Integration of Realtime Ray Tracing Into Interactive Virtual Reality Systems

Hilko Hoffmann[1], Dmitri Rubinstein[1], Alexander Löffler[2], Michael Repplinger[1], Philipp Slusallek[1, 2]

(1) German Research Center for Artificial Intelligence (DFKI)
(2) Computer Graphics Group, Saarland University (UdS)

Abstract

Current processors provide high performance through parallelism by integrating more and more computational cores on a single chip instead of increasing the clock rate. This is true for both the CPU (multi-core of up to 8 cores) and even more so for the GPU (many-core of up to 240 cores). GPUs are still being programmed in vendor specific languages (like Nvidia's CUDA) but cross-vendor initiatives like OpenCL will allow for providing performance on a standard desktop PC that was previously only possible on supercomputers. With is upcoming Larrabee processor, Intel goes one step further and tries to combine the concepts and advantages of multi-core CPUs with that of many-core GPUs. It moves the entire rendering process into software providing more flexibility to realtime graphics applications like games or visualization applications.

In this paper we present a highly parallel System consisting of the completely new Realtime Ray Tracing engine „RTfact" and the Realtime Scene Graph „RTSG" that allow making good use of modern parallel hardware. RTfact accelerates rendering via ray tracing to the point where it can be used for interactive Virtual reality applications, while RTSG allows for flexible and high-level descriptions of 3D environments on the basis of the X3D standard that enable the description of 3D objects and their behavior. RTSG is thus the interface between Virtual Reality systems and a number of different rendering modules that includes ray tracing as well as fast rasterization via the OGRE library.

RTSG currently is the fastest X3D browser that optimally supports construction and design decisions through high image quality, exceptional visual realism, as well as the high degree of detail in scenes.

Introduction

One application area for Virtual Reality (VR) applications is he support for visual decision making for design, product development, as well as architectural visuali-

zation. Those use cases need the ability to accurately visualize the object properties such as the type of material, roughness, color, shape, or curvature. This in turn requires an accurate simulation of the illumination in an environment, possible from many light sources.

Current VR systems mostly use classical rasterization techniques to generate images. However, in contrast to computer games where the content development and optimization of visual display may require several man years, VR applications must be able to display and interact with a 3D scene almost immediately and essentially without much preprocessing or manual tuning of scene and display parameters. Even though, the results must abide to even higher standards, as the displayed images should not only look nice but are the basis for possibly far reaching decision – and therefore should be reliable and correct.

This is where traditional rasterization technology reaches its limitations, despite its high, hardware-accelerated rendering performance: The predefined graphics pipeline (Möller und Haines, 2002) makes it hard to flexibly use it also for other rendering techniques. Most of the available techniques put visual effects before physical realism, and the development effort as well as the rendering cost increases tremendously when realism for arbitrary scenes must be supported. A good example is Nvidia's demo "Medusa" (Golem, 2008), which requires a total of 120 separate rendering passes per frame.

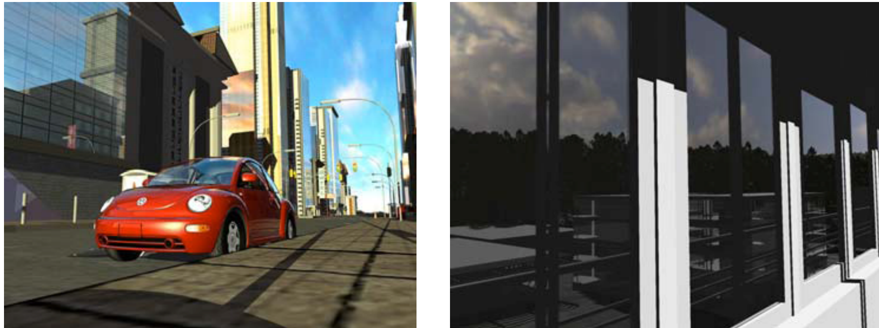

Figure 1: Ray tracing of realistic reflections of glass and varnish surfaces

Realtime Ray Tracing (Wald, 2001) is an alternative approach to rasterization that supports the physically correct simulation of illumination, reflection, hard and smooth shadows, dynamic light sources, and measured materials and luminaires. The resulting images are of high visual quality and with good input data hard are hard to distinguish from reality (Figure 1). However, ray tracing has very high computational demands that are not supported by specialized hardware.

The Realtime Ray Tracing Engine „RTfact"

Modern CPUs offer high performance through parallelism in the form of many cores and SIMD (Single Instruction Multiple Data) processing, which makes them well suited to accelerate ray tracing. New algorithmic approaches replace the predefined graphics pipeline by the flexibility and modularity of software that enables new kinds of visual realism. SIMD instructions are most effective when many similar computations have to be done simultaneously. We can exploit this by tracing an entire packet of rays together instead of individual rays because similar rays typically require very similar computations. However, the implementation of these advanced ray tracers is significantly more complex, these SIMD instructions must still be manually coded due to limitations in current C/C++-Compiler that have difficulties to automatically convert code to SIMD form. Instead, a programmer in the past had to code the core algorithms as well as the individual shaders in what is essentially assembly level programming using so called "instrinsics". Such ray tracing application were fast but inflexible, hard to extend, and largely non-portable to new processor versions. More flexible software approaches using common object-oriented design techniques and external libraries simplify programming but suffer significantly in performance.

The goal of the newly developed ray tracing engine "RTfact" (Georgiev, 2008) was to maximize performance on modern CPUs as well as GPUs without compromising flexibility. RTfact is no complete rendering system but a template C++ library that offers building blocks for the assembly of optimized and well adapted ray tracers. Through the use of C++ temples we combine the performance of SIMD code with the flexibility of object-oriented programming. However, in contrast we typically separate the algorithms from the concrete data representations that would normally put into the same class into separate template constructs ("concepts"), which follows the design principles used by the STL and Boost C++ libraries.

However, the result is no longer a binary ray tracing library but a set of source code files from which the compiler selects and combines the suitable features during compile time, based on very high level C++ template instantiations. This code is then automatically inlined into big basic blocks, which would be hard to write and maintain by hand but which offer great optimization opportunities for a compiler. In order to still provide traditional binary interfaces and shared libraries (DLLs), we create preconfigured binary libraries that cover the most used and optimized applications scenarios. For other cases the developer can directly use the original templated interface.

The Ray Tracing Scene Graph „RTSG“

The ray tracing engine RTfact offers rendering functionality on a level comparable to OpenGL or DirectX. However, most applications including VR systems prefer to work on higher level of abstractions – so called scene graphs. They allow for organizing a 3D scene using a set of hierarchically organized elements (e.g. geometric objects, light sources, cameras, sensors, etc.) instead of working with low-level API calls. For OpenGL there exist a number of scene graph libraries, such as OpenSceneGraph (Kuehne et al., 2007), or OpenInventor (Wernecke, 1994).

The significantly different rendering approaches makes it largely impossible to integrate RTfact with any of them as they have been optimized for OpenGL-style rendering that is hard to emulate in a ray tracing engine. The resulting system would be too inefficient and slow to support realtime VR applications (Dietrich, 2004). Instead we designed a new scene graph better suited also for ray tracing. The resulting library RTSG is based on the ISO standard “X3D” (Kloss et al., 2009).

The current version supports both RTfact for ray tracing as well as the “Object-Oriented Graphics Rendering Engine (OGRE)“ (Junker, 2006) which in turn uses OpenGL und DirectX for rendering. The programming interface of RTSG is based on the “X3D Scene Access Interface” (SAI) and is completely renderer agnostic. This level of abstraction also allows for creating hybrid rendering systems that simultaneously talk to multiple renderers and may even combine their results. No changes are required in the application as the rendering configuration can be specified in a separate application-independent configuration. Despite its separation between scene graph and rendering RTSG today is clearly one of the fastest X3D viewers available.

The Distribution Framework “URay”

RTSG and in particular RTfact are able to fully exploit the computational power of a PC but for very large scenes and highly realism the processors of a single PC may not be sufficient. In those cases it would be advantageous to be able to also exploit the capacity of other computers in the form of an on-the-fly or dedicated cluster of PCs. For that purpose we developed the “URay” framework (Repplinger et al., 2008) that can distribute and synchronize the computation across the Internet. This framework is based on the “Network-Integrated Multimedia Middleware (NMM)” (Lohse et al., 2008) operating on distributed flow graphs (see Figure 2).

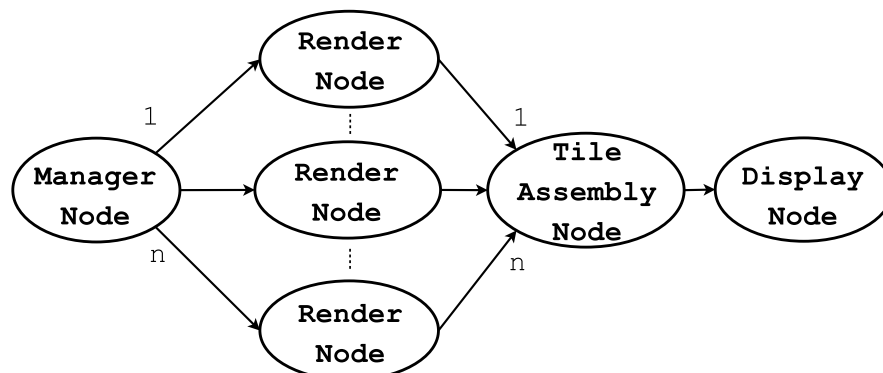

Figure 2: Example of a NMM flow graph within URay

The nodes of this graph represent individual processing steps (like ray tracing, tone mapping, or display) and can be distributed across multiple machines. For distributed rendering, a master node splits the frame buffer into tiles that are sent to a number of rendering nodes, which are typically separate machines so we can use all of their combined computational power. The resulting tile of pixels data are finally composited and displayed in additional nodes.

URay offers a large number of rendering and display options, including displaying the same image on multiple displays or tiling the display independently from the tiling used for rendering, or full stereo across all displays. Additional image processing nodes can be inserted in the flow graph to perform operations such as tone mapping, image warping, edge blending, and others. All displays can be synchronized across the network using NMM's built-in high-quality distributed synchronization framework. This general distributed rendering approach makes use of the fact that pixels are independent in ray tracing, which leads to a highly efficient and linearly scalable distribution approach.

Integrating Ray Tracing into a VR System

The software architecture of established VR systems, e.g., Lightning (Bues et al., 2008) or VR-Juggler, generally consists of three layers: an application and interaction layer (high-level), a scene graph system (mid-level), and a rendering layer (low-level). State-of-the-art systems based on rasterization mainly use OpenGL for low-level rendering libraries because of its platform independence and wide hardware support. The layer above, responsible for graphic abstraction and organization, usually uses classic scene graphs mostly on an OpenGL base, such as OpenSceneGraph or OpenSG. Only the application development on the top layer discriminates between the concepts of different VR systems, in this way defining their target group.

To integrate realtime ray tracing into an existing VR system, one can theoretically begin on any of the three layers. However, the application layer should remain unchanged because a change of this layer would affect all existing and future applications on the VR system and would necessitate writing them anew.

Using a rendering engine for realtime ray tracing inside the rendering layer instead of an engine based on OpenGL must fail in practice (as described in Section 3). The reason for this is that the structure of the scene graphs used is unsuitable for a reasonable integration of ray tracing (Rubinstein, 2005). Our approach within the VR system Lightning therefore uses the scene graph RTSG, which is completely independent of a specific rendering technology and supports both ray tracing and rasterization. RTSG already includes the ray tracing engine RTfact, as well as the rasterization engine OGRE.

The rendering specific components of the scene graph OpenSceneGraph originally used by Lightning are completely replaced by RTSG in our approach. Figure 3 shows the applied architecture and the different levels of integration.

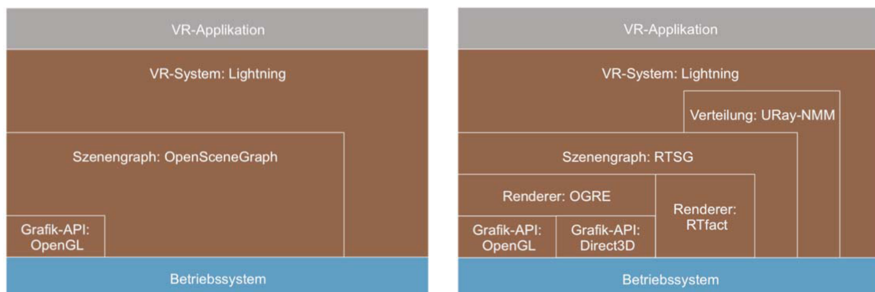

Figure 3: Architecture of a classic VR system (left) and the integrated system (right)

RTSG implements the X3D standard, providing standardized interfaces for an easy integration into existing VR and visualization systems. X3D also provides interfaces for powerful languages for defining application logic, for instance by the *scene authoring interface* (SAI).

In a typical VR system, there are many different possibilities for the realization of application logic. For example, the VR system Lightning has a C++ and a TCL/TK script interface. The developer has to decide on which level applications or components of application should be defined.

The proposed approach separates the application logic into two fundamental areas: object related components that can be directly assigned to the behavior of an object within the scene (e.g., a traffic light pre-emption). Components spanning different objects define the behavior of the entire system (e.g., traffic simulation). The previous ones are implemented directly in X3D and can be reused in different applications, while the latter are developed on the level the VR system.

Conclusions and Future Work

Through the development of multi core processors that can be programmed flexibly, ray tracing is a genuine alternative to image rendering using graphic boards. In VR applications, ray tracing provides an inherent image quality and a degree of realism that the classic methods cannot achieve without enormous effort. Today's flexible hardware increasingly dissolves the borders between rasterization and ray tracing. This makes hybrid systems imaginable benefit from the advantages of both approaches.

The integration described above shows that the use of ray tracing in immersive, interactive applications with a justifiable amount of hardware. However, there is still a need for high performance clusters to achieve a smooth interaction and reaction of the system. The possibilities of visualizing applications with highly dynamic scene content are currently very limited, though the image quality is high.

The big breakthrough is probably only to be expected when ray tracing can be used by computer games on desktop computers and is integrated into common graphic drivers or development libraries, respectively.

References

Akenine-Möller T., Haines E. (2002): *Real-Time Rendering*. AK Peters, Natick, Mass. 2002.

Bues M., Gleue T. and Blach R. (2008): *Lightning - Dataflow in Motion*, in Proceedings of the IEEE VR 2008 workshop "SEARIS-Software Engineering and Architectures for Interactive Systems", 2008.

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

Golem (2008): *Nvidia Tech-Demo „Medusa“*; http://www.golem.de/0806/60425.html

Georgiev, I. and Slusallek P. (2008): *RTfact: Generic Concepts for Flexible and High Performance Ray Tracing*. Proceedings of IEEE Interactive Raytracing Symposium.

Lohse M., Winter F., Repplinger M. und Slusallek P. (2008): *Network-Integrated Multimedia Middleware (NMM)*. Proceedings of ACM Multimedia 2008.

Junker G. (2006): *Pro OGRE 3D Programming*, Apress

Kloss J, Schickel P. (2009): *Programmierung interaktiver Multiuser-Welten für das Internet*; Addison-Wesley.

Kuehne B., Martz P. (2007): *OpenSceneGraph Reference Manual v2.2*; Skew Matrix Software and Blue Newt Software.

Repplinger M., Löffler A., Rubinstein D. und Slusallek P.(2008): *URay: A Flexible Framework for Distributed Rendering and Display*. Technical Report 2008-01, Informatik, Universität des Saarlandes.

Rubinstein D. (2005): *RTSG: Design and Implementation of a Scene Graph Library based on Real-Time Ray Tracing*, diploma thesis, Saarland University.

Wald I. Benthin C, Wagner M., Slusallek Ph. (2001): *Interactive Rendering with Coherent Ray Tracing*. Computer Graphics Forum 20, 3 (Sep. 2001): 153–164.

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