This is demonstrated in the below Scala excerpt:

```
addHandler[RenderNextFrame]
{
    case RenderNextFrame(sender) =>
        val command = new RenderCommand(scene, viewer)
        internalContext.render(Array(command))
        Actor.self ! RenderNextFrame(Actor.self)
}
```

Certain of the above instructions seem unusual, but are Scala language features. For instance, the ! operator allows for sending object instances to running actors by the default messaging mechanism.

The implemented concept of automated, self-triggered rendering is best suited for later performance testing. Yet a later enhancement might integrate a waiting instruction to avoid the system load of imperceptible, excess frame rates. In this context, one must again note it is imperative that the `render` call does not even block within the Java interfaces. Otherwise, the message processing loop of the raytracing actor could stall.

Scene and viewer variables in the above code excerpt are initialized within the remaining two handlers. These handlers are customized indirectly by means of overloading, and notify the rendering actor about new entities and state variables of a certain raytracer-specific aspect. The choice of a separate aspect for raytracer state variables is discussed in detail but in the next section.

On notification about a new Simulator X entity, the raytracing actor updates its internal OptiX scene accordingly. In particular, a new OptiX wrapper puppet is created — which in turn corresponds to a JNI-bound C++ instance that holds all OptiX API data. The puppet is initialized with any parameters that accompany the entity creation message. Thereafter, the puppet is placed into an internal lookup map

alongside its bound entity. Finally, scene and viewer variables are set appropriately once an object of either type is created.

The below code example demonstrates the insertion of a light-typed puppet within the handler for new entities:

```
def createEntity(e : Entity, c : TypedCreateParamSet)
{
    c.aspectType match
    {
        case SemanticSymbols.aspects.optixLight =>

            val spot   = c.valueFor(Optix.lightSpotFlag)
            val pos    = c.valueFor(Optix.lightPosition)
            val dir    = c.valueFor(Optix.lightDirection)
            val col_a  = c.valueFor(Optix.lightColorAmbient)
            val col_d  = c.valueFor(Optix.lightColorDiffuse)
            val col_s  = c.valueFor(Optix.lightColorSpecular)
            val shadow = c.valueFor(Optix.lightShadowFlag)

            val p = new OptixLightPuppet
                (spot, pos, dir, col_a, col_d, col_s, shadow)

            scene.add(p)

            entityToPuppetMap = entityToPuppetMap + (e -> p)

        /* ... */
} }
```

State variables of entities are created but after the respective entity has already been reported to the raytracing actor. Therefore, there already is a corresponding, construction-initialized puppet instance inside the entity-to-puppet lookup map. Consequentially, the raytracing actor retrieves the puppet for the owner entity of the state variables from its lookup map. Finally, all state variable are bound to the puppet, each by means of two functions: one function updates a puppet parameter from a state variable, the other updates a variable from puppet data. Both functions are later on called in terms of general application glue logics.

The next code excerpt illustrates state variable handling for few light parameters:

```
def processSVar(e : Entity, s : Symbol, c : TypedCreateParamSet)
{
    c.aspectType match
    {
        case SemanticSymbols.aspects.optixLight =>

            val p = entityToPuppetMap(e)
```

```
            val pos = e.get(Optix.lightPosition).get
            addSVarUpdateFunctions
                (pos, p.setPosition( _ : Vector3f ),
                      p.getPosition)

            val dir = e.get(Optix.lightDirection).get
            addSVarUpdateFunctions
                (dir, p.setDirection( _ : Vector3f ),
                      p.getDirection)

        /* ... */
} }
```

The last of the handlers within the OptixRenderActor is invoked once an entity is to be removed from the raytracing actor's local representation. In this case, the actor clears the entity from its internal map, and explicitly destroys any associated instances of OptiX wrapper puppet.

8.2.2 Application integration

In contrast to the effortless creation of the final raytracing actor, the integration of the raytracing component into existing applications required more work.

As already mentioned in chapter 4, most of the current applications already bind hard-coded jVR-specific state representations onto their entities. Consequentially, it is not sufficient to just replace a single class within a constructor call to translate all applications to the new raytracer. Instead, three alternative routes have been identified: Integration of a new, general aspect type, integration of a new, raytracing-specific aspect type, or conversion of aspects from the current jVR rasterizer.

The integration of a general aspect type for use both with the jVR renderer and the OptiX raytracer promises most benefits: Rendering is decoupled from the back-end implementation, and any further rendering components are intuitively integrated. However, this approach requires at least a partial incision into overall system design. Hence, it was not realized in terms of this thesis, but deferred as a suggestion for later enhancement.

Choice between a raytracer-specific aspect type and conversion of the existing jVR aspects has been tough. The former gives a more clean design and acts as further proof for the intuitive use of the Simulator X environment. The latter allows for instant application of raytracing to all existing applications, and does not risk the discarding of even more code once a general rendering aspect is devised.

In the end, the integration of a raytracer-specific aspect type was decided, mainly under the argument of glitches introduced by the conversion of jVR-specific structures. Consequently, a new aspect had to be integrated alongside the other aspects of the chosen example application. The new aspect type already has been included in the development of the core raytracing actor. Thus, the remaining task was to

step through all entity interactions within the example application while inserting raytracer counterparts to all rasterizer state variables.

For example, the below world description entity, as instantiated within the main application loop, has been outfitted both with a jVR and an OptiX representation:

```
new EntityDescription(
    EntityAspect(
        Symbols.graphics,
        new TypedCreateParamSet
            (SemanticSymbols.aspects.shapeFromFile,
             JVR.geometryFile <= levelfile,
             JVR.initialPosition <= ConstMat4(Mat3x4.Identity),
             JVR.manipulatorList <= shader,
             JVR.scale /**/<= ConstMat4(Mat3x4.scale(100)) ),
        Ontology.transform isProvided),
    EntityAspect(
        Symbols.raytracer,
        new TypedCreateParamSet
            (SemanticSymbols.aspects.optixModel,
             Optix.modelPath <= levelfile,
             Optix.initialTransform <= ConstMat4(Mat3x4.Identity)),
        Ontology.transform isProvided),
    /* ... */
    NameIt("world")
).realize
```

In this case study, the OptiX model implementation is not compatible with an explicit jVR shader (or manipulator in the above excerpt). Instead, material information is directly extracted from the input Collada level file.

Similar operations had to be performed on each graphics-based entity description. Thereafter, raytracing support had fully been integrated. Figure 8.1 shows a still image from the final, raytraced application.

The connection of existing Simulator X applications with the raytracing component concludes all implementation and integration tasks. The raytracer attained a state suited for evaluation — as performed in the next chapter.

8.3 Caveats

As a counterpoint to the above successful integration, the following section names certain problematic integration aspects that have not completely been solved during the wrapping process.

8.3.1 Shared pointer wrapping

The most problematic point was the translation of the shared pointer semantic for automated object management. As mentioned before, even the current implementa-

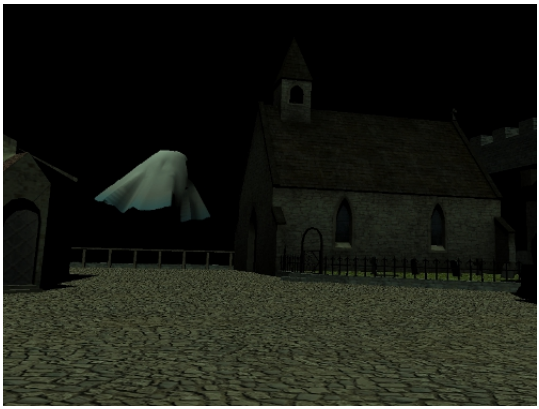


Figure 8.1: The example application, as raytraced by the OptiX kernel. Note that scene lighting has been modified for better visibility.

tion is not free from issues. In the following, the respective problems are elaborated in greater detail.

The actual difficulties are caused by differing life-time management strategies in C++ and Java:

C++ shared pointers define a rather stringent, reference-counted strategy, where destruction of objects is ensured once the last reference is dropped. On object destruction, certain OS- or library-level operations must be performed to release unmanaged resources — such as OptiX API objects. For this purpose, there are user-definable object destructors that are automatically invoked on object deletion.

In contrast, Java has a rather relaxed garbage cleanup system that releases unreferenced objects at some unspecified later time. Even worse, Java provides no reliable notifier once any object actually is deleted. There is an overridable `finalize` method, but its use generally is disadvised due to stability reasons.

This leaves but two choices: Either force manual destruction of resources — which contradicts the general Java language concept of automated resource management — or specify error prone finalization handlers.

Consequentially, both options have been implemented in this thesis. As stated before, the client application must manually destroy each Java object before destruction. In this case, the later finalizer recognizes a zero value inside the internal pointer representation, and takes no further action. Therefore, stability issues on normal shutdown are avoided. Yet, if the finalizer detects an alive pointer, the corresponding native object still is deleted. In turn, any leaked resources are freed.

The above tradeoff has long been deliberated. The final decision was made under consideration of accepted resource management strategies within the Java language. Java was not originally designed for low-level resource management. Corresponding

patterns — most notably exception safety under general runtime exceptions — are not common knowledge amongst Java programmers. Therefore, the dual strategy of manual release and automatic finalization offered good prospects.

Yet, even with the in-place resource management strategy one problem remains for leaked Java objects: The Java VM makes no guarantees on the destruction order of objects during garbage collection. This conflicts with the suggested order of destruction for resources and the owning RenderContext in the native raytracer kernel. Thus, the detach functionality — already integrated in the preceding interface discussion — had to be implemented.

8.3.2 Large data transfers

Unlike shared pointer wrapping, large data transfers did not cause a stability, but a performance problem. In particular, the transfer of runtime-generated resource data to the rendering system already requires at least a single copy into the client-side state buffer. However, even when transferred through the more efficient Java JNI interface, at least another copy is involved. Depending on the implementation of the underlying Java virtual machine, this copy may even run through unoptimized Java bytecode instead of an optimized CPU copy operation. While this is a basic language problem, there is an intuitive workaround: Any code working with resource data should be implemented in C++ and exposed via another Java wrapper. This is a reasonable suggestion considering that CPU resource modification code anyways means more number crunching than the higher-level Java language was designed for.

8.3.3 Exception wrapping

Exception wrapping turned out to be a problem as well. While it is possible to wrap C++-style exceptions into Java exceptions, this also means that each C++ exception explicitly has to be wrapped into a Java class. Thus, any additional exception in turn requires an appropriate modification to the Java wrapping code. Within the current design, this has been solved by catch-all blocks within native code. These map any incoming C++ exception into the Java **OptixException** type, and the original cause of the exception is lost. The most problematic aspect of this approach is that certain non-critical error conditions — such as a non-existent resource file — are signaled by exceptions as well. A better solution of this wrapping mechanism is left for future work.

9 Evaluation

In the preceding chapter, the OptiX-based raytracing component has been integrated into the Simulator X environment next to the existing jVR rasterization component. Furthermore, a major example application of the Simulator X platform has been connected to both components for later evaluation.

Said evaluation is performed within the following chapter. At first, the example application is introduced in more detail. Thereafter, the actual evaluation takes place: In consequent sections, both components are analyzed in regards to their general features, graphics quality, rendering performance, and overall stability. Each section ends with a short conclusion on the respective advantages and disadvantages of raytracing and rendering.

9.1 SimThief example application

The example application that forms the framework for later evaluation is a game named SimThief. It realizes a first person ghost hunting scenario: The player and a series of ghosts compete against each other in the courtyard of a medieval castle.

The AI-controlled ghosts pursue two different goals: On the one hand, few ghosts hunt the player. If a ghost manages to touch the player, the player's health is reduced. On the other hand, the remaining ghosts collect powder barrels distributed throughout the virtual environment to destroy a drawbridge at the castle exit. The game is lost either if the user's health is depleted, or if the exit has been destroyed.

The player must use magic to fend off all ghosts while avoiding ghost contact and while protecting the bridge exit. Once all ghosts have been chased away, the player wins, and the game starts anew.

Figure 9.1 provides some in-game screenshots for further illustration.

From a more technical point of view, SimThief acts as a proof-of-concept client application for the Simulator X framework. It employs several, loosely coupled components to simulate the virtual environment: An OpenAl component handles sound, a separate physics component detects environment collisions, and an input component allows for connection of various controlling devices from WiiMotes to traditional computer keyboards. In terms of higher-level program flow, a general AI component controls all ghosts, while a standalone game component encapsulates remaining logics such as victory conditions. Finally, the jVR rasterization component has been providing graphics up to now.

9.2 General features

In terms of general features, both jVR and the OptiX raytracer have their respective benefits and disadvantages. In this section, various feature aspects will be revisited for a conclusion: At first, language-specific differences in-between the competing modules are reviewed and portability concerns are presented. Thereafter, the general interface design and multithreading strategy of the components are contrasted.

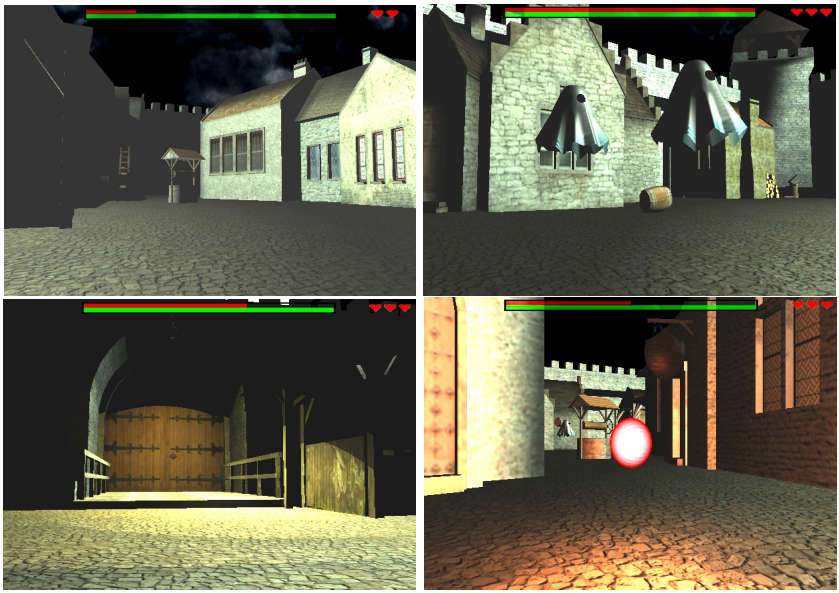


Figure 9.1: The above screenshots show various components of the SimThief game: The castle courtyard, two ghosts and a barrel, the drawbridge exit, and a fireball spell. Once more, scene lighting has been modified for better visibility.

9.2.1 Language-specific aspects

Most of the differences between the rasterizer and the raytracer component are motivated by the choice of implementing language:

The new OptiX raytracing kernel has been written in C++. Consequentially, it inherits all benefits and drawbacks of a native language implementation: Offline, thorough program optimization promises performance advantages over just-in-time compiled or interpreted bytecode. Yet, the entire rendering kernel must explicitly be ported and recompiled on any alternative operating system.

Further disadvantages are induced by the integration of the native C++ raytracing library with the Scala Simulator X environment. Functional shortcomings have already been discussed in the preceding chapter. However, other aspects of development suffer from the integration as well: For instance, application debugging is seriously complicated. In the main development environment, two separate development environments have been used for native and cross-platform parts of the OptiX raytracer. Consequentially, it was not possible to monitor stack traces over language boundaries. Instead, the native language IDE remotely had to connect to the running Java process. During integration of the raytracer within the SimThief game and during final debugging, this error-prone strategy interacted poorly with the less than

optimal stability of the OptiX API

In contrast, jVR is written entirely in Java. Therefore, it can conveniently be developed from within the same IDE as the core Simulator X system. Debugging over the language barrier between Scala and Java is not a problem, since both rely on the same virtual machine and data representation.

In terms of portability, interpreted languages are the ideal showcase — the jVR renderer in theory works on any system that has an appropriate Java VM implementation. There are two caveats involved though: Console porting and back-end jOGL support.

For one, a large part of the market for entertainment and virtual reality titles currently consists of gaming consoles. These do not support a Java VM, but only feature other interpreted and native languages. Porting for console thus means a complete rewrite of the entire Java-side simulator, while the C++ rendering interface could be retained with but the introduction of a new back-end.

Second, even the jVR renderer relies on back-end, native-level OpenGl binding libraries. These are imported by means of JNI in a similar way to the raytracer interfaces — albeit at a much lower abstraction level. Therefore, the jVR component requires system-level support for the jOGL add-on library, which currently is restricted to run-of-the-mill customer computers and Solaris SPARC workstations.

9.2.2 Architecture comparison

In terms of interface and architecture, both components are reasonably intuitive.

jVR's rendering pipelines allow for higher-level customization on the application code layer. In contrast, the OptiX raytracer component is not open to run-time customization of the raytracing process due to limitations within the OptiX API. Instead, a fixed raytracing implementation is combined with user-programmable and runtime plug-able materials to achieve at least partial visual flexibility.

In the context of the SimThief game, only a single jVR pipeline has been used throughout the development of this thesis. Consequentially, differences in rendering flexibility have not been obvious. Yet, these probably will become more prominent once other jVR rendering strategies — such as anaglyph mode — are to be integrated into the OptiX raytracer. With the base RenderTechnique class, the general C++ rendering interface at least provides an appropriate expansion point already.

Similar to the increased flexibility of rendering pipelines, jVR's exposed scene graph system provides greater customizability at the expense of intuitive use. Unlike rendering pipelines, the SimThief application however does not even consider more advanced scene constellations than flat object pools. Thus, the general interface of the OptiX raytracer seemed more appropriate here.

In contrast to its less flexible scene and resource management facilities, the OptiX raytracer provides a more elaborate multithreading scheme than the jVR rasterizer. In particular, the per-window thread within the jVR rasterizer is opposed to arbitrary client threads for the OptiX raytracer. Yet, SimThief again only takes advantage of basic out-of-thread rendering to avoid a stall on the message processing loop of the

Figure 9.2: Unsupported mipmapping induces moire effects in the left-hand side OptiX rendering. In contrast, texture filtering within the right-hand side jVR image creates smooth surfaces even for far-away objects. Images have been brightened for better visualization.

rendering actor. Therefore advanced multithreading features currently can not be evaluated in depth.

In total, general features seem balanced over both components. Within the development process of this thesis, the jVR rendering component appeared to be slightly more easy to handle. Still, this might contribute to the fact that the jVR component was readily integrated into the system — while the raytracer was actually coded and debugged.

9.3 Image quality

Image quality between both approaches is comparable, with but little benefits for the established jVR renderer. Contrary to expectations, these are not a result of the more flexible jVR rendering system, but a consequence of certain missing features within the OptiX API or the raytracer implementation.

For instance, one of the most notable differences in image quality results from missing support for state-of-art texture filtering and mipmapping techniques within the OptiX API. This has already been mentioned in chapter 7: Unlike OpenGl (e.g. jVR) and many other rendering platforms, OptiX only supports a single texture level and linear filtering on this texture level. Consequentially, high-contrast textures on distant objects or angled surfaces tend to produce aliasing artifacts or moire patterns. This becomes particularly evident with the low altitude camera within SimThief and the brick pattern on the floor of the castle courtyard. Figure 9.2 illustrates this problem.

Other graphical glitches are caused by duplicate and z-fighting triangles, missing OptiX shader implementations for spell surfaces and user interface components, and small inconsistencies in specular highlight calculations. Apart from these and few other artifacts, the OptiX raytracer and the jVR rasterizer create comparable images.

This is attributed to the fact that both implementations use the same Blinn-Phong lighting model, the same surface model, and similar hard-edged shadowing strategies.

9.4 Performance

Features and image quality are relevant for any kind of rendering software. However, for interactive applications, acceptable performance is an even more important criterion.

Two different machines have been involved in performance tests: On the one hand, a more recent computer with a Core-i5 CPU from Intel's current Sandy Bridge lineup in combination with one of the latest NVIDIA 5xx consumer graphics boards represents a modern, high-end gaming system. On the other hand, a two-year-old system corresponds to the average consumer hardware. Here, a legacy Core-i7 Lynnfield CPU is complemented by a GPU from NVIDIA's predecessor 4xx board line. Both systems run on Windows 7 Ultimate, the latest Detonator graphics driver 275.33 has been installed.

The first relevant performance test had been executed even before the first rendered image was displayed on screen. In particular, the startup time for both components — including the loading process of the initial scene — has been measured.

On the faster testing system, the jVR rasterizer took about 10 seconds for startup. In contrast, the OptiX raytracer required but half the time on the same system. Similar timings have been achieved on the legacy testing rig, with an increase of loading times by less than a second for either rendering component.

Differences in timing between both components are attributed to two facts: For one, general loading requires much number-crunching, which is handled better by machine-specific optimizations in the C++ compiler. Second, the jVR renderer has to load and compile additional shaders that currently are not available for the OptiX component.

In general, both loading times seem high under consideration of the terse SimThief geometry with but ten-thousands of triangles. No further in-depth timing was performed apart from basic startup measurements. Still, long loading times potentially are induced by the complexities of the Collada XML geometry input: For the separation of whitespace-separated coordinate streams within the XML, both C++ and Java implementations repeatedly create small and inefficient String objects on the heap. The problems of Collada import become even more evident if the C++ OptiX raytracer is compiled in debug mode: All memory allocations then are supervised by the C runtime, the corresponding checks increase loading times by an order of magnitude to little less than a minute.

After startup timings, online performance testing commenced. For each performance test, the SimThief game was played for exactly five minutes. The resulting total number of frames was accumulated to determine the average framerate over the entire testing run. At the same time, a running fps counter was calculated to find the respective maximum and minimum framerate over the entire testing run.

The results of performance testing are shown in the below tables:

Intel Core i5-2500, 8 GB RAM, NVIDIA GeForce GTX 580:

Resolution	jVR fps			OptiX fps		
	min	avg	max	min	avg	max
800 × 600	64	83	85	32	55	59
1024 × 768	53	62	64	21	30	31
1280 × 1024	41	46	51	19	22	24
1680 × 1050	39	41	42	15	17	19
1920 × 1080	31	34	35	11	12	12

Intel Core i7-860, 4 GB RAM, NVIDIA GeForce GTX 460:

Resolution	jVR fps			OptiX fps		
	min	avg	max	min	avg	max
800 × 600	59	76	81	27	48	52
1024 × 768	46	54	54	24	27	32
1280 × 1024	34	41	43	18	19	22
1680 × 1050	31	39	45	12	14	17
1920 × 1080	33	36	37	9	10	13

Average framerates for both the new raytracer and the traditional rasterizer are acceptable over lower resolutions. In these cases, there is almost no human-perceivable difference in performance between both algorithms — even though there already is quite some absolute framerate distance. In contrast, for the largest resolutions the higher per-pixel cost of raytracing seems to outweigh the decreased triangle setup costs even more and framerates drastically break in on the raytracer.

Analysis of minimum and maximum framerates over the testing course indicates that there are no special bottleneck situations either for the jVR or for the OptiX component. Yet, this is attributed to the restricted SimThief game environment: The castle level geometry is rather balanced, and viewing depth is restricted. Consequentially, the weak points of both algorithms are not captured. On the one hand, there is no highly tesselated or depth-overlapping object that challenges the rasterizer component. On the other hand, there are no spatially varying tesselation depths that could disturb raytracer tree generation.

In total, performance evaluation leads to the conclusion that jVR and the Optix component are but little different in terms of rendering speed for low to medium resolutions, with a huge benefit for rasterization on large resolutions.

9.5 Stability

The preceding evaluation has demonstrated that interactive raytracers are more and more catching up to traditional rasterization approaches. While there still are ob-

```
! VoidApi::OptixContext::render: Render: Unknown error (Details: Function "_rtCo
ntextLaunch2D" caught exception: Encountered a CUDA error: driver().cuGraphicsMa
pResources( static_cast<unsigned int>( m_interopResourceVector.size( ) ), &m_int
eropResourceVector[ 0 ], stream ) returned (208): Already mapped, [3735709])
```

Figure 9.3: This OptiX-internal error message indicates that sampling from a 3-byte RGB texture currently is not supported.

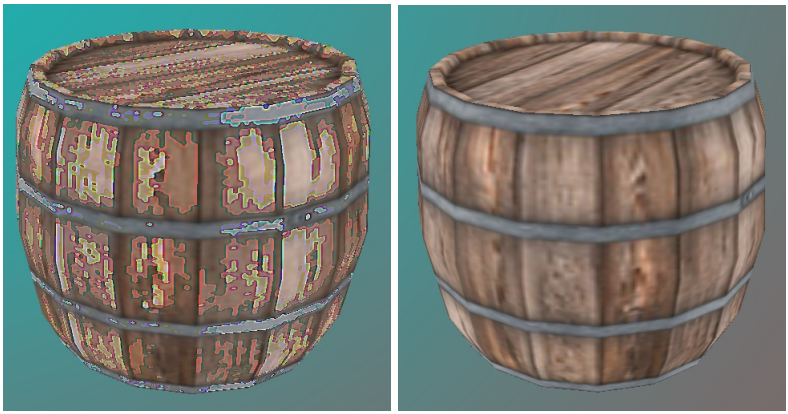


Figure 9.4: The left-hand sampling from a 4-byte RGBA texture does not entirely crash rendering, but produces visible sub-texel sized artifacts on the barrel model. The right-hand image shows the correct rendering with an uncommon, memory intensive floating point texture.

servable gaps in features, image quality, and performance, the current generation of interactive raytracers already comes close to established rasterizers. Within the foreseeable future, the continuing trend will have raytracing on par to traditional rasterization in these aspects.

Yet, another vital argument against the general use of raytracing in productive scenarios has not been named yet: In terms of stability, raytracers — and in particular the OptiX API — still have quite some deficits.

When compared to the original jVR raytracer, the OptiX API requires much more boiler-plate code for much less functionality. This implies more potential for errors. At the same time, stability of the OptiX-internal, GPU-based raytracing kernel lacks in comparison to the robustness of the mature OpenGl API.

For further justification, one should reconsider the OptiX scene hierarchy. It has already been stated in chapter 7 that particular care is required to ensure a complete hierarchy.

Most noteworthy, if any type of expected node is missing, behavior ranges from completely blank screens to outright system crashes. The former are caused by errors

found during automated OptiX-internal validation and come with a rather general, unhelpful error message alongside aborted rendering. The latter are produced by problems that are not identified within the validation preprocess.

Yet, a virtual scene from productive software could potentially consist of thousands of nodes. Enforcing scene consistency in this example is not trivial — even less so under the influence of a pure C API and the presence of C++ exceptions. Further complications are caused by the prohibition of empty nodes. In effect, once a single OptiX scene node has been removed, all higher-level nodes must be purged, and removed if empty. Any slight mistake in client-side code can require a complete system reboot. This error robustness must definitely improve, otherwise no serious software studio will consider using the OptiX API for critical programs.

In contrast, the mature OpenGl API provides a better tradeoff between input validation and stability. If any of the input GL object identifiers is invalid, the respective object is rendered with artifacts, or not rendered at all. However, the remaining rendering process continues without constraints, and the application remains stable. This approach definitely benefits the final application user: Slight rendering artifacts are recognized, and there still is enough time to safely conclude working and restart the application. The only way to provoke OptiX-like hard freezes from OpenGl involves host-side pointer errors, which are easily avoided within higher-level languages.

Apart from fragile scene management, many other stability issues within the OptiX API have been identified in the course of this thesis as well. These range from bugs over missing and not implemented features to misleading diagnostic messages: Not implemented texture sampler types for popular 3-byte or 4-byte image formats lead to visual artifacts and random crashes (figures 9.3 and 9.4). Mipmapping is exposed by the OptiX API, but not implemented. Invalid combination of ray types yields hard system freezes. Data buffer resize and reassignment randomly fail...

In total, the stability and usability concept behind the OptiX API should be the most important focus of future development for further gains on traditional rasterization.

Other, more general future perspectives both for interactive raytracing and the raytracer implementation within Simulator X are discussed in the next chapter.

10 Conclusion

This chapter provides a review over the design of the raytracing component, its integration into the Simulator X system, and its evaluation. Thereafter, a preview on further research topics and open implementation tasks is given. Finally, a last, short conclusion on interactive raytracing is presented.

10.1 Review

Within this thesis, a raytracing component for Simulator X has been developed and integrated.

At first, the current research on interactive raytracing had been examined. While research on that field is still ongoing, it was concluded that there are current strategies that make interactive raytracing of dynamic scenes feasible on modern hardware. Yet, investigation of middleware platforms revealed NVIDIAs OptiX API as the only publicly available interactive raytracer.

On a different focus point, general rendering engines had been reviewed in terms of their multithreading and raytracing compatibility. The review ended with the insight that both aspects must be considered in the initial design phase.

After the state-of-art research, the existing, Scala-based framework Simulator X has been introduced. Based on the design principles of minimal coupling with maximal cohesion, the Simulator X system combines an event-based communication scheme with an actor-based entity model. The existing jVR-based rendering system was found tightly integrated into existing application logics, no general rendering interface had been defined. Multithreading is handled by the general actor concept within the framework — effectively granting only a single thread access to each instance of the rendering system.

Apart from the Simulator X framework, the OptiX API for GPU-based raytracing has been studied as well. Within OptiX, the user builds a scene hierarchy from certain predefined node types. Programmable components and device-side variables are attached to scene nodes and control parts of the entire raytracing process. Access to the OptiX API is performed by two separate languages: A thread-unsafe C library for host control is complemented by a CUDA-based interface for hardware-side programmability.

The actual task of raytracer implementation began with the definition of a general rendering interface. Focus points were intuitive extensibility, support for both raytracing and rasterizing back-ends, native multithreading capabilities, and intuitive client-side interfaces. At first, abstract render-side resources and their functional behavior were specified. Then, a device-independent command-based control mechanism for the rendering process was introduced. Finally, multithreading and blocking rules were defined over all functional elements. In particular, behavior was chosen as to maximize performance for frequent operations and to minimize implementation difficulties for rare operations. Effortless thread-safety for arbitrary client threads was maintained as well.

Once the general rendering interface had been defined, its implementation with an OptiX-based back-end was elaborated. As the language of choice, C++ was preferred over Java due to its direct integration with the C-based OptiX API. Each of the previously defined, functional interfaces was replaced by an appropriate C++ interface class. Additional, OptiX-specific implementing classes were integrated into the class hierarchy. These internally capsule OptiX programs or scene nodes, and realize the general interface functionality via the OptiX C API. Finally, a default raytracing pipeline based on Blinn-Phong shading with hard-edged shadows was integrated into the raytracer implementation.

In the following course, the finished raytracer was integrated into the Simulator X framework. Because the main implementation had been composed in the C++ language, all its classes had to be encapsuled in respective Java interfaces. These were connected to their C++ correspondents by a sophisticated pointer wrapping scheme. Particular efforts were required to unify C++ shared pointer lifetime management with Java automatic garbage collection. The final Java classes in turn intuitively integrated into the Scala-based framework via a separate raytracing component. As proof of concept, this component was linked into a major demonstration application of the Simulator X framework.

Finally, the OptiX-based rendering component had been compared to its jVR counterpart in regards to various aspects. General performance and image quality were comparable, with certain benefits to the traditional OpenGl implementation. At the same time, various stability problems within the OptiX kernel made it obvious that GPU-accelerated raytracing still is a rather new contender to established graphics APIs.

10.2 Preview

Although much has already been achieved in the course of this thesis, there always remain open points of interest, both in more formal research topics as well as implementation tasks. As a guideline for future work, the most important of these are listed in this section.

10.2.1 Research

In regards to open research areas, two separate topics come to mind: interactive raytracer features, and general renderer design.

On the features side, a desirable request for drastically improved realism involves the calculation of higher-level surface-to-surface light transfers. The current raytracer only captures light transfers from a single point or directional light to a single surface accurately. It supports neither light-emitting surfaces nor surface-to-surface light transfers. Both of these, however, are a vital element of the rendering equation. In other words, the rendering equation is to be fully solved for each point within a dynamic scene and on each rendering frame. In this context, the term global illumination often is used. Global illumination particularly adds to the perceived realism of partially lit, high-contrast scenes. As a further gain, any raytracer implementa-

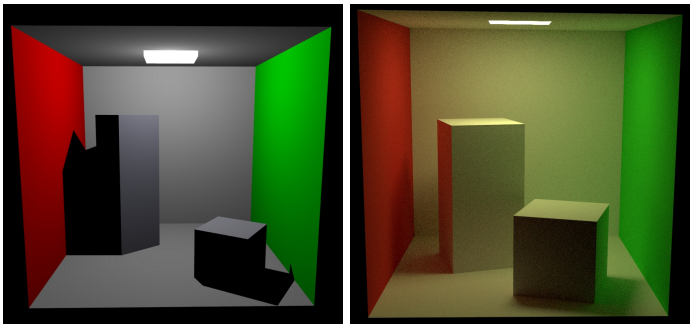


Figure 10.1: Unlike the basic raytraced image on the left hand side, the right hand side Cornell Box [CU98] was calculated by a path-based raytracer [NV11d] and considers surface-to-surface light transfers. Thus, the initially white light tints the respective sides of the box objects after diffuse reflection from the colored walls. Smooth shadows and ambient occlusion are intrinsically calculated within path tracing.

tion that considers higher order light transfers automatically allows for realistic soft shadows. A more visual impression of these benefits is given in figure 10.1.

There currently even are path-based raytracer implementations for OptiX that sample secondary light transfers [NV11d] for approximative global illumination. Unfortunately, depending on the actual approach, either image quality or rendering framerate suffer and cannot be accepted for an interactive application. Research for more practicable strategies, especially considering the peculiarities of graphics hardware, is potentially a rewarding endeavor.

Within the environment of a conventional rasterizer, any of the global illumination effects are even more difficult to achieve. As a workaround, there currently are several offline pre- and post-processing techniques that at least allow for the restricted approximation of higher-order light interactions. For instance, pre-calculated radiosity transfers have been used alongside lightmapping [AW00] for quite some time to provide an exact, but static solution to the rendering equation. In more recent times, lightmaps have been extended by wavelet-compressed depth maps [CU08] to allow for arbitrary, but fully simulated interactions between dynamic lights and static surfaces. Perhaps a raytracer could employ any of these techniques for improved image quality until interactive framerates becomes feasible with completely dynamic, raytraced radiosity.

In regards to general renderer design, the final renderer interface has certain design-induced flaws. While blocking behavior and thread safety has been designed for optimal performance and ease of use, there could be a more intuitive solution. For instance, the messaging approach that has been dismissed for the final design may still be worthwhile for further investigation.

10.2.2 Implementation

Currently, the OptiX raytracer implementation is not fully featured. While the main raytracer kernel has completely been finished, there still is much extension and polishing work left ahead. Mainly, this refers to Scala integration, auxiliary renderer features, and example applications.

Most importantly, the Scala integration of the current raytracer kernel has been created for intuitive compatibility with existing example applications within the Simulator X codebase. In turn, advanced rendering capabilities not supported by the original jVR renderer, such as skeletal animation or resource updates, are currently not accessed from within Simulator X. The next goal here should be a unified, fully-featured renderer interface within the Scala framework.

On side of renderer features, with only two different model types supported, clients are severely restricted in the scenes that can be displayed intuitively. Additional model types, such as terrains or procedural geometry, could even provide additional insights into interactive raytracer requirements.

Consider the example of a large-scale terrain: Distant triangles require very small screen estate, typically less than a single pixel. This leads to graphical moire effects, where multiple differently colored triangles fight for the same on-screen position over multiple frames. A traditional rasterizer solves this problem by adding LOD levels to the terrain, with the additional benefit of reduced per-triangle rasterization overhead. An offline raytracer works around depth moire by a combination of clever camera positioning, hand-pruned terrain triangles, and anti-aliasing methods. However, both of these approaches are less than ideal for an interactive raytracer: LOD-ing introduces the requirement to mess with a potentially large part of the scene hierarchy, while anti-aliasing drastically reduces performance. Currently, there is no alternative solution available. Perhaps a viable approach might be the translation of raytracing from triangle primitives to purely procedural representations. This allows for a distance-adaptive shape representation at the increased cost of tests between rays and arbitrary surfaces. While there already exist established mathematical methods [TW10] that could be applied here, their feasibility in interactive raytracing has not been studied yet.

Another feature that potentially needs implementation once large-scale scenes become available is a more sophisticated OptiX scene update strategy. In the current design, any updates to the OptiX internal optimization hierarchy are triggered automatically by the raytracing kernel once any update to the scene has been recognized. However, for large scenes, this mechanism is less then optimal. Instead, a manual scheme could be devised that updates objects more reasonably. For example, updates for skeleton-animated objects far from the viewer could potentially be skipped each other frame without any visual artifacts.

Finally, one might also re-evaluate the choice of OptiX as middleware raytracer platform. Albeit OptiX is the only alternative for interactive raytracing at the time of this writing, further APIs are bound to be developed in the future. These potentially provide a different point of view on the raytracing process, and thus might be able to work around some of the shortcomings within OptiX. For instance, reduced programmability that is more suited to raytraced rendering instead of general

raytracing calculations might drastically reduce implementation clutter. Likewise, a more restricted API might also be more stable. As any such platform switch has been considered in the initial design, the renderer interface can be reused without efforts. Thus, one only has to write a new context implementation and evaluate that one in regards to the original OptiX version.

10.3 Conclusion

As seen from the results of this thesis, interactive raytracing methods are no longer completely infeasible. Yet these still are not as evolved and widely accessible as their rasterizer counterparts. Thus, conventional rasterizer implementations will continue to dominate the interactive market for the foreseeable future.

In medium terms, the acceptance of interactive raytracing into general virtual reality software could be coupled with the aggregation of GPU and CPU features on a single, multi-architecture processor [TS09]: CPU-level general computations could be performed at GPU-level parallelism. Thus, the way would be opened for completely customized software rendering kernels that dispose even the last remnants of the fixed-function pipeline on current hardware. In turn, developers are free to write arbitrary, custom rendering kernels that are open to any combination of raytracing or rasterization strategies.

Appendices

A Framework Overview

The rendering kernel is based on a general application framework that has been created during the study course of the thesis author. In this chapter, an overview over important framework modules past the rendering core is given.

Note that an exhaustive coverage over all module functionality exceeds the scope of this thesis. For a more in-depth functionality description, refer to the Doxygen code documentation contained in the source code delivery.

A.1 Exception module

This module provides general exception handling support within the framework. This includes exception classes derived from std::exception, and a description of exception handling support.

Unlike the common practice in interactive applications, the framework must be compiled with mandatory exception support. In particular, the often named performance bottleneck that is caused by reduced compiler optimizations in the presence of exceptions has been proven a myth on more modern C++ compilers.

Exceptions are used to signal all errors within any framework calls. Particular attention has been invested into maintaining exception safety and associated requirements. This includes non-throwing destructors and cleanup methods, as well as exception safety method specifications where appropriate. Further details on exceptions in C++ and their pitfalls are provided in [HS00].

A.2 ScopeGuard module

Within this module, a C++ template facility to execute arbitrary code on scope exit has been implemented.

In particular, a ScopeGuard may at any time be generated on the stack, and executes a compile-time associated function or member method with stack-saved parameters on destruction at stack unwinding. This simulates a Java-esque finally construct that is natively not available in C++ exception handling clauses.

ScopeGuards are especially useful when interfacing with pure C code, such as the OptiX SDK, that does not support RAII patterns. Sadly, ScopeGuards are often overlooked and have not even been considered for the coming C++0x standard.

The framework implementation closely follows the original ScopeGuard suggestion [AA00], with certain adaptions inspired by [AA01].

A similar pattern is available from the Boost libraries [BS11]: The `BOOST_ON_EXIT` macro provides lamda-like expressions at the cost of an additional preprocessor step. The next C++ standard even provides native lamda expressions for small inline code, yet these are not widely supported yet, and cause code bloat when used for scope exit functions. Thus both of these alternatives have not been adopted.

A.3 Container module

The container module provides a full set of C++ container classes and associated helpers, such as iterators or smart pointers.

Most of the provided functionality overlaps with the STL. Usually, the STL should be the primary choice for cross-platform development. However, there are certain design flaws within the STL for which the Container module points out alternative implementation ideas.

For instance, the STL does not provide a basic data buffer for storage of BLOB (binary large object) data — such as the point datastream within a mesh geometry, or the colour array of an image. While std::vector may be used, its implementation performance is often poor, and it is not suited for interfacing with standard C functions (such as the Optix SDK) without code hacks.

See [YK09] for further critical STL coverage.

A.4 Log module

The Log module provides helper functionality for printing debug and progress output to a client-specified HTML log file.

In comparison to existing C++ and Java logging solutions, it comes with three focus advantages: For one, logging satisfies NOTHROW requirements to allow for output from within destructors and cleanup methods. Next, the logging module can be compiled out completely for improved performance in release builds. Finally, it solves the static initialization and shutdown dilemma that holds for C++ static instances, while at the same time it allows for logging actions at any time — even during static initialization.

A.5 Geometry and image modules

The geometry and image modules provide helper functionality for dealing with general in-memory triangle meshes and images. This includes image rescaling, image operations, mesh collision detection and on-CPU skeletal animation. Also included in the geometry module is a mathematics framework for general 2D and 3D primitives, such as rectangles, boxes, vectors, matrices, and quaternions.

A.6 File module

The File module abstracts actual client-side resource operations from the operating system and target environment. In particular, clients just specify relative POSIX pathes — dubbed virtual pathes — when interfacing with the framework. The framework then resolves these pathes to appropriate file instances.

For example, an internet-distributed module implementation may map a virtual path to a file on a remote server, while another implementation might look for the file in a resource JAR archive within the current working directory or the user's LINUX home folder. As the client-side virtual path is the same in both cases, the implementations can be exchanged without effort.

A.7 Thread module

The Thread module provides OS-independent support for multithreading, mutexing, and signalling. Particular design goal was to provide a slim, but exception safe layer atop the existing low-level OS solutions such as pthreads or Windows threads. The interface of the Thread module has been based on the threading proposal for the new standard, and may be replaced once the C++ standard threading becomes available.

B Bibliography

The following sources have been considered in the creation of this master thesis:

[AA00] *"Generic: Change the Way You Write Exception-Safe Code — Forever"*
Andrei Alexandrescu, Petru Marginean
Web release, `http://drdobbs.com/184403758`, December 2000

[AA01] *"Modern C++ Design: Generic Programming and Design Patterns Applied"*
Andrei Alexandrescu
17th Printing, Addison-Wesley, Pearson Education, 2009

[AA68] *"Some techniques for machine rendering of solids"*
Arthur Appel
AFIPS Conference Proceedings 32, pages 37–45, 1968

[AM03] *"Dense matrix algebra on the GPU"*
Adam Moravanszky
Direct3D ShaderX2, Wordware Publishing, 2003

[AS06] *"Massive Model Rendering with Super Computers"*
Abe Stephens
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[AW00] *"3D Computer Graphics"*
Alan Watt
Third Edition, Addison-Wesley, 2000

[BM11] *"iRT: An Interactive Ray Tracer for the CELL Processor"*
Barry Minor, Mark Nutter, Joaquin Madruga
Web release, `http://www.alphaworks.ibm.com/tech/irt`, 2007

[BL11a] *"Blender"*
Blender Foundation
Web release, `http://www.blender.org`, 2011

[BL11b] *"Blender rendering pipeline recode"*
Blender Developer Wiki
Web release, `http://wiki.blender.org/index.php/Dev:Source/Render/Pipeline`, 2011

[BS06a] *"Raytracing und Szenengraphen"*
Björn Schmidt
Diplomarbeit, Johann Wolfgang Goethe-Universitaet Frankfurt am Main, Fachbereich Informatik und Mathematik, Professur Grafische Datenverarbeitung

[BS06b] *"The Hacks of Life — VBOs, PBOs, and FBOs"*
Benjamin Supnik
Web release, `http://hacksoflife.blogspot.com/2006/10/vbos-pbos-and-fbos.html`, 2006

[BR10] *"Brook and BrookGPU"*
Stanford University
Web release, `http://graphics.stanford.edu/projects/brookgpu`, 2010

[BS11] *"Boost C++ libraries"*
Boost.org
Web release, `www.boost.org`, 2011

[CA99] *"Advanced Programming for the Java 2 Platform — Chapter 5: JNI Technology"*
Calvin Austin, Monica Palwan
Web release, `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/index.html`, November 1999

[CB09] *"Eficient Ray Traced Soft Shadows using Multi-Frusta Tracing"*
Carsten Bentin, Ingo Wald
Proceedings of High-Performance Graphics 2009

[CL09] *"Fast BVH Construction on GPUs"*
Christian Lauterbach
Eurographics, volume 28, number 2, pages 375–384, 2009

[CS08] *"COLLADA — Digital Asset Schema Release — Version 1.5.0"*
Mark Barnes, Ellen Levy Finch
Web release, `www.collada.org`, April 2008

[CU98] *"The Cornell Box"*
Cornell University Program of Computer Graphics
Web release, `http://www.graphics.cornell.edu/online/box`, February 1998

[CU08] *"Smooth and non-smooth wavelet basis for capturing and representing light"*
Cameron Upright, Dana Cobzas and Martin Jagersand
Proceedings of the 3DPVT 08, Fourth International Symposium on 3D Data Processing, Visualization, and Transmission, 2008

[CW67] *"Halftone Perspective Drawings by Computer"*
Chris Wylie et al.
AFIPS Conference Proceedings, volume 31, Fall Joint Computer Conference, AFIPS Press, Montvale, pages 49–58, 1967

[DH07] *"Interactive k-D Tree GPU Raytracing"*
Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, Pat Hanrahan
Proceedings of the 2007 symposium on Interactive 3D graphics and games, 2007

[DL76] *"Theories of Vision from Al-Kindi to Kepler"*
David C. Lindberg
Book, Chicago, 1976

[DM06a] *"Interactive Ray Tracing: Higher Memory Coherence"*
Dinesh Manocha, Sung-Eui Yoon
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[DM06b] *"Ray Tracing Dynamic Scenes Using BVHs"*
Dinesh Manocha, Sung-Eui Yoon
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[DP09] *"Quake Wars: Ray Traced"*
Daniel Pohl, Intel Corporation
Web release, `http://www.qwrt.de`, 2009

[DP10] *"Wolfenstein: Ray Traced"*
Daniel Pohl, Intel Corporation
Web release, `http://www.wolfrt.de`, 2010

[DW10] *"Ein Konfigurierbares World-Interface zur Kopplung von KI-Methoden an Interaktive Echtzeitsysteme"*
Dennis Wiebusch, Marc Erich Latoschik, Henrik Tramberend
Virtuelle und Erweiterte Realität, 7. Workshop of the GI special interest group VR/AR, pages 47-–58, 2010

[EC74] *"A Subdivision Algorithm for Computer Display of Curved Surfaces"*
Edwin Catmull
Dissertation, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, 1974

[EG04] *"Design Patterns. Elements of Reusable Object-Oriented Software."*
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
First edition, Addison-Wesley, Reprint, 1994

[FA09] *"When Will Ray Tracing Replace Rasterization?"*
Fedy Abi-Chahla
Web release, `http://www.tomshardware.com/reviews/ray-tracing-rasterization,2351.html`, 22. July 2009

[GS06a] *"Scattering Secondary Rays"*
Gordon Stoll
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[GS06b] *"Ray Tracing Performance — Zero to Millions in 45 Minutes"*
Gordon Stoll
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[HS00] *"Exceptional C++, 47 Engineering Puzzles, Programming Problems, and Solutions"*
Herb Sutter
17th Printing, Addison-Wesley, Pearson Education, 2009

[IK07] *"Ancient Theories of Vision and Al-Kindi's Critique of Euclid's Theory of Vision"*
Ika Putri
Composition, Proseminar: History of Computational Science, Vision, and Medical Science, Technische Universität München, 2007

[IW06a] *"Ray Tracing Animated Scenes"*

Ingo Wald
Presentation sheets, SIGGRAPH Course, SIGGRAPH 2006

[IW06b] *"Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies"*
Ingo Wald
ACM Transactions on Graphics, 2006

[IW07a] *"On fast Construction of SAH-based Bounding Volume Hierarchies"*
Ingo Wald
IEEE Symposium on Interactive Ray Tracing, pages 33-40, 2007

[IW07b] *"State of the Art in Ray Tracing Animated Scenes"*
Ingo Wald, William R. Mark, Johannes Guenther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, Peter Shirley
Eurographics 2007

[IW08] *"Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes"*
Ingo Wald, Thiago Ize, Steven G. Parker
Computers and Graphics 32, pages 3–13, 2008

[JB70] *"A procedure for generation of three-dimensional half-toned computer graphics presentations"*
Jack Bouknight
Communications of the ACM 13, 9, pages 527–536, September 1970

[JF98] *"Open source Java projects: Java Native Access"*
Jeff Friesen
Web release, `http://www.javaworld.com/javaworld/jw-02-2008/jw-02-opensourcejava-jna.html`, May 1998

[JK68] *"The rendering equation"*
James Kajiya
Siggraph, 1986

[JP10] *"Hierarchical LBVH Construction for Real-Time Ray Tracing"*
Jacopo Pantaleoni, David Luebke
High Performance Graphics, June 2010

[IW10] *"Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture"*
Ingo Wald
IEEE Transactions on Visualization and Computer Graphics, 2010

[LL10] *"Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture"*
Ingo Wald
7. Workshop of the GI special interest group VR/AR, Shaker Verlag, Virtuelle und Erweiterte Realität, pages 145–156, 2010

[MC10] *"C++ FAQ Lite — Frequently Asked Questions"*
Marshall Cline
Web release, `http://www.parashift.com/c++-faq-lite`, 2010

[ML10] *"Engineering Realtime Interactive Systems: Coupling and Cohesion of Architecture Mechanisms"*
Marc Erich Latoschik, Henrik Tramberend
Proceedings of the Joint Virtual Reality Conference of Euro VR — EGVE — VEC, EG Symposium Proceedings, pages 25–28, 2010

[ML11] *"Simulator X: A Scalable and Concurrent Architecture for Intelligent Realtime Interactive Systems"*
Marc Erich Latoschik, Henrik Tramberend
In proceedings of the IEEE Virtual Reality 2011 Conference, 2011

[MO10] *"Programming in Scala — A comprehensive step-by-step guide"*
Martin Odersky, Lex Spoon, Bill Venners
Second edition, artima, 2010

[MR10] *"Entwicklung einer shaderbasierten Grafik-Engine für den Einsatz in der Lehre"*
Marc Rossbach
Master thesis, Beuth Hochschule fuer Technik Berlin, FB VI – Informatik und Medien, Fachgebiet Medieninformatik, October 2010

[MR11] *"RaytracerCLGL — An OpenCl raytracer implementation"*
Maximilian Reischl
Presentation on term project, Angewandte Informatik 5, Intelligent Graphics Group, Term 2010/2011

[MS09] *"Spatial Splits in Bounding Volume Hierarchies"*
Martin Stich, Heiko Friedrich, Andreas Dietrich
Proceedings of High-Performance Graphics, 2009

[MS11] *"A Scala Tutorial for Java programmers"*
Michel Schinz, Philipp Haller
Web release, Version 1.3, `www.scala-lang.org`, 2011

[NV10] *"OptiX Raytracing Engine — Quickstart Guide"*
NVIDIA corporation
Web release, `http://developer.nvidia.com`, Version 2.1, December 2010

[NV11a] *"OptiX Raytracing Engine — Programming Guide"*
NVIDIA corporation
Web release, `http://developer.nvidia.com`, Version 2.1, February 2011

[NV11b] *"OptiX Raytracing Engine — API Reference"*
NVIDIA corporation
Web release, `http://developer.nvidia.com`, Version 2.1, February 2011

[NV11c] *"CUDA C Programming Guide"*
NVIDIA corporation
Web release, `http://developer.nvidia.com`, Version 4.0, March 2011

[NV11d] *"OptiX SDK sample application: Cornell Box Scene"*
NVIDIA corporation
Web release, `http://developer.nvidia.com`, March 2011

[OE11] *"OGRE — Open Source 3D Graphics Engine"*
Various contributors

Web release, `http://www.ogre3d.org`, 2011

[OS11] *"OpenSceneGraph"*
Various contributors
Web release, `http://www.openscenegraph.org/projects/osg`, 2011

[RG71] *"3-D Visual Simulation"*
Robert Goldstein, Roger Nagel
Simulation — Transactions of The Society for Modeling and Simulation International, Thousand Oaks, California, pages 25-–31, January 1971

[PS06] *"State of the Art in Interactive Ray Tracing"*
Philipp Slusallek
Presentation sheets, SIGGRAPH course on interactive raytracing, SIGGRAPH 2006

[SB06] *"Interactive Distribution Ray Tracing"*
Solomon Boulos
Presentation sheets, SIGGRAPH interactive raytracing course introduction, 2006

[SG06] *"State of the Art in Interactive Ray Tracing"*
Various authors
Presentation sheets, SIGGRAPH interactive raytracing course introduction, 2006

[SG07] *"Broad-Phase Collision Detection with CUDA"*
Scott Le Grand, NVIDIA Corporation
GPU Gems 3, Second Edition, Pearson Education, 2007

[TI08] *"Ray tracing with the BSP tree"*
Thiago Ize, Ingo Wald, Steven G. Parker
Proceedings of the IEEE Symposium on Interactive Ray Tracing, 2008

[TS09] *"The End of the GPU Roadmap"*
Tim Sweeney
Presentation sheets, SIGGRAPH 2009

[TF05] *"KD-Tree Acceleration Structures for a GPU Raytracer"*
Tim Foley, Jeremy Sugarman
Graphics Hardware, 2005

[TF10] *"Scene Graphs - just say no"*
Tom Forsyth
Web Release, `http://home.comcast.net/~tom_forsyth/blog.wiki.html`, 2010

[TW79] *"An improved illumination model for shaded display"*
Turner Whitted
Proceedings of the 6th annual conference on Computer graphics and interactive techniques, 1979

[TW08] *"Implementation of Data Parallel Algorithms on Graphics Accelerators"*
Tobias Werner
Composition, Lehrstuhl Angewandte Informatik III, Universität Bayreuth, Winter Term 2008 / 2009

[TW10] *"Interior Point Methods with Second Derivatives for Solving Large Scale Nonlinear Programming Problems"*
Tobias Werner
Diploma Thesis, Lehrstuhl Angewandte Informatik VII, Universität Bayreuth, Summer Term 2010

[TW11a] *"Survey of Rendering Systems"*
Tobias Werner
Composition, Lehrstuhl Angewandte Informatik III, Universität Bayreuth, Summer Term 2011

[TW11b] *"Design and implementation of a multithreaded OpenGL 3.0 rendering kernel"*
Tobias Werner
Master project, Lehrstuhl Angewandte Informatik III, Universität Bayreuth, Summer Term 2011

[UE11] *"Unreal Technology"*
Epic Games, Inc.
Web release, `http://www.unrealengine.com`, 2011

[VH00] *"Heuristic Ray Shooting Algorithms"*
Vlastimil Havran
Dissertation Thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000

[WPa] *"Rendering (computer graphics)"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/Rendering_(computer_graphics)`, 2011

[WPb] *"Ray casting"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/Ray_casting`, 2011

[WPc] *"Ray tracing"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/Ray_tracing_(graphics)`, 2011

[WPd] *"The rendering equation"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/Rendering_equation`, 2011

[WPe] *"OpenRT Real Time Ray Tracing Project"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/OpenRT`, 2011

[WPf] *"Blinn–Phong shading model"*
Wikipedia, the free encyclopedia
Web release, `http://en.wikipedia.org/wiki/Blinn-Phong_shading_model`, 2011

[YF11] *"Yo Frankie!"*

Apricot Open Game Project
Web release, `http://www.yofrankie.org`, 2011

[YK09] *"C++ FQA Lite — Frequently Questioned Answers"*
Yossef Kreinin
Web release, `http://yosefk.com/c++fqa`, 2009

C Compact Disc Contents

This section details the contents of the delivery medium which accompanies the thesis:

- `Thesis`

  A folder containing the latex files and jpeg images that compose this thesis paper. Also provided is a TeXnicCenter project `Thesis.tcp`, and a Windows-only building helper script `FetchResult.bat`.

- `Bibliography`

  Any available electronical versions of the literature sources that have been used in the creation of this thesis.

- `Software`

  All software source code written to accompany the thesis is stored within this directory. Note that required third party software components have not completely been included for licensing and size reasons.

- `Software/OptixExa`

  The OptiX example application from chapter 5.

- `Software/OptixWrap`

  The native C++ code of the general renderer interface, the OptiX raytracer implementation, and the JNI-based Java wrapper. The actual source code is distributed over various modules, and in general resides within the `Code` directories. Included are a Visual Studio 2010 workspace for building within each `VisualStudio` subfolder, and complete Doxygen code documentation within the `CodeDoc` subfolders.

- `Software/siris_ray`

  A snapshot of Simulator X from the siris project repository, with integrated functionality for the Optix wrapper module.

Attention: The compact disc is only provided for convenience, and has not been approved for public release.

D Deutschsprachige Zusammenfassung

Nach Prüfungsordnung ist Masterarbeiten in einer anderen als der deutschen Sprache eine deutschsprachige Zusammenfassung beizulegen. In diesem Abschnitt werden daher die Ergebnisse meiner Arbeit kurz auf Deutsch erläutert.

D.1 Übersicht

Ziel der Arbeit war die Implementierung eines echtzeitfähigen Raytracing-Kerns und die Integration in die bestehende, Scala-basierte VR-Simulationsumgebung Simulator X.

Als wissenschaftliche Basis wurden zuerst der allgemeine wissenschaftliche Stand interaktiven Raytracings sowie das bestehende Design ausgewählter Open-Source-Komponenten untersucht.

Im Rahmen der Arbeit mussten anschliessend die bestehende Architektur von Simulator X sowie die existierende Java-Rendering-Komponente analysiert werden.

Erst dann konnte ein allgemeines Rendering-Interface auf Basis minimaler Kopplung und maximaler Kohäsion erstellt werden. Weitere Anforderungen an dieses Interface ergaben sich aus allgemeinen Entwurfsregeln, aus der Multithread-Nutzung auf modernen Mehrkernsystemen, und aus den Vorraussetzungen für den Betrieb eines Raytracing-Moduls. Zusätzlich sollte zur späteren Erweiterbarkeit ein klassischer Rasterisierer unterstützt werden.

Für die Implementierung der Schnittstelle musste eine geeignete Programmier- Plattform ausgewählt werden. Dazu wurde nach Kandidaten für echtzeit-fähiges Raytracing gesucht. Schliesslich wurde eine Implementierung auf Basis der OptiX-API von NVIDIA beschlossen.

Nach Abschluss der Implementierungsarbeiten musste die Rendering-Schnittstelle in Simulator X über einen Satz an Java-Wrappern eingebunden werden. Auch hier standen mit JNA und JNI zwei verschiedene Lösungen zur Wahl, die Entschiedung fiel aufgrund besserer Performance für JNI.

Zum Abschluss der Arbeit wurde die neue Rendering-Schnittstelle in Bezug auf Bildqualität und Performance mit ihrem jVR-Vorgänger verglichen.

Im Folgenden werden die bereits umrissenen Schritte näher erläutert.

D.2 Stand der Forschung

Die wissenschaftliche Recherche für diese Arbeit stützte sich auf zwei verschiedene Themen: Zuerst wurden allgemeine wissenschaftliche Arbeiten über interaktives Raytracing studiert, um einen Einblick in die Materie und die Realisierbarkeit von Raytracing auf moderner Konsumenten-Hardware zu gewinnen. Anschliessend wurden verschiedene bestehende Rendering- und Raytracing- Engines auf ihre grundlegende Architektur sowie Multithreading- Fähigkeiten untersucht.

Im Rahmen wissenschaftlicher Veröffentlichungen fanden sich zahlreiche neue Verfahren zum Umgang mit interaktivem Raytracing. Insbesondere erwähnenswert ist

die Entwicklung neuer Algorithmen, die von den traditionellen Methoden für Offline-Raytracing abweichen. Im herkömmlichen Raytracing wird zwar eine optimale Baumstruktur (zumeist ein kd-Baum) zur Optimierung der Dreieck-Strahlen-Verschneidung herangezogen – diese is jedoch bei dynamischen Änderungen in der virtuellen Szene nur schwer anzupassen.

Spezielle Verfahren für interaktives Raytracing in dynamischen Szenen setzen daher auf andere Ansätze mit weniger optimalen, dafür schneller zu aktualisierenden Optimierungsstrukturen. Der vielversprechendste Vertreter ist hierbei der sogenannte BVH-Ansatz, kurz für Bounding Volume Hierarchy oder übersetzt Umspannende-Volumen-Hierarchie. In diesem Ansatz wird jedes Primitivum der Szene in ein Volumen eingeschlossen – zumeist wird in praktischen Realisierungen ein Dreieck in einen achsen-orientierten Quader eingespannt. Aus den entsprechenden Volumina wird eine Baumstruktur aufgebaut. Im Vergleich zu einem kd-Baum enthält ein Knoten dieser Struktur nur Abhängigkeiten zu allen untergeordneten Knoten. Dementsprechend bedingt die Aktualisierung eines Primitivums nur Anpassungen an den Volumina einiger übergeordneter Knoten. Eine grundlegende Änderung an der Struktur des Baumes ist nicht notwendig. Über verschiedene Ausprägungen der BVH-Verfahren lassen sich insgesamt interaktive Bildwiederholraten für das Raytracing dynamische Szenen selbst auf aktueller Konsumenten-Hardware erzeugen.

Im Gegensatz zu den wissenschaftlichen Erkenntnissen findet sich bislang keine vollständige 3D-Engine, die auf interaktives Raytracing gründet. Allerdings ergaben sich bei der Analyse bestehender Engines dennoch einige Einblicke in die entsprechenden Anforderungen an einen Renderkern. Insbesondere bei wurde bei Entwicklung der frei verfügbaren Ogre3D-Engine darauf Wert gelegt, eine möglichst generelle Rendering-Schnittstelle zu schaffen. In diesem Sinne werden innere Details des Renderers – etwa Optimierungsstrukturen, Szenen-Traversierung oder Lichtberechnung – nicht auf der Anwenderseite offengelegt. Dies ermöglicht folglich ein problemloses Austauschen der eigentlichen Renderer-Umsetzung. Dementsprechend existiert für die Ogre3D-Engine auch ein nicht-öffentlicher Prototyp einer Raytracer-Implementierung.

Im Gegensatz zu allgemeinen Architekturkriterien finden sich zur Unterstützung von paralleler Abarbeitung oder Benutzung in bestehenden 3D-Engines kaum Informationen. Bei frei verfügbaren Systemen wird die Unterstützung von Multithreading in Hinblick auf die damit einhergehenden Umsetzungsprobleme nicht integriert. Im Gegensatz dazu bieten viele kommerzielle 3D-Platformen bereits explizite Multithreading-Fähigkeiten, beispielsweise über Rendering- und Ressourcen-Lader-Threads. Allerdings veröffentlichen die entsprechenden Entwickler abgesehen von grundlegenden Merkmals-Listen keine detailierten Informationen zur Architektur oder Implementierung.

Die Recherche ergab folglich zwei Erkenntnisse: Zum einen ist interaktives Raytracing selbst dynamischer Szenen auf aktueller Konsumenten-Hardware prinzipiell möglich. Zum anderen muss die Implementierung von Multithreading- und Raytracing-Fähigkeiten von Anfang an in der Integration eines allgemeinen Rendering-Kerns berücksichtigt werden – eine spätere Nachrüstung ist in der Regel nicht problemlos möglich.

D.3 Architektur des VR-Rahmenwerks Simulator X

Ehe die Integration eines Raytracers ins VR-Rahmenwerk Simulator X beschrieben werden kann, ist zuerst ein grober Überblick über den Entwurf des bestehenden Rahmenwerks nötig. Das bestehende VR-Rahmenwerk ist dabei vollständig in der Hochsprache Scala gehalten. Scala ist eine funktionale Programmiersprache, die auf der Java-VM aufsetzt, und mit dieser binär-kompatibel ist.

Simulator X wurde nach zwei grundlegenden Architekturkriterien aufgebaut: Maximale Kohäsion bei minimaler Kopplung. Das heisst, das System zerfällt in maximalgrosse funktionale Einheiten, die untereinander nur die nötigsten Verbindungen unterhalten. Solche Systeme eignen sich sehr gut für die Parallelisierung.

In diesem Sinne setzt Simulator X auf ein sogenanntes Entity-Modell auf. Bei einem Entity-Modell wird innerhalb eines logischen Objekts keine funktionsspezifische Objektrepräsentation – Audiodaten, Geometrie, AI, ... – vorgehalten. Stattdessen werden diese Informationen in jeweils dem zugehörigen Funktionsmodul – Renderer, Audio-System, AI-Steuerung – in einer modulabhängigen Darstellung verwaltet und lediglich aus den logischen Objekten referenziert.

In der Realisierung von Simulator X wird das Entity-Modell durch zwei weitere Paradigmen vervollständigt: Ein Actor-Modell und ein Event-Modell.

Ein Actor-Modell zerlegt ein bestehendes System zur Parallelisierung in unabhängige, funktionale Einheiten, auch Aktoren genannt. Jeweils ein Aktor entspricht in der Regel einer eigenständigen, parallelen Ausführungseinheit. Dadurch wird funktionale Kohäsion gefördert.

Die harte Kopplung einzelner Aktoren lässt sich durch das Event-Modell vermeiden. Hierbei wird statt direkten Methodenaufrufen ein allgemeines Nachrichtenschema verwendet, um die Schnittstelle zwischen Aktoren schmal und kontrollierbar zu halten.

Die Integration von Entity-, Actor- und Event-Modell in Simulator X wird auf verschiedenen Abstraktionsebenen vorgenommen.

Das zugrundeliegende Implementierungskonzept verlangt zuerst eine Kapselung jedes aktor-internen Zustands in sogenannte Zustandsvariablen. Jede Zustandsvariable gehört dabei einem eindeutig bestimmten Aktor. Anderen Aktoren ist der direkte Zugriff auf fremde Zustandsvariablen nicht gestattet. Jedoch können Aktoren innerhalb ihres eigenen, lokalen Zustands Referenzen auf die Variablen anderer Aktoren anlegen. Bei Änderungen am Wert einer Variablen werden über das Event-Modell alle Referenzen aktualisiert. Insgesamt werden durch dieses Schema ein globaler, geteilter Weltzustand und damit einhergehende Parallelisierungsprobleme vermieden.

Die eigentlichen Entitäten des Entity-Modells entstehen schliesslich durch die Kombination von Zustandsvariablen. Dabei ist die Zugehörigkeit einzelner Variablen innerhalb einer Entität zu einem einzelnen Aktor nicht zwingend vorgeschrieben. Im Gegenteil werden sogar einzelne Zustandsvariablen innerhalb einer Entität nach Funktionalität und somit oft dem zugeordneten Aktor zu sogenannten Aspekten zusammengefasst. Beispielsweise gibt es Aspekte für das Rendering, die Soundausgabe, oder für AI-Berechnungen.

Auf einer höheren Abstraktionsstufe steht das World Interface von Simulator X – die Schnittstelle zwischen einzelnen funktionalen Komponenten sowie der späteren Anwendung. Das World Interface nimmt dabei verschiedene Aufgaben war. Darunter fallen das Verarbeiten und Verschicken von Nachrichten über Änderungen an Zustandsvariablen, das Verbinden von funktionalen Komponenten über allgemeine Nachrichtentypen, sowie die Konfiguration des gesamten Nachrichtensystems.

Einen speziellen Platz im Konzept von Simulator X nehmen die funktionalen Komponenten ein: Eine Komponente ist dabei eine Sammlung von zusammengehörigen Aktoren mit streng festgelegten funktionellen Umfang. Beispielsweise gibt es eine Komponente, die nur die graphische Darstellung einer Anwendung übernimmt.

Aktuell ist die Komponente zur graphischen Darstellung in Simulator X auf Basis des Java-Rasterisierers jVR implementiert. Der bestehende Renderer setzt dabei auf zwei verschiedene Konzepte: Zum einen wird von jVR ein Szenengraph verwaltet, in dem die virtuellen Objekte zusammengestellt werden müssen. Zum anderen wird das Konzept einer benutzerdefinierten Rasterisierungs-Pipeline zur Ansteuerung und Kontrolle des Rendering-Prozesses genutzt.

Auch ein Konzept zur parallelen Abarbeitung mehrerer Rendering- Prozesse sowie eines einzelnen Hauptthreads wird von jVR vorgeschrieben. Insbesondere nutzt jVR intern mehrere selbständige Threads zur Abarbeitung eingehender Pipeline-Objekte: Es wird ein Arbeiter-Thread pro Ausgabefenster erstellt. Auf Anwendungsseite wird jedoch nur ein einzelner Thread unterstützt, der den Szenengraphen aktualisiert und die Pipeline-Eingaben für die Render-Threads erzeugt.

In Simulator X wird der eigentlich Java-basierte jVR-Renderer hauptsächlich über zwei kooperierende Aktoren eingebunden: Der eine Aktor kapselt eine einzelne jVR-Instanz inklusive Szenenverwaltung und Pipeline-Erzeugung, während der andere Aktor eine ganze Arbeitsgruppe von jVR-Renderern synchronisiert.

Die Zusammenarbeit dieser beiden Aktoren mit dem restlichen System wird über Entities und Zustandsvariablen realisiert. Insbesondere existiert ein eigener Aspekt für render-spezifische Parameter. Allerdings fällt auf, das Teile der Benutzerlogik nicht vollständig auf Anwenderseite integriert sind – so findet sich in einem jVR-Aktor etwa Code zur Verwaltung des Bildschirmmenüs einer Beispielanwendung. Ebenso ist der Renderer selbst nicht komplett von der eigentlichen Anwendung getrennt. Zwar wird durch die Architekturkonzepte von Simulator X eine Trennung auf Datenebene erreicht, dennoch wird über den jVR-spezifischen Render-Aspekt eine semantische Kopplung zwischen Anwendung und Renderer-Implementierung hergestellt.

D.4 Die allgemeine Render-Schnittstelle

Im Gegensatz zur festen Schnittstelle des jVR-Moduls richtete sich die Architektur der Raytracer-Komponente innerhalb dieser Arbeit auf ein universelles Renderer-Front-End mit austauschbarer Implementierung aus. Im Folgenden wird das entwickelte Font-End vorgestellt, ehe eine entsprechende Raytracer-Realisierung präsentiert wird.

Die allgemeine Schnittstelle zum Renderer zerfällt in eine Reihe von untergeordneten Funktionsmodulen: Zum einen gibt es ein Szenenmodul, zum anderen ein Ressourcenmodul, und schliesslich ein Steuerungsmodul. Dabei gehen all diese Untermodule

im Funktionsumfang über den Umfang der bestehenden Scala-Anbindung hinaus, um später eine leichte Erweiterbarkeit zu garantieren. Im Folgenden werden alle drei Module beschrieben, ehe das Kern-Interface dargestellt wird.

Im Szenenmodul werden die eigentlichen Objektinstanzen verwaltet. Im Gegensatz zu einem normalen Rasterisierer kann ein Raytracer nicht über ein Traverser-Pattern auf dem Logik-Szenegraphen implementiert werden, da für gute Performance unbedingt eine implementierungsspezifische, zeitkoherente Optimierungsstruktur verwendet werden muss. Dementsprechend werden durch das Szenenmodul die clientseitige Arbeit mit Objektinstanzen von der implementierungsseitigen Optimierung getrennt.

Der Client arbeitet in diesem Zusammenhang nur mit allgemeinen Objekten, RenderPuppets (zu deutsch: Render-Marionetten) genannt. Diese Objekte können nach Belieben über das Kompositions- Muster in den clientseitigen Logikgraphen eingebunden werden. Zudem werden RenderPuppets vom Client zu logischen Render-Szenen zusammengefasst. Eine Szene ist die kleinste Einheit, die in einem Durchgang auf dem Bildschirm dargestellt werden kann. Die Implementierung der Szene und der einzelnen Render-Puppets wird dabei über das Entwurfsmuster Interface vor dem Anwender verborgen. Vom Client veranlasste Änderungen an den Marionetten werden so unsichtbar an die implementierungsspezifische Representation – etwa die hierarchische Optimierungsstruktur eines Raytracers oder den Portalgraphen eines traditionellen Renderingsystems – übertragen.

In typischen Anwendungen gibt es statische, grosse Datensätze, die von zahlreichen Marionetten innerhalb einer Szene verwendet werden. Dabei handelt es sich zumeist um Geometrie-, Textur- und Shader-Daten. Um Speicherplatz einzusparen, müssen diese Daten zwischen allen beteiligten Objekten geteilt werden. Typischerweise ist für einen Performancegewinn zudem eine Speicherung in einem implementierungsspezifischen Format nötig – beispielsweise unter Verwendung von schnellem Speicher auf der Grafikkarte. Dieses Konzept wird über das Funktionalitätsmodul für renderseitige Ressourcen abgedeckt. Insbesondere gibt es verschiedene Ressourcentypen, die alle eine bestimmte Mindestfunktionalität unterstützen. Beispielsweise speichert die Ressource Texture eine 2D-Grafik zur Verwendung als Oberflächentextur, während die Ressource Model ein skelettanimiertes Modell inklusive Texturzuordnung verwaltet. Um das clientseitige Interface möglichst einfach und implementierungsfrei zu halten, ist bei fast jeder Ressource ein direktes Laden aus einer vorgegebenen Datei möglich.

Nachdem der Anwender nun eine Szene aus Marionetten zusammengestellt und mit Ressourcen verknüpft hat, muss dem Rendermodul nur noch mitgeteilt werden, dass diese Szene auch im aktuellen Bild dargestellt werden soll. Um zusätzliche Flexibilität zu erlauben, wird im zugehörigen Steuerungsmodul hierfür das Konzept einer Befehlsschlange implementiert. Der Anwender kann dabei eine Reihe von Kommandos zusammenstellen, die jeweils eine einzelne Szene von einem bestimmten Sichtpunkt aus in einem bestimmten Teil des Fensters darstellen.

Auf Basis der drei Funktionsmodule fällt die Schnittstelle des eigentlichen Renderkerns nun sehr schmal aus. Zum Erzeugen implementierungsspezifischer Szenen, Marionetten und Ressourcen werden abstrakte Funktionsaufrufe nach dem Factory-Entwurfsmusters geboten. Ein einzelner weiterer Funktionsaufruf nimmt eine Schlange an Render-Befehlen an, und löst so den Render-Vorgang aus.

D.5 Anforderungen in Multithreading-Umgebung

Bislang wurde bei der Vorstellung der Rendering-Schnittstelle noch nicht auf die Multithreading-Fähigkeiten eingegangen, da der Entwurf von Multithreading-Verhalten immer ein Gesamtbild über alle betroffenen Programmteile erfordert.

Im Folgenden wird daher nun abschliessend der Multithreading-Entwurf vorgestellt, ehe zur eigentlichen Implementierung übergegangen wird.

Grundsätzlich ist bei den zusätzlichen Risiken und dem erhöhten Aufwand eines Multithreading-Ansatzes immer die Frage nach dem Gewinn gegeben. Im Zusammenhang mit einem Rendering-Kern ergeben sich durch Multithreading jedoch zwei entschiedende Vorteile.

Zum einen ist in modernen Mehrkern-Systemen Multithreading der einzige Weg, die maximale Systemleistung zu nutzen. Entsprechend verspricht Multithreading bei einem CPU-zeitaufwendigen Rendervorgang einen Performance-Gewinn. Dieser Punkt ist bei modernen Hardware-Rasterisierern weniger bedeutend, da der Hauptaufwand für die Bildberechnung an die GPU abgegeben wird. Bei einem Raytracer fällt jedoch durch die Verwaltung der nötigen Optimierungsstruktur auch ein höherer CPU-Aufwand an, der durch Multithreading besser bewältigt werden kann.

Zum anderen lässt sich durch geschicktes Multithreading eine flüssigere Simulation erreichen, selbst auf alten Einkern-Prozessoren. Dieser Punkt ist nicht sofort ersichtlich, ergibt sich aber bei Betrachtung der für Echtzeitanwendungen typischen Hauptschleife. In der Regel findet dabei eine Folge von Simulationsschritten und Darstellungsschritten statt. Wird kein Multithreading genutzt, so muss der Aufruf der Darstellung zwangsweise blockieren, bis das Rendering abgeschlossen ist. Je nach Zeitaufwand der Darstellung werden die Simulationsschritte unregelmässig über die Zeit verteilt. Folge sind Mikroruckler in der Bildwiederholung, Stabilitätsprobleme in der Simulationsnumerik, eine pulsierende Netzwerkbandbreite, oder andere störende Faktoren. Durch Multithreading jedoch lässt sich der unkontrollierbare Zeitbedarf der Darstellung aus der Anwendungslogik herauslösen. Somit wird die Simulationsabfolge regelmässiger und berechenbarer, die Anwendung läuft flüssiger.

Aus dem letzten Punkt ergibt sich auch das entschiedende Kriterium für das Multithreading-Verhalten der Renderer-Schnittstelle: Je nach Operation muss unterschieden werden, ob die Operation an die Bildwiederholrate (also potentiell bis zum Abschluss eines Bildes) blockiert, oder nicht. Wird kein Blockieren erwünscht, muss ein alternatives Vorgehen zur Vermeidung von Gegenläufigkeits- Problemen und Inkonsistenzen zwischen Anwendungslogik und dem Renderer gefunden werden.

D.6 Multithreading-Verhalten

Zuerst wird das Multithread-Verhalten von Szeneobjekten und Render-Marionetten festgelegt. Sowohl auf Marionetten wie auf Szenen erfolgen häufige konkurrente Lese- und Schreibzugriffe – potentiell auch von mehreren Client-Threads aus. Diese Aufrufe sind auf Clientseite schwer ohne übermässigen Aufwand zu kontrollieren. Verständlicherweise muss nämlich nach jeder dynamischen Objektsimulation – Bewegung, Drehung, Skelettposen-Änderung – auf eine Marionette zugegriffen werden. Entsprechend darf auch kein Blockieren an die Wiederholrate stattfinden.

Die Lösung ist intuitiv gefunden: Jede Marionette und Szene hält zwei getrennte Zustände. Zugriffe durch den Anwender ändern oder lesen einen clientseitigen Zustand, der durch einen clientseitigen Semaphor gegen konkurrenten Zugriff geschützt ist. Gleichzeitig existiert ein renderseitiger Zustand, der zur Darstellung in einem separaten Rendering-Thread genutzt wird. Über einen speziellen client- sowie einen renderseitigen Aufruf können ferner die beiden internen Zustande durch Kopieren des clientseitigen Zustands abgeglichen werden. Dadurch wird ein Blockieren an die Bildwiederholrate bei Änderungs- oder Lesezugriffen verhindert. Ein Blockieren findet lediglich bei Konflikt zwischen Synchronisations- und Leseoperationen statt, beides verhältnismässig schnelle Operationen, so dass keine unnötige Wartezeit eingeführt wird.

Auch das Erzeugen sowie das Löschen von Marionetten oder Szenen blockiert nicht an die Bildwiederholrate. Beide Operationen werden intern ohne Blockierung im Renderer gepuffert, und erst dann durchgeführt, wenn der Renderingthread seine Arbeit verrichtet.

Im Gegensatz zu Rendermarionetten zeigen Ressourcen eine andere Zugriffsstatistik. Insbesondere werden Ressourcen nur in seltenen Fällen ausgelesen oder durch client-seitigen Code verändert. Stattdessen wird in typischen Anwendungsfällen die Ressource einmalig nach Initialisierung eingelesen, und dann immer weiter verwendet. Ferner bedingt eine Puffer-Strategie bei Ressourcenänderungen eine teuerer Allokation eines grossen, dynamischen Zwischen-Puffers, was bei vielen Aktualisierungen sogar zu einem Speicherüberlauf führen kann. Im Hinblick auf einen potentiellen Hintergrund-Ladethread wurde dennoch für die Zuweisung oder das Laden von Ressourcen ein nicht-blockierendes, gepuffertes Verhalten festgelegt. Der Speicherverbrauch wird durch das Freigeben des Puffers nach verzögerter Anwendung auf die renderseitigen Ressourcendaten minimiert. Lediglich die – äusserst seltenen – Operationen zum Auslesen einer Ressource können potentiell an die Bildwiederholrate blockieren.

Im Gegensatz zur Arbeit mit Ressourcendaten sind jedoch Ressourcenanfragen beim Renderer sowie das Freigeben einzelner Ressourcenreferenzen häufige Operationen, da beides an die Arbeit mit Rendermarionetten gekoppelt ist. Entsprechend muss in diesen beiden Fällen wieder nichtblockierendes Verhalten durch Puffern realisiert werden.

Schliesslich ist noch das Multithread-Verhalten bei Abarbeitung eines Kommandopuffers zu definieren. Obwohl es möglich ist, den Rendering-Thread direkt in den Renderer zu integrieren, so dass die Übergabe der Kommandos nicht blockiert, wurde aus Flexibilitätsgründen ein anderer Ansatz gewählt. Insbesondere wurde für die Kommandoübergabe an den Renderer selbst blockierendes Verhalten zwingend gefordert. Stattdessen wird zur Entkopplung ein separates Modul zur Steuerung eines eigenen Rendering-Threads mitgeliefert. Über eine Anfrage an dieses Modul lässt sich erfahren, ob der Rendering-Thread gerade bereits mit einem neuen Bild beschäftigt ist. Falls nein setzt die Anwendung ihre Simulationsarbeit fort, ansonsten wird nichtblockierend ein neuer Kommandopuffer an den Rendering-Thread übergeben. Dieser gibt den Puffer wiederum intern und blockierend an den normalen Renderer-Aufruf weiter. Schliesslich lassen sich mit diesem Verfahren auch die verbleibenden blockierenden Ressourcen-Leseoperationen durch Einfügen zwischen Anfrage und Komman-

doübergabe an den Rendering-Thread entkoppeln.

D.7 Auswahl einer Raytracer-Implementierung

Zur Implementierung des beschriebenen Interfaces wurde nach einer geeigneten Raytracing-Platform gesucht. Allerdings ergab sich dabei schnell ein Problem: Mit Ausnahmen von NVIDIAs OptiX-API sind keine Alternativen für interaktives Raytracing öffentlich verfügbar. Ein entsprechendes Hardware-Projekt der Uni Saarland wurde 2007 abgeschlossen, die zugehörige Forschungsgruppe arbeitet derzeit an einem geschlossenen CUDA-basierten Raytracing-System. Ein System fürs interaktive Raytracing von IBM steht zur freien Verfügung, allerdings ist dieses System nur mit Cell-Prozessoren (Playstation 3, Blade-Server) kompatibel. Der Quellcode ist ebenfalls nicht frei verfügbar, daher kann dieser Raytracer in Simulator X nicht zur Anwendung kommen. Schliesslich gibt es von Intel verschiedene Videos zur Demonstration von interaktivem Raytracing eines Computerspiels auf Server-CPUs oder Cloud-Rechnern. Jedoch wird auch hier keine zugehörige Implementierung zur freien Verfügung gestellt.

Daher wurde in dieser Arbeit eine Implementierung auf Basis von NVIDIAs OptiX-API beschlossen. Dennoch wurde zuerst eine allgemeine Renderer-Schnittstelle entwickelt, so dass ein späterer Tausch des Renderkerns gegen einen anderen Raytracer oder Rasterisierer problemlos ohne Anpassungen an clientseitigem Framework ermöglicht wird.

D.8 Implementierung der Schnittstelle auf Optix-Basis

Die Implementierung der Raytracer-Schnittstelle auf Basis von NVIDIAs OptiX-API ging intuitiv von statten. OptiX verwendet bereits intern eine szenengraphen-ähnliche Struktur zur Beschreibung der virtuellen Szene. In der Implementierung wurden zu jeder render-seitigen Ressource und zu jeder render-seitigen Marionette entsprechende Knoten aus der OptiX-Struktur hinzugefügt. Sofern eingehende Änderungen an den client-seitigen Objekten anstehen, werden diese bei Synchronisation an die OptiX-internen Objekte weitergeleitet. Somit wird die Szene aktualisiert. Ebenso werden Beziehungen zwischen client-seitigen Objekten – beispielsweise die Verwendung einer Modell-Ressource innerhalb einer Modell-Marionette – in Beziehungen zwischen OptiX-Objekten übersetzt. Im Sinne des Beispiels wird etwa ein OptiX-interner Geometrieknoten mit Modelldaten innerhalb der Modell-Ressource an einen Transformknoten innerhalb der Modell-Marionette angefügt.

Bei der Übersetzung in die OptiX-Schnittstelle traten dabei verschiedene Besonderheiten auf:

Hauptsächlich ist ein grosser Teil der Abarbeitung innerhalb des OptiX-Raytracing-Vorganges programmierbar. Dadurch lassen sich auch nicht-bildgebende Verfahren – Kollisionserkennung oder AI-Berechnungen etwa – mit OptiX realisieren. Auf Implementierungsebene steht hier das Konzept sogenannter "programmierbarer Komponenten", also einzelne benutzerdefinierte Schritte im Raytracing-Prozess. Beispielsweise wird zu Beginn der Bilderzeugung ein vom Benutzer zu schreibendes Strahlen-Erzeugungs-Programm gestartet. Dieses bestimmt, welche Strahlen durch die virtuelle Umgebung geschickt werden, sammelt zurückgegebene Daten (z.B. Farbinformatio-

nen) auf, und speichert die Ergebnisse in einem Ausgabepuffer ab. Analog können auch die Kollisionsberechnung mit Primitiven (zur Unterstützung von prozedurellen Oberflächen neben traditionellen Dreiecken) sowie die Reaktion auf Strahlen-Oberflächen-Berührung nach individuellen Anforderungen programmiert werden. Im Sinne der Raytracer-Komponente werden einige der programmierbaren Komponenten hart vorgegeben, andere können durch client-seitige Ressourcen neu zugewiesen werden. Beispielsweise ist das Programm zur Strahlenerzeugung in der Implementierung fest kodiert, während alle Schnitt-Reaktionen zur Unterstützung von prozedurellen Oberflächen austauschbar gestaltet sind.

Neben individueller Programmierbarkeit verdient auch das Kommunikations-Schema zwischen der Ausführung von OptiX auf der Grafikkarte und der CPU-seitigen Anwendung besondere Aufmerksamkeit. Insbesondere existieren zwei getrennte Schemen zur Datenübertragung: Geräte-seitige Variablen und geräte-seitige Datenpuffer. Geräte-seitige Variablen werden für kleine Datenmengen genutzt, und werden pro Objekt in der OptiX-Szene zugewiesen. Innerhalb jeder programmierbaren Komponente wird dabei auf den Variablensatz des aktuellen Objektes zugegriffen. In der Raytracer-Implementierung wird beispielsweise über solche Variablen auf jedem Modell-Objekt die zugehörige Textur des Modells referenziert. Im Gegensatz zu geräte-seitigen Variablen dienen geräte-seitige Datenpuffer zum Speichern grosser, statischer Datenmengen. Die Implementierung nutzt Datenbuffer unter anderem für Texturdaten oder für Modellpunkte und zugehörige Dreiecks-Indizes.

Nach Realisierung aller voreingestellten programmierbaren Komponenten und aller render-seitigen Objekte fällt die Implementierung des eigentlichen Raytracing-Verfahrens innerhalb des Raytracers leicht: Der eingehende Kommandopuffer wird Element für Element abgearbeitet, zugehörige Startknoten innerhalb der OptiX-Strukturhierarchie für jedes Kommando werden extrahiert. Schliesslich wird der Raytracing-Vorgang für jede Szene einzeln gestartet. Die Ergebnisse werden in einem weiteren Datenpuffer aufgesammelt, und schliesslich mit OpenGL in einem Fenster dargestellt.

D.9 Einbettung des Renderkerns in Simulator X

Aufgrund der C-Schnittstelle der OptiX-API wurden alle vorangegangenen Entwicklungsschritte in der Sprache C++ ausgeführt. Dadurch wurden zahlreiche Probleme bei der Kollaboration zwischen einer systemnahen Sprache mit einer abstrakten Hochsprache – Ressourcenmanagement und Wrapping – auf die wesentlich schmäleren Schnittstellenklassen des Renderers reduziert.

Nach Abschluss aller Arbeiten am Raytracing-Kern mussten dennoch alle Schnittstellenklassen für die Verwendung in der Scala-basierten Simulator X-Umgebung in einer Java-Version zur Verfügung gestellt werden. Die Verwendung der JNI-Schnittstelle zur Verbindung zwischen C++ und Java ergab sich aus den Geschwindigkeits-Vorteilen, und dem im Vergleich zur JNA-Implementierung wesentlich vielseitigeren Anbindungsverfahren.

In der Portierung wurde dabei für jede Schnittstelle in C++ ein entsprechendes Java-Interface definiert. Intern hält jedes Java-Objekt einen gekapselten Zeiger auf eine entsprechende native Objektinstanz. Alle Funktionsaufrufe auf ein Java-Objekt werden

unvermittelt und mit den selben Verhaltensregeln an die gekapselte Instanz weitergereicht. Entsprechend werden Rückgabewerte über die JNI-Schnittstelle wieder in Java-Klassen portiert und an den Aufruf in der eigentlichen Anwendung weitergeleitet.

Grundsätzliche Probleme bei der Implementierung ergaben sich auf zwei verschiedenen Feldern: Lebenszeit-Management und Ausnahmebehandlung.

Das Problemfeld Lebenszeit-Management wird durch unterschiedliche, inkompatible Speicher- und Ressourcenverwaltung in Java und C verursacht. Während C++ Strategien für manuelles Ressourcenmanagement selbst unter Ausnahmebedingungen bereitstellt, benutzt Java komplett automatisierte Ressourcenfreigaben. Entsprechend ergeben sich zwei Alternativen für die Bindung der Lebenszeiten von nativen Instanzen an die Lebenszeiten der zugehörigen Java-Objekte. Zum einen wird ein manuelles Zerstören aller Objekte über explizite Java-Funktionsaufrufe von jeder späteren Client-Anwendung erwartet. Zum anderen werden aus Sicherheitsgründen spezielle Java-Finalisierer implementiert, die bei vergessener Ressourcenfreigabe versuchen, das native Objekt dennoch zu löschen.

Das Problemfeld Ausnahmebehandlung bedingt eine Übernahme aller Ausnahmen aus dem C++-Programm in die Java-seitige Schnittstelle. Daraus wiederum folgt aufgrund der zwingenden Ausnahmespezifizierer in Java ein Anpassungsaufwand auch für potentielle Endanwender. Um solche Schwierigkeiten zu begrenzen, ist aktuell nur einen einzelne Java-seitige Ausnahme definiert, alle auftretenden C++ Ausnahmen werden entsprechend konvertiert. Dadurch gehen jedoch leider weitergehende Ausnahmeinformationen verloren. So ist beispielsweise in Java-Code nicht mehr zwischen einer fehlenden Ressourcen-Datei, einer korrupten Datenübertragung, oder einem Zugriffsfehler zu unterscheiden.

Mit den genannten Lösungen für Lebenszeit-Management und Ausnahmebehandlung war die Konvertierung der nativen Schnittstellen nach Java abgeschlossen.

Die folgende Implementierung der Raytracing-Komponente innerhalb von Scala und dem Simulator X-Umfeld war schliesslich intuitiv zu bewältigen. Eine einzelne Aktor-Komponente wurde erstellt, eine spezielle Nachricht zur Konfiguration der Komponente wurde definiert. Das Verarbeiten eingehender Eintities und Zustandsvariablen wurde analog zur bestehenden Integration des jVR-Renderers durchgeführt. Schliesslich wurde die Raytracing-Komponente zu Demonstrations- und Testzwecken mit einer bedeutenden Beispielanwendung des Simulator X-Projekts verknüpft.

D.10 Bewertung des Raytracing-Kernels

Nach Abschluss der Integrationsarbeiten wurde das Raytracing-Kernel in der genannten Beispielanwendung dem jVR-Rasterisierer gegenübergestellt. Dabei wurden unter anderem Bildqualität, Leistung und Stabilität verglichen.

Bei Bildqualität und Leistung ergaben sich nur wenige Unterschiede zwischen beiden Implementierungen mit einem leichten Vorsprung für den klassischen Rasterisierer. Wie erwartet fiel der Raytracing-Ansatz dabei in hohen Auflösungen aufgrund eines höheren Aufwands pro Bildpunkt deutlich hinter der jVR-Darstellung zurück. Weitere

kleine Abzüge gab es für fehlende Unterstützung von nativer Texturen-Filterung oder Bild-Filterung in der OptiX-API.

Im Gegensatz zum Beinahe-Gleichstand bei Bildgüte und Geschwindigkeit zeigten sich in der OptiX-API insbesondere im Rahmen einer weitläufigeren Szene deutliche Stabilitätsprobleme. Beispielsweise führten leere Gruppenknoten innerhalb der OptiX-internen Szenenhierarchie bereits zu einem Abbruch des kompletten Renderverfahrens oder einem Anwendungsabsturz. Ebenso drastische Folgen hatten nicht-definierte, geräte-seitige Variablen, oder die Verwendung einiger Standard-Bildformate. Sofern überhaupt Fehlermeldungen ausgegeben wurden, waren diese zumeist schwer zu verstehen und deuteten auf interne Programmier-Fehler in der OptiX- oder CUDA-Schnittstelle hin. Wünschenswert wäre hier ein Verhalten ähnlich zu bestehenden Rendering-APIs wie OpenGl oder DirectX, die bei einem Fehler lediglich kurzfristige Bildfehler erzeugen.

D.11 Ausblick

In diesem Sinne fällt auch der Ausblick auf die Zukunft interaktiven Raytracings aus. Insbesondere erreichen aktuelle Implementierungen auf Grafikkarten bereits interaktive Geschwindigkeit und akzeptable Bildqualität. Allerdings steht für interaktives Raytracing noch ein wichtiger Meilenstein aus: Akzeptanz in veröffentlichten Software-Titeln.

Diese Akzeptanz setzt einen fortgesetzten Reifungsprozess auf dem Gebiet der Programmierschnittstellen voraus. Ein Schritt in richtige Richtung könnte beispielsweise eine standardisierte, hersteller-unabhängige API-Spezifikation für interaktives Raytracing darstellen. Daraus würden sich potentiell Alternativen zur OptiX-API ergeben. Mit zunehmendem öffentlichen Interesse könnten schliesslich Qualität und Stabilität der gerätenahen Implementierungen ansteigen.

Auch im Rahmen der Raytracing-Komponente von Simulator X bestehen offene Punkte. Zum einen ist im Moment nur ein grundlegendes Lichtberechnungs-Verfahren auf Basis von Blinn-Phong-Shading mit harten Schatten implementiert. Zukünftige Erweiterungen könnten beispielsweise pfad-basierte globale Beleuchtungssimulationen oder fortschrittliche Schattenberechnung integrieren. Zum anderen müssen auch die grundlegende Schnittstelle sowie die Integration in Simulator X erweitert werden. So werden fortgeschrittene Animationsverfahren – etwa Skelettanimation – bereits grundlegend im Raytracer-Backend unterstützt, allerdings noch nicht nach aussen hin zur Verfügung gestellt.

Eine abschliessende, langfristige Perspektive für interaktives Rendering wird von Tim Sweeney – Entwickler der kommerziellen Unreal Engine – vertreten: Dieser hofft in der kommenden Dekade auf eine Integration von GPU- und CPU-spezifischen Fähigkeiten in einem einzelnen, frei programmierbaren Kernprozessor. Dementsprechend könnten sowohl Rasterisierungs- wie Raytracing-Kernel in einer Hochsprache geschrieben und je nach Anforderungen einer Anwendung miteinander verwoben werden.

E Software Tools

The following software tools have been used in the creation of this thesis document and the accompanying programs and media documents:

- **Blender 2.5 Test Build**
  `http://www.blender.org`
- **Doxygen 1.7.4**
  `http://www.doxygen.org`
- **GIMP 2.6**
  `http://www.gimp.org`
- **JetBRAINS IntelliJ IDEA 10 Community Edition**
  `www.jetbrains.com/idea`
- **LaTeX 2ϵ**
  `http://www.latex-project.org`
- **MiKTeX 2.9**
  `http://miktex.org`
- **NVIDIA Cuda and Optix SDKs**
  `http://developer.nvidia.com`
- **OpenOffice 3**
  `http://www.openoffice.org`
- **TeXnicCenter**
  `http://www.texniccenter.org`
- **Visual Studio 2010 Express**
  `http://www.microsoft.com/germany/visualstudio`
- **Visual Assist**
  `http://www.wholetomato.com`

Thanks go out to all involved parties for their great work !

F Erklärung zur Authentizität

Hiermit bestätige ich, Tobias Werner, dass ich diese Masterarbeit inklusive zugehöriger Programmquellen selbständig angefertigt habe. Lediglich die unter Anhang B genannten Quellen und die unter Anhang E genannten Hilfsmittel wurden bei Erstellung der Arbeit einbezogen. Entsprechende Verwendungsstellen im laufenden Text sind ausdrücklich mit einer Referenz versehen. Ebenso habe ich diese Arbeit nicht bereits an einer anderen Universität zur Erreichung eines Mastergrades abgegeben.

Unterschrift: _____

(Tobias Werner)