Real Time Ray Tracing on Many-Core-Hardware

Iliyan Georgiev[1], Dmitri Rubinstein[1],
Hilko Hoffmann[2], Philipp Slusallek[1],

{georgiev, rubinste}@cs.uni-sb.de

{Hilko.Hoffmann,Philipp.Slusallek}@dfki.de

(1) Computer Graphics Group, Saarland University, Germany
(2) DFKI and Saarland University, Germany

(3) DFKI (German Research Center for Artificial Intelligence)

Abstract

With the shift from highly clocked, single thread processors to chips that have dozens or even hundreds of smaller processors, we are witnessing a fundamental change in the computing hardware we use. This paper describes an approach to use modern many-core-hardware to apply real time ray tracing to Virtual Reality environments. We show that current hardware can speed up real time ray tracing to interactive frame rates that make ray tracing interesting for the integration into existing industrial Virtual Reality systems.

Keywords: Real Time Ray Tracing, Rendering, Visual Realism, Virtual Reality Systems

1 Introduction

The introduction of many-core-architectures in CPUs as well as in GPUs can be considered as a revolution in terms of the available computing power of upcoming standard PC hardware. Increasingly, users have massive parallel compute power on their desks that is capable of running even highly compute intensive applications on commodity hardware. In the context of graphics applications the new hardware will, for example, enable the use of ray tracing based rendering systems at interactive frame rates. Real time ray tracing (RTRT) is thus complementing and maybe even replacing the classical rasterisation approach for certain types of industrial applications. Due to its straight-forward simulation of the physical process of light transportation, ray tracing is well suited to achieve realistic global lighting, shadowing, refractions, and reflections. In particular, it is the ease of use of ray tracing that makes it a relevant to industry. Ray tracing enables a declarative scene description, in which a description of the geometry of objects and its optical material properties is sufficient, to automatically generate images that are known to be correct (within the used model of light exchange and the accuracy of the input model). This guaranteed visual and physical realism directly addresses a key problem in the industry especially in the field of product development. Short product development cycles and the increasing number of product variants make it inefficient and sometimes impossible to build and evaluate hardware prototypes for each product configuration. Consequently, the industry is looking for technologies that support early and well-founded design and market decision. This applies for instance to evaluating the readability and ergonomic quality of instruments and displays in cockpits, reviewing the appearance of head and back lights of cars, or assessing the danger of reflections on windshields and other parts of the car under a variety of lighting conditions.

This paper describes the design of an RTRT system that allows the rendering of highly realistic, physically-correct visualizations on modern many-core-hardware at interactive frame rates. The RTRT system creates the basis for the integration into a commercial Virtual Reality (VR) system to be used in existing VR-installations.

2 Rasterization versus Ray Tracing

The majority of the currently available VR frameworks use the rasterization algorithm for rendering the virtual environments because it is the only hardware-supported and thus fast solution. The major feature of this algorithm is that it processes geometry primitives independently from each other, which enables highly efficient hardware implementations. However, this feature makes it rather difficult to implement any visual effect that requires interaction between primitives like reflection or refraction. In order to circumvent these limitations modern VR systems must employ complex multi-pass techniques to realize each one of these effects. Unfortunately, using multiple effects generally leads to a combinatorial explosion of rendering passes. For example, the latest nVidia Tech-Demo “Medusa” for GTX 200 GPUs use 120 rendering passes to achieve all of the desired effects.

The use of more rendering passes increases the computing power the graphics hardware must provide. On the other side, we see a tremendous improvement in the hardware support for ray tracing like algorithms. This in turn improves the performance of ray tracing which can trivially implement these types of high-quality and physically-correct images. This increasingly allows for replacing the complexity of many layered multi-pass algorithms with a simple, robust, and well understood algorithm. Instead of combining many complex and widely varying approximations that make it difficult to evaluate the validity of the visual results, we have only a single algorithm that closely resembles the physical process and can easily be analysed in terms of the accuracy of its results. This guarantee of the visual results is one major reason prohibiting the more widespread use of VR techniques in industry, where decision makers that cannot be expected to understand the intricacies of the rendering process currently cannot trust what they see in VR systems, which makes these systems largely useless to them. The rasterization hardware sequentially processes the geometric primitives leading to a basically linear complexity. Newer hardware features allow for spatial queries and conditional processing of primitives that together can be exploited to build system that exhibit logarithmic complexity. However, since the entire process cannot (yet) be implemented on the GPU, it easily becomes a complex interplay between algorithms and data structures distributed between the CPU and the GPU. Conversely, ray tracing exhibits an (average case) logarithmic complexity in the number of geometry primitives via its tight integration of a spatial index structure (Bentley, 1975). This leads to a very good and build-in scalability of ray tracing with respect to huge geometry data sets (Wald et al, 2004; Dietrich et al, 2006). It is important to note, that we can expect rasterization to increasingly move to a similar integrated approach – which makes it more and more similar to ray tracing, including all its advantages and disadvantages.

The first rendering framework that provided RTRT for the industry was OpenRT (Dietrich et al, 2003). Even today it is the only ray tracing library that can easily be integrated into applications.

OpenRT has been extensively used in the industry by e.g. Volkswagen, BMW, Audi, DaimlerChrysler, and EADS/Airbus. However, this framework was created when the modern features of today's CPUs and GPUs like wide-"SIMD" and multi- and many-core architectures were not yet available on the mass market. Another drawback of the OpenRT was its performance hit when used for rendering of highly dynamic scenes and its use of cluster architectures for achieving high performance and scalability.

Probably the biggest issue for all existing realtime ray tracing implementations is the reliance on hard-coded processor and platform specific optimizations to reach the best possible performance. As a result developers have to choose between implementation flexibility and high-performance, which often leads to awkward and mostly arbitrary compromises. In order to achieve the maximum performance, some systems compromised flexibility, remaining fast only in very specific configurations of algorithms and data structures. Others systems allowed flexibility by exposing object-oriented interfaces and sometimes even external APIs. However, the overhead of these approaches is generally high and these systems suffered from significant performance drawbacks.

3 Software Architecture for Realtime Ray Tracing

In order to overcome the limitations of these systems and to develop an RTRT system for the use in industrial VR-systems we decided to take a completely new approach that allows for offering both flexibility as well as highest performance – within the same ray tracing framework. The new architecture targets the simultaneous and full exploitation of all features of modern CPUs and GPUs to increase performance on today's PC architectures that typically combine many CPU and GPU cores.

3.1 RTfact: A Generic and Fast Ray Tracing Engine

The new ray tracing engine "RTfact" (Georgiev, 2008) has the goal to provide a maximum performance on the latest generation of CPUs and GPUs without compromising flexibility.

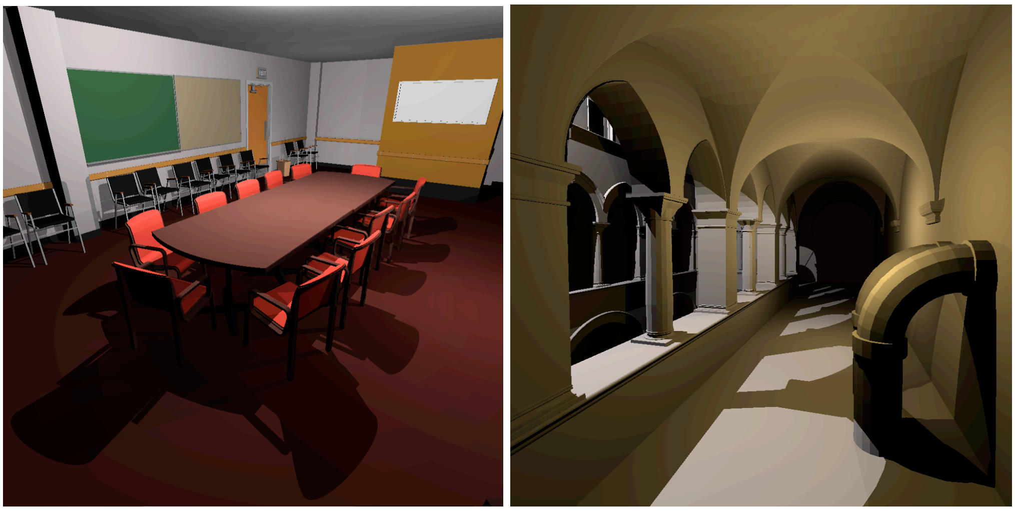

Figure 1: Example images rendered using RTfact at 1024x1024 resolution on a mobile single core of a 2.6 GHz Core 2 Dual processor at 8.2 and 7.1 fps.

RTfact is a ray tracer prototyping library that provides the building blocks for creating custom ray tracing-based solutions, rather than providing a complete rendering system. Employing generic programming paradigms (template programming), the library combines the flexibility of off-line rendering systems with the performance of modern ray tracing algorithms. Separating the many different algorithms building or operating on ray tracing data structures from each other as well as from the concrete representation of the data, allows us to achieve seamless component integration and composability not previously seen in other RTRT systems.

Separating the basic ray tracing functionality from the rendering functionality gives us the ability to employ ray tracing not only for visualization, but also for many other VR tasks. Examples for such tasks are collision detection and object interaction, which today are commonly implemented through with RT-like approaches but usually employ their own algorithms and data structures. Figure 1 shows some example images rendered with RTfact.

A number of scene graph libraries exist for OpenGL, with OpenInventor (Wernecke, 1994) probably being the best known. Ideally, we would use an already existing scene graph library and adapt it for our ray tracing engine. However, previous approaches that went this way (Dietrich et al, 2004) experienced among others too much runtime overhead when animations were present in the scene, resulting in low rendering performance. The reason for this overhead is that all existing scene graph libraries are deeply integrated and optimized to use rasterization-based APIs and are not designed to allow for incrementally updating scene data after small or local changes somewhere in the graph. Typically, scene graphs perform complete retraversals of the entire graph, which can be quite costly. For ray tracing applications, which need to maintain spatial index structures, we are interested in the smallest possible set of operations that update these indices.

This difference between the two rendering techniques makes a simple integration of a ray tracing engine into a traditional rasterization-based scene graph library inefficient. Therefore we have developed “RTSG – Real-Time Scene Graph” library, based on the X3D ISO standard. We chose X3D because it is the only ISO ratified standard for interactive real-time 3D graphics and it allows for flexible extensions. Initially designed for 3D graphics for the Web, its importance for the industry is growing, as it also supports CAD, geospatially referenced geometry, and distributed interactive simulation (DIS).

In contrast to other implementations, RTSG has been designed from scratch to work well with the RTRT technology. Nevertheless, RTSG is fully independent of the underlying rendering architecture and framework. Therefore, applications that use RTSG can directly take advantage of both rasterization and ray tracing back-ends for visualization purposes.

Figure 2 shows a comparison between an OpenInventor- and RTSG-based application. An OpenInventor-based application is restricted to use OpenGL for rendering, also similar limitations have OpenSG and OpenSceneGraph libraries. Even DirectX cannot be used instead of OpenGL. Using RTSG gives to the application full flexibility to render the same virtual environment with different rendering back ends e.g. RTfact, OpenRT, OpenGL, DirectX, etc. Our first rendering backend used OpenRT API, current implementation uses RTfact and we are finishing our OGRE-based renderer which can use OpenGL and DirectX for rendering. We are also working on a hybrid rasterization and ray tracing rendering approach.

From the application's developer perspective the changes for moving from OpenInventor to RTSG are minimal. With our scene graph API developer can load, store, and render scenes, without deep knowledge of the rendering backend.

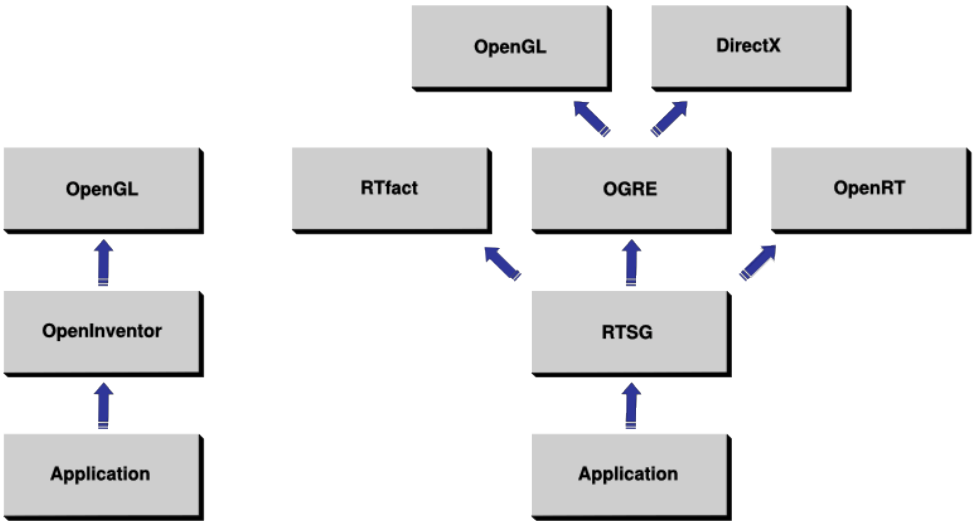

Figure 2: Comparison of OpenInventor (left) and RTSG (right) application scheme

The API is based on the X3D scene authoring interface (SAI) and thus is fully rendering backend independent. The way how the scene is processed depends fully on the renderer. For ray tracing we use a more efficient approach than full scene retraversal like other scene graphs. Instead the scene is traversed only once after the scene graph has been loaded. We register callbacks to all the parts of the scene graph which may change. When these parts are modified, e.g. as a result of animation, our callback handlers propagate changes directly to the underlying ray tracing library. For the RTfact framework as well as OpenRT geometry updates will cause rebuild of the spatial index structure, but changes in the material appearance will only cause update of a the parameters of surface shaders. Note that all updates are performed incrementally and we plan to use this feature in the future work in order to incrementally rebuild spatial index structure and thus additionally reduce the rebuild time overhead.

The disadvantage of the original OpenInventor, VRML, and X3D specifications is their use of an OpenGL model for material representation. This representation is too limiting for ray tracing, as for example reflective and refractive materials are not supported. Initially we added OpenRT-specific nodes in order to represent new material parameters. However we plan now to use latest extension to the X3D standard, which allow defining arbitrary parameters for programmable shaders, also for ray tracing materials. See Rubinstein, 2005 for more details.

4 Performance Evaluation

Direct comparisons of ray tracing performance are often quite difficult due to differences in scenes, view points, index build and traversal algorithms, etc. For the following comparison, we used a simplified scene representation that is well supported by all systems but does not show the advantages of each one. In particular, we compare the performance of RTfact to that of the original OpenRT system, to Arauna (Bikker, 2007), Manta (Bigler, 2006), and Wald (Wald 2007) for both kd-tree (K) and bounding volume hierarchies (B). See Figure 1 for example images. Table 1 gives a good overview of the performance numbers of RTfact and how they compare to other existing ray tracing systems. RTfact achieves even better performance than some of the other highly optimized ray tracing systems – despite its greater composability and flexibility. Since RTfact is still a fairly new system we are confident that we can increase performance even more – particularly, because it is easy to add new algorithms to it. In addition, we expect to integrate fast GPU ray tracing code into RTfact in the near future.

	SPONZA	CONFERENCE	SODA HALL
OpenRT (K)	4.5	4.2	5.1
Manta (K)	4.7	4.2	5.4
RTfact (K)	**6.8**	**6.4**	**6.5**
Wald (B)	n/a	9.3	11.1
Manta (B)	4.5	4.8	5.6
Arauna (B)	13.2	11.3	n/a
RTfact (B)	**13.1**	**11.6**	**11.4**

Table 1: Performance comparisons of different RTRT systems

Given the current trend to increasing numbers of cores both on the CPUs and the GPUs in combination with the good scalability properties of ray tracing, will allow us to achieve full realtime performance on systems that have 8 or more cores, while even quad-core systems show pretty good results already. We are currently building a prototype system that incorporates an eight-socket quad-core CPU system with two dual-GPU add-in card and 128GB of main memory, which should offer enough rendering performance for even highly challenging VR tasks, including global illumination.

5 Conclusions and future work

The first results made with RTfact and RTSG are showing that the many-core hardware boosts RTRT so that reasonable frame rates are possible even on commodity hardware. We focus on the development of RTfact so that the ray tracing engine is using many-core CPUs as well as GPUs at the same time to increase rendering speed. Furthermore the combination of RTRT and rasterization can be an option to reach frame rates even suitable for games.

6 References

Bentley J. L. (1975): *Multidimensional Binary Search Trees Used for Associative Searching.* Communications of the ACM 18, 9, 509-517.

Bigler, A. Stephens, and S. G. Parker (2006): *Design for Parallel Interactive Ray Tracing Systems*. IEEE Symposium on Interactive Ray Tracing. Dietrich A., Marmitt G., and Slusallek P. (2006)*: Terrain Guided Multi-Level Instancing of Highly Complex Plant Populations*. Proceedings of the IEEE Symposium on Interactive Ray Tracing, pp. 169-176.

Bikker (2007): *Real-time Ray Tracing through the Eyes of a Game Developer*, in IEEE Symposium on Interactive Ray Tracing.Dietrich A., Wald I. and Slusallek P. (2003): *The OpenRT Application Programming Interface - Towards A Common API for Interactive Ray Tracing,* in Proceedings of the 2003 OpenSG Symposium, pages 23-31.

Dietrich A., Wald I., Wagner M. and Slusallek P. (2004): *VRML Scene Graphs on an Interactive Ray Tracing Engine.* Proceedings of IEEE VR 2004.

Georgiev, Iliyan and Slusallek P. (2008): *RTfact: Generic Concepts for Flexible and High Performance Ray Tracing.* Proceedings of IEEE Interactive Ray Tracing Symposium, Los Angeles, USA, August, to appear.

Rubinstein, D. (2005): *RTSG -- Design and Implementation of a Scene Graph Library based on Real-Time Ray Tracing.* Diploma Thesis. http://graphics.cs.uni-sb.de/~rubinste/works/diplom.pdf

Wald, I. and Kollig, T. and Benthin, C. and Keller, A. and Slusallek P. (2002): *Interactive Global Illumination using Fast Ray Tracing.* Rendering Techniques, Proceedings of the 13th Eurographics Workshop on Rendering.

Wald I., Dietrich A., and Slusallek P. (2004): *An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models.* Rendering Techniques, Eurographics Symposium on Rendering.

Wald I., Boulos S., and Shirley P. (2007). *Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies*. ACM Transactions on Graphics, 26(1):6.

Wernecke J. (1994): The Inventor Mentor, Addison-Wesley

Video of nVidia's Tech-Demo “Medusa”, http://www.golem.de/0806/60425.html

X3D Specifications, http://www.web3d.org/x3d/specifications/