Realtime Ray Tracing for Current and Future Games

Jörg Schmittler, Daniel Pohl, Tim Dahmen, Christian Vogelgesang, and Philipp Slusallek
{schmittler,sidapohl,morfiel,chrvog,slusallek}@graphics.cs.uni-sb.de

Abstract: Recently, realtime ray tracing has been developed to the point where it is becoming a possible alternative to the current rasterization approach for interactive 3D graphics. With the availability of a first prototype graphics board purely based on ray tracing, we have all the ingredients for a new generation of 3D graphics technology that could have significant consequences for computer gaming. However, hardly any research has been looking at how games could benefit from ray tracing.

In this paper we describe our experience with two games: The adaption of a well known ego-shooter to a ray tracing engine and the development of a new game especially designed to exploit the features of ray tracing. We discuss how existing features of games can be implemented in a ray tracing context and what new effects and improvements are enabled by using ray tracing. Both projects show how ray tracing allows for highly realistic images while it greatly simplifies content creation.

1 Introduction

Ray tracing is a well-known method to achieve high quality and physically-correct images, but only recently its performance was improved to the point that it can now also be used for interactive 3D graphics for highly complex and dynamic scenes including global illumination [Wa04, WDS04, WBS03, BWS03, GWS04].

While the above systems still rely on distributed computing to achieve realtime performance, a first prototype of a purely ray tracing based graphics chip [SWWS04] shows that efficient hardware implementations are indeed possible and provide many advantages over rasterization

This encourages the research on possible effects of ray tracing technology for computer games. In this paper we describe our experiences with two games which use ray tracing for rendering and the physics engine.

2 Computer Games Based on Ray Tracing

Ray tracing and rasterization technology are basically two different algorithms to solve the same problem: the visibility calculation. While rasterization uses a set of potential visible triangles which are rendered sequentially into the Z-buffer, ray tracing starts at the virtual camera and for every pixel shoots rays into the scene. Since rays are terminated as soon

as they hit an object, the visibility calculation is highly efficient and fully output sensitive. As shading is performed after visibility calculation, you only pay for what you see.

While ray tracing has access to the entire scene database and only reads what it needs to, current rasterization technology operates on a stream of independent triangles sent by the application. Therefore it cannot efficiently and accurately render *global effects* such as shadows, reflections, and indirect illumination on demand, i.e. after finding out that these effects are actually visible. Every effect has to be split into several render passes by the application and relies on tricks and approximations (e.g. shadow and reflection maps) which are inaccurate and break down in many situations (e.g. multiple reflections).

In contrast ray tracing trivially supports global effects by shooting on demand additional rays for shadows, reflections, and refractions. This output sensitivity allows for efficiently rendering even highly complex scenes. For every pixel this recursive approach automatically combines all visible shading effect correctly without involving the application or the need for separate rendering passes. Even memory management of the graphics card's memory is handled automatically by the ray tracer [SLS03].

Basic shading computations are the same as for rasterization. Thus, the same shaders (e.g. for texture filtering and calculation of light intensities) and image filters (e.g. for anti-aliasing) can be used. However, ray tracing allows to adaptively shoot new rays as required. While both techniques can eventually achieve similar results, this requires complex and costly operation by both the graphics hardware and the application. In contrast ray tracing handles most effects automatically and internally. This greatly simplifies content creation for games, which is increasingly becoming a limiting factor for the gaming industry.

2.1 Traditional Ego Shooter

We started our research with adapting the existing, well-known ego-shooter *Quake 3: Arena* by Id-Software to use ray tracing for rendering. We concentrated our efforts on adapting shading effects and general game management because most otherwise difficult rendering effects (e.g. shadows and reflections) were automatically handled by the ray tracer.

The game engine was written from scratch and supports player and bot movement including shooting and jumping, collision detection, and many special effects like jump-pads and teleporters. The main development was done by a single student in less than six months.

The game engine interfaces with the ray tracer through the OpenRT-API [DWBS03], which is very close to the OpenGL. OpenRT manages all ray tracing events fully transparent to the application, making it unaware of the underlying ray tracing implementation, which may run on a single computer, a cluster of PCs, or a dedicated ray tracing hardware.

Figure 1 shows several example images from the game with many shading effects. The engine supports *all* of the standard effects of traditional computer games like dynamic placement of (blood) splats, texture animation and blending, volumetric fog, and pre-computed light maps (if desired).

Figure 1: Screen shots of the ray traced version of *Quake 3: Arena*.

Screen shots of live game play. While most images are taken from the PC-cluster-based version, the right-most image was rendered on the hardware prototype (1024x768, 32bit). All images were rendered at fully interactive rates of 5-20 fps.

Some of the effects supported by ray tracing: a portal providing a view into distant places, light effects in the power-up, and correct reflections in the ammo-box and on some spheres.

More ray tracing specific effects like dynamic lighting including shadows and physically-correct reflections and refractions are trivially supported by simply specifying the corresponding material attributes. This also holds for camera portals and surveillance cameras, which are automatically rendered correctly by default even if they recursively see each other.

Since ray tracing is output sensitive there is no need for any level-of-detail mechanism to reduce scene complexity. This allows for highly crowded scenes with many players, monsters, and complex trees in a forest. Furthermore as ray tracing efficiently supports multiple instantiations, even crowded scenes have negligible memory requirements and scene complexity has only a minimal impact on performance.

In summary, we were able to support all the traditional effects of Quake 3 while most effects were significantly simpler to implement. Looking at newer engines such as *Unreal 3*, we still see no effects that could not be supported easily by ray tracing.

2.2 Novel Game Design for Ray Tracing

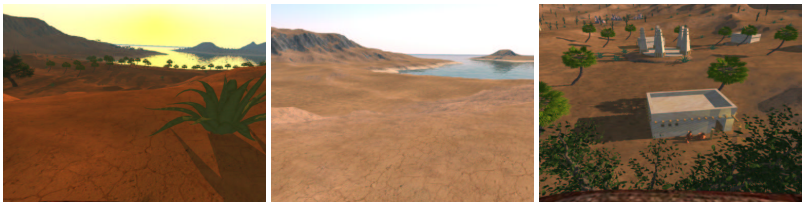
Ray tracing offers new ways to design a game which led us to the development of *Oasen* game. Oasen operates in a fully open space on a huge world consisting of several islands and includes day time simulation with changing sky and light situations (see Figure 2). The player takes the role of a salesman on a flying carpet visiting different places, buying and selling goods while fighting off other players or bots.

While ray tracing is basically a method for visibility calculation, we were also able to use it for the physics engine, acoustics, and collision detection. Similar to a radar system it uses rays to determine the distance to nearby objects. By reusing the existing fast ray tracer on the original geometry we avoided having to build special algorithms and data structures for those tasks.

No level-of-detail mechanisms, clipping-planes, or fog have been used to reduce scene complexity, since ray tracing efficiently handles huge amounts of geometry and objects automatically. This avoids any artifacts such as popping and results in smooth flights. Huge numbers of light sources are efficiently handled by exploiting the restricted range of illumination of each light and organizing them in a spatial index structure. For each pixel we can then efficiently locate light sources that contribute to its illumination, allowing physically-correct illumination at very low costs even in the presents of hundreds of visible light sources.

Figure 2: Screen shots of the ray tracing based game *Oasen*.

Typical life screen shots showing correct shadows, nicely rendered water including caustic-effects, and volumetric clouds.



The two left-most images show the inherent scene management capability of ray-tracing: vegetation and buildings add 40-times triangles over the basic landscape geometry while the performance drops by less than 10% – without the use of any level-of-detail or clipping techniques.

3 Conclusion

In this paper we briefly summarized the experience we gained from implementing two games using a ray tracing based game engine. We have been able to easily port all of the exiting rendering effects to ray tracing, where their implementation has been much simpler and more efficient. Furthermore ray tracing adds many novel aspects that help designing more realistic and compelling game contents.

Any shading effect can be approximated with rasterization, but every combination of sha-

ders requires special support and complex programming for both the application and shaders. In contrast, ray tracing automatically handles all shader interaction allowing for plug-and-play use of arbitrary shading effects. As a result, content creation is greatly simplified and game designers can again concentrate on the content and game experience instead of working around the many limitations of current technology. Furthermore ray tracing offers new ways to design the physics engine including collision detection and acoustics.

The main limitation of ray tracing has been its still limited support for dynamic scenes [WBS03], but ongoing research will soon remove this constraint. Even though LOD mechanisms were not needed for supporting complex scenes, new approaches are required to efficiently handle resulting geometric aliasing.

The highly realistic and physically-correct images together with a greatly simplified rendering engine make ray tracing an interesting technology for future computer games. Today, the system requirements for ray tracing based games seem to be very high as a cluster of PCs forming a virtual CPU with 30 GHz is required to render the images present here at interactive rates (5-20 fps for 640x480 pixels). But future hardware will allow to have even higher performance on a single PC board similar to today's graphics cards [SWWS04]. This encourages further research on ray tracing and computer games. More details and videos are available at `http://graphics.cs.uni-sb.de/RTGames/`

Literatur

[BWS03] Benthin, C., Wald, I., und Slusallek, P.: A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*. 22(3):621–630. 2003. (Proceedings of Eurographics).

[DWBS03] Dietrich, A., Wald, I., Benthin, C., und Slusallek, P.: The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In: *Proceedings of the 2003 OpenSG Symposium*. S. 23–31. Darmstadt, Germany. 2003. Eurographics Association.

[GWS04] Günther, J., Wald, I., und Slusallek, P.: Realtime caustics using distributed photon mapping. In: *To appear in Proceedings of Eurographics Symposium on Rendering*. 2004.

[SLS03] Schmittler, J., Leidinger, A., und Slusallek, P.: A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Volume 27, Graphics Hardware*. S. 693–699. 2003.

[SWWS04] Schmittler, J., Woop, S., Wagner, D., und Slusallek, P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In: *To appear in Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*. 2004.

[Wa04] Wald, I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis. Computer Graphics Group, Saarland University. 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WBS03] Wald, I., Benthin, C., und Slusallek, P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In: *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*. 2003.

[WDS04] Wald, I., Dietrich, A., und Slusallek, P.: An interactive out-of-core rendering framework forvisualizing massively complex models. In: *To appear in Proceedings of Eurographics Symposium on Rendering*. 2004.