# A Virtual Memory Architecture for Real-Time Ray Tracing Hardware

Jörg Schmittler [*], Alexander Leidinger, Philipp Slusallek

*Computer Graphics Lab, Saarland University, Im Stadtwald, Bld. 36.1, 66123 Saarbrücken, Germany*

**Abstract**

Real-time ray tracing offers a number of interesting benefits over current rasterization techniques. However, a major drawback has been that ray tracing requires access to the entire scene data base. This is particularly problematic for hardware implementations that only have a limited amount of dedicated on-board memory.

In this paper we propose a virtual memory architecture for ray tracing that efficiently renders scenes many times larger than the available on-board memory. Instead of wasting large dedicated memory on a graphics card, scene data is stored in main memory and on-board memory is used only as a cache. We show that typical scenes from computer games only require less than 8 MB of cache memory while 64 MB are sufficient even for scenes with GBs of geometry and textures. The caching approach also minimizes the bandwidth between the graphics subsystem and the host such that even a standard PCI connection is sufficient.

*Key words:* Real-time ray tracing, hardware architectures, memory management

## 1 Introduction

Over the last years the advantages of real-time ray tracing have attracted significant interest in the research community. Real-time ray tracing has first been realized with software implementations on shared-memory supercomputers [1,2] and on PC clusters [3,4]. But even the most optimized software implementations currently cannot achieve the required performance unless large numbers of CPUs are used in parallel, making hardware support a desirable goal.

[*] Correspondence author
*Email addresses:* `schmittler@cs.uni-sb.de` (Jörg Schmittler), `leidinger@graphics.cs.uni.sb.de` (Alexander Leidinger), `slusallek@cs.uni-sb.de` (Philipp Slusallek).

Last year Purcell et al. [5] presented a proof of concept that ray tracing can be implemented entirely on modern GPUs using advanced programmable shading processors. In addition Schmittler et al. [6] developed a custom hardware architecture for ray tracing. Simulations showed that this architecture is able to achieve a performance comparable to current rasterization hardware with significant savings in off-chip memory bandwidth due to on-chip caching.

While software implementations can flexibly use the main memory in a system, hardware implementations (e.g. on a separate graphics board) typically have very limited memory resources. Even worse this memory is dedicated to graphics use and cannot be used by the rest of the system for other tasks. Thus it is highly beneficial to minimize the amount of required on-board memory.

However, small on-board memories become a problem because ray tracing requires access to the entire scene data base. This requirement is mainly due to the unpredictable access to scene data by secondary rays, i.e. for reflections, transparencies, and shadows. However, it is this “feature” that allows ray tracing to compute global effects and thus provides many of its advantages. Without access to the entire scene data recent techniques such as interactive lighting simulation [7,8] would not be possible. Small memory resources would thus impose severe limits on the size of scenes that could be rendered on a given hardware.

We address the issue of limited on-board memory using a caching scheme that uses this memory only for caching scene data that is primarily stored in main memory of the host. Since ray tracing trivially splits into lots of independent tasks, memory access latencies can be hidden by using multi-threading on the ray tracing chip. Together this allows for rendering scenes in real-time that are many times larger than the available memory on-board resources.

Due to multi-threading and the use of coherent ray tracing, the caching scheme causes almost no overhead or slowdown (see Section 4) even though a cache miss is very costly as it has to transfer the required data from main memory via a slow system bus. Additionally, this approach minimizes the bandwidth across this bus such that even a standard PCI bus is sufficient for all of our test scenes.

The scheme is completely transparent to the application and even the core ray tracing hardware. It operates solely in the memory controller of the ray tracing chip where it simulates a large virtual memory for the rest of the ray tracing core. Being transparent to the ray tracer also means that the caching scheme uniformly includes all types of scene data such as geometry, shader programs, shader parameters, textures, etc.

We start the presentation with a brief review of the basic hardware architecture in which this caching scheme has been integrated and which is later used for its evaluation. We then present the proposed virtual memory architecture in Section 3 and evaluate the performance using cycle-accurate simulations of the approach on a register transfer level in Section 4. At the end we conclude and discuss future work.

## 2 The SaarCOR Hardware Architecture

Our hardware architecture (see Figure 1) consists of a custom ray tracing chip connected to several standard SDRAM chips, a separate frame-buffer, and a bridge to the system bus all placed on a single board. The bus bridge is used to transfer all scene data from the host memory under the control of the virtual memory subsystem. The SDRAM chips are used as level-2 caches and store the current working set of the scene including its geometry, the spatial index structures for fast ray traversal, material data, and textures. The image is rendered into a dedicated frame-buffer and is displayed via a VGA port.

The ray tracing implementation is based on the *SaarCOR* architecture as presented in [6]. The main focus of this initial version of the architecture has been on the implementation of the core ray tracing algorithm. For this paper we have extended the architecture to also include support for a fixed function shading unit and a common unified memory interface for all components of the chip.

The SaarCOR architecture consists of three main components: The ray-generation controller (RGC), possibly multiple ray tracing pipelines (RTP), and the memory interface (MI). Each RTP consists of a ray-generation and shading unit (RGS) and the ray tracing core (RTC). The RGS generates primary rays and hands them over to the RTC for computing the ray triangle intersections. Within the RTC, the traversal unit traverses the ray through the spatial index structure (a kd-tree in our case) until a leaf-node is reached. Leaf-nodes store lists of triangle addresses, which are then fetched by the list unit. The intersection unit then loads the data of a triangle and performs the intersection computation. Its results are sent back to the traversal unit, which either continues ray traversal through the kd-tree or sends the intersection results back to the RGS.

The RGS is responsible for shading the ray, which might generate new recursive rays. We use an extended Phong reflection model that can access two textures: a standard image texture and a bump map. In addition the shader implements shadows, reflection, and refraction effects by spawning new rays as needed. Note that we use this simple shader only for studying the memory behavior in general. We expect that a custom ray tracing implementation will have a more flexible programmable shading unit.

All memory requests by the pipelines are handled by the unified memory interface. This unit contains four different first-level caches, one for each type of functional unit. All functional units of the same type share their cache. This works well since we perform all ray tracing and shading operation on packets of rays instead of single rays. This significantly reduces the number of memory requests from any unit and allows us to scale the performance simply by increasing the number of RTPs.

In order to keep the pipelines busy all the time, memory access latencies are hidden by using multi-threading with several independent packets of rays per RTP. This works very well as even a small number of threads suffices to achieve very high utilization (see [6] for more details).

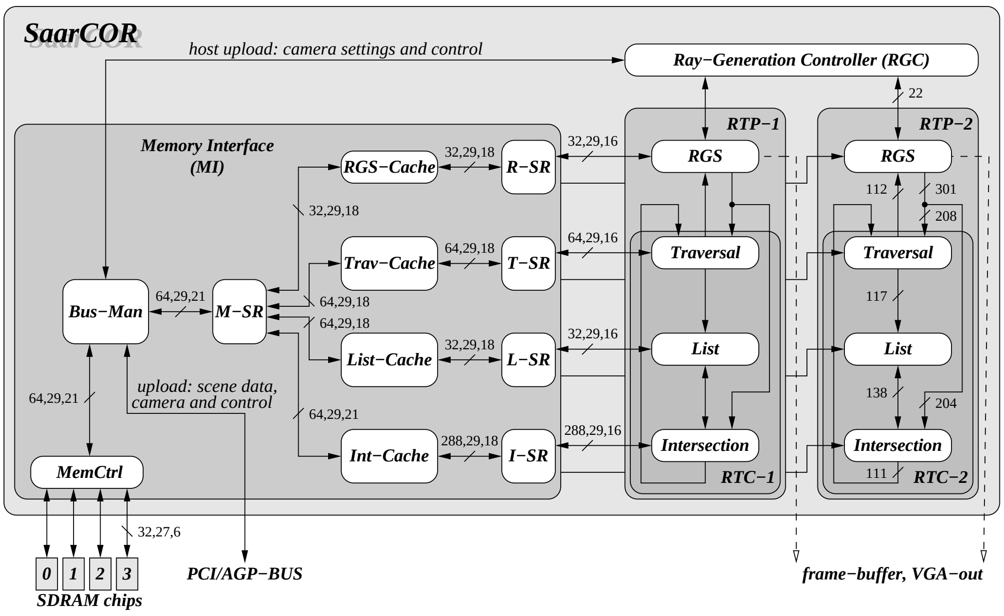

Fig. 1. The SaarCOR architecture consists of three components: The ray-generation controller, multiple ray tracing pipelines (RTP) and the memory interface. Each RTP consists of a ray-generation and shading unit (RGS) and the ray tracing core (RTC). Please note the simple routing scheme used: it contains only point-to-point connections and small busses, whose width is also shown separated into data-, address- and control-bits.

## 3 Virtual Memory Architecture

In order to minimize the amount of on-board memory we need to explicitly manage the scene data base. When using OpenGL this task resides within the application. If the size of the scene exceeds the graphics memory, the application must download only those textures and possibly display lists that are relevant to the current view. Some graphics cards offer hardware support only for virtual texture memory [9].

In contrast, a fully automatic and uniform virtual memory system for all types of data is possible with ray tracing due to its demand driven structure and its coherent access to scene data. In the following we assume that the entire scene data is stored in main memory and that the graphics subsystem can fetch this data independently of the application (DMA). Later, we briefly discuss options for application-based management of the scene data.

The virtual memory management is built into the bus managing circuit of SaarCOR (Figure 1: *Bus-Man*), allowing transparent address translation for memory requests from the ray tracer and automatic loading of missing scene data from the PC memory.

Since caching of single triangles and kd-tree nodes has proven to work very well for on-chip caches [6], we apply a similarly simple caching scheme using a standard $n$-way set-associative cache to manage the level-two cache. Using $n$-way caches reduces the probability of collisions, i.e. of different addresses mapping to the same cache entry. Another way to reduce collisions in certain access patterns is to use address hashing. Experiments with both strategies showed that in our case the combination of a 4-way set-associativity and simple address hashing works extremely well over a wide variety of benchmark scenes (see below). In standard caches, the lower bits of the memory address are used as the address into the cache memory. Our hashing function extents this scheme by simply adding the upper bits of the memory address to the lower ones, which skews the regular access pattern.

We divide the on-board memory into cache lines each consisting of $k$ bytes. A larger $k$ results in a coarse subdivision of the memory, which – depending on the memory layout used – is likely to increases the probability of collisions and thus the penalty of a cache miss due to longer bus transactions. The optimal choice of $k$ is also influenced by the amount of meta data required for cache management. The relevant meta data must be kept readily available on-chip as it is required for each memory access.

Since we use address hashing, the cache tags must store the full host address of the cache line together with several bits for managing purposes. The total size of the meta data is thus given as *5 bytes * size(on-board memory) / k*. 16 MB of cache memory with cache lines of 128 bytes already require 640 KB (3.9%) of meta data, which is reduced to 80 KB (0.4%) with cache lines of 1024 bytes each. Even though larger cache lines would significantly reduce memory requirements for meta data, our measurements indicate that a line size of 128 bytes is optimal for our architecture.

With the significant size of the meta data we also explored ways to reduce the on-chip memory requirements by storing it externally in the on-board memory. A small additional on-chip cache of just a few KB is used to hold the most recently used entries. It reduces latency and external memory bandwidth due to meta data lookup (much like a TLB in CPUs). Section 4 discusses the performance impact of both of theses architectural variants: *version A* stores all meta data in on-chip memory, while *version B* uses the TLB-like approach.

## 4 Results

In order to analyze and evaluate the performance of our virtual memory architecture, we used the same cycle-accurate simulator as in [6]. Similar to the previous paper, we define a standard SaarCOR-chip as having roughly the same hardware resources as a traditional GPUs. Thus the architectural parameters of the SaarCOR-chip are: 4 pipelines with a core frequency of 533 MHz and a 128-bit wide SDRAM memory running at 133 MHz, delivering a theoretical bandwidth of 2 GB/s. The L1-caches are 4-way set-associative and their size is 336 KB split into 64 KB for shading, 64 KB for kd-tree nodes, 64 KB for triangle addresses, and 144 KB for triangle data. 16 threads have been used per RTP. We believe that the amount of on-board caches are not problematic because even for L1-caches high latencies can be tolerated due to multi-threading and many of today's CPUs already include up to 1 MB of on-chip cache.

These simulations on the register transfer level also include the system bus for loading the graphics data. For the simulated standard 32 bit, 33 MHz PCI-bus we assume a latency of roughly 550 core clock cycles for loading a 128 byte cache line from PC memory.

We performed measurements on a wide variety of benchmark scenes. In order to minimize the simulation times we used a two step approach: In the first step we perform a complete walk-through of the benchmark scenes with a sequential ray tracer without level-one caches, but including the virtual memory architecture. The results of these measurements show for each frame the number of different cache lines addressed (the working set), the number of cache-collisions, and the resulting number of cache lines being loaded from the host (including multiple loading of the same cache line due to cache-collisions).

These graphs were used to find hot-spots in the walk-through sequences, i.e. frames where most collisions occurred or where the working set was largest (see Figure 3). In the second step we used the cycle accurate simulator to simulate in detail how the SaarCOR architecture performs at these hot-spots. Consequently, most results presented here refer to worst-case situations. Much better performance can be achieved for other parts of the walk-through.

Most of our benchmark scenes are similar to the ones used previously [6]. As shown in [6] the performance of the ray tracer scales linearly with the number of rays shot and mostly independent of the type of the ray (i.e. whether it is a primary or secondary ray). Thus to evaluate the performance of our virtual memory architecture we used scenes where the available bandwidth and memory latencies are important. Table 1 lists details of the scenes while Figure 2 provides an overview of the walk-through performance and the location of the hot-spots. Images and videos of the walk-through sequences can be found at `http://www.SaarCOR.de/VMA4RT`.

| Scene | #triangles | #lights | size on hard-disk | |
|---|---|---|---|---|
| | | | geometry | textures |
| pQuake3-nlnt | 46 356 | 0 | 16MB | — |
| Conference-nl | 282 000 | 0 | 88MB | — |
| Sodahall | 1 510 322 | 0 | 429MB | — |
| pQuake3-nt | 46 356 | 1 | 16MB | — |
| Conference | 282 000 | 2 | 88MB | — |
| Conference-8l | 282 000 | 8 | 88MB | — |
| pQuake3-nl | 46 356 | 0 | 16MB | 28MB |
| pQuake3 | 46 356 | 1 | 16MB | 28MB |
| Cruiser | 3 637 101 | 1 | 540MB | 340MB |

Table 1
The scenes and their parameters as used for benchmarking (suffix *nl*: without lighting, suffix *nt*: without textures).

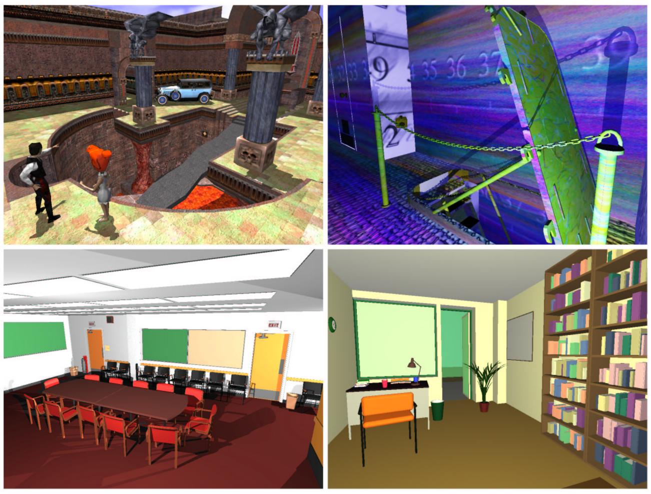

Fig. 2. Some of our benchmark scenes: pQuake3, cruiser, conference and Sodahall (from left to right and top to bottom)

Figure 3 shows graphs resulting from the first simulation step for the pQuake3 scene with 8 MB and the cruiser scene with 64 MB of on-board cache, respectively. The amount of on-board memory should be chosen such that the working set fits nicely into, otherwise performance penalties due to multiple loading of cache lines are unavoidable. As the results indicate, even as little as 8 MB are sufficient for many scenes with a maximum of 64 MB required for extremely large scenes with huge textures.

Table 2 shows the step two simulation results: the cycle-accurate simulation of the hot-spots for the variants A and B, where the latter uses a 4-way set-associative on-chip cache of 64 KB for meta data. For the simulations we

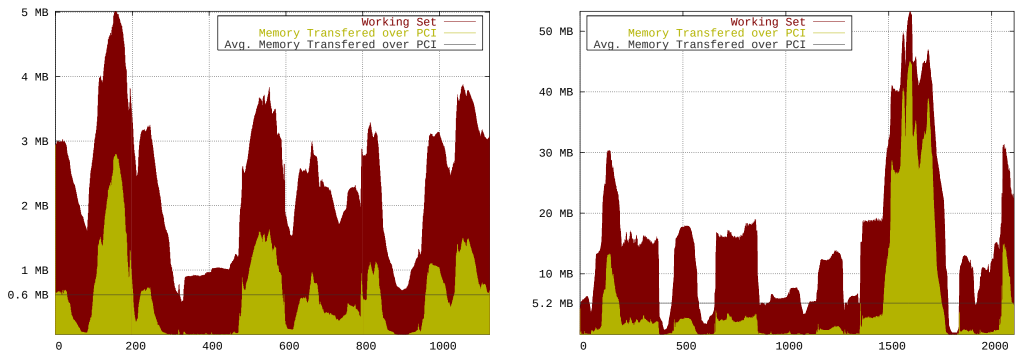

Fig. 3. Results of the first simulation step for the pQuake3 scene (left, with textures and light) using 8 MB card memory and the cruiser scene (right, with textures and light) with 64 MB. For each frame of the sequence it plots the size of the working set and the amount of memory transfered.

have assumed a standard PCI bus to transfer scene data from the host to the graphics card. For comparison the rightmost column shows the results of a simulation with unlimited local memory where the entire scene was stored on the card and no virtual memory was used. Because the SaarCOR architecture contains several independent threads running in parallel some small, non-obvious variations can be seen in the results due to scheduling issues (e.g. pQuake3-nlnt, frame 1046).

Table 2 clearly shows that for most scenes the rendering performance is hardly influenced by the addition of virtual memory. This is even true for version B of the architecture that uses only a small on-chip cache to manage the larger meta data stored in slow off-chip SDRAM.

By looking at the amount of memory transfered per frame from the host's memory, it is obvious that even a bus as slow as standard PCI does not limit the performance even for highly complex models. This was confirmed by simulations with a faster PCI bus (64 bit, 66 MHz) that provided essentially the same performance. This clearly shows that our approach of hiding latency by using several independent threads within the RTC units of our architecture works very well for hiding the latency of slow SDRAM memory as well as the even larger latency of PCI accesses in the case of level-two cache misses.

Besides the normal benchmark scenes, we also included measurements of the cruiser which is a very difficult scene, consisting of 1 GB of data due to highly localized complex geometry and huge textures and bump-maps (added manually for testing purposes). In [6] it was shown that the cruiser data set needs a larger cache for triangles in order to avoid a drastic performance drop. We therefore increased only the triangles cache to 576 KB.

The results shown in Table 2 were taken for the worst-case hot-spots of Figure 3. It showed that the performance of the cruiser scene is limited by the

| Cache variant: | Version A | | | Version B | | | no VM | |
|---|---|---|---|---|---|---|---|---|
| Scene, Frame | fps | PCI | Mem | fps | PCI | Mem | fps | Mem |
| pQuake3-nlnt, 150 | 131.2 | 0.04 | 0.9 | 130.8 | 0.04 | 1.1 | 131.5 | 0.8 |
| pQuake3-nlnt, 556 | 170.0 | 0.02 | 0.4 | 170.3 | 0.02 | 0.4 | 171.3 | 0.3 |
| pQuake3-nlnt, 1046 | 130.5 | 0.01 | 0.2 | 130.4 | 0.01 | 0.2 | 130.3 | 0.2 |
| Conference-nl, a | 90.6 | 0.27 | 7.0 | 84.8 | 0.30 | 9.4 | 94.7 | 6.7 |
| Conference-nl, b | 82.4 | 0.41 | 7.5 | 76.9 | 0.43 | 10.0 | 86.6 | 7.0 |
| Sodahall, a | 116.7 | 0.03 | 1.0 | 116.4 | 0.03 | 1.3 | 117.2 | 1.0 |
| Sodahall, b | 183.3 | 0.01 | 0.4 | 182.9 | 0.01 | 0.5 | 183.8 | 0.4 |
| Sodahall, c | 129.1 | 0.01 | 0.5 | 128.6 | 0.01 | 0.6 | 129.4 | 0.5 |
| pQuake3-nt, 150 | 81.3 | 0.04 | 1.3 | 81.2 | 0.04 | 1.6 | 81.7 | 1.3 |
| pQuake3-nt, 556 | 90.3 | 0.02 | 0.5 | 90.3 | 0.02 | 0.5 | 90.5 | 0.4 |
| pQuake3-nt, 1046 | 80.5 | 0.01 | 0.2 | 80.4 | 0.01 | 0.3 | 80.5 | 0.2 |
| Conference, a | 31.6 | 0.78 | 19.1 | 29.7 | 0.91 | 25.2 | 33.0 | 18.1 |
| Conference, b | 26.9 | 1.24 | 25.3 | 24.9 | 1.29 | 33.8 | 28.4 | 23.7 |
| Conference-8l, a | 11.1 | 1.15 | 30.7 | 10.9 | 1.15 | 40.9 | 11.4 | 29.3 |
| Conference-8l, b | 11.6 | 1.51 | 31.4 | 11.3 | 1.72 | 41.8 | 11.9 | 29.5 |
| pQuake3-nl, 150 | 126.3 | 0.48 | 7.0 | 126.6 | 0.43 | 7.9 | 130.8 | 6.8 |
| pQuake3-nl, 556 | 170.4 | 0.10 | 2.5 | 169.9 | 0.18 | 2.9 | 170.9 | 2.4 |
| pQuake3-nl, 1046 | 130.1 | 0.20 | 3.1 | 129.8 | 0.13 | 3.4 | 130.3 | 2.9 |
| pQuake3, 150 | 80.1 | 0.53 | 6.7 | 80.2 | 0.44 | 7.4 | 80.7 | 6.1 |
| pQuake3, 556 | 89.7 | 0.10 | 2.6 | 89.7 | 0.21 | 3.1 | 89.8 | 2.5 |
| pQuake3, 1046 | 79.0 | 0.20 | 3.1 | 79.1 | 0.13 | 3.5 | 79.4 | 2.9 |
| Cruiser, 142 | 24.9 | 4.05 | 43.5 | 17.3 | 4.11 | 52.7 | 37.1 | 37.4 |
| Cruiser, 500 | 43.7 | 0.99 | 21.4 | 36.9 | 0.97 | 25.7 | 53.5 | 19.7 |
| Cruiser, 1625 | 2.7 | 3.09 | 350.0 | 2.8 | 3.86 | 404.6 | 4.1 | 337.0 |
| Cruiser, 2080 | 17.5 | 1.72 | 52.5 | 14.8 | 1.80 | 61.7 | 23.4 | 50.1 |

Table 2
Simulation results of the largest hot-spots in each benchmark. The achievable frame-rate as well as the amount of memory transfered over the PCI-bus and between SaarCOR and the on-board memory are listed. All memory transfers are measured in MB per frame. All measurements are performed with a standard SaarCOR-chip, except for the cruiser scene. The on-board memory for cache variants A and B was 8 MB for all pQuake3 scenes and the Sodahall. For the conference 32 MB and for the cruiser scene 64 MB were used.

bandwidth to local memory (the L2-cache) due to the large working sets for triangles. This also holds for the hot-spot around frame 1625 where the hit-rate of the triangle-cache goes down to 13%. This hot-spot shows very drastically the difference between simulations with and without L1-caches and proves that a combination of two cache levels – small L1-caches with small cache-lines and a large L2-cache with larger cache-lines – is very well suited to minimize external bandwidth. The issue of reducing the large working set for triangles can be addressed by using an improved algorithm to build the kd-tree, which already works very well for our software ray tracer, but has not yet been implemented for the hardware simulations.

## 5 Conclusion and Future Work

In this paper we have demonstrated that a fully automatic mechanism for scene-management can be implemented for ray tracing-based graphics cards. The virtual memory architecture is efficient and simple to implement while it allows to render scenes that are many times larger than the available on-board memory. These results invalidate the common assumption that ray tracing is impractical because it would need to store the entire scene in local memory.

The architecture is fully transparent to the core ray tracer and the application giving the illusion of a large and fast local memory storing the entire scene data. However, only a small local memory is required as a cache for the scene data that is stored in general purpose main memory. Any savings in on-board memory can thus be more efficiently used to enlarge main memory.

Our cycle-accurate simulations show that the addition of virtual memory management hardly influences the performance of the ray tracer. This demonstrates nicely that simple multi-threading is perfectly suited to hide even large memory access latencies. The small on-chip caches together with the level-two caches in local SDRAM reduce the on-board memory requirements to the size of the per-frame working set. The bandwidth requirements for loading scene data from the host's memory during rendering are very low, such that even a standard PCI-bus is sufficient for all of our benchmark scenes.

Our virtual memory architecture fully removes the necessity of applications to explicitly manage the on-board memory of the graphics subsystem, which has become a major issue with current rasterization technology [10]. Our approach provides essentially the same positive effects for ray tracing graphics technologies as the ubiquitous virtual memory management in operating systems today. It is unclear how these results could be applied in the context of rasterization since the rendering system usually has insufficient information about the entire scene.

Although our approach significantly reduces the amount of memory on the graphics board, the scene still has to be available somewhere in host's memory. However, the demand driven approach of ray tracing also allows some simple solutions to this problem: We can mark some "proxy" geometry (e.g. LOD nodes) and send asynchronous notifications to the application containing information about their respective ray hit rates. The application can then react appropriately and replace the proxy geometry by other data.

Future work will also include the support for dynamic scenes [11] and fully programmable shading. Programmable shading would also be the prerequisite for implementing full real-time lighting simulation [7] for a single chip ray tracing solution.

## References

[1] M. J. Muuss, Towards real-time ray-tracing of combinatorial solid geometric models, in: Proceedings of BRL-CAD Symposium '95, 1995.

[2] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P. P. Sloan, Interactive ray tracing, in: Interactive 3D Graphics (I3D), 1999, pp. 119–126.

[3] I. Wald, C. Benthin, M. Wagner, P. Slusallek, Interactive Rendering with Coherent Ray Tracing, Computer Graphics Forum (Proceedings of Eurographics 2001 20 (3).

[4] I. Wald, P. Slusallek, C. Benthin, Interactive Distributed Ray Tracing of Highly Complex Models, in: Proceedings of the 12th Eurographics Workshop on Rendering, 2001, london.

[5] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, Ray Tracing on Programmable Graphics Hardware, in: Proceedings of SIGGRAPH 2002, 2002.

[6] J. Schmittler, I. Wald, P. Slusallek, SaarCOR – A Hardware Architecture for Ray Tracing, in: Proceedings of Eurographics Workshop on Graphics Hardware, 2002, pp. 27–36.

[7] I. Wald, T. Kollig, C. Benthin, A. Keller, P. Slusallek, Interactive Global Illumination using Fast Ray Tracing, Rendering Techniques 2002 (2002) 15–24(Proceedings of the 13th Eurographics Workshop on Rendering).

[8] C. Benthin, I. Wald, P. Slusallek, A Scalable Approach to Interactive Global Illumination, to be published at Eurographics 2003 (2003).

[9] 3Dlabs, Virtual Textures – a true demand-paged texture memory management system in silicon (1999).
URL `http://www.merl.com/hwws99/hot3d.html`

[10] 3Dlabs , OpenGL 2.0 - Minimizing Data Movement and Memory Management in OpenGL (2002).
URL `http://www.3dlabs.com/support/developer/ogl2/whitepapers`

[11] I. Wald, C. Benthin, P. Slusallek, A Simple and Practical Method for Interactive Ray Tracing in Dynamic Scenes, Submitted for publication, also available as a technical report at http://graphics.cs.uni-sb.de/Publications (2002).