Realtime ray tracing of dynamic scenes on an FPGA chip

Conference Paper · August 2004

DOI: 10.1145/1058129.1058143 · Source: DBLP

Sven Woop
Intel
25 PUBLICATIONS **904** CITATIONS
SEE PROFILE

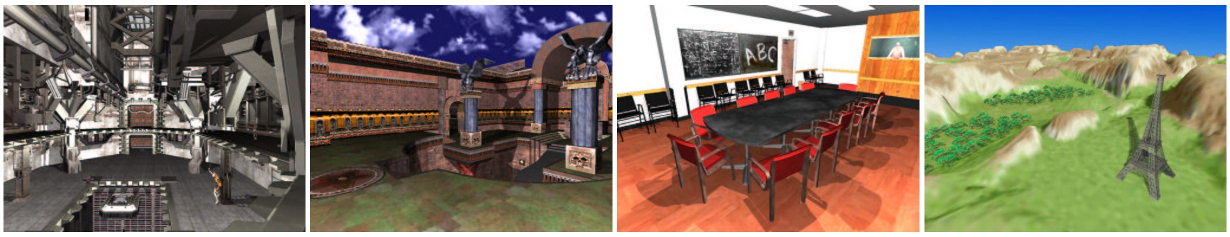
Philipp Slusallek
Universität des Saarlandes
320 PUBLICATIONS **5,660** CITATIONS
SEE PROFILE

Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip

Jörg Schmittler, Sven Woop, Daniel Wagner[†], Wolfgang J. Paul, and Philipp Slusallek[‡]

Computer Science, Saarland University, Germany



Several example scenes rendered in realtime with 20 to 60 frames per second on the SaarCOR prototype hardware. From left to right: UT2003, Quake3, Conference and Terrain. Adding support for dynamically modifying these scenes actually reduces the hardware requirements.

Abstract
Realtime ray tracing has recently established itself as a possible alternative to the current rasterization approach for interactive 3D graphics. However, the performance of existing software implementations is still severely limited by today's CPUs, requiring many CPUs for achieving realtime performance.
In this paper we present a prototype implementation of the full ray tracing pipeline on a single FPGA chip. Running at only 90 MHz it achieves realtime frame rates of 20 to 60 frames per second over a wide range of 3D scenes and includes support for texturing, multiple light sources, and multiple levels of reflection or transparency.
A particular interesting feature of the design is the re-use of the transformation unit necessary for supporting dynamic scenes also for other tasks, including efficient ray-triangle intersection as well as shading computations. Despite the additional support for dynamic scenes this approach reduces the overall hardware cost by 68 %.
We evaluate the design and its implementation across a wide set of example scenes and demonstrate the benefits of dedicated realtime ray tracing hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Hardware Architecture]: Graphics processors, Parallel processing I.3.7 [Three-Dimensional Graphics and Realism]: Ray Tracing, Animation

1. Introduction

Over the last two decades rasterization has become the dominant approach for interactive 3D graphics. However, due to its inability to directly access more than a single triangle at a time, rasterization is increasingly limiting the advancement of interactive 3D graphics and content creation. Without access to the entire scene even such apparently simple things as shadows, reflections, and indirect lighting are difficult to implement correctly and require non-trivial support from an application.

Application programmers must spend an increasing amount of their time working around these limitations instead of concentrating on their main job. Similarly, content creators struggle setting up special tricks and fakes necessary for at least approximating the required effects. As an example, rendering reflections of close-by objects requires to dynamically generate reflection maps depending on the current view and scene configuration. If applied to large or curved surfaces these maps must be applied carefully in order to avoid exposing the inherent inaccuracies of this approach.

[†] {schmittler,woop,cebron}@graphics.cs.uni-sb.de
[‡] {wjp,slusallek}@cs.uni-sb.de

Ray tracing does not suffer from the same issues because it can directly, efficiently, and accurately simulate the path of light through an entire scene. While this allows highly realistic rendering and much simpler content creation, the required computations have been hugely expensive in the past and only allowed for off-line rendering. Recently, however, improved ray tracing algorithms and optimized software implementations already achieved realtime performance even on a single PC for simple scenes [Wal04]. But realtime performance in larger scenes or with non-trivial lighting effects still requires the combined power of many processors [Wal04].

Realtime ray tracing can become a viable alternative graphics technology if it can be offered within a single desktop computer. This requires highly parallel hardware that delivers the necessary floating point performance. In the past a number of proposals have been made, including the use of GPUs [PBMH02] and custom hardware [Gre91, GH96, MKS98]. However, none of these proposals was able to deliver realtime performance for non-trivial scenes while supporting all the usual ray tracing features.

In this paper we present an implementation of a fully featured ray tracing pipeline in hardware. Using a single FPGA chip running at 90 MHz it offers realtime rendering performance of 20 to 60 frames per second while supporting all important ray tracing features, like Phong-like shading, textures, reflections, transparencies, shadows, and indirect lighting even in non-trivial scenes.

The hardware also supports interactively changing the scene through an object-based approach that allows to affinely transform entire groups of triangles. The same approach can also be used to instantiate these objects multiple times at arbitrary locations in the scene.

This technique requires frequent transformations of rays between different coordinate spaces using a dedicated transformation unit. Unfortunately, this unit requires significant hardware resources while remaining idle for long periods of time during which other units perform different ray tracing computations.

Consequently, we use special algorithms for ray-triangle intersection computations as well as for important parts of the shading computations that re-use the available but idle transformation unit. This strategy greatly simplifies the intersection and shading units and reduces the overall hardware cost by up to 68 %.

1.1. Previous Work

Ray tracing offers many benefits in the context of a hardware implementation. In particular, it scales trivially and can easily be parallelized. For a detailed overview of the state-of-the-art in interactive ray tracing see [WPS*03].

Several approaches have already been realized on MIMD and SIMD architectures [GP89, GP90, LS91]. Exploiting this scalability by massive parallelization has recently allowed interactive ray tracing to be achieved in software. It was first realized on supercomputers [Muu95, KH95, PSL*99] and more recently, interactive performance has been brought to clusters of standard PCs [WBWS01, WPS*03].

The availability of interactive ray tracing required efficient algorithms to support dynamic scenes. Thus [RSH00] covered dynamic updates to scene data structures and [LAM01] investigated lazy evaluation for scene data. More recently [WBS03] presented dynamic scene management on a distributed rendering cluster. Furthermore there is the BART suite [LAM00], a well designed benchmark suite to evaluate implementations of dynamic ray tracing systems.

Our hardware implementation is based on an extended version of the software approach outlined in [WBWS01, WBS03] which has been shown to efficiently handle the BART suite. Section 5 shows that the prototype's performance is several times faster than the software version on a single CPU over a wide range of different dynamic scenes. Thus although we did not implement the BART suite for the prototype we expect it to run well on our hardware.

Besides using existing architectures, several special purpose hardware architectures for ray tracing have been developed. Initially hardware support was provided only for the intersection operation [Gre91]. Later DSPs were used to build PC-card based ray tracing accelerators [GH96]. Several volume ray casters on PC-cards have been developed [MKS98, PHK*99, Pfi01] and there is a commercially available hardware architecture for high quality off-line ray tracing [Hal01].

While these projects achieved remarkable results and work well as accelerators for ray tracing, none of them is capable of delivering full-screen realtime frame rates comparable to those of current rasterization hardware. Recently, a proposal for a complete hardware architecture for ray tracing has been simulated that supports static scenes [KiSSO02].

Instead of designing special purpose hardware for ray tracing, another interesting approach maps it to more general future architectures: Using a multi-processor system on a single chip [MPJ*00], Purcell has developed a highly optimized distributed ray tracer that would be capable of delivering interactive performance [Pur01]. More recently, ray tracing was also mapped to rasterization hardware using their programmable pipelines [PBMH02]. These two projects show that ray tracing can in principle be realized on such multi-processor single chip hardware architectures.

1.2. Overview of the Paper

In this paper we explore the design of special purpose hardware for ray tracing. In particular, this allows us to explore the performance and efficiency with which ray tracing can be implemented in hardware. As we will see later, the hardware efficiency of those implementions on GPUs and CPUs is rather low compared to our custom hardware implementation.

The design for our ray tracing hardware is based on the SaarCOR [SWS02] architecture and adapts it for an implementation on modern FPGAs. We discuss the necessary changes to the architecture and evaluate the implementation and its performance using example scenes from several different application scenarios.

This paper is structured as follows: We start with the presentation of a ray tracing algorithm for dynamic scenes which uses transformations and discuss the re-use of these transformations for other tasks. The architecture of our design and its prototype implementation is then described in the Sections 3 and 4. We analyze and evaluate the design in Section 5 before discussing possible consequences and future work in Section 6.

2. Ray Tracing of Dynamic Scenes

The basic ray tracing algorithm intersects rays with triangles and stores the intersection that is closest to the ray's origin. To increase performance, a *spatial index* is used that subdivides space and allows to quickly locate objects in space. Rays then only have to be intersected with objects located in those spatial regions that are pierced by the ray. With hierarchical indices the expected runtime of ray tracing becomes logarithmically in the scene complexity. This compares favorably to the linear complexity of a rasterization unit (unless the application implements similar techniques on top of the rasterization engine, i.e. using occlusion queries).

However, the creation of the index is at least linear in the number of scene primitives, which seems to remove the advantage for ray tracing in the case of dynamic scenes. Fortunately, often large parts of dynamic scenes remain static for long periods of time allowing for amortizing the cost of building a spatial index over many frames. Additionally, we can trade off the building cost against the quality of an index depending on its expected live time allowing us to spend less time for building an index that cannot be amortized over many frames.

We base our approach on [WBS03] where primitives are split into separate *objects* that are either completely replaced by modified content or which move under a single affine transformation with respect to the scene. We focus on the second approach which allows us to pre-compute and re-use a separate *bottom-level spatial index* for each such object. Each such index can then be positioned in the scene with an affine transformation and is inserted into a *top-level spatial index*, which must be rebuild whenever any object moves.

When a ray encounters an object in the top-level index it is transformed into the object's coordinate space and can now continue traversing the pre-computed index of the object [WBS03]. Objects can be instanciated multiple times simply by storing multiple transformations together with references to the original object. This approach maintains the logarithmic cost in scene complexity even for dynamic scenes with instances.

The same approach can also be applied to a hardware implementation of ray tracing but this requires that a functional unit is provided for performing the transformation of the rays. This unit is expensive in the number of floating-point units that are required but remains idle most of the time. Consequently, we investigated other uses of the transformation unit in ray tracing, which are discussed below.

2.1. Ray Triangle Intersection

As the core part of ray tracing is the computation of the intersection between a ray and a triangle, this operation has received much attention in research (see [MT97, Eri97, Wal04]). In this section, we present a slightly extended version of Arenberg's algorithm [Are88] that is optimal for our purposes as it utilizes the transformation unit to perform most of the ray-triangle intersection calculations and also computes the dot product between ray direction and the normal of the triangle for free.

Figure 1 illustrates the *unit triangle intersection method* which consists of two stages: First the ray is transformed using a triangle specific *affine triangle transformation* to a coordinate system in which the triangle is the *unit triangle* Δ_{unit} with the vertices $(1,0,0), (0,1,0)$, and $(0,0,0)$. In the second stage, a much simplified intersection test of the transformed ray with the unit triangle is performed.

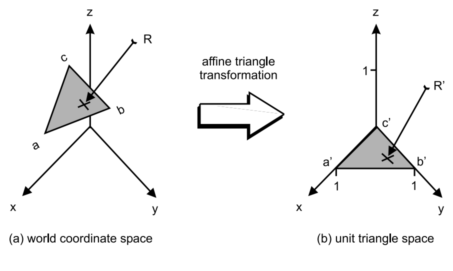


Figure 1: *The unit triangle intersection method consists of two stages: First the ray is transformed, using a triangle specific affine triangle transformation. In the second stage, a simple intersection test of the transformed ray with the unit triangle is performed.*

2.1.1. Affine Triangle Transformation

The affine triangle transformation to a triangle $\Delta = (A,B,C)$ with $A,B,C \in \Re^3$ is an affine transformation $T_\Delta(X) = m \cdot X + N$ with $m \in \Re^{3,3}$ and $X,N \in \Re^3$ that maps the triangle Δ to the unit triangle Δ_{unit} such that the normalized normal $N = \frac{(A-C)\times(B-C)}{|(A-C)\times(B-C)|}$ of the triangle is mapped to the normal $N_{unit} = (0,0,1)$ of the unit triangle.

The inverse T_Δ^{-1} of T_Δ can easily be described by the following equations:

$$T_\Delta^{-1}\begin{pmatrix}1\\0\\0\end{pmatrix} = A \quad T_\Delta^{-1}\begin{pmatrix}0\\1\\0\end{pmatrix} = B$$
$$T_\Delta^{-1}\begin{pmatrix}0\\0\\0\end{pmatrix} = C \quad T_\Delta^{-1}\begin{pmatrix}0\\0\\1\end{pmatrix} = N$$

These equations map the vertices of the unit triangle to the vertices of the triangle Δ and the normal N_{unit} to N. The solution for T_Δ^{-1} takes the form:

$$T_\Delta^{-1}(X) = \begin{pmatrix} A_x - C_x & B_x - C_x & N_x - C_x \\ A_y - C_y & B_y - C_y & N_y - C_y \\ A_z - C_z & B_z - C_z & N_z - C_z \end{pmatrix} \cdot X + \begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix}$$

The transformation T_Δ^{-1} is unique and well defined. If the triangle is not degenerate its inverse T_Δ exists and is a one-to-one affine transformation.

2.1.2. Unit Triangle Intersection

A ray $R = (O,D)$ with the origin $O \in \Re^3$ and direction $D \in \Re^3$ is intersected with a triangle Δ by transforming R using T_Δ to the *unit triangle space* and intersecting the transformed ray with the unit triangle. We do not directly compute the point of intersection P but the *intersection parameter* $t \in \Re$, such that $P = O + t \cdot D$. The parameter t and the barycentric coordinates of P within Δ do not change under a one-to-one affine transformation. Thus it is equivalent to compute the ray-triangle intersection in world coordinate space or in unit triangle space.

This transformation greatly simplifies the intersection computation of the ray with the triangle. Let $R' = T_\Delta(R) = T_\Delta(O,D) = (m \cdot O + N, m \cdot D) = (O', D')$ be the ray transformed to the unit triangle space, then the intersection can be computed by:

$$t = -\frac{O'_z}{D'_z}, \; u = O'_x + t \cdot D'_x, \text{ and } v = O'_y + t \cdot D'_y.$$

2.1.3. Dot Product Preservation

An additional analysis of Arenberg's algorithm [Are88] allows to compute the dot product between the ray direction D and the normal of the triangle for free. Since the normalized normal of the triangle was mapped to the normal N_{unit} of the unit triangle, the dot product is simply $D' \cdot N_{unit} = D'_z$. This property can be exploited if designing shading units, as shaders typically require the cosine between ray direction and geometry normal for color calculation and ray generation (see next section).

2.1.4. Further Applications

This concept of first transforming a ray to a generic coordinate space before intersecting it can also be applied to many other types of geometric primitives, e.g. boxes, ellipsoids, cylinders, etc. The advantage is that only a single representation (the transformation) needs to be stored together with a flag indicating the type of primitive. Additionally only a much simpler and smaller primitive-specific second stage intersection unit must be added.

2.2. Shading

Shading a ray consists of two parts: computing the local scattering of light and spawning new rays to gather the incident light from certain directions. The transformation unit is well suited for the second part as discussed below.

2.2.1. Transformation of the Normal

For advanced shading effects such as reflection and refraction, the normal of the triangle is used to calculate secondary rays. This normal is specified in object coordinate space, but needs to be transformed to world coordinate space. Obviously this transformation can be performed using the transformation unit.

2.2.2. Ray Generation

Spawning of a ray requires several floating-point operations. These operations can be specified using a transformation $T(X) = (\, A \;\, B \;\, C \,) \cdot X + D$. For simplicity reasons we write $T = [A,B,C;D]$ to specify a transformation.

Using transformations for ray generation allows for using of the transformation unit and thus reduces the complexity of the shading unit. Therefore we compute a new ray R_{new} by providing a transformation T_{new} and an initial ray R'_{new} to the transformation unit which then calculates R_{new} as input to the remaining ray tracing units.

Primary Rays We specify a camera by its position C_p and an orthonormal basis $\{C_r, C_u, C_d\}$ formed by the right-vector, the up-vector, and the viewing direction. The values $x,y \in [-1,\ldots,1]$ parameterize the screen space with a unit view frustum of 45 degree. For each pixel (x,y) on the screen the initial ray $R'_{init} = ((0,0,0),(x,y,1))$ is mapped using the transformation $T_{init} = [C_r, C_u, C_d; C_p]$ to the primary ray $R_{init} = (C_p,\; x \cdot C_r + y \cdot C_u + C_d)$.

Shadow Rays Shadow rays can be calculated very easily given the incident ray $R = (O,D)$, its intersection parameter t, and the position of the light source L. Using $T_{shadow} = [L,O,D;0]$ and $R'_{shadow} = ((1,0,0),(-1,1,t))$ yields the shadow ray $R_{shadow} = (L,\ O+t\cdot D-L)$.

Reflection Rays The calculation of the reflection ray requires the normal N of the triangle, the normalized ray direction D, the cosine c between N and D, and a small positive value e to avoid self intersections. The ray is computed using $T_R = [D,N,D;O]$ and the initial ray $R'_{refl} = ((-e,0,t),(1,-2\cdot c,0))$ yielding the reflection ray $R_{refl} = ((O+t\cdot D-e\cdot D),(D-2\cdot c\cdot N))$.

Transparency Rays For calculating a transparency ray simply the same transformation T_R as for the reflection ray can be used with the initial ray $R'_{transp} = ((e,0,t),(1,0,0))$. A transparency ray then evaluates to $R_{transp} = ((O+t\cdot D+e\cdot D),D)$.

Refraction Rays Unfortunately, the situation for refraction rays is a bit more complicated but some part of the nonlinear refraction calculation can be performed using the transformation T_R as in the reflection case. In a first step we compute $\mu = \eta\cdot c-\sqrt{1-\eta^2+(\eta\cdot c))^2}$ using $c = N\cdot D$, which has been computed by the triangle intersection for free. The initial ray $R'_{refr} = ((t,0,e),(-\eta,\mu,0))$ is then mapped by T_R to the refraction ray $R_{refr} = (O+t\cdot D+e\cdot D,\ \mu\cdot N-\eta\cdot D)$ of a surface with the index of refraction η.

2.3. Implementation Issues

The transformation unit must perform a 3×3 matrix multiplication and a vector addition for the origin of the ray as well as another matrix multiplication for the ray's direction. Since a matrix multiplication is very costly in terms of hardware resources, only one matrix multiplication unit together with the vector addition is implemented. The transformation of the direction is performed by feeding zeros to the correct coefficients. This requires two steps of the transformation unit to completely transform a ray. Fortunately, in many cases the transformation unit is used to transform a sequence of several rays sharing the same origin (e.g. primary rays or shadow rays). In those cases this allows for reducing the workload of the transformation unit for a sequence of n rays from $2n$ to only $n+1$ steps.

In order to evaluate the hardware savings, we compare the number of floating point units of four different ray tracer designs. The first variant RT_{static} consists of a static ray tracing pipeline using a ray-triangle intersection algorithm I_w based on Wald [Wal04] and simple shading unit S that only performs the geometric calculations on rays with floating point operations. The second variant RT_{Dyn1} is a dynamic ray tracer using the optimizations described above and therefore only contains floating point units in the transformation unit T and in the small primitive intersection unit. Variant RT_{Dyn2} is a dynamic ray tracer build with standard methods using the units T, S and I_w. Similar to this variant, RT_{Dyn3} only differs in the ray-triangle intersection algorithm I_p based on Plücker [Eri97], which is a bit cheaper than Moeller-Trumbore's [MT97]. All designs have the traversal unit K in common.

Table 1 shows that with the optimizations presented in this paper, a dynamic ray tracing chip is already significantly cheaper in terms of floating point units than an optimized static ray tracer even though it offers additional new functionality. Furthermore, re-using the transformation unit for other purposes cuts the number of floating point units at least in half.

FP cost ratio	without Traversal	full design
RT_{Dyn1} / RT_{static}	66%	75%
RT_{Dyn1} / RT_{Dyn2}	46%	57%
RT_{Dyn1} / RT_{Dyn3}	23%	32%

Table 1: *Comparisons of the cost of four different ray tracers measured in floating point units. It shows that the optimizations presented in this paper allow to significantly reduce the hardware costs.*

3. SaarCOR Hardware Architecture

We build on the SaarCOR ray tracing architecture first presented in [SWS02], which we extend with support for dynamically moving objects and multiple instantiations of objects. In [SWS02] the focus was on the ray tracing core of a static ray tracer. It was shown, that the SaarCOR architecture scales well in the number of ray tracing pipelines, in particular due to its good caching behavior and consequently low memory bandwidth requirements. With approximately the same amount of hardware it achieved roughly the same performance compared to a rasterization based graphics card.

In order to remove the need to have the entire scene in graphics memory, [SLS03] presented a virtual memory architecture for ray tracing. This fully transparent technique uses graphics memory only as a cache and loads necessary scene data from host memory on-demand. Over a wide variety of scenes even a slow standard PCI bus is able to support a ray tracer with only a small impact on the rendering performance.

3.1. Dynamic SaarCOR Architecture

The SaarCOR architecture for dynamic ray tracing (see Figure 2) is designed for a custom chip that is connected to the host system bus and to several external memory chips, all placed on a single PC board. Camera settings and scene data are uploaded by the host. The memory chips store the scene

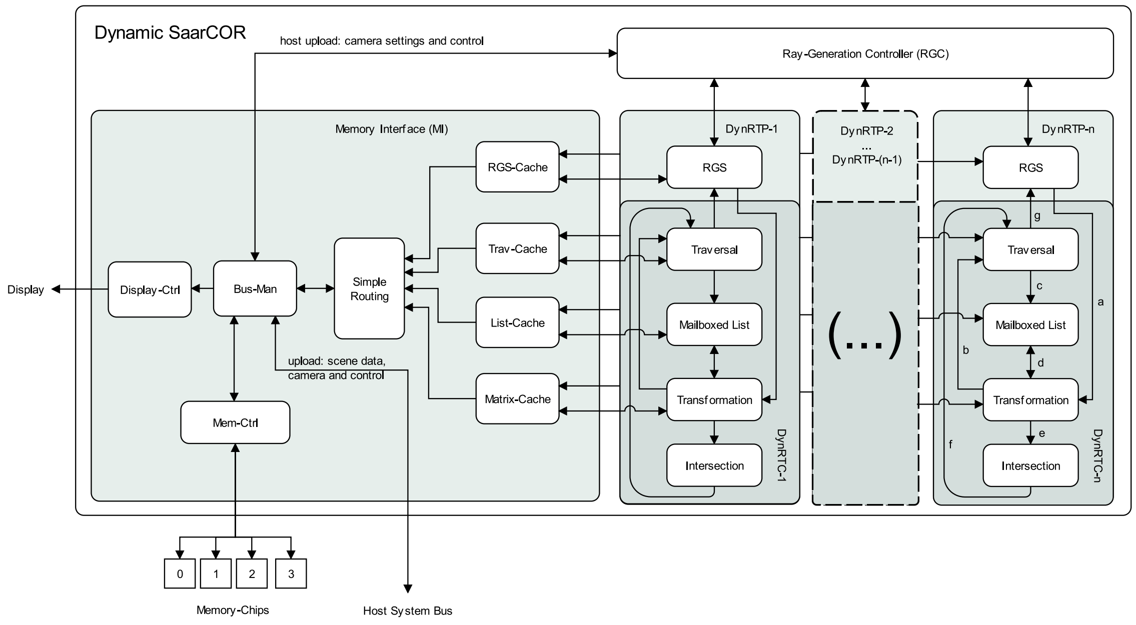


Figure 2: *The SaarCOR architecture for dynamic ray tracing is built into a custom chip connected to external I/O and memory. The chip is split into three functional sections: The dynamic ray tracing pipeline (DynRTP) which contains the dynamic ray tracing core (DynRTC) and the ray generation and shading unit (RGS). The RGS-units are controlled by a ray generation controller (RGC). The memory interface (MI) manages the connection between the rendering units, the memory, and the host system bus. Scalability is achieved by supporting several DynRTPs in parallel. Please note the simple routing scheme that uses only point-to-point connections.*

data including geometry, top- and bottom-level spatial index structures (in our case KD-trees), materials and shaders, as well as the frame buffer. The rendered image is displayed by the *Display-Controller*.

The core ray tracing algorithm is contained in the dynamic ray tracing pipeline (DynRTP). The tracing of a ray is started at the ray generation and shading unit (RGS) which generates an initial ray R' and transformation T which are sent to the dynamic ray tracing core (DynRTC) via path *a* of Figure 2. The transformation T is applied to the ray R' by the transformation unit and the transformed ray is sent to the traversal unit via path *b*.

The traversal unit then starts traversing the ray through the top-level KD-tree until a leaf node is found. The ray with the list of objects is then forwarded via path *c* to the mailboxed list unit, which sequentially fetches objects referenced in the list while omitting objects that have already been visited by the same ray. This technique greatly improved performance by avoiding the cost of multiple traversals of the same object.

Path *d* is used to send the ray together with the address of a transformation to the transformation unit which maps the ray into object space. The ray is sent to the traversal unit via path *b* to start the bottom-level traversal.

The same process starts over except that leaf nodes now contain lists of triangle transformations that are used to transform the ray into triangle space before being forwarded to the intersection unit via path *e*. The intersection results are returned to the traversal unit via path *f* and the operation continues at the list unit until all elements have been processed (similarly at the object level). Finally, results are handed back to the RGS unit for shading via path *g*.

In order to hide memory and other latencies multiple rays are being processed simultaneously using a multi-threading approach. Optionally packets of rays are used to further reduce the memory bandwidth in the ray tracing core. Scheduling of rays in this complex pipeline is trivial requiring only minimal state memory and simple logic to route rays to the correct next stage.

All memory accesses of the DynRTP are handled by the memory interface (MI) which contains individual caches for each type of data (i.e. shading data, KD-tree nodes, lists, and matrices for dynamical objects and triangles). All caches share the same connection to the memory via the *Simple Routing* unit. This unit acts like a multiplexer for memory requests and uses a labeled broadcast for memory responses [SWS02].

The memory controller (Mem-Ctrl) manages accesses to external memory chips and uses a simple reordering algorithm that optimizes the effective bandwidth of the memory chips by avoiding unnecessary switching of memory-banks. This performs close to optimally because the memory access pattern of tracing many rays simultaneously are almost random and thus statistically well distributed across the banks, which allows for keeping them all busy.

For scalability several DynRTPs are supported. The work performed in each DynRTP is controlled by the ray generation controller (RGC) which schedules screen-space pixels. This performs dynamic load balancing between the ray tracing pipelines while requiring only minimal bandwidth between the RGC and each of the DynRTPs.

The architecture exploits coherence between adjacent rays by using packets of independent rays as described in [SWS02]. This significantly reduces the bandwidth to the caches as only one memory request is performed per packet while the fetched data can be used for all rays. This greatly simplifies the interface to the cache and increases the scalability of the architecture.

3.2. Shading

The SaarCOR architecture is designed to support different types of shading processors: multiple general purpose processors like standard RISC-CPUs or shading units similar to today's rasterization hardware. A detailed description of those features is beyond the scope of this paper.

In this paper we focus on a fixed function shading pipeline that was implemented in the prototype. This shader supports multiple light sources, multiple levels of reflection or transparency and Phong-like shading with bilinearly filtered textures. For each ray that is shaded, shading data is fetched from memory, including material color, normals, texture-coordinates, and the texture base address. Additional memory requests are done for texture lookup. Each material and texture color consists of RGB data and values specifying the degree of specularity and opacity.

Rays are calculated using the formulas presented in Section 2.2 by forwarding the transformation and initial ray to the transformation unit via path *a* in Figure 2. Since for simplicity all colors are currently processed as 24 bit integers no floating point operations are required in the shading units.

4. Prototype and Implementation

While the original SaarCOR architecture already provided promising simulation results, it was never proven in the real world. Consequently, we decided to develop a prototype that demonstrates the usefulness and performance of a complete hardware implementation of ray tracing. The prototype was developed on FPGA hardware in less than six month by a small dedicated team of 3 students. It already provides realtime frame rates and all important features of ray tracing. We describe and evaluate this prototype in the following sections.

4.1. Development Platform

The SaarCOR prototype is build using a Xilinx Virtex-II 6000-4 FPGA [Xil03], that is hosted on the Alpha Data ADM-XRC-II PCI-board [Alp03]. The board contains six independent banks of 32-bit wide SRAM (each 4MB) running at the FPGA clock speed, a PCI-bridge, and a general purpose I/O-channel. This channel is connected to a simple digital to analog converter implementing a standard VGA output supporting resolutions of up to 1024×768 at 60 Hz.

The SaarCOR design was specified using JHDL [Bri03], an OpenSource high-level hardware description language allowing for fast and painless prototyping with the flexibility of parameterized designs. Additionally, we used Xilinx tools for schematic entry and low level synthesis. The system was completely developed under Linux.

4.2. Implementation Issues

The floating-point units of SaarCOR implement single operations, such as addition, comparison, and multiplication. They were modeled in JHDL with a parameterized precision in order to evaluate its effect on performance and rendering quality. The results of several benchmark scenes showed that 24 bit floating-point numbers with 16 bit mantissa provide a good compromise between accuracy and hardware cost. This format is similar to those used in current graphics boards by ATI.

Two of the six banks of SRAM are used to implement a double-buffered frame buffer with resolutions of up to 1024×768 pixels. A third SRAM stores all shading data, while the remaining three banks contain the KD-tree and the object and triangle transformation matrices.

The memory bandwidth for shading is rather low because only a small amount of shading data is required (28 byte per ray with bilinear texture filtering), and only the final pixel color needs to be written to the frame buffer. No bandwidth is required as for read-modify-write cycles of a Z-buffer or the overhead of overdraw. The resulting bandwidth is low enough (see Section 5) that it did not even require caches (which nonetheless would be fairly effective as shown by our simulations).

In [SWS02] it has been shown that 4 times more traversal operations than intersection operations need to be computed during the rendering of typical scenes. Since a single FPGA can only hold one intersection unit (see Section 4.3) we use packets of four rays which are traversed in parallel and are intersected sequentially. If all units would be fully utilized, they would require a raw bandwidth of 2 GB/s when running at 90 MHz. Due to small direct mapped caches that already yield very good hit rates and non-perfect utilization we easily reduce the bandwidth to a small fraction of the 1 GB/s available to the DynRTC (see Section 5).

4.3. Hardware Complexity

Table 2 lists the hardware resources required by a single ray tracing pipeline measured in the number of floating-point units for addition, multiplication, division, and comparison. The rightmost column also provides the amount of internal memory used for instance for ray-data and stacks that store 64 KD-tree traversal states per packet. These numbers include all additional index structures of the caches and dual port memory bits are counted as 2 bits. It is obvious that the arithmetic complexity and the internal memory requirements are extremely low.

Unit	Add	Mul	Div	Cmp	Mem
Traversal	4	–	4	13	44.5 KB
Mailboxed List	–	–	–	–	0.8 KB
Transformation	9	9	–	–	9.3 KB
Intersection	3	2	1	–	–
DynRTC-Cache	–	–	–	–	15.6 KB
Shader	–	–	–	–	4.8 KB
Total	16	11	5	13	75.0 KB

Table 2: *Complexity of one ray tracing pipeline measured in floating-point units for addition, multiplication, division, and comparison, respectively. The rightmost column also lists the internal memory requirements including any meta-data and global state, such as parameters for 8 light-sources. Each DynRTP uses 32 threads and contains caches that store 512 data-items each.*

Since the FPGA provided more capacities than required, we implemented a system with 64 threads, shading with support of 256 light sources, PCI interface, VGA interface, and performance counting infrastructure. Still our design only utilizes 56% of the FPGA's logic cells and 78% of the FPGA's memory resources including wasted resources due to memory layout and mapping constraints. The prototype runs at a frequency of 90 MHz and delivers a total of 4 billion FLOPs.

Much larger FPGAs are already available today that have about 60% more logic cells and four times more memory and multiplier blocks. This would support at least two additional ray tracing pipelines because the memory and multiplier resources have been the most limiting factor in the design of the prototype.

Our design also compares well to today's high end rasterization hardware. For instance, Nvidia's GeForce 5900FX [Nvi04] contains 125 million transistors (3-times more than Intel's Pentium-4). Its 400 FP-units running at a frequency of 500 MHz yield 200 billion FLOPs, which is 50-times the performance of the SaarCOR prototype. Despite that a direct comparison is not really possible, these resources would allow for quite a large number of parallel ray tracing pipelines.

As shown below the memory bandwidth of SaarCOR including *un-cached* shading is mostly far less than 300 MB/s plus additional 135 MB/s to display the image in 1024×768 at 60 Hz. As today's graphics boards offer more than 30 GB/s external memory bandwidth, this would support more than 100 independent ray tracing pipelines.

Together this indicates that SaarCOR could be scaled by one to two orders of magnitude going from our current FPGA technology to one used by today's rasterization engines.

5. Benchmark-Scenes and Results

The previous incarnations of SaarCOR [SWS02, SLS03] presented a ray tracer for static scenes and a virtual memory architecture, which were simulated and evaluated on the register transfer level using a cycle accurate simulator. We used the same simulator also for the Dynamic SaarCOR architecture as presented above. This simulation used a "standard" SaarCOR configuration with four parallel DynRTCs and provided essentially the same results as for the previous architectures: high cache hit rates, very low external memory bandwidth due to the use of larger packets and caching, good usage of all pipeline stages, high scalability, and lower hardware requirements for similar performance when compared to rasterization. Please refer to these papers for more detail.

In the following we focus on providing results of measurements of the FPGA prototype as discussed in Section 4. Even though the specific implementation and its parameters (such as memory latencies, etc.) are very different, these measurements confirm and verify the simulation results.

In particular we provide comparisons of the hardware prototype with the highly optimized software implementation of the OpenRT realtime ray tracing system [Wal04]. It uses SSE-optimized code on packets of four rays and algorithms almost identical to those of the prototype. The software runs on a Intel Pentium-4 with 2.66 GHz and 1 GB RAM. The SaarCOR prototype runs at 90 MHz on the configuration described above. The host computer's influence on the results are completely negligible since all computations are performed directly on the SaarCOR hardware except for uploading changes to the scene between frames. This uploading can be performed in parallel to rendering.

The configurations for the measurements differ in two ways from results presented in earlier papers: we use new algorithms for building KD-trees that greatly improve the quality and thus the traversal cost of KD-trees [Wal04]. This speedup is included in both the software and the hardware measurements given below. Furthermore all measurements include fully textured shading, which is quite costly on CPUs [Wal04].

5.1. Scenes used for Benchmarking

Table 3 lists the scenes and its parameters used to evaluate our prototype. Screen shots of some example scenes are show in the images on the first page of this paper. Full resolution images and videos can be found on the project's home page at `http://www.SaarCOR.de`.

To cover a wide range of 3D applications, we use a variety of scenes ranging from simple scenes, such as a simple room with table and chairs as in *Scene6* up to huge scenes with hundreds of millions of triangles using instantiation of complex objects as in *SunCOR* ($5,622$ sunflowers each consisting of $33,288$ triangles). We render the full details of all scenes with no level-of-detail mechanisms, as ray tracing handles such complex scenes easily due to its logarithmic computational complexity and its output sensitive computation that only ever touches visible parts of the scene.

Other more realistic examples are taken from computer games, such as *Castle* [Act02], *Quake3* [Id-99], and *UT2003* [Epi03]. While *Quake3* and *UT2003* only contain static meshes, *Quake3-p* also includes 16 moving players and monsters. The *Castle* scene also shows some nice ray tracing effects including multiple reflections. The videos to this paper include a small shooter game running on top of the ray tracer using the *UT2003* environment.

The other three scenes, *Office*, *Conference*, and *Terrain* provide more examples of indoor and outdoor scenes and show an office, a conference room with many light sources, and a large outdoor scene with complex trees casting detailed shadows.

The plug-and play concept of ray tracing allows to quickly and easily create such benchmark scenes: every object is described by its own self-contained shader independent of any other objects or shaders. All global effects resulting from interactions of multiple objects and shaders are computed correctly and on-demand during ray tracing. No manual tweaking or preprocessing is required except for building the spatial index structures.

This property of ray tracing greatly simplifies and speeds up content creation compared to rasterization, which requires many tricks to obtain approximations of global effects and care must be taken to avoid exposing their limitations.

5.2. Results

Table 3 compares the performance of the SaarCOR prototype to the OpenRT software implementation when rendering the benchmark scenes at 512×384 pixel using primary rays only, but including fully textured shading. It shows, that although the CPU is clocked 30 times faster than the SaarCOR prototype, the hardware is still 3 to 5 times faster. Thus, the 90 MHz prototype is theoretically equivalent to an 8 to 12 GHz CPU.

Looking at the raw FLOPs of the underlying hardware and comparing the resulting frame rates shows that SaarCOR uses its floating point resources 7 to 8 times more efficiently than the SSE-optimized OpenRT software on a Pentium-4. This means that even the highly optimized software ray tracing code uses the available floating point hardware only to a small fraction, indicating that the current CPU designs are non-optimal for ray tracing (even ignoring their insufficient maximum floating point performance).

In comparison, the fastest published ray tracer on GPUs delievers 300K to 4M rays per second on an ATI Radeon 9700PRO [Pur04]. In contrast, our simple FPGA prototype already achieves 3M to 12M rays per second at a much lower clock rate and using only a fraction of both the floating point power and the bandwidth of this rasterization hardware. An implementation in a comparable ASIC technology should allow us to significantly scale the ray tracing performance even further by at least an order of magnitude.

			#frames per second		
Scene	#triangles	#objects	SaarCOR	OpenRT	Speed-Up
Scene6	806	1	60.8	12.9	4.7
Castle	20 891	8	23.8	9.2	2.6
Office	34 312	1	48.9	10.4	4.7
Quake3	39 424	1	33.6	11.1	3.0
Quake3-p	52 790	17	26.7	7.9	3.4
UT2003	52 479	1	25.4	8.0	3.2
Conference	282 805	54	22.1	8.1	2.7
Terrain	10 469 866	264	15.9	3.5	4.5
SunCOR	187 145 136	5622	32.1	7.5	4.3

Table 3: *This table lists our fully textured benchmark scenes with their complexity (number of triangles and dynamic objects). The three rightmost columns compare the performance of the SaarCOR prototype with only one rendering pipeline running at 90 MHz to the OpenRT software ray tracer with SSE-optimized code on an Intel Pentium-4 2.66 GHz. The images were rendered at* 512×384 *pixel using primary rays only but including fully textured shading. It shows, that although the CPU is clocked 30 times faster than the SaarCOR prototype, the hardware is still 3 to 5 times faster. While all scenes use bilinear filtered textures, Scene6 and Office were rendered with unfiltered textures since otherwise the shading bandwidth would slightly limit the performance (10% at most).*

Table 4 provides a more detailed view on the external memory bandwidth and the cache hit rates. These measurements were taken at a screen resolution of 1024×768. In addition this table lists the usage ratios for the units of the DynRTC.

		Usage Rate [in %]				Cache Hit Rate [in %]			Ext. Bandwidth [in MB/s]		
Scene	fps	Trav	List	Transf	Int	Trav	List	Tranf	DynRTC	Shading	Total
Scene6	15.3	68	16	85	41	99	97	99	8	218	226
Castle	5.9	73	18	74	48	99	82	94	50	138	188
Office	12.4	76	16	72	36	99	71	90	69	246	246
Quake3	8.5	87	15	45	21	99	49	79	104	197	301
Quake3-p	6.8	92	13	35	16	99	66	82	65	157	222
UT2003	6.5	82	19	66	42	98	63	86	105	152	257
Conference	5.7	89	25	51	28	98	63	78	135	132	267
Terrain	4.2	80	27	34	18	97	27	43	283	98	381
SunCOR	8.7	46	36	29	12	90	2	6	513	202	715

Table 4: *This table provides details on the performance of the SaarCOR prototype running at 90 MHz and with a resolution of* 1024×768 *pixels using primary rays only but with bilinear-filtered textured shading. We provide the usage for each unit of the DynRTC, the hit rates of the all caches, as well as the external memory bandwidth (excluding frame buffer readout for display). It shows that multi-threading allows to efficiently keep most of the units busy and high hit rates are achieved even with tiny caches of only 4, 2 and 6 KB for traversal, list, and transformation, respectively. Please note that shading is uncached and while all scenes use bilinear filtered textures, Scene6 and Office were rendered with unfiltered textures since otherwise the shading bandwidth would slightly limit the performance (10% at most).*

They provide insight into the hardware efficiency looking at the ratio between the number of cycles a unit was busy versus the total number of cycles it took to render the image.

Our multi-threading approach results in high usage rates. Multi-threading increases performance almost linear up to 32 threads per DynRTP. Using 64 threads further improves performance by only about 10%. However, since the resources were still available on the FPGA we used the larger number of threads for our measurements.

Even in complex scenes the external bandwidth in total is small (mostly well below 300 MB/s, ignoring the fixed 135 MB/s required for frame buffer readout due to image display at a resolution of 1024×768 with 60 Hz). The bandwidth of the DynRTC is very efficiently reduced already by tiny caches of only 12 KB total. The bandwidth requirements for shading are constant per frame as ray tracing shades every pixel exactly once and no caches are used. Only a single texture with bilinear filtering is used in the examples because the need for complex multi-texturing is greatly reduced in ray tracing as light, reflection, and environment maps are replaced by tracing the necessary rays. Bandwidth is further reduced by not having to generate these maps in the first place.

We only provide measurement data for primary rays as the performance measured in rays per second for secondary rays is identical, as it only depends on the specific arrangement of geometry in particular scenes, e.g. the placement of light sources and complexity of the scene visible through reflections. This means that switching on secondary rays can even improve the overall performance measured in rays per second.

These results confirm the results of earlier simulations [SWS02] and are obvious since the number of operations required to trace a ray and the amount of memory transfered only depends on the location of the ray in a specific scene and not on its type. The same holds for the memory bandwidth as ray coherence is mostly preserved when generating secondary rays. Secondary rays do influence the cache depending on the amount of additional data that is accessed, which again depends on the specific scene.

When using packets of rays, the performance also depends on the coherence of rays within a packet. However, coherence has been much higher than generally expected. Shadow and reflection rays are mostly coherent except for extreme cases. But even global illumination computations can be designed to use highly coherent rays [BWS03]. Our measurements on the prototype using 200 (virtual) point light sources for approximating the indirect illumination in a scene showed that the number of rays computed per second are constant compared to rendering the scene without shadow rays.

6. Conclusion and Future Work

Ray tracing is still perceived by many as an offline technique for high-quality images. Even though realtime software implementations are available for some time now, their dependence on larger clusters of PCs for good performance has been a major drawback.

With this paper we present what we believe to be the first realtime ray tracing hardware. With the prototype hardware we demonstrate that ray tracing is at least as well suited for hardware implementation as the ubiquitous rasterization approach. Even the rather simple prototype implementation of ray tracing already achieves realtime performance for a wide variety of scenes. Furthermore due to its performance which is several times higher than any current CPU, it could also be used to accellerate ray tracing in offline rendering packages.

Ray tracing hardware allows to overcome two of the main limitations of rasterization hardware. The external memory bandwidth requirements of ray tracing are only a tiny fraction compared to rasterization, where bandwidth has been a major limiting factor. Furthermore, ray tracing offers almost unlimited and very efficient scalability by adding multiple pipelines per chip, multiple chips per board, and/or multiple boards per PC.

Scalability is mainly limited by the bandwidth to the scene data. However, exactly this bandwidth can easily be reduced using packets of rays, caching, or (cached) replication of the read-only data. Ray tracing greatly benefits from its demand-driven and output-sensitive type of processing that minimizes the processing to only the relevant parts of the data.

Ray tracing hardware allows to overcome many limitations of the current rasterization approach. As demonstrated it supports huge and complex scenes, accurately computes many advanced rendering effects, and simplifies content creation significantly.

In particular we have shown that integrating a shared transformation unit into the ray tracing pipeline provides many benefits. It offers support for object-based changes to a scene by applying affine transformations to entire groups of primitives. In addition, the same unit can be re-used for greatly simplifing intersection computation as well as for implementing important parts of the shading and ray generation process.

Surprisingly, the addition of this new functionality actually *reduces* the overall hardware requirements because it factors several costly but underutilized parts of the pipeline into a single well utilized component. We were able to reduce the hardware costs by 68% compared to other implementations for dynamic ray tracers and still 25% compared to a simple static implementation with less functionality.

Of course, our prototype implementation still leaves room for many additional features and improvements. Performance seems mainly a question of getting access to more capable hardware technologies and adapting the parameters of the architecture to the capabilities of the new environment. Still, there is certainly room for extending and further improving the efficiency and performance of several parts of the current architecture.

Regarding additional features hardware ray tracing greatly benefits from the research in software implementations of realtime ray tracing. Most of the techniques developed there can be carried over to hardware with only minor changes or adaptations. This is particularly important in the context of an API for ray tracing. We believe that the OpenRT API [DWBS03] would also work well for a hardware ray tracing engine. It is currently being ported to the prototype.

Even though object-based dynamics already coveres the majority of cases, the support for dynamically changing scenes is still too limited. More flexibility is required by many applications, most notably computer games and visual simulations.

Probably the important missing feature is programmable shading, which would finally provide the full capabilities of ray tracing to applications and users. Together with the ability to easily implement global effects this seems like the killer application for ray tracing.

Finally, it remains to be seen what will be the preferred platform for realtime ray tracing in the future. Available are high-performance general purpose CPUs, large parallel programmable processing engines such as GPUs or arrays of RISC-like CPUs, or finally custom hardware. Custom hardware seems to offer the most benefits for the core ray tracing pipeline especially since it uses its floating point resources most efficiently, while other parts of ray tracing, such as shading, seem better suited for or even require general purpose-like engines. As a consequence a combination of custom hardware and more flexible engines seems like a promising approach.

Acknowledgement

We thank both Alpha-Data and Celoxica for their support of this project through providing the FPGA card. Furthermore we like to thank Ingo Wald and Carsten Benthin for their valuable contributions in our discussions.

References

[Act02] ACTIVISION: Return to Castle Wolfenstein. *http://games.activision.com/games/wolfenstein/* (2002).

[Alp03] ALPHA-DATA: ADM-XRC-II. *http://www.alphadata.uk.co* (2003).

[Are88] ARENBERG J.: Ray/Triangle Intersection with Barycentric Coordinates. *http://www.acm.org/tog/resources/RTNews/html/-rtnews5b.html* (1988).

[Bri03] BRIGHAM YOUNG UNIVERSITY, USA: BYU JHDL. *http://www.jhdl.org* (2003).

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

[DWBS03] DIETRICH A., WALD I., BENTHIN C., SLUSALLEK P.: The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*

(Darmstadt, Germany, 2003), Eurographics Association, pp. 23–31.

[Epi03] EPIC GAMES: Unreal Tournament 2003. *http://www.unrealtournament.com/ut2003/* (2003).

[Eri97] ERICKSON J.: Pluecker coordinates. *Ray Tracing News* (1997). http://www.acm.org/tog/resources/ RTNews/html/rtnv10n3.html#art11.

[GH96] GREG HUMPHREYS C. S. A.: *TigerSHARK: A Hardware Accelerated Ray-tracing Engine*. Tech. rep., Princeton University, 1996.

[GP89] GREEN S. A., PADDON D. J.: Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications 9*, 6 (1989), 12–26.

[GP90] GREEN S. A., PADDON D. J.: A highly flexible multiprocessor solution for ray tracing. *The Visual Computer 6*, 2 (1990), 62–73.

[Gre91] GREEN S. A.: Parallel processing for computer graphics. *MIT Press* (1991), 62–73.

[Hal01] HALL D.: The AR350: Today's ray trace rendering processor. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1* (2001).

[Id-99] ID-SOFTWARE: Quake3-Arena. *http://www.quake3arena.com/* (1999).

[KH95] KEATES M. J., HUBBOLD R. J.: Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum 14*, 4 (1995), 189–202.

[KiSSO02] KOBAYASHI H., ICHI SUZUKI K., SANO K., OBA N.: Interactive Ray-Tracing on the 3DCGiRAM Architecture. In *Proceedings of ACM/IEEE MICRO-35* (2002).

[LAM00] LEXT J., ASSARSSON U., MÖLLER T.: *BART: A Benchmark for Animated Ray Tracing*. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2000. Available at http://www.ce.chalmers.se/BART/.

[LAM01] LEXT J., AKENINE-MÖLLER T.: Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations* (2001), pp. 311–318.

[LS91] LIN T. T., SLATER M.: Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer* (1991), 187–199.

[MKS98] MEISSNER M., KANUS U., STRASSER W.: VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Eurographics/Siggraph Workshop on Graphics Hardware* (1998).

[MPJ*00] MAI K., PAASKE T., JAYASENA N., HO R., DALLY W., HOROWITZ M.: Smart Memories: A Modular Recongurable Architecture. *IEEE International Symposium on Computer Architecture* (2000).

[MT97] MOELLER T., TRUMBORE B.: Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[Muu95] MUUSS M. J.: Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95* (June 1995).

[Nvi04] NVIDIA: GeForce FX. *http://www.nvidia.com* (2004).

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2002* (2002).

[Pfi01] PFISTER H.-P.:. SIGGRAPH course on Interactive Ray Tracing, 2001.

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The VolumePro real-time raycasting system. *Computer Graphics 33* (1999).

[PSL*99] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P. P.: Interactive ray tracing. In *Interactive 3D Graphics (I3D)* (April 1999), pp. 119–126.

[Pur01] PURCELL T.: The SHARP Ray Tracing Architecture. SIGGRAPH course on Interactive Ray Tracing, 2001.

[Pur04] PURCELL T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering* (Brno, Czech Republic, June 2000), pp. 299–306.

[SLS03] SCHMITTLER J., LEIDINGER A., SLUSALLEK P.: A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Volume 27, Graphics Hardware* (2003), 693–699.

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2002), pp. 27–36.

[Wal04] 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., 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).

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001 20*, 3 (2001).

[WPS*03] WALD I., PURCELL T. J., SCHMITTLER J., BENTHIN C., SLUSALLEK P.: Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports* (2003).

[Xil03] XILINX: Virtex-II. *http://www.xilinx.com* (2003).