SaarCOR

A Hardware-Architecture for Realtime Ray Tracing

Jörg Schmittler
Computer Graphics Group
Saarland University
Saarbrücken, Germany

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (*Dr.-Ing.*)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Betreuender Hochschullehrer / Supervisor:

Prof. Dr.-Ing. Philipp Slusallek
Universität des Saarlandes
Saarbrücken, Germany

Gutachter / Reviewers:

Prof. Dr.-Ing. Philipp Slusallek
Universität des Saarlandes
Saarbrücken, Germany

Prof. Dr. Wolfgang J. Paul
Universität des Saarlandes
Saarbrücken, Germany

Prof. Dr. Wolfgang Straßer
Universität Tübingen
Tübingen, Germany

Dekan / Dean:

Prof. Dr.-Ing. Thorsten Herfet
Universität des Saarlandes
Saarbrücken, Germany

Eingereicht am / Thesis submitted:

30. Januar 2006 / January 30th, 2006

Datum des Kolloquiums / Date of defense:

6. Dezember 2006 / December 6th, 2006

Jörg Schmittler
Computer Graphics Group
Saarland University, B36.1
66123 Saarbrücken, Germany
Schmittler@SaarCOR.de

Kurzfassung

Seit vielen Jahrzehnten ist Strahlverfolgung (engl. Ray-Tracing) eine bekannte und viel genutzte Technik, um hochrealistische Bilder zu erzeugen. Dieses Verfahren simuliert den physikalische Transport von Licht, wodurch auch hochkomplexe optische Zusammenhänge und Beleuchtungssituationen korrekt dargestellt werden können.

Verwendet wird das Verfahren deshalb beispielsweise in der Werbung und Filmindustrie, in der Architektur sowie bei der industriellen Prototypenentwicklung. Gerade für letztere Anwendung ist es von besonderer Bedeutung, dass Ray-Tracing auch mit hochkomplexen Modellen sehr gut arbeiten kann.

Um diese hohe Bildqualität zu erreichen benötigt das Verfahren relativ komplexe Berechnungen mit einem nahezu unstrukturierten Speicherzugriffsverhalten. Aus diesem Grund dauert die Berechnung eines Bildes auf Standard-Prozessoren in der Regel zwischen mehreren Minuten für einfache Szenen bis zu vielen Stunden für komplexe Simulationen. Die Verwendung von modernen, hochgetakteten CPUs und großen Multi-Prozessor-Maschinen hat hierbei das Problem nicht grundlegend lösen können und wird auch in den nächsten Jahren keine Echtzeitanwendungen ermöglichen.

Die gleichen Gründe – komplexe Berechnungen und unstrukturierte Speicherzugriffe – haben auch dazu geführt, dass bisherige Versuche, Ray-Tracing mit Spezial-Hardware zu beschleunigen, für allgemeine Anwendungen nicht zu der nötigen Leistung geführt haben.

Diese Arbeit stellt einen umstrukturierten Ray-Tracing Algorithmus vor, der das Problem der unstrukturierten Speicherzugriffe löst und entwickelt detailiert eine komplette Hardware-Architektur für Echtzeit-Ray-Tracing. Diese Architektur wird in genauen Simulationen untersucht und eine Prototypenimplementierung zeigt weltweit erstmals Ray-Tracing von komplexen Szenen und optischen Effekten in Echtzeit auf nur einem einzelnen Chip.

Abstract

For many decades, ray tracing is known and well used for rendering highly realistic images. Since ray tracing simulates the physical transport of light even highly complex optical properties and illumination conditions can be rendered correctly.

Due to these features ray tracing is used e.g. for commercials and movies, in architecture, and for visualizations of industrial prototypes. Especially for latter application it is of great advantage that ray tracing can handle highly complex models very well.

However, achieving this high standard in image quality requires relatively complex calculations and a rather unstructured memory access behavior. For these reasons rendering an image of a simple scene already takes several minutes on standard processors, while complex simulations can run for many hours. Using high-end processors and multiprocessor machines does not solve the issue of rendering time in general and therefore will not be an option for realtime applications in the next years.

Due to the same reasons – complex calculations and unstructured memory access patterns – previous attempts to built special hardware to accelerate full featured ray tracing for general applications did not provide the necessary processing power.

This thesis presents how the ray tracing algorithm can be restructured to allow for structured memory accesses. Using these modifications a complete hardware architecture for realtime ray tracing is developed and verified using cycle-accurate simulations. Finally, these new techniques allowed for the world's first prototype implementation of full featured ray tracing of complex environments on a single chip.

Zusammenfassung

Seit vielen Jahrzehnten ist Strahlverfolgung (engl. Ray-Tracing) eine bekannte und viel genutzte Technik, um hochrealistische Bilder zu erzeugen. Dieses Verfahren simuliert den physikalische Transport von Licht, wodurch auch hochkomplexe optische Zusammenhänge und Beleuchtungssituationen korrekt dargestellt werden können.

Um diese hohe Bildqualität zu erreichen benötigt das Verfahren relativ komplexe Berechnungen mit einem nahezu unstrukturierten Speicherzugriffsverhalten. Aus diesem Grund dauert die Berechnung eines Bilders auf Standard-Prozessoren in der Regel zwischen mehreren Minuten für einfache Szenen bis zu vielen Stunden für komplexe Simulationen. Die Verwendung von schnelleren CPUs und großen Multi-Prozessor-Maschinen kann das Problem nicht grundlegend lösen und wird deshalb auch in den nächsten Jahren keine Echtzeitanwendungen ermöglichen.

Aus diesen Gründen – komplexe Berechnungen und unstrukturierte Speicherzugriffe – ist es bislang noch nicht gelungen Ray-Tracing mit Spezial-Hardware für allgemeine Anwendungen derart zu beschleunigen, so dass damit Echtzeitanwendungen ermöglicht würden.

Diese Arbeit stellt einen umstrukturierten Ray-Tracing Algorithmus vor, der das Problem der unstrukturierten Speicherzugriffe löst. Der Kern dieser Umstrukturierung ist die Verwendung von Paketen von Strahlen anstelle von einzelnen Strahlen. Hierbei werden solche Strahlen zu einem Paket zusammengefasst, von denen anzunehmen ist, dass sie die gleichen Bereiche in der virtuellen Welt durchqueren und damit auch zu großen Teilen die gleichen Daten benötigen. Mit dieser Zusammenfassung zu Strahlenpaketen lässt sich deshalb die Anzahl der Speicherzugriffe drastisch verringern.

Darüber hinaus ermöglicht dieses Zusammenfassen, dass lange Wartezeiten auf Speicheranfragen überbrückt werden, so dass die zur Verfügung stehenden Funktionseinheiten sehr gut genutzt werden können. Zusammen mit einer Multi-Threading Implementierung erlaubt der Einsatz von Strahlpaketen so die effiziente Verwendung von modernen Prozessortechniken mit hohen Frequenzen und vielen Berechnungsstufen.

Die Umstrukturierung des Strahlverfolgungsalgorithmus wird im Detail sehr Hardwarenah beschrieben und die sich ergebenden Auswirkungen auf Berechnungen und Speicherzugriffe untersucht. Der umstrukturierte Algorithmus wird dann in mehrere Funktionsblöcke gegliedert und eine Struktur für eine Hardware-Architektur erarbeitet. Hierbei ergeben sich vielfältige Möglichkeiten zur Optimierung, die im Einzelnen diskutiert werden.

Die vorgestellte Hardware-Architektur gliedert sich im wesentlichen in drei Bereiche: Die Strahlverfolgung selbst, die Berechnung der Farbe die auf den Bildschirm gezeichnet wird und die Außenverbindungen zum Speicher, zum Wirtsrechner und zum Monitor. Die Strahlverfolgung enthält neben verschiedenen Optimierungsmöglichkeiten auch die beiden Teilgebiete statische und veränderliche Welten, die mit wenigen Einschränkungen vom gleichen System behandelt werden können.

Das Berechnen der Farben, die auf dem Bildschirm angezeigt werden sollen, ist ein komplexes Forschungsgebiet und daher nicht Gegenstand dieser Arbeit. In dieser Arbeit wird vielmehr ein System entwickelt, mit dem beliebige Berechnungen effizient unterstützt und an den Prozess der Strahlverfolgung angekoppelt werden können. Im Rahmen dieser Unterstützung wird ein minimalistischer Prozessor entwickelt, der exemplarisch verschiedene Techniken vereint und als Beispiel für einen Ray-Tracing-Prozessor dienen kann.

Der Aufbau der Außenverbindungen eines Chips ist immer ein kritischer Punkt, der die Leistungsfähigkeit des Gesamtsystems stark beeinflusst. Erfreulicherweise kann gezeigt werden, dass die Verwendung des umstrukturierten Strahlverfolgungsprozesses die Anforderungen an die Außenverbindung zum Speicher stark reduziert.

Ein weiteres Problem bisheriger Hardware-Lösungen des Ray-Tracing-Verfahrens war, dass die darzustellende Welt vollständig in den zur Verfügung stehenden Speicher passen musste. Diese Arbeit stellt ein Konzept vor, dass den lokalen Speicher nur als Zwischenspeicher benutzt und fehlende Daten bei Bedarf vom Wirtsrechner nachläd. In genauen Simulationen wird gezeigt, dass selbst ein relativ langsamer Standard-PCI-Bus in der Lage ist, die für Echtzeitanwendungen benötigten Daten zu übertragen.

Die in dieser Arbeit vorgestellte Gesamtarchitektur für Echtzeitstrahlverfolgung wurde darauf ausgerichtet, durch Parallelisierung eine sehr hohe Möglichkeit zur Leistungssteigerung zu ermöglichen. Dabei wurde darauf geachtet, dass sich konzeptbedingt kaum Engpässe ergeben. In genauen Untersuchungen wird gezeigt, dass eine sehr gute Leistungssteigerung tatsächlich möglich ist.

Im Rahmen dieser Arbeit werden viele detaillierte Untersuchen angestellt. In vielen Fällen wird dabei analysiert, wie sich eine Designentscheidung auf das Gesamtsystem auswirkt. Bisher verfügbare Simulationssysteme setzen allerdings voraus, dass die zu untersuchenden Schaltungen auf Gatterniveau implementiert sind und benötigen meist mehrere Tage für die Simulation eines vollständigen Chips.

Um dennoch die Konsequenzen von Designentscheidungen schnell und effizient untersuchen zu können, wurde ein neues Simulatorkonzept entwickelt, das Zyklen-exakte Ergebnisse von vollständigen Chips innerhalb weniger Stunden berechnet. Dieser Simulator ermöglicht dabei Simulationen des Zeitverhaltens von Schaltungen, ohne dass diese auf Gatterniveau implementiert werden müssen und wird ebenfalls in dieser Arbeit vorgestellt.

Diese Simulationsergebnisse über die Architektur zur Strahlverfolgung werden dann genutzt, um den weltweit ersten Prototypen zur Strahlverfolgung in Echtzeit zu entwickeln. Dieser Prototyp basiert auf FPGA-Technologie von 2003 und erlaubt es bereits ein Vielfaches der Leistung eines 30-fach schneller getakteten Standard-Prozessors zu erreichen. Diese Arbeit enthält eine genaue Beschreibung und Analyse dieses Prototypen.

Zusammenfassend zeigt diese Arbeit, dass mit relativ einfachen Techniken, wie einer statischen Lastbalancierung, einfachen Verbindungstechniken, geringer Speicherbandbreite, Standard-Speichertechnologie und kleinen Speicher-Caches, bereits ein sehr leistungsfähiges System für Echtzeitstrahlverfolgung möglich ist. Dieser Ansatz ist deshalb sehr vielversprechend, da er viele Möglichkeiten zur Leistungssteigerung offen lässt.

An vielen Stellen konnten Techniken, die sich in Software bereits bewährt hatten, direkt in Hardware übernommen werden. Mittlerweile sind sehr viele interessante neue Erweiterungen und Verbesserungen veröffentlicht worden, die in weiteren Arbeiten direkt an die in dieser Arbeit vorgestellten Konzepte anknüpfen können. Einige Beispiele für derartige Erweiterungen werden an geeigneter Stelle aufgezeigt.

Danksagung

Im Nachfolgenden möchte ich mich bei Personen bedanken, die mir auf vielfältige Art und Weise geholfen haben und so diese Arbeit ermöglichten. Mein herzlichster Dank gilt:

- Meiner Frau Anja, die mir mit viel Liebe und großem Verständnis den Rückhalt für meine Arbeit gegeben hat,
- meinem Vater, dem Ingenieur, von dem ich viel über Technik und praktisches Arbeiten gelernt habe,
- meiner Mutter, die mir erlaubt hat, meinen eigenen Rhythmus zu leben,
- meinen Schwiegereltern, die mir ihr Ferienhaus als Exil zum Aufschreiben dieser Arbeit zur Verfügung gestellt haben,
- Philipp Slusallek für die kompetente Betreuung meiner Arbeit, für gute Ideen und Anregungen,
- Wolfgang Paul für die erstklassige Ausbildung in Rechnerarchitektur und Hardwaredesign,
- Ingo Wald für das umfangreiche und geduldige Erklären von Ray-Tracing Techniken, wodurch diese Arbeit erst ermöglicht wurde,
- Sven Woop, Daniel Wagner, Patrick Dreker und Alexander Leidinger für die erfolgreiche Zusammenarbeit beim SaarCOR-Projekt,
- Timothy Purcell für die produktiven Gespräche über Ray-Tracing Hardware,
- Tim Dahmen, Daniel Pohl und Raoul Plettke für die interessanten Diskussionen über den Einsatz von Ray-Tracing in Computerspielen,
- Andreas Dietrich für die gemeinsame Arbeit am Sample-Cache-Projekt,
- Carsten Benthin für die interessanten Gespräche über Ray-Tracing Algorithmen,
- Peter Bach, Cédric Lichtenau, Michael Bosch und Michael Braun für die gute Zusammenarbeit und praktische Ausbildung in mehreren Hardware Projekten,
- Georg Demme, Rainer Jochum und Maik Schmitt für den Aufbau und die Pflege der technischen Infrastruktur des Lehrstuhls, die ein vernünftiges Arbeiten ermöglicht hat,
- allen Mitarbeitern und Studenten des Lehrstuhls Slusallek für das angenehme Arbeitsklima.

When Leo Fender first invented an electric guitar one could have said: "But to what extend is this real music?" To which the answer is: "All right, we're not going to play Beethoven on it, but at least let's see what we can do."

Douglas Adams [Gai02]

Contents

1 Introduction

Albrecht Dürer (1471–1528)
Inventor of Ray Tracing [Hof92]

Two-dimensional images of three-dimensional real and virtual worlds are at least as old as the first cave-paintings. But during the European Renaissance for the first time a photo-realistic level was achieved. Albrecht Dürer (1471–1528, a formative artist of that epoch) developed a technique that subdivides the image using a lattice and helps to paint three-dimensional objects exactly where they project on to a two-dimensional painting. In other variants of that method he used strings to project points on real objects directly onto a canvas (see Figure 1.1). These revolutionary techniques allowed him to achieve astonishing results and his work on perspectively correct drawings made him famous.

Figure 1.1: Albrecht Dürer developed a technique (see [Alb25]) using a lattice of threads that divides a frame into squares and a finder through which the scene to be painted is looked at (see left image). Additionally, a string was used to project points on the three-dimensional object onto the two-dimensional canvas (the image on the right). Using these techniques allowed Dürer to create paintings with astonishing visual realism and accuracy.

These basic ideas developed by Albrecht Dürer and improved by other artists in the following centuries have been adapted to computers by Arthur Appel in 1968 [App68]. In Appel's variant the lattice of threads is replaced by an image plane such that every pixel of the image represents a square of the lattice. Then for each pixel a ray originating at a virtual camera is intersected analytically with the objects of the computer representation of a three-dimensional world to check which object projects onto the corresponding pixel.

During the last decades Appel's technique was improved and extended constantly [Gla89] and the method of casting rays was applied recursively to also evaluate the lighting conditions and material properties like reflections and refractions on the objects projected

to a pixel. This recursive technique is well known under the name *ray tracing* and allows for achieving photo-realistic and physical correct images which can be found almost everywhere: In the newspaper, on packages of products, in commercials, in movies, on cellular phones, and in computer games.

But although the basic idea of ray tracing is very simple it has two decisive drawbacks since it requires complex calculations and has a rather unstructured memory access pattern. Due to these requirements ray tracing could not be used for interactive applications and therefore was restricted to high quality offline rendering.

Thus for interactive applications a different technique was required and found in *rasterization* [AMH02]. In contrast to ray tracing where for every pixel it must be checked which object projects onto that pixel, the rasterization method sequentially projects all objects of a scene onto the image plane and checks which pixels are covered by each object. Additionally, for each pixel the distance to the closest object is kept such that only objects closer to the camera overwrite pixels already covered by a different object (see Figure 1.2).

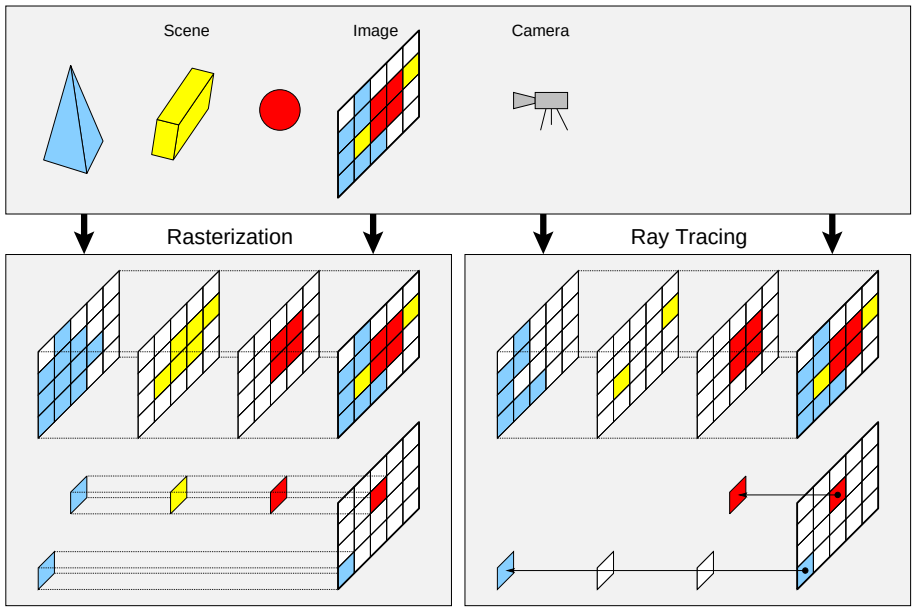


Figure 1.2: Comparison of the basic algorithms of rasterization and ray tracing. For the sake of simplicity in this example the camera displays the scene using an orthographic projection. The left part illustrates rasterization of every object in the scene which requires multiple writes and reads on the framebuffer to store, read back and modify the color and the distance to the triangle seen through a pixel. The right hand side illustrates rays being sent through the scene for each pixel of the image. Using a spatial index structure allows for terminating rays as soon as they hit an object and objects that are not pierced by a ray are not considered at all. The framebuffer is only written once and there is no need for a read back.

Comparing the requirements of rasterization and ray tracing shows that for small scenes containing only few objects with simple materials and lighting situations the hardware requirements of rasterization are much lower than for ray tracing. This is the reason why

the first graphics boards built more than twenty years ago used rasterization techniques. Since then chip technology has improved very much but these new technologies were only used to speed up rasterization graphics because still ray tracing is considered “too expensive”.

This thesis will show that realtime graphics is possible with ray tracing at reasonable costs using current technologies the next section discusses some of the fundamental properties of both techniques and motivates why it is beneficial to invest further research on ray tracing.

1.1 Why Ray Tracing?

There are many reasons to prefer ray tracing over rasterization although in general both calculate the visibilty of objects in a virtual world. But while rasterization is linear in the number of objects[1] and amortizes over the number of pixels rendered, ray tracing is logarithmically in scene complexity (see Chapter 2) but linear in the number of pixels rendered[2].

While this gives a bad performance for ray tracing compared to rasterization when rendering small worlds with simple shading, things drastically change with complex worlds and advanced shading effects applied on a per pixel basis.

The limitations of rasterization and the advantages of ray tracing become most obvious when comparing the layers of any application using either rendering system (Figure 1.3). In both systems the application controls the scene-graph to manage the objects in the scene. Since current rasterization based graphics supports only triangles as geometric primitives we apply the same restriction to scenes used for ray tracing[3].

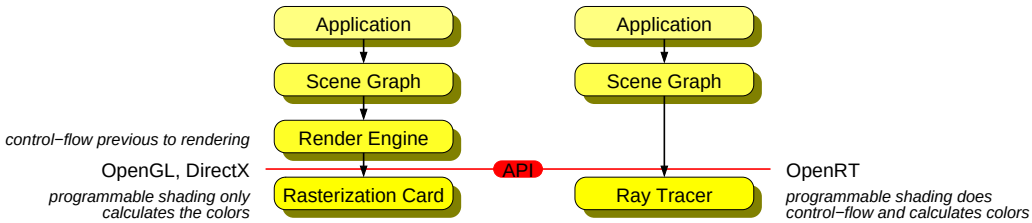


Figure 1.3: Structure of rasterization and ray tracing applications.

Using rasterization a render-engine manages the control-flow and it decides which parts of the scene might be visible to minimize the workload on the graphics board. Then it sends all potentially visible triangles to the graphics board using either the OpenGL– or DirectX–API. The graphics board computes the image on-the-fly while triangles are

[1] This linearity actually occurs on two positions of the cost measurement: First every object has to be projected onto the image plane and then for every pixel covered the current distance has to be checked and overwritten if closer than any previously projected object. Therefore depending on the size of the object on the image plane either task might become the limit for the rendering speed. However, there are techniques to reduce the number of triangles that need to be rasterized but nevertheless this linear term still applies to all potentially visible triangles.

[2] Actually ray tracing is linear in the number of rays rendered but in general every pixel requires at least one ray.

[3] Nevertheless Section 4.5 removes this restriction again.

being sent – and this is also the main disadvantage of rasterization since it restricts the graphics system to a local view on the global scene data:

1. To be able to render an image on-the-fly every information which might be required for this computation has to be generated prior to the on-the-fly rendering (e.g. illumination, reflection– and light-maps).
2. All elements of a scene are processed independently which does not allow for interaction between objects (e.g. casting shadows from one triangle onto another).
3. Since the image is computed on-the-fly it is finished with the last triangle being send, but unfortunately only at this time it becomes clear which triangles should have been sent or which information should have been generated in the first place.

Thus the fundamental problem of rasterization is that the render-engine decides which parts of the scene should be drawn without knowing what is visible since visibility is calculated by the graphics board using only the information given by the render-engine. Nevertheless current software technologies allow for solving primary visibility that checks which objects can be seen from the camera sufficiently enough[4]. But unfortunately this problem becomes worse with every further level of visibility calculation like those required for lighting and multiple reflection and refractions. Especially in complex scenes and non-planar surfaces it is non-trivial and often requires manual tweaking by skilled artists to approximate those secondary effects convincingly.

In contrast to rasterization the concept of ray tracing is fundamentally different: Here the application only specifies the world including all objects and the corresponding material properties, lighting conditions, and the settings of the camera. Then the graphics system decides which rays are required to calculate the primary visibility and only the data required to trace those rays is accessed. This property holds also for multiple reflections, refractions, and even indirect lighting effects as in global illumination since rays for secondary (and further) effects are generated recursively on demand. Thus the entire rendering process is independent of anything above the API-layer and highly efficient as only things which are required to correctly render the image are calculated. In Chapter 6 we will see that a ray tracer even manages its memory automatically, while in general memory management with rasterization is still a largely unsolved problem.

But the crucial point is not that either rendering system is not capable of doing some specific effects. It is only about how efficient things are handled by the rendering system and how complicated it is to achieve certain effects. With rasterization a lot of effort has to be put into programming the render-engine and by artists to approximate effects[5] . In contrast, with ray tracing all effects are correctly handled automatically and highly

[4] Recent advances in rasterization technology allow for *occlusion queries* (see Chapter 6.2.2) which enable the application to check the contribution of an object to the image. Nevertheless this technique greatly simplifies and improves finding the set of potentially visible triangles in general it does not change the control-flow of rasterization based graphics. However, it allows for rendering rather large scenes by rendering the bounding boxes of a spatial index structure (e.g. the bounding boxes of the nodes of a kd-tree) and using these results to determine whether to render the content of the box. But again this technique requires some extra work by the application and introduces other problems (see [BWPP04, SBS04]).

[5] For example with rasterization rendering shadows already becomes rather complex: Although pixel-accurate shadows can be realized by *shadow volumes* [Cro77, AMH02] these do not allow for colored shadows and light through semi-transparently textured objects. Colored shadows can be realized with *shadow maps* [Wil78] but only *perspective shadow maps* [SD02] and *trapezoidal shadow maps* [MT04]

efficient by default but at a higher initial cost. Thus, if we could build ray tracing based graphics boards fast enough for realtime applications and at a reasonable price this should have a major impact on interactive computer graphics – and this is what this thesis is about.

1.2 Previous Work

The focus of this thesis is on ray tracing and how to build efficient hardware support for it. Therefore this is also the main focus of the previous work presented in this section. Nevertheless for further readings and comparisons the next section gives references to rasterization based graphics and corresponding hardware architectures.

The remainder of this section will focus on ray tracing and is split into two parts: first the research that focuses on making ray tracing interactive by using parallel computers, hardware accelerators and by improving the underlying algorithms are summarized. The second part presents the complete framework for realtime ray tracing developed at the Saarland University of which this thesis is one part. A more general and detailed overview of the state-of-the-art in interactive ray tracing can be found in [WPS+03, Wal04].

1.2.1 Rasterization Based Graphics

The basic algorithm of rasterization and its many improvements are contained in almost any current textbook on computer graphics, e.g. [AMH02, Shi02, Wat00, ESK97]. Additional information and documentation on the OpenGL–API can be found at [`www.opengl.org`], while the most important manufacturers also provide many technical documentation and concepts how to implement various effects and applications [`www.nvidia.com`, `www.ati.com`, `www.sgi.com`].

Much work has been done on designing and implementing rasterization using dedicated hardware. Since any comprehensive summary is far beyond the scope of this thesis only the most interesting publications and architectures are listed.

One key paper in the field of computer graphics in the seventies has been [SSS74] which deals with the characterization of ten hidden-surface algorithms. It is interesting to note, that the eleventh algorithm listed only in the appendix of this paper and described as "ridiculously expensive" is the basis for all rasterization based graphics available today [AMH02].

[MCEF94, Eld01] present classifications on how parallelization of rasterization based graphics can be done. Parallel rendering architectures are RealityEngine Graphics [Ake93], SGI's InfiniteReality [Bur96, Kil97, MBDM97], Pixel-Planes from UNC [FPE+89], PixelFlow [MEP92, EMP+97], and the scalable "Pomegranate" [EIH01] architecture.

deliver pixel-accurate shadows. Furthermore since rasterization does not handle transparencies correctly by default, *depth peeling* [Eve01] has to be used. This sums up to a rather complex system which can deliver pixel-accurate colored shadows and light but is rather complex to handle especially when compared to simply generating rays and having a ray tracing system handling all interactions automatically.

1.2.2 Making Ray Tracing Interactive

Due to the concept of rasterization it is missing a 3D spatial index to quickly locate the relevant triangles. Instead, the application must provide the missing functionality in software (i.e. frustum and occlusion culling). This splits the rendering process, adds overhead and complexity while eliminating the options for complete hardware acceleration.

In contrast, ray tracing is fundamentally based on a 3D spatial index structure in object space. The traversal operation through this spatial index is a conservatively approximated enumerator for the set of triangles hit by the ray in front to back order. The index imposes no limits on the allowable set of rays and can answer even single ray queries efficiently. In most cases the spatial indices are *hierarchical* in order to better adapt to the often uneven distribution of triangles in space. Efficient hardware support for ray queries in hierarchical indices is a prerequisite for accelerated ray tracing. It would allow for a fully declarative scene description by integrating the entire rendering process into hardware including any global effects.

One drawback of spatial indices in general are dynamic changes to the scene, as this would require partial or full re-computation of the index. This, however, applies to any rendering algorithm that uses a spatial index, including advanced rasterization. Little research on spatial indices for dynamic scenes has been performed in the past [RSH00, LAM00, WBS03a].

For a long time hardware support for ray tracing has been held back by three main issues: the large amount of floating-point computations, support for flexible control flow including recursion and branching (necessary for traversal of hierarchical index structures and for programmable shading), and finally the difficulty to handle the memory bandwidth and access patterns to an often very large scene data base.

On the *software* side significant research has been performed on mapping ray tracing efficiently to parallel machines, including MIMD and SIMD architectures [GP90, LS91]. The key goal has been to optimally exploit the parallelism of the architecture in order to achieve high floating-point performance [Muu95, PSL+99, Neb97, BP90, KH95].

Realtime ray tracing performance has recently been achieved even on *single high-performance CPUs* [WSBW01, WPS+03, Wal04]. However, higher resolutions, complex scenes, and advanced rendering effects still require a cluster of CPUs for realtime performance [Wal04].

This large number of CPUs is also the main drawback of these software solutions. The large size and cost of these solutions is preventing a more widespread adoption of realtime ray tracing. We speculate that the ray tracing performance needs to be increased by up to two orders of magnitude compared to a single CPU in order to achieve realtime, full-resolution, photo-realistic rendering for the majority of current graphics applications.

One solutions could be *multi-core CPUs* announced by all the major manufacturers. However, based on the publically announced road maps for multi-core chips, reaching the above goal will take at least another 5 to 10 years.

On the other hand, the computational requirements of ray tracing do not require the complexity of current CPUs. Smaller hardware that satisfies the minimum requirements but allow for greater parallelism seems to be a more promising approach. First examples are ray tracing on a DSP and the simulation for the SmartMemories architecture [GH96, MPJ+00, Pur01].

One particularly interesting example is the use of *programmable GPUs* already available in many of today's PCs. With more than twenty SIMD units, they offer excellent raw floating-point performance. However, the programming model of these GPUs is still too limited and does not efficiently support ray tracing [CHH02, Pur04]. In particular, GPUs do not support flexible control flow and only very restricted memory access. The proposed *Cell architecture* will offer similar parallelism but a much more flexible programming model, if currently available information is correct [Son05, Zim03].

On the other extreme, several *custom hardware* architectures have been proposed, both for volume [MKS98, PHK+99, HKR00] and surface models. Partial hardware acceleration has been proposed [Gre91] and a different implementation is commercially available [Hal01]. In addition a complete ray tracing hardware architecture for static scenes has been simulated [KSSO02]. The first complete, fully functional *realtime ray tracing chip* was presented by Schmittler et al. [SWS02, SWW+04]. However, all of these specialized hardware architectures only support a fixed functionality and cannot be programmed, an essential property for advanced rendering.

1.2.3 The Saarland OpenRT Realtime Ray Tracing Project

In 2000 the Saarland OpenRT Realtime Ray Tracing project was started. At the beginning of this project the aim was to evaluate whether ray tracing can be implemented efficiently on standard processors. Additionally, the requirements for realtime ray tracing on a hardware architecture should be evaluated.

Ray Tracing in Software

Soon it was shown that the computations for ray tracing can be rewritten to perform more than *30-times faster* on a single processor compared to other ray tracers [WSBW01]. Additionally, it was shown that the *performance scales almost linear with the number of processors* in a cluster of standard PCs. This allowed for interactive frame-rates even for highly complex models [WSBW01, WSB01] (see Figure 1.4). These results motivated further research in many fields, which in the following years achieved several important results.

Figure 1.4: Highly realistic image synthesis depends on the correct simulation of the lighting conditions of a virtual world of which especially indirect illuminations causing effects like the color bleeding in the middle images is important. These images are calculated in realtime using ray tracing for global illumination simulations and rendering [BWS03, WBS03b, WKB+02]. The system runs fully interactive and also allows for moving objects with immediate updates on the lighting situation. Since the system still performs well on highly complex models it is very interesting for interior design and architectural applications.

The most advanced application of realtime ray tracing was its use for *global illumination* calculations, which allowed for the first time to receive immediate updates on the illumination of even complex models. These simulations include all direct and indirect illumination effects such as color bleeding even in dynamically changing worlds. Furthermore they have been extended do calculate *caustic effects* using photon tracing [WKB+02, BWS03, WBS03b, WGS04, GWS04, Wal04] (see Figure 1.4).

Recently it has been shown that it is possible to efficiently render even *highly complex models* of an airplane consisting of 350 million individual triangles [WDS04] (Figure 1.5). Alternatively to using highly triangulated models it has been shown that rendering *free-form surfaces* directly can deliver better performance and higher visual quality while saving storage and additionally allowing for efficient animations of objects [BWS04] (see Figure 1.6). [WS05] shows that also point based models can be ray traced efficiently and support for ISO-surfaces in *volume rendering* was presented in [MFK+04, WFM+05].

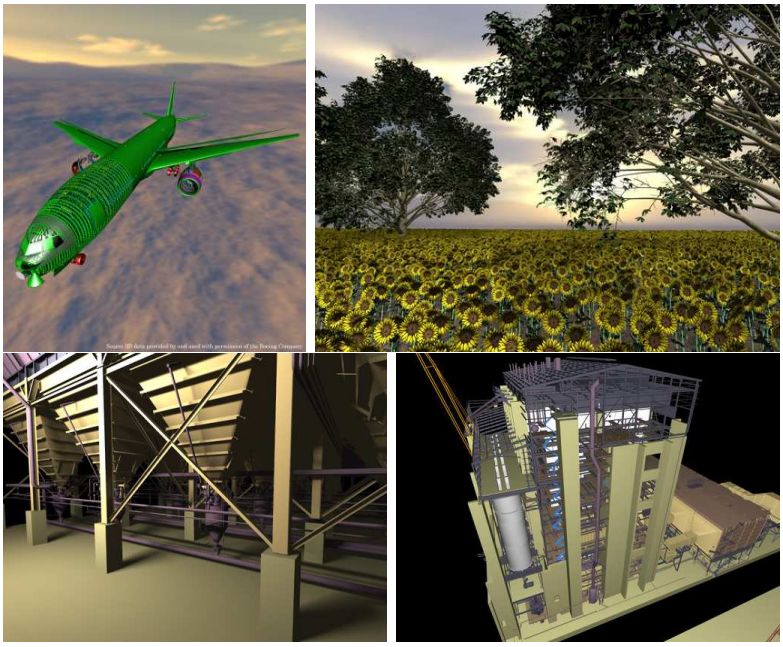


Figure 1.5: Ray tracing allows for rendering even extraordinary huge models like the Boeing 777 (top left image) consisting of over 350 million individual triangles interactively and without any mesh simplification or level-of-detail algorithm [WDS04]. In contrast to the airplane the huge field of sunflowers (top right image) consists of over 1 billion triangles but uses instantiated geometry. The lower row shows images of a highly detailed model of a power plant consisting of 13 million individual triangles. The image on the bottom left demonstrates that ray tracing allows for interactive global illumination calculations even in these highly complex scenes. Images of the sunflowers and the power plant courtesy of Andreas Dietrich respectively Carsten Benthin.

Besides these techniques that allow for highly detailed geometric scenes with complex lighting simulations also standard techniques for animations including benchmarks of the BART suite have been presented [WBS03a, WBDS03].

Efficient support of applications requires an easy to use and flexible programming model for ray tracing. This led to the so called *OpenRT*-API [WBS02, DWBS03, Wal04], which

Figure 1.6: Ray tracing allows for simulating the transport of light physically correct and is therefore ideally suited for visualization of prototypes. The left image still uses a car model constructed of triangles but the image in the middle demonstrates that new advances in ray tracing of free-form surfaces remove the requirement for triangulation [BWS04]. The right most image shows the visualization of a headlight [BWDS02]. Image of car courtesy of Andreas Dietrich.

was adapted from OpenGL to allow for porting existing applications easily and to shorten the period of vocational adjustment. Using this API several applications have been implemented ranging from prototype visualizations over a browser for VRML and applications for virtual studios to computer games [BWDS02, Wag02, PMWS03, PHS04, PS04, DWWS04, SDP+04].

Ray Tracing in Hardware

In addition to these developments using standard computers and software programs it has been evaluated how ray tracing can be implemented using dedicated graphics hardware [SWS02, SLS03, SWW+04]. This thesis presents this line of research.

There have been several Master and Bachelor theses, which base on work of the author who also co-supervised those theses. The Master thesis of Sven Woop [Woo04] focuses on the development and implementation of a dynamic ray tracing core on a FPGA-based prototype. The Bachelor and Master theses of Patrick Dreker [Dre05a, Dre05b] evaluate fixed function and programmable shading and presents a first implementation of programmable shading on a FPGA.

The Master thesis of Alexander Leidinger [Lei04] presents virtual memory management for ray tracing hardware architectures. The not yet finished Bachelor thesis of Daniel Wagner develops and evaluates streamlined floating-point circuits of variable precision and presents a highly optimized FPGA-based implementation.

Ray Tracing in Computer Games

Besides the research on how to accelerate ray tracing using standard and special purpose hardware it has been evaluated how ray tracing can be used in computer games [SDP+04] (see Figure 1.7).

Daniel Pohl [Poh04] adapted the rasterization-based first-person shooter *Quake3: Arena* [IS04] to the OpenRT ray tracing environment. Raoul Plettke [Ple05] has shown how ray tracing simplifies the use of large urban environments like those in racing games *GTA:Vice City* [Roc03] and additionally how ray tracing allows for greatly improving

the visual quality of those environments. Both theses were developed in cooperation with Professor Marc Stamminger of the University of Erlangen-Nürnberg (Germany) and also co-supervised by the author.

Figure 1.7: Besides the two full featured games *Oasen* [SDP+04] and *Q3RT* also two case studies *Ray City* and *Rabbit* have been developed. (Images from left to right: courtesy of Tim Dahmen, Daniel Pohl, Raoul Plettke and the author.)

1.3 Overview of This Thesis

As this thesis deals with hardware accelerated ray tracing as a possible future technology for interactive computer graphics its focus is not limited to the presentation of a single special prototype. Instead a general hardware architecture is discussed including fully programmable shading and it is evaluated how graphics cards using this technology can be used for a wide range of applications.

Thus, this thesis contains three different parts: the design of the ray tracing core (i.e. the visibility calculation using rays), the design of an infrastructure supporting multi-threaded and multi-core CPUs in a shared memory system (for shading), and the evaluation of this technology using various applications and content of computer games. The main focus is on the first part as it showns that to great extents the other parts can be realized by adaptations of known techniques.

Outline

The next chapter presents algorithms for ray tracing of statical and dynamically changing scenes. It includes details, optimizations, and finally evaluations of the algorithms in an implementation independent way. The results of these evaluations are guidelines on how to build a hardware architecture for ray tracing.

Chapter 3 discusses the design decisions leading to the key features of the SaarCOR hardware architecture for realtime ray tracing and presents an overview. The following Chapters 4, 5, and 6 then handle in detail the hardware architectures for tracing and intersecting rays, shading, and the memory interface, respectively.

The design process from ray tracing algorithms to a working ray tracing hardware prototype is documented in Chapter 7. It includes the presentation of the various variants of the SaarCOR hardware architecture and how they are simulated, implemented, and optimized depending on technology specific parameters. The detailed evaluation of these variants can be found in Chapter 8. Finally, Chapter 9 concludes this thesis, gives an outlook, and presents directions for future work.

2 Ray Tracing Algorithms

In the previous chapter the basic principle of ray tracing was summarized and its key features were presented. In this chapter the principles of ray tracing are explained in more detail before more advanced techniques and algorithms for ray tracing are discussed. Additionally, hardware oriented implementation details and extensions are presented before a high-level evaluation of the algorithms concludes this chapter.

The basic principle of ray tracing can be seen in Figure 2.1: The task is to generate the two dimensional image PIC of a three dimensional world or object OBJ which can be seen on the virtual screen SCR through the virtual camera CAM. For each pixel (examples are P1 and P2) on the virtual screen a ray (e.g. R1 and R2) starting at the camera is generated. Then it is checked which object is intersected by this ray first.

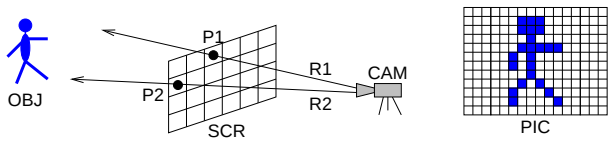


Figure 2.1: Basic Principle of Ray Tracing

This basic algorithm covers only what can be seen directly from the camera and is called *ray casting*. For advanced image synthesis also indirect effects such as the illumination of objects, reflections in mirrors, or refractions in glas materials need to be calculated. These issues are covered by the recursive *ray tracing* algorithm.

In the following the process of ray tracing is explained using the example of Figure 2.2. Here the scene to be rendered is described by specifying geometric and material properties for all objects of the scene and additionally positions and characteristics of light sources. All specifications can contain various constants but also functions or *shader programs* which are evaluated on-demand during rendering. Although in general arbitrary geometric primitives can used with ray tracing (e.g. [BWS04, MFK+04, DWBS03]), in this thesis only triangles are used to simplify scene descriptions and algorithms. But this is no hard limitation since every geometric object can be approximated using triangles and most scenes used in interactive computer graphics today use triangles only. Nevertheless Sections 4.5.4 and 5.3 deal with extensions to support other primitives as well.

Rendering of an image starts with ray casting of *primary rays* for each pixel of the image. If a ray pierces at least one object a *hit-point* is returned which is the closest intersection between this ray and all objects of the scene. Using this hit-point (respectively the information that there is none) the ray is shaded to calculate the color of the pixel.

Shading a ray might generate *secondary rays* recursively to evaluate global effects at the hit-point, e.g. direct or indirect illumination, reflections, or refractions. In the example above the primary ray hits the blue sphere which has a shiny surface and therefore requires the evaluation of a *reflection ray*. Additionally, the direct illumination from the

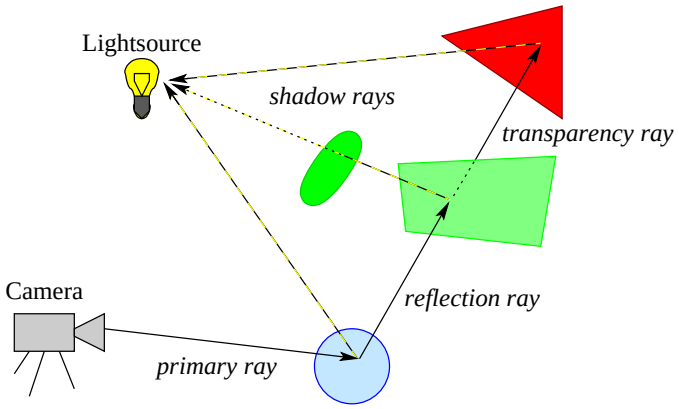


Figure 2.2: Recursive Ray Tracing

light source is checked by a *shadow ray*.

Since the shaders might spawn arbitrary rays, this concept allows also for more advanced shading effects such as diffuse reflections using multiple reflection rays to stochastically sample the environment. Similarly, area light sources can be sampled using multiple randomly chosen point-light sources [WKB+02, BWS03]. In general ray tracing allows for approximating the integral part of the incoming light over a hemisphere using multiple rays and thus allows for rendering highly realistic images of virtual worlds.

2.1 Ray Tracing of Static Scenes

The ray tracing algorithm as described above is rather slow as every triangle in the scene has to be intersected with every ray to find out the intersection closest to the ray's origin.

In general a procedure can be accelerated by either exchanging parts of the algorithm by more efficient techniques or by finding ways to achieve the same result by doing less work. In the spirit of the first way one could exchange the ray-triangle intersection method by a more efficient one. Although there are many different algorithms available (see Section 4.5) the procedure still is linear in the number of triangles in the scene.

But for static scenes, one can put some amount of work to build up data structures helping to quickly find triangles in the vicinity of a given ray. While this has some initial cost, it amortizes well during rendering and highly pays off over multiple frames.

2.1.1 Spatial Index Structures

The number of ray-triangle intersections is reduced by using a *spatial index structure* which spatially subdivides the volume of the scene into smaller volumes (so called *voxels*) each containing only a few triangles. The triangle closest to the ray's origin which is pierced by the ray is then found by *traversing* the acceleration structure which enumerates all voxels pierced by the ray starting at the ray's origin. Since only the triangles found in such voxels are intersected this drastically reduces the number of ray-triangle intersections and speeds up the ray tracing process. Therefore spatial index structures are often refered as *acceleration data structures*.

Since intersection computation starts closest to the ray's origin and continues along the ray it can be terminated after the intersection of a voxel is finished and a valid intersection has been found. This property of ray tracing using a spatial index structure is called *early ray termination*. Since for each ray early ray termination is checked individually this leads to automatic and highly efficient *occlusion culling* on a per ray basis. An example using a *regular grid* which subdivides scene space uniformly can be seen[1] in Figure 2.3.

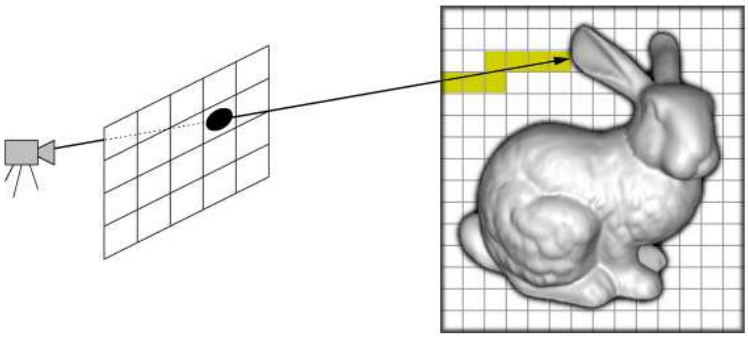


Figure 2.3: Example for traversal of a static scene using a regular grid.

Compared to grids, hierarchical spatial index structures such as *octrees* and *binary space partition trees* (BSP-trees) [SS92, Sub90, Hav01] provide an adaptive way to subdivide scene space. A survey of acceleration structures and techniques can be found in [Gla89, Wal04, Hav01] while the latter one also shows that hierarchical subdivision methods reduce the average computational complexity of scenes with n triangles from $O(n)$ to $O(log\ n)$.

A special case of BSP-trees are axis-aligned BSP-trees which are called *kd-trees*. These spatial index structures hierarchically subdivide 3D scene space with planes orthogonal to one axis of the coordinate system. Therefore the position of the so called *splitting plane* can be described by a flag denoting the axis and a scalar value specifying the position on the axis. A number of heuristics of how to position these planes to achieve optimal space subdivision using as few subdivisions as possible have been published [Hav01, Wal04].

Figure 2.4 shows three examples for kd-trees of different qualities. Each inner node of the tree contains a description of a splitting plane and a pointer to its two children. Each leaf node forms a voxel and contains a reference to a list of triangles. Generally speaking a kd-tree leading to high performance ray tracing has large empty voxels for quickly skipping empty space, has as few triangles per voxel as possible to reduce the overhead and has as few nodes as possible to reduce the cost of traversal.

Due to their low storage requirements, simplicity (see next section), and good performance kd-trees are used on the OpenRT software ray tracer as well as on the SaarCOR hardware architecture. In particular, the SaarCOR hardware directly uses the kd-trees built by the OpenRT software.

As of today, the best known method for building cost-optimal kd-trees (i.e., kd-trees that yield very good traversal performance) is the *Surface Area Heuristic* [Hav01, Wal04].

[1] Although all algorithms and hardware units presented in this thesis are designed for operations in three dimensions for the sake of simplicity throughout this thesis most examples are drawn in 2D only and some of them show regular grids as a representation for any type of spatial index structure.

However, the benchmarks presented in this thesis use kd-trees of two different qualities: *good* kd-trees using the Surface Area Heuristic by Ingo Wald [Wal04] and *standard* kd-trees using a different heuristic also by Ingo Wald but implemented earlier at the beginning of the OpenRT project (for details see Chapter 8).

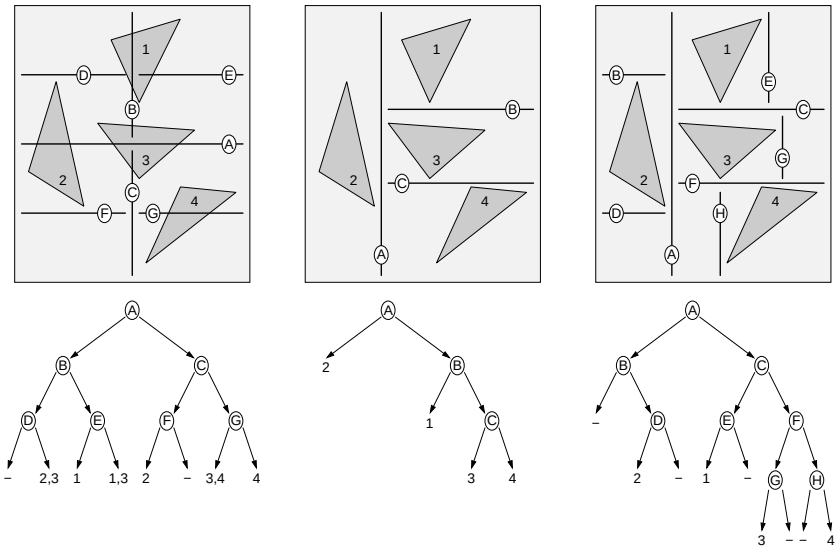


Figure 2.4: Examples for kd-trees of different qualities: the left-most tree simply subdivides in the middle of the longest dimension and yields the worst tree with many triangles per voxel. The example in the middle uses more advanced heuristics and yields a tree with exactly one triangle per leaf. The right-most tree subdivides even further allowing to quickly skip empty space. Traversing a ray through the right-most tree generally requires more traversal steps but less ray-triangle intersections than tracing a ray through the kd-tree of the example in the middle.

2.1.2 Traversal of kd-Trees

This section presents an algorithm for traversal of kd-trees. It is adapted from [WBWS01, Wal04] which in return are based on [Kel98, Hav01].

Let a ray $R = (O, D)$ with origin O and direction D be defined as

$$R(t) = O + t \cdot D \text{ with } t \in [0, \infty) \text{ and } O, D, R(t) \in \Re^3$$

and let *near, far* $\in \Re$ with $0 \leq near \leq far < \infty$. Traversing a ray is then cutting the interval of $t \in [near, far]$ to the bounding box of the volume the ray enters. Since volumes are defined by splitting the current volume on axis aligned planes, the traversal operation can be perform in 2D. Therefore in every *traversal step* the intersection $R(d)$ with the splitting plane is calculated. As a splitting plane is described using the axis $k \in \{x,y,z\}$ orthogonal to the plane and the position $s \in \Re$ on this axis, calculating the hit point $R(d)$ is done using:

$$d = \frac{s - O_k}{D_k} \text{ with } k \in \{x,y,z\} \text{ and } d, s \in \Re$$

Here for $P \in \Re^3$ P_k denotes the component of P on the axis k. We identify the volume

formed by all points $P \in \Re^3$ for all $P_k \leq s$ as the *negative half space*. In an analog way $P_k \geq s$ forms a *positive half space*. Please note that this definition is conservative which means that objects lying on the splitting plane are contained in both half spaces[2]. Since traversal operations are performed in 2D only for the sake of simplicity all examples are drawn in 2D using a splitting plane parallel to the y-axis independent of the actual splitting axis. Furthermore we identify the natural order of written text with the 2D coordinate system, that means the y-axis points from the bottom of a page to its top and the x-axis points from the left to the right side (see Figure 2.5). Therefore we can describe the negative half space formed by a splitting plane as being *left* of the plane and the positive half space as being *right* of the splitting plane.

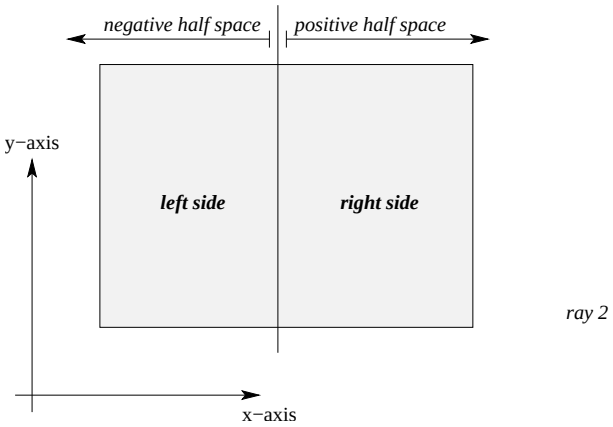


Figure 2.5: Simplified labels for kd-trees shown in examples: all splitting planes are drawn parallel to the y-axis independent on their actual orientation and the *negative half space* formed by the splitting plane is labeled as being *left of the splitting plane*.

Depending on the direction of the ray, it crosses the splitting plane either from left to right or vice versa. For each direction there are three possible cases for the position of $R(d)$ relative to the interval [*near, far*] as depicted in Figure 2.6 and described in Table 2.1. Additionally, depending on the implementation of floating-point operations special care has to be taken of rays parallel to the splitting plane (leading to division by zero errors in the calculation of d).

The side of the splitting plane which contains the origin of the ray is called the *near*-side while the other is the *far*-side. If a ray wants to visit both children the *near*-side is always entered first.

Initialization

While this distinction of cases is quite simple it requires *near* and *far* to be initialized corresponding to the bounding box of the scene. This initial cutting of *near* and *far* is called *clipping* and although no errors can occur if done inproper or not at all it might lead to unnecessary traversal steps and performance deterioration (see Figure 2.7a).

For rays which start and end inside the bounding box of the scene clipping is not necessary.

[2] Obviously, this is suboptimal and therefore methods for building kd-trees avoid having objects on the splitting plane by adjusting the position of the plane.

ray crosses splitting plane from left to right:

Fig.2.6a)	$d > far$	visit only left side: keep interval [*near, far*]
Fig.2.6b)	$near \leq d \leq far$	visit both sides: left=[*near, d*], right=[*d, far*]
Fig.2.6c)	$d < near$	visit only right side: keep interval [*near, far*]

ray crosses splitting plane from right to left:

Fig.2.6a)	$d < near$	visit only left side: keep interval [*near, far*]
Fig.2.6b)	$near \leq d \leq far$	visit both sides: right=[*near, d*], left=[*d, far*]
Fig.2.6c)	$d > far$	visit only right side: keep interval [*near, far*]

Table 2.1: The six different cases of kd-tree traversal.

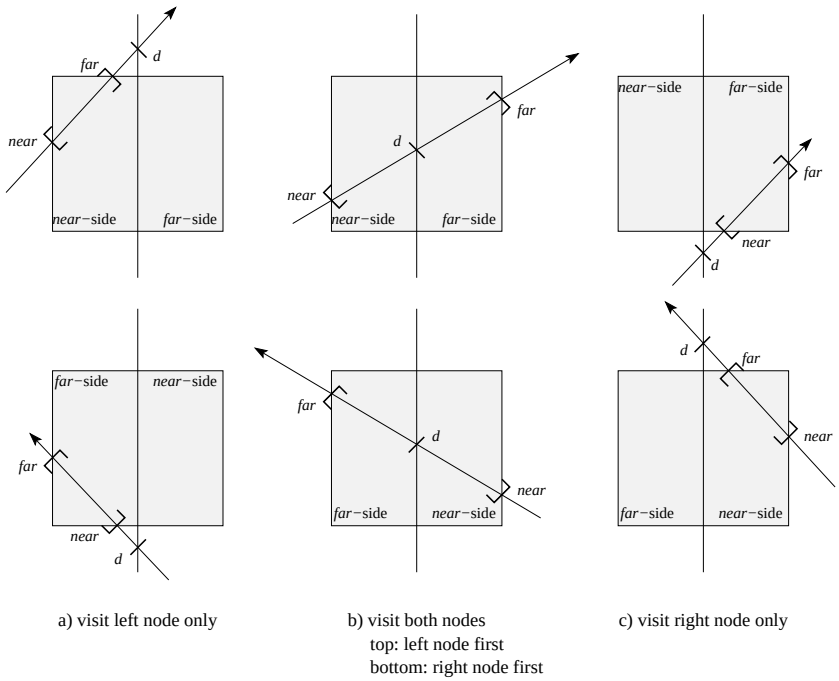
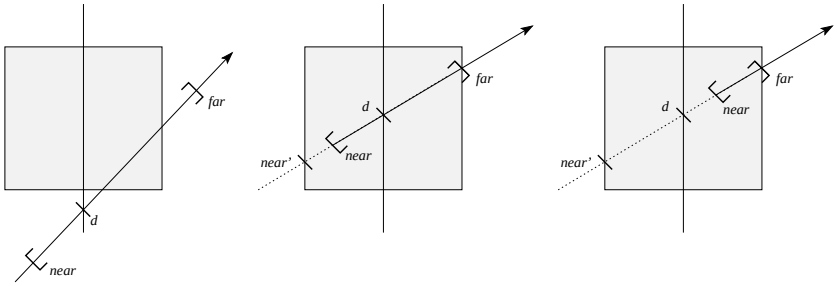


Figure 2.6: The six different case of kd-tree traversal.

Unfortunately clipping those rays can actually lead to unnecessary traversal steps as *near* might become negative during clipping as depicted in Figure 2.7b. This can be avoided by initializing $near = 0$ and $far = \infty$ and only accepting updates *near'* and *far'* if *near*<*near'* and *far*>*far'* respectively. Obeying this update policy becomes even more important when tracing only segments of rays[3], that means when starting with a given interval for t.

In typical ray tracers written in software clipping is performed using special subroutines. For a hardware based ray tracer a dedicated unit for clipping is expensive and used only once per ray. Instead of using a dedicated unit clipping can also be performed by adding six nodes at the top of the kd-tree. These nodes contain splitting planes containing the six sides of the axis-aligned bounding box of the scene (see Figure 2.8). Especially

[3]Tracing segments of rays is useful for many applications e.g. for prototype visualization when cutting an object into slices without actually modifying the 3D data.



a) inproper clipping b) ray origin inside voxel

Figure 2.7: Two special cases of kd-tree traversal: a) traversal step with incorrectly set *near* and *far* values, b) a ray starting inside a voxel. All cases can be handled trivially by correctly initializing *near* and *far* to the bounding box of the scene and at the same time rejecting any updates on *near* if $d <$ *near*. Inproper handling of these cases does not cause any errors but may lead to unnecessary traversal steps.

with optimizations given in Section 2.4.2 clipping using traversal steps is most efficient in hardware.

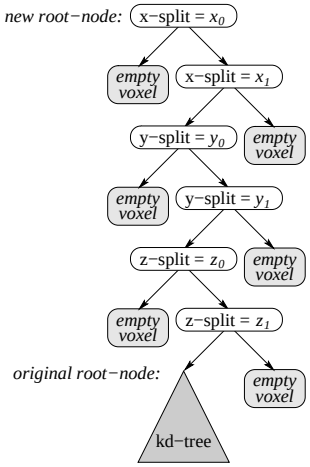


Figure 2.8: Clipping of a ray to the axis-aligned bounding box $((x_0, y_0, z_0), (x_1, y_1, z_1))$ of a scene by putting six additional nodes on top of the scene's original kd-tree.

Termination

Traversal of a ray can be terminated as soon as a valid hit-point $R(t)$ is found. The validity of a hit-point has to be checked since there are two cases in which a hit-point is either

invalid	$t <$ *near*, e.g. behind the viewer, or
not yet valid	$t >$ *far*, e.g. Figure 2.9

The test $t < near$ can be optimized if support of tracing segments of rays is not necessary. Then any hit-point with $t \geq 0$ is either valid or not yet valid. Thus instead of performing the full test $(t - near) \geq 0$ which requires a floating-point subtraction it suffices to check the sign of t. This optimization is especially suited for hardware implementations.

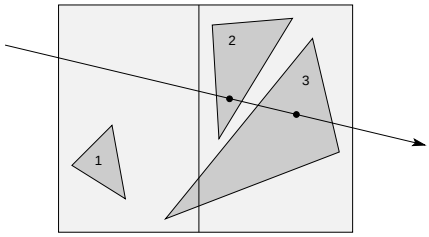


Figure 2.9: Example for traversing a kd-tree that shows the necessarity to check whether a hit-point is in the current voxel: In the first step the left voxel is intersected yielding a hit-point on triangle 3. Since this hit-point is not inside the intersected voxel traversal continues with the voxel on the right leading to a valid and correctly identified final hit-point with triangle 2.

2.1.3 Implementation Details

During the traversal of a kd-tree a three bit *traversal decision*$=(gl, gr, fc)$ needs to be calculated for each node which specifies whether to go left (gl), to go right (gr) or, in case both children are visited, which one is the first child (fc) to be visited. As each node has exactly two children, child(0) denotes the left child and child(1) the right one.

The description of the six cases of kd-tree traversal in Table 2.1 is not suitable for direct implementation in hardware. Therefore Table 2.2 presents a reformulation of the traversal algorithm using standard notations (see Appendix A). Please note that except for the multiplication and the subtractions (which will be dealt with below) in the description of the hardware only single bits and gates are used. Further note that the computation of entering either side is computed in two steps: first it is checked what a ray crossing the splitting plane from left to right (*norm.ray*) would do. Then if the actual ray is orientated in the opposite direction left and right are permuted.

Since multiplications are much faster to perform than divisions to speed up the traversal steps $I_k = \frac{1}{D_k}$ with $k \in \{\text{x,y,z}\}$ is computed only once per ray during generation. Furthermore since the computation of l and r requires only the sign of the subtraction no full floating-point subtractors are required but two integer comparisons on the exponent and the mantissa suffice. This allows for an optimized implementation in hardware. Thus besides at most three floating-point subtractions and one floating-point multiplication only 13 gates are necessary to compute the traversal decision.

Performing the Traversal Step

After the calculation of the traversal decision performing the traversal step is straight forward, if only one side needs to be visited. If both sides are to be visited, then *far* and the pointer to child($/fc$) to need to be put to the stack. After that, we set $far = d$ and enter child(fc).

	Algorithmic Operation	Hardware Version
distance	$d = \frac{(s-O_k)}{D_k}$	$d = (s - O_k) \cdot I_k$
go left norm.ray	$l = \begin{cases} 1 & \text{if } d > far \\ 0 & \text{else} \end{cases}$	$l = SIGN(far - d)$
go right norm.ray	$r = \begin{cases} 1 & \text{if } d < near \\ 0 & \text{else} \end{cases}$	$r = SIGN(d - near)$
go both	$b = \begin{cases} 1 & \text{if } near \leq d \leq far \\ 0 & \text{else} \end{cases}$	$b = /l \wedge /r$
right to left	$rtl = \begin{cases} 1 & \text{if } D_k < 0 \\ 0 & \text{else} \end{cases}$	$rtl = SIGN(D_k)$
first child	$fc = \begin{cases} 1 & \text{if } (D_k < 0)\ AND\ (near \leq d \leq far) \\ 0 & \text{else} \end{cases}$	$fc = rtl \wedge b$
go left act.ray	$gl = \begin{cases} 1 & \text{if } (\ (d > far)\ AND\ (D_k \geq 0) \\ & \quad OR\ (d < near)\ AND\ (D_k < 0) \\ & \quad OR\ (near \leq d \leq far)\ ) \\ 0 & \text{else} \end{cases}$	$gl = (\ l \wedge /rtl\ ) \vee (\ r \wedge rtl\ ) \vee b$
go right act.ray	$gr = \begin{cases} 1 & \text{if } (\ (d < near)\ AND\ (D_k \geq 0) \\ & \quad OR\ (d > far)\ AND\ (D_k < 0) \\ & \quad OR\ (near \leq d \leq far)\ ) \\ 0 & \text{else} \end{cases}$	$gr = (\ r \wedge /rtl\ ) \vee (\ l \wedge rtl\ ) \vee b$

Table 2.2: Algorithmic description of the operations performed during a traversal step and their representation suited of hardware implementation. It shows that besides three floating-point subtractions and one multiplication only 13 gates are necessary to compute the traversal decision. This gate count is interesting for extending general purpose processors for efficient ray traversal (see Section 5.3).

If a voxel has been intersected but no valid hit-point has been found set $near = far$ and pop *far* and the pointer from the stack and enter the corresponding node. If the stack is empty traversal is finished – maybe even without finding a hit-point.

Every voxel is intersected by sequentially intersecting the ray with every triangle contained in that voxel. Each ray-triangle intersection returns the *hit-information* = (*hit*, *ID*, *dist*, u, v) with the boolean *hit* specifying that the triangle was hit (1) or missed (0)[4]. If it was a hit, *ID* specifies which triangle was hit and *dist* is its distance along the ray, i.e. $R(dist)$ is the position of the hit-point.

Most ray-triangle intersection algorithms calculate the barycentric coordinates (u, v) of the position where the ray pierces the triangle (see Figure 2.10). Since these coordinates are often required for shading, e.g. for texturing, recomputation can be avoided by re-

[4]For hardware implementations this flag can be omitted as */hit* can be coded directly into the sign-bit of the distance since negative distances are invalid anyway.

turning (u, v) together with the hit-information. During the ray-triangle intersections only the intersection closest to the ray's origin, i.e. with minimal *dist* is kept. However, an extension which returns a sorted list of the n closest intersections is straight forward. This might be of use if geometry can be tagged as *non-occluder*, e.g. mostly transparent objects that are approximations of non-geometric objects like fire and smoke as used in current computer games.

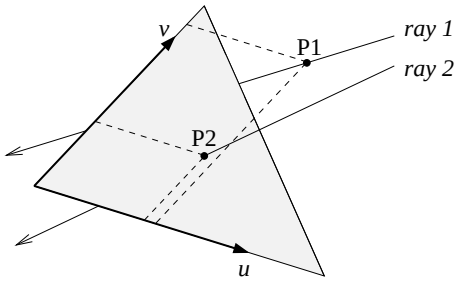


Figure 2.10: Examples for ray-triangle intersection tests using the barycentric coordinates of the position where the ray pierces the plane spanned by the triangle. The ray *hits* the triangle if for the coordinates (u, v) it holds $0 \leq u, v \leq 1$ and $(u + v) \leq 1$ as can be seen for ray 2. Ray 1 *misses* the triangle since P1 has $(u + v) > 1$.

Data Layout

The layout of the data structures of the kd-tree is important to achieve good performance as it influences the caching behavior and the memory bandwidth. Therefore the data layout has been carefully evaluated by Ingo Wald [Wal04] for the OpenRT software ray tracer. The data layout used for the SaarCOR architecture is basically the same, but adds support for culling empty voxels (see Section 2.4.2) and is presented in the following.

A node in the kd-tree needs to store pointers to its children and two flags denoting whether they actually contain a valid node (used for the extension presented in Section 2.4.2). Additionally, it stores the splitting plane (one floating-point value and a flag describing the corresponding axis) and a flag to denote whether it is an *inner node* or a leaf (*voxel*). If the node is a leaf, only the number of triangles contained in the voxel and a single pointer to the list of the corresponding triangle IDs needs to be stored.

The memory layout is simplified by using a unified data structure which stores inner nodes and leafs in *records* of the same size (64 bits). This is achieved by restricting the address space of the kd-tree to 4 GB (32 bit addresses for byte-addressable memory) which suffices for most applications. Storing the floating-point value of a splitting plane of an inner node requires 32 bits, which can be used to alternatively store the number of triangles contained in a voxel.

Additionally, the flags require 4 bits for inner nodes and 2 bits for voxels. Since every inner node has exactly two children storing both children aligned in memory allows for specifying the address of the children using a single pointer (see Figure 2.11). To allow for relocation of the kd-tree in memory, no absolute pointers but relative offsets *ofs* are used. Thus as each record is 64 bit wide, storing the flags is for free as the alignment property of pairs of inner nodes guarantees that $ofs[3:0] = 0000$ which gives 4 bits to store the flags. The pointers to 32 bit triangle IDs stored in voxels have the alignment

property $ofs[1:0] = 00$ which gives the 2 bits required. The data structures for inner nodes and leafs are described in Figure 2.12.

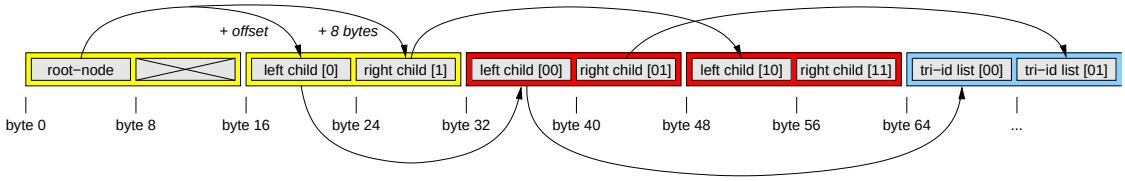


Figure 2.11: Memory layout of the left-most kd-tree in Figure 2.4

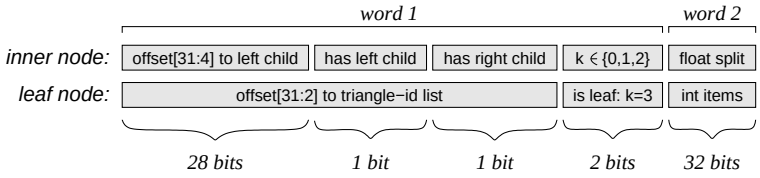


Figure 2.12: Unified data structure of the kd-tree nodes (inner nodes and leafs).

2.2 Ray Tracing of Dynamic Scenes

The visual realism and especially the immersion of a virtual world depends to large extends on dynamically changing conditions (such as movement of the sun and weather) and dynamically modifications of the world (such as moving people and objects).

In current VR and computer games those dynamics are realized mainly by moving, modifying, inserting, and removing triangles and changing their material properties accordingly. While updates to material properties are always trivial any change on the geometry only works well for immediate mode rendering. But since ray tracing heavily relies on spatial index structures to achieve reasonable speed, supporting arbitrary modifications on the geometry requires recalculation of the corresponding data structures. Unfortunately building spatial index structures is costly in terms of computations and memory bandwidth and thus realtime generation of kd-trees is only possible for small numbers of triangles even on high-end computers (see [WBS03a]).

Fortunately, a look into the real world shows that many objects do not perform unstructured motion. For example a city with houses, traffic lights, and cars can be represented nicely by rigid bodies. While most houses stand still, cars change their position and orientation and traffic lights might get bent by the wind[5] but these motions can be produced by an affine transformation matrix using translation, rotation, and shearing respectively and without rebuilding the kd-trees of any rigid object. This observation by Wald et al. [WBS03a, Wal04] was used in the OpenRT software for ray tracing of dynamic scenes. The techniques used there are also used in the SaarCOR architecture (with minor extensions) and presented in the following.

[5]Please note that here not arbitrary bending but a combination of rotation and sheering is meant.

2.2.1 Dynamic Scenes Using Rigid Objects

The previous section presented ray tracing for static scenes using a kd-tree as spatial index structure. This kd-tree subdivides a region enclosed by a bounding box. A *rigid object* in the sense of this thesis is a static scene with a kd-tree, a bounding box, and some associated[6] geometric and material data. Furthermore, we allow the references in the kd-tree leafs to point to lists of triangles as well as to lists of *instances of rigid objects*. An instance of an object is a reference to a rigid object and an affine transformation matrix. Using this specification *hierarchical dynamic scenes using rigid objects* can be described.

Obviously cyclical references are forbidden and therefore the structure of these hierarchical dynamic scenes can be drawn as a *directed acyclic graph* (DAG). The level of the DAG containing the parent rigid object is called the *top-level* while the *bottom-level* is formed by all leaf nodes containing rigid objects with references to triangles only. We call a rigid object containing only references to other rigid objects a *meta object* and rigid objects with only references to triangles[7] a *geometric object*.

Since all objects are rigid any change to triangles or transformation matrices requires recalculation of the kd-trees of all higher levels. Figure 2.13 shows an example of a dynamic scene and presents DAGs illustrating the levels and their dependencies. Further details on the hierarchical structure and the corresponding levels are given in Section 4.1.

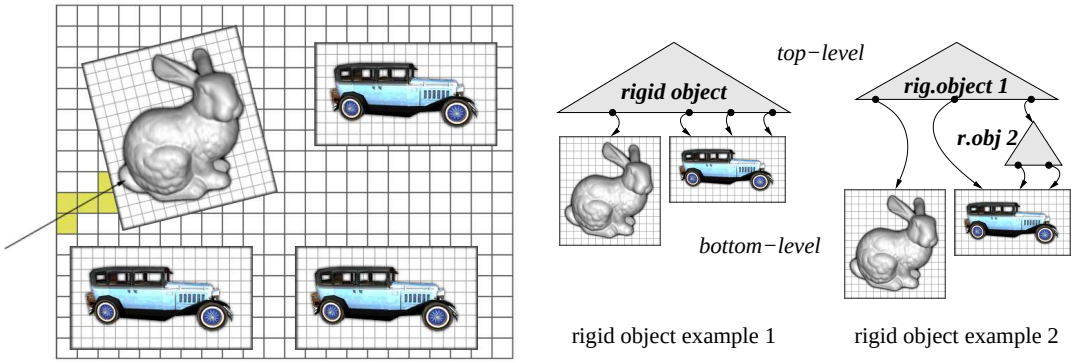


Figure 2.13: Example for traversal of a dynamic scene using rigid objects. On the left the execution of the traversal algorithm is illustrated while on the right there are two examples for the hierarchical structure of the scene. In the second example two cars are grouped together in a rigid object allowing for simultaneous movements. On any change of a rigid object (either its triangles or its transformation matrix) the kd-trees of all rigid objects above them have to be recomputed.

2.2.2 Traversal of Dynamic Scenes

Traversal of dynamic scenes made of rigid objects is identical to the traversal of a static scene. The only difference is the intersection of a voxel since additionally to intersecting

[6] This definition does not specify where and how associated data is stored and whether it is shared by other objects as well since the data layout depends on the implementation only.

[7] Chapters 4.5 and 6.2 will show how the restriction to triangles only can be removed allowing for supporting other geometric primitives as well with only a few minor changes.

rays with triangles also intersections between rays and rigid objects need to be calculated. This latter intersection can be implemented by first transforming the ray into the coordinate space of the object using the affine transformation matrix and the traversing the kd-tree of the object.

When a voxel is reached during traversal the state of the ray in the current kd-tree is saved. Then for each object i contained in the voxel the ray $R_m = (O_m, D_m)$ is transformed from the coordinate space of the current meta object into ray $R_i = (O_i, D_i)$ in the local coordinate space of the object.

This transformation uses the affine transformation matrix $\mathrm{T}_i = (M_i, N_i)$ of the object i with $M_i \in \Re^{3,3}$ and $N_i \in \Re^3$. Since a ray consists of two components two transformations have do be performed: $O_i = M_i \cdot O_m + N_i$ and $D_i = M_i \cdot D_m$.

The ray R_i is then traversed and intersected with object i, which either intersects further objects or finally triangles. The latter intersection returns the hit-information$_i$ for object i which is compared to the hit-information$_m$ of the meta object one level higher and only the information with the closest distance is kept. Here the *ID* in the hit-information contains not only the ID of the triangle but also the IDs of all objects from the top-level object to the bottom-level object.

Directly comparing the hit-distances resulting from intersections of geometric objects at different levels is valid as the distance is measured in terms of the length of the ray which is preserved under affine transformations (as long as the ray direction is not normalized after the transformation) [WBS03a, Wal04]. The criteria for termination of traversal is equal to traversal of static scenes.

This concept of instantiated rigid objects also allows for having several cars of the same type but with different paints without duplicating the triangle data. When rendering static scenes the triangle-ID of the hit-information is mapped to a material which is then used to shade the triangle. Shading of instantiated rigid objects can use the triangle-ID and the object-ID which allows for identifying the same triangle in different instantiations of the object.

Additionally, simple animations using key-frames can be realized using instantiated rigid objects. Here every key-frame is represented by a separate geometric object and animation is performed by exchanging the pointers to the different poses of the object. Furthermore using hierarchical dynamic scenes also allows for doing skeletal animation in a highly efficient way[8].

Thus while dynamic scenes using rigid bodies do have strict limitations they still suffice for a wide range of applications and games (some examples are shown in Section 1.2.3).

2.2.3 Building kd-Trees for Meta Objects

In Section 2.1.1 criteria for good kd-trees of geometric objects have been presented which also hold for kd-trees of meta objects. But in contrast to kd-trees of geometric object which are built in a preprocessing step kd-trees of meta objects typically have to be rebuilt more often (after every change to a transformation matrix). Thus in general the

[8] However, there is a problem with the joints of the various bones, but these can be approximated. Alternatively rebuilding a kd-tree for only the few triangles of the joints in a separate object is also feasible.

effort put into building a kd-tree of a meta object can only be amortized over a single frame. Therefore simpler heuristics which can be performed faster are used to build these kd-trees [WBS03a, Wal04].

For example when building kd-trees of geometric objects for every triangle it is checked exactly whether it is contained in the current voxel (see [Wal04]). For kd-trees of meta objects an exact check would be very complex as every triangle (or even worse: referenced rigid object) had to be transformed and checked. Therefore the check is approximated by only checking whether the axis aligned bounding box of the transformed bounding box of the meta object is contained in the current voxel. This leads to an overhead as Figure 2.14 illustrates even when compared not to a direct check on the transformed triangles but on the much simpler check on the transformed bounding box.

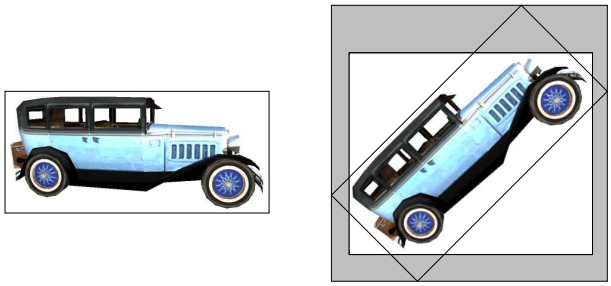


Figure 2.14: Example for an rigid object with its axis-aligned bounding box and the same object and its bounding boxes rotated by 45 degrees. It shows that the axis-aligned bounding box of the rotated object is much larger (the grey part) than required.

Since a correct check simply would be too costly for kd-trees of meta objects one has to use the rather coarse approximations. Nevertheless using clipping in the referenced objects reduces this overhead again.

2.3 Packets of Rays

So far a rather simple ray tracing algorithm has been presented. This algorithm requires a very high bandwidth to memory as for each traversal step unpredictable and for each ray-triangle intersection partially prefetchable new data has to be loaded. This leads to a bad compute-to-bandwidth ratio and long latencies for memory accesses.

In general the performance of memory bound algorithms can be improved by

A) Increasing the memory bandwidth,

B) Using a cache, or

C) Reducing the memory accesses by rearranging the algorithm.

Solving the issue by using faster memory chips is usually the last choice as this can increase the costs drastically. Adding a cache helps in many cases but without exploiting any features of the algorithm. It just stores as many of the previous memory accesses as possible and there is hope that a new request can be served by delivering the old data stored in the cache.

Fortunately the number of memory accesses can be greatly reduced by exploiting the coherence between neighboring rays. Since coherent rays are likely to access the same memory we can group individual rays into *packets of rays*. Grouping takes place during ray generation and the whole packet stays together until all rays are terminated. For each step of the algorithm memory data is fetched only once and used for all rays in the packet, which are still computed individually [WBWS01, Wal04].

Thus ideally using a packet of n rays should reduce the required bandwidth by a factor of n. But since a packet needs to visit every node that any ray of the packet wants to visit some rays might be idle during a step of the packet. Therefore the savings and the additional costs of packets of rays have to be evaluated using careful profiling (see Section 2.5).

2.3.1 Traversing Packets of Rays

For every ray of the packet the traversal decision is calculated individually. Finally, the logical *OR* over all traversal decisions is calculated which forms the group's traversal decision=(gl_g, gr_g, fc_g). Here gl_g and gr_g specify whether any ray of the packet wants to visit the left respectively the right child. If we assume that all rays of a packet cross the splitting plane in the same direction then fc_g specifies in which child is visited first. The case where this assumption is violated is handled in the next paragraph.

Except for the trivial cases where the packet only wants to enter either side Table 2.3 lists all possible cases of different decisions and the action that is performed. When using packets, rays may be forced to enter a child that they do not want to visit. Those rays must be set to be *inactive* in those nodes not only since operations performed on the ray in that node are useless but also since any contribution to the group's traversal decision by the inactive ray may lead to children being visited no ray of the packet wants to visit (see Figure 2.15).

packet decision	ray decision	individual action of ray	collective action of group
(1,1,*fc*)	(1,0,*fc*)	no individual action	visit child(*fc*)
(1,1,*fc*)	(1,1,*fc*)	1. put *far* to stack 2. set *far* = *d*	1. put child(/*fc*) to stack 2. visit child(*fc*)
(1,1,*fc*)	(0,1,*fc*)	set ray to inactive	visit child(*fc*)

Table 2.3: Skipping the trivial cases where the packet only wants to enter either side this table lists all possible cases of different decisions and the action that is performed.

Thus during traversal some rays may put values to the stack and some do not. When a voxel has been intersected and not all rays are terminated the stack is popped. To successfully restore the state of all rays for every ray R_i the stack must contain the flags $active_i = gl_i$ respectively gr_i depending on the side that has been put to the stack, $both_i = gl_i \wedge gr_i$ and if $both$=1 also the value of far. Then during popping only those rays with $both_i$=1 update $near_i$=far_i and set far_i to the popped value of far_i.

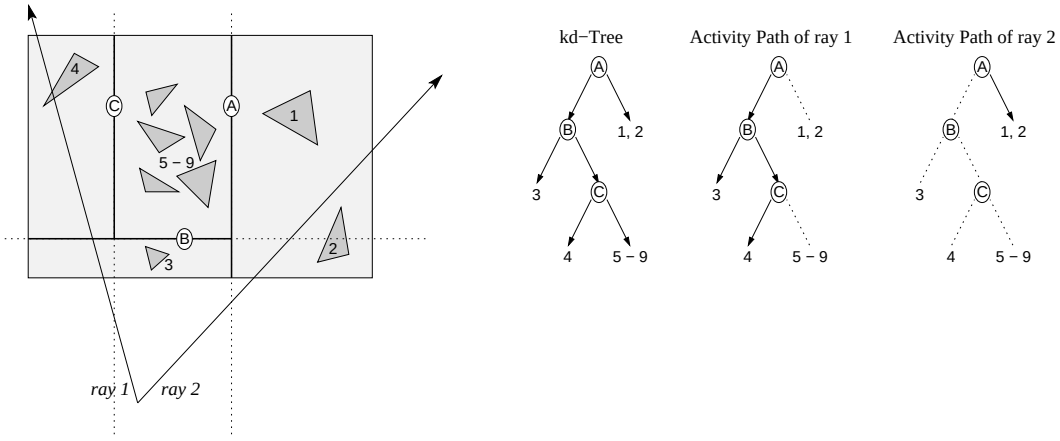


Figure 2.15: This example for tracing packets of rays illustrates the importance of marking those rays inactive that do not want to visit the child the packet currently enters. In the middle the corresponding kd-tree is shown which is fully visited when the activity of the rays is not taken into account. With proper marking of the activity the paths both rays take in the kd-tree is shown on the right. Note that triangles 5–9 should not be visited but would if ray 2 was still active when checking plane C.

Consistency of Packet Decisions

Unfortunately the assumption that all rays of a packet always cross splitting planes in the same direction does not hold in general. This assumption can be simplified since actually we only assume that all rays cross the splitting plane *consistently*. That means that all rays of a packet that do cross the splitting plane do it in the same direction and for all other rays we do not care. Nevertheless even the simplified assumption still does not always hold and in these cases not only an overhead occurs but also errors are made as Figure 2.16 illustrates.

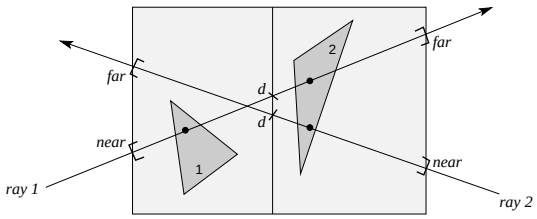


Figure 2.16: Example for a packet of two rays that cross a splitting plane in different directions causing the unmodified traversal algorithm to produce errors. The packet's traversal decision is (1,1,1) specifying to visit both children and enter the right one first. There the intersection with triangle 2 yields valid hit-points (since they are in the current voxel) for both rays and traversal terminates. But ray 1 would have pierced triangle 1 and terminated correctly if traversed in single ray traversal.

The problem of crossing rays can be addressed in two ways: First one could allow only those rays to be grouped in a packet that do not cross. Secondly the traversal algorithm could be changed in a way such that arbitrary rays can be packed together without leading to any errors.

Forcing Consistency

Following the first way requires to be able to detect rays that do cross splitting planes in different directions. After the detection of *invalid packets* they are split into several valid packets. For efficiency reasons detection should take place right after or even during ray generation.

But which packets are valid? Obviously those that only contain rays that have the same direction regarding to the splitting planes, i.e. since we use an axis-aligned kd-tree if the signs of the ray directions are pairwise equal. Furthermore rays that cross but are only defined on one side of the crossing, e.g. rays sharing the same origin, are also valid[9]. The Figures 2.17 and 2.18 illustrate those cases for secondary rays while Section 5 will address in general coherence properties when shading packets.

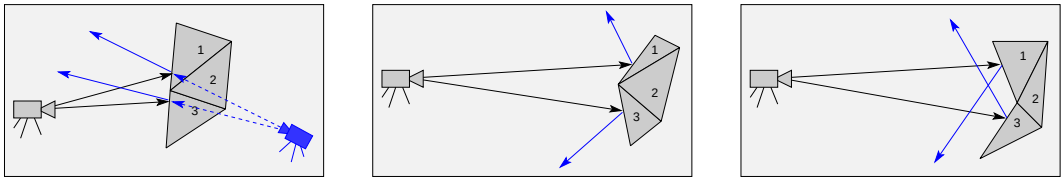


Figure 2.17: Tracing reflection rays splits into three cases (from left to right): the trivial case of a planar mirror which yields the same rays as if the camera would have been mirrored on the planar surface, reflections on strictly convex surfaces also do not lead to any problems (except for cases with numerical instabilities), but concave objects can cause rays that cross, which might lead to errors.

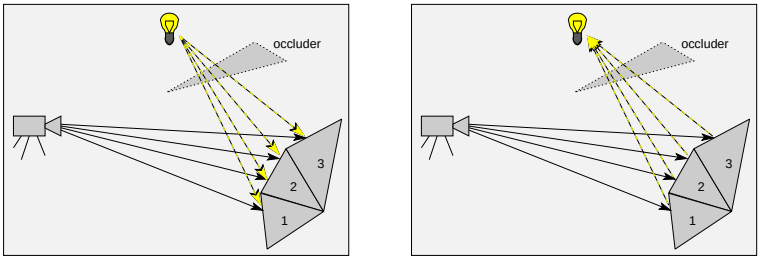


Figure 2.18: Tracing shadow rays of point light sources always generates valid packets of rays due to the limited interval the rays are defined in. Here a trade-off has to be made whether to trace the rays using the same origin (starting at the point light) or using the same destination (starting at the hit-points). Both cases have their advantages: if there is an occluder close to the light source in most cases it is cheaper to start at the light source, but if the light source is located in a different room data that would never have been access is loaded and might trash the cache.

[9] The OpenRT software ray tracer only supports rays with the same directions [Wal04] since SSE-instructions are rather inflexible and generating the traversal order of the packet is trivial if all rays have the same traversal order. For valid packets of crossing rays the traversal order has to be taken from any ray that wants to visit both children which is performed correctly when using the *OR*-ed traversal decision as presented above.

Especially on architectures built around large packets of rays splitting a single packet into several sparsely used packets due to issues of validity might have a severe impact on the performance. Thus adapting the algorithm to support invalid packets seems to be an interesting option.

Modified Traversal Algorithms

Unfortunately besides the problem of terminating too early (with hit-points that are actually not valid) there is another one: If during single ray traversal both children of a node want to be visited the voxel on the *far*-side is put to the stack and the *near*-side is entered. But when traversing invalid packets it might be necessary to put the *near*-side to the stack and enter the *far*-side first (see Figure 2.16). But this requires some adaptions to the stack-handling routines as not only the correct child has to be put to the stack but also when popping from the stack the correct values for *near* and *far* have to be restored[10]. In the following two variants are presented that handle invalid packets correctly.

Brute Force Consistency

The first algorithm also uses the detection of invalid packets during ray generation but does not change the packet. In case of a valid packet traversal is handled like before. For invalid packets traversal is also performed like before but early ray termination is omitted. Instead traversal continues until the stack is empty and all voxels throughout the scene pierced by the rays have been visited.

The performance of this brute force variant can be improved if any ray R_i is only intersected with a voxel if $near \leq f_i$ with $R_i(f_i)$ being the best hit-point already found for ray R_i. This improvement is valid since we never skip any voxel that could give a better hit-point than $R(f_i)$[11]. This improvement has no effect on the overhead in traversal but eliminates many unnecessary intersection operations.

Smart Consistency

The second variant to handle invalid packets does not require to detect the validity of a packet during ray generation. Instead traversal is performed as in the basic version but the first child *fc* given in the traversal decision is replaced by the two bits *lcf* (left child first) and *rcf* (right child first) specifying explicitly which child to visit first:

$$lcf = gl \wedge gr \wedge /fc \text{ and } rcf = gl \wedge gr \wedge fc = fc.$$

Again the *OR* over all traversal decisions forming the group's decision is calculated and additionally the flag $dirty = lcf_g \otimes rcf_g$. Then if *dirty*=0 traversal continues as before

[10] If not addressed properly situations like in Figure 2.16 might end up having *near=far* which does not allow for useful traversal as whole subtrees can not be entered anymore.

[11] By the way, it does not matter how we found the hit-point in the first place, i.e. whether the hit-point is contained in a voxel that was already visited or if the triangle with the hit-point is contained in several voxels and the hit-point was found during intersection of a voxel other than the one containing the part of the triangle with the hit-point. In the latter case it is guaranteed that we will also visit the voxel the hit-point is in since we visit all voxels pierced by the ray.

using rcf_g as the group's traversal decision. If *dirty*=1 then we define[12] that the group's decision is to enter the left child first and therefore every ray R_i with $rcf_i = 1$ becomes an invalid ray in this step[13].

Every stack operation additionally stores (or restores) for every ray its validity. Thus a ray that became invalid stays invalid until a valid state is popped from stack again. Valid and invalid rays do perform the same operations just as in the original traversal algorithm with the exception that early ray termination takes place only for valid rays.

This algorithm returns only valid results since it is guaranteed that every subtree or voxel that should have been visited but was not due to other rays forcing a different traversal order will be visited before termination. Since early ray termination is only checked after intersection of a voxel it is guaranteed that every node popped from stack (and maybe restoring validity of a ray) will be correctly visited before any termination is checked. The is important as the example in Figure 2.19 illustrates.

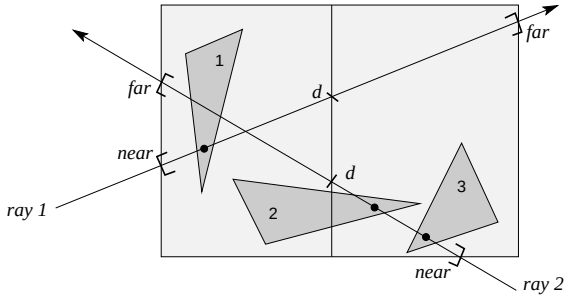


Figure 2.19: This example illustrates why termination of a ray may not be checked right after popping from stack but only after a voxel has been intersected using a valid state. The left child is entered first and the invalid ray 2 has a hit-point with triangle 2. When the right child is popped ray 2 becomes valid and the hit-point on triangle 2 becomes a valid criterion for early termination. If ray termination is checked before the right voxel was intersected triangle 3 will be missed.

Just like in the brute force variant the performance of this algorithm can be improved if any ray R_i is only intersected with a voxel if $near \leq f_i$ with $R_i(f_i)$ being the best hit-point already found for ray R_i.

2.3.2 Data Structures For Handling Packets

The previous section has shown that for every ray in the packet several flags signaling the state of the ray have to be kept. Since flags are of boolean type we can write every set of flags as a *bit-vector* which is a string $\{0|1\}^+$ where the letter i (counted from right to left, starting with 0) in the string denotes the state of ray i. Thus if referring to the state of ray i we simply write *bit-vector*[i]. Appendix B describes the operations performed on bit-vectors and presents their implementation details. According to the flags presented in the previous section we distinguish between several bit-vectors (Table 2.4).

[12] Instead of arbitrarily choosing the left side to be the valid side, a better way would be to count how many rays want to visit either side first and declare the side with the most votes to be the valid side.

[13] It is interesting to note that exactly for those rays that became invalid in the current step the *near*-side

active-vector	state of activity in current node/voxel
not-terminated-vector	state of termination
invalid-vector	state of invalidity
gl-vector	specifies which rays want to go left
gr-vector	specifies which rays want to go right
both-vector	specifies which rays want to go into both childen

Table 2.4: Most types of bit-vectors used in the SaarCOR architecture. Those vectors belong to the set of *ray-vectors* since they represent the state of rays. Besides ray-vectors there are also *packet-vectors* which specify the state of packets. Using a not-terminated-vector instead of a terminated-vector simplifies many computations, e.g. masking any active but terminated ray can be implemented using a single *AND* gate.

2.3.3 Implementation Details on Packets of Rays

This section presents a hardware optimized pseudo-code for traversing packets of rays. It illustrates in detail the various operations and the data management performed when tracing rays and thus gives an overview of the functional units that need to be build into hardware.

The implementation of transforming a ray and calculating the ray-triangle intersection is straight forward. For the sake of simplicity this pseudo-code only handles static scenes and the ray-triangle intersection is performed by the black-box instruction `intersect( triangle, ray )`. Furthermore the management of the stack is also specified using two black-box instructions `put_to_stack(` *variable* `)` and `get_from_stack(` *variable* `)`.

Typically in higher level languages memory accesses are performed implicitly by accessing arrays or components of objects to allow for compiler level optimizations. In the pseudo-code listed below all memory accesses are explicitly specified using `memory[` *address* `]`, which allows for quickly understanding the access behavior of the algorithm. For the sake of simplicity there is no support to handle invalid packets[14] .

The routine requires the rays including the corresponding active-vector to be initialized correctly. The data structures and other initializations are given at the beginning of the pseudo-code. Since for floating-point numbers multiplications are cheaper to perform than divisions the traversal loop is sped up by initially calculating the reciprocal of the ray's direction `ray[i].rcpdir = 1.0 / ray[i].direction`.

```
int i, j

bool go_left, go_right, fc
bool active-vector[n:0], not-terminated-vector[n:0], rtl[n:0], b[n:0]

float d[n:0], l[n:0], r[n:0], far[n:0], near[n:0]

struct Vector3D
{
```

has to be put to the stack. This simplifies case switching in the stack handling routines.

[14]There is also no support for mailboxing and empty voxels (see Section 2.4).

```
  float x,y,z  // coordinates in 3D space
}

struct ray-data
{
  Vector3D origin      // origin of ray
  Vector3D direction   // direction of ray
  Vector3D rcpdir      // reciprocal of direction
}

struct hit-information
{
  float distance       // f with R(f) specifies the hit-point
  int   triangle-ID    // ID of triangle
  float u,v            // barycentric coordinates
}
hit-information hit[n:0], pot-hit

struct kd-tree-node
{
  int   adr     // absolute address of kd-tree node
  int   lchild  // absolute address of left child
  int   rchild  // absolute address of right child
  int64 data    // 64 bit data stored in kd-tree node
                // contains: flag  is_inner-node
                // if inner node:
                //       float split : position of splitting plane
                //       flag  k     : splitting plane axis {x,y,z}
                //       int   ofs   : address offset to left child
                // if leaf node:
                //       int   items : number of triangles in voxel
                //       int   ofs   : address offset to list of triangle-IDs
}
kd-tree-node node
```

```
 1 traverse_packet(ray[n:0], active-vector[n:0], root_node_address)
 2 {
 3  node.adr = root_node_address
 4  not-terminated-vector[n:0] = active-vector[n:0]
 5  hit[n:0].distance = +infinity
 6  far[n:0] = +infinity
 7  near[n:0] = 0
 8
 9  while ( OR( not-terminated-vector[n:0] ) )
10  {
11    node.data = memory[ node.adr ] // Load node from memory
12
13    if ( node.data.is_inner-node ) // type of node is contained in node.data
14    {
15      // k={x,y,z}, split, and ofs are contained in node.data
16      node.lchild = node.adr + ofs                // address of left child
17      node.rchild = node.lchild + size_of( node ) // address of right child
18
19      fc = 0    // Init first child to be visited by packet
20      for i = all active rays in active-vector[n:0]
21      {
22          d[i] = ( split - ray[i].origin.k ) * ray[i].rcpdir.k
23          l[i] = SIGN( far[i] - d[i] )
24          r[i] = SIGN( d[i] - near[i] )
```

```
25          b    = NOT( l[i] ) AND NOT( r[i] )
26        rtl[i] = SIGN( ray[i].direction.k )
27          fc   = fc OR ( rtl[i] AND b )
28
29        gl[i] = ( l[i] AND NOT( rtl[i] ) ) OR ( r[i] AND rtl[i] ) OR b
30        gr[i] = ( r[i] AND NOT( rtl[i] ) ) OR ( l[i] AND rtl[i] ) OR b
31      }
32
33      go_left  = OR( gl[n:0] )
34      go_right = OR( gr[n:0] )
35
36      if ( go_right AND go_left ) // visit both children
37      {
38        both-vector[n:0] = gl[n:0] AND gr[n:0]
39        put_to_stack( both-vector[n:0] )
40
41        for i = all active rays in both-vector[n:0]
42        {
43          put_to_stack( far[i] )              // save value of far
44          far[i] = d[i]                       // update value of far
45        }
46
47        if ( fc ) // if first child is right child
48        {
49          put_to_stack( gl[n:0] )      // save active-vector of left child
50          put_to_stack( node.lchild )  // save pointer of left child
51          active-vector[n:0] = gr[n:0] // set new active-vector to right child
52          node.adr = node.rchild       // set current pointer to right child
53        }
54        else
55        {
56          put_to_stack( gr[n:0] )      // save active-vector of right child
57          put_to_stack( node.rchild )  // save pointer of right child
58          active-vector[n:0] = gl[n:0] // set new active-vector to left child
59          node.adr = node.lchild       // set current pointer to left child
60        }
61      }
62      else
63      {
64        if ( go_left ) // visit left child only
65        {
66          active-vector[n:0] = gl[n:0] // set new active-vector to left child
67          node.adr = node.lchild       // set current pointer to left child
68        }
69
70        if ( go_right ) // visit right child only
71        {
72          active-vector[n:0] = gr[n:0] // set new active vector to right child
73          node.adr = node.rchild       // set current pointer to right child
74        }
75      }
76    }
77
78    else // node is leaf: Intersect all triangles in voxel with all active rays
79    {
80      // items_in_voxel and ofs are contained in node.data
81      tri_list.adr = node.adr + ofs    // pointer to list of triangles in voxel
82
83      for j = 0 to items_in_voxel // for all triangles in list
84      {
```

```
 85        tri_id = memory[ tri_list.adr + j ] // Load ID of triangle from memory
 86        triangle = memory[ tri_id ]         // Load triangle from memory
 87
 88        for i = all active rays in active-vector[n:0]
 89        {
 90          pot_hit = intersect( triangle, ray[i] )
 91          if ( NOT( SIGN( pot_hit.distance ) )  // if pot_hit is valid
 92          {
 93            if ( SIGN( pot_hit.distance - hit[i].distance )
 94               hit[i]=pot_hit // keep intersection with closest distance
 95          }
 96        }
 97      }
 98
 99      // Check for termination of rays
100      for i = all active rays in active-vector[n:0]
101        not-terminated[i] = SIGN( far[i] - hit[i].distance )
102
103      if ( OR( not-terminated-vector[n:0] ) )  // Pop from stack if not terminated
104      {
105        active-vector[n:0] = 0 // guarantees that the while-loop is entered
106        while ( NOR( active-vector[n:0] ) AND OR( not-terminated-vector[n:0] ) )
107        {
108         if ( stack_is_empty ) not-terminated-vector[n:0] = 0
109         else
110         {
111           get_from_stack( node.adr )
112           get_from_stack( both-vector[n:0] )
113           for i = all active rays in both-vector[n:0]
114           {
115             near[i] = far[i]
116             get_from_stack( far[i] )
117           }
118           get_from_stack( active-vector[n:0] )
119           active-vector[n:0] = active-vector[n:0] AND not-terminated-vector[n:0]
120         }
121        }
122      }
123    }
124  }
125  return(hit[n:0]) // return hit-results
126 }
```

Details on the Pseudo-Code

In the pseudo-code above besides the case switching depending on the traversal decision there are hardly any `if` statements. This is achieved by adjusting the algorithm such that special cases are correctly handled by default. The most important example is the handling of termination and the efficiency by not visiting unnecessary nodes.

Both is achieved using only two lines of code: In line 4 the `not-terminated-vector` is initialized with the `active-vector` and therefore guarantees that we never wait for a ray to finish that was never active. Line 119 masks rays inactive that were active when the stack was written but have been terminated in the meantime. Obviously this masking can result in having no active ray in the current step which requires further popping from the stack.

Care should be taken of numerical inaccuracies especially in the calculations of `l[i]` and `r[i]` (lines 23 and 24), and in the test for termination `far[i]>hit[i].distance` (line 101). To avoid visible errors, a small $\epsilon > 0$ should be used as a security distance. Please note that ϵ is not a general constant but dependent on the scene (or more exactly on the dimensions of the object being traversed). Section 5.3.5 shows how floating-point comparisons can be implemented with little hardware efforts.

2.4 Optimizations

So far this chapter presented the basic algorithms for ray tracing, some implementation details and several improvements, which are used in the SaarCOR hardware architecture. Further discussion on fast ray tracing in general can be found in [Wal04, Hav01]. Finally, there are two more optimizations important for efficient hardware implementations which are summarized in the following.

2.4.1 Mailboxing

Due to the construction of kd-trees, a single item, i.e. a triangle or a rigid object, might overlap several voxels (for example see Figures 2.9 and 2.3). Thus during traversal the same triangle respectively object might be found and intersected several times, which obviously causes overhead. A technique to avoid those multiple intersections is called *mailboxing* [AW87, Gla89].

The general idea behind mailboxing is caching for every ray the triangles respectively objects that have been intersected. Then before performing an intersection it is checked whether the item has been intersected already. However, there are several ways to manage the data structure of the cache.

For software ray tracers on standard computers mailboxing can be implemented by numbering all rays and storing for each item the number of the ray intersected last. While this easy solution gives a good performance it has the disadvantage that the data structures of the scene are not read-only anymore, which breaks on parallel computers.

Alternatively to storing the number of the ray in the data structure of the triangle respectively object the numbers of the items are stored in the data structure of the ray [Wal04]. Therefore when intersecting with an item the whole list of previously intersected items has to be checked. While this works very well in parallel software solutions, a hardware implementation suffers from the fact that those lists could reach arbitrary lengths (worst case: the number of items in the scene).

A Hardware Implementation of Mailboxing

Therefore the hardware implementation of mailboxing used in the SaarCOR architecture supports only limited lengths of lists. This reduces the amount of on-chip memory, removes the requirement for external data paths for swapping or paging and gives a fixed worst case time to check whether an item is in the list[15].

[15] In hardware lists can be checked either sequentially or in parallel while the latter variant is only feasible for lists with few items. These lists can be implemented as a k-way set associative cache where a full associativity is equal to a standard list and lower associativities obviously result in lower efficiencies

But since the lists can store only a limited amount of previously intersected items in general not all unnecessary intersections can be removed. Furthermore a policy is needed on how to manage, which items are kept in the list.

Experiments with the SaarCOR prototype (see Chapter 7) have shown that a round robin replacement strategy performs well in scenes where only short sequences of pairwisely identical items occur (e.g. when traversing a single triangle mesh). On the other hand scenes consisting of several nested objects are handled very well by using the "full-is-full" strategy, which means that new items are only added to the list if there is space left. Since larger objects surrounding smaller ones are visited earlier they are likely to be cached in the mailbox. This allows for removing the most expensive intersections of large objects. Examples for these replacement strategies are given in Figure 2.20.

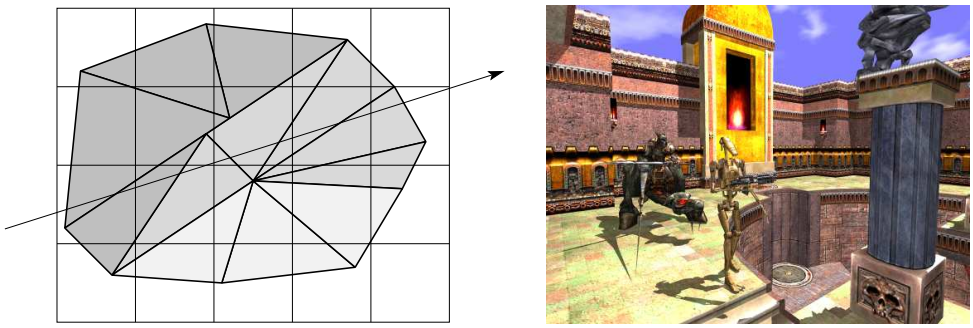


Figure 2.20: The left image shows an example of a standard triangle mesh for which the round robin replacement strategy for mailboxing typically gives the best results. According to our experiments the "full-is-full" strategy performs best for dynamic scenes with nested objects like in the image on the right where the whole building is a single rigid object with several monsters running around holding guns (modeled as separate objects) in their hands. (The image uses geometry and textures from *Quake3* [IS04] and some additional monsters.)

Instead of using linear lists to store the IDs (i.e. numbers or addresses) of the previously intersected items also standard caching algorithms could be used. Those algorithms give a constant time to check whether an item had been intersected already but since they use a hashing scheme to sort the IDs of the items into cache memory the quality of mailboxing depends on the ID an item has. However, in general not all available cache memory is used due to the hashing scheme, which simply does not map to all memory cells equally. Thus a hashed scheme for mailboxing becomes efficient (in terms of logic resources) for large lists, but for lists with only few items linear lists are better (if their implementation is feasible).

Details on the Implemention

Since typically a mailbox with rather few items (e.g. four or eight) already performs very well (see Section 2.5 and Chapter 8) the SaarCOR architecture uses linear lists for mailboxing. This allows for a parallel implementation that has the advantage that initialization and check for an item can be performed in a single cycle.

due to conflicts in address mapping.

However, since packets of rays are used, per packet not only the number of the item intersected last has to be stored but also the active-vector that was valid during this intersection. If an item needs to be intersected with a packet and the item was found in the mailbox the current active-vector is compared to the stored active-vector and only those rays are intersected which are currently active but were not during the previous intersections.

The implementation of this check is trivial in hardware when the active-vector is stored as string-encoded bit-vector (see Appendix B): the current active-vector and the inverse of the previous active-vector are *AND*-ed yielding the active-vector for the current intersection. If the resulting active-vector does not contain any active rays the intersection of this item can be skipped and otherwise only the rays in the new active-vector have to be intersected. In the latter case also the active-vector in the mailbox is updated.

2.4.2 Empty Voxels

Section 2.1.1 has shown that good kd-trees have many empty voxels to skip empty space in the scene fast and efficiently. The next section evaluates several high-level characteristics of the ray tracing algorithm on several scenes. There the example of a typical game-like scene *UTID* (Figure 2.22) will show that during rendering between 76% and 87%[16] of all visited voxels do not contain any triangles. However, visiting an empty voxel still requires some amount of bandwidth and work but does not yield any new results. For example in the *UTID* scene this wasted bandwidth is up to twice the bandwidth for reading the lists of triangle Ids and accounts for up to 11% of the total bandwidth to the caches.

In particular before entering a voxel in most cases the other child is stored on the stack, then the empty voxel is fetched, intersected and finally after popping from the stack we are at the point we should have visited in the first place. Especially in a hardware architecture with dedicated units for traversal and intersection empty voxels cause long latencies and even require special logic to handle cases where a packet of rays is passed to intersection but there is nothing to intersect with.

Therefore if we knew in advance that the voxel is empty we would not visit it at all. Thus efficient support for empty voxels can be achieved trivially by adding two flags to the data structure of the node denoting whether the node has a left respectively a right child (see Figure 2.12). Then to calculate the corrected traversal decision simple set

$$gl'_g = gl_g \wedge \mathit{has\_left\_child} \text{ and } gr'_g = gr_g \wedge \mathit{has\_right\_child}.$$

Special Cases

Unfortunately skipping voxels requires to adapt the interval [*near*,*far*] like it would have been adapted when entering and leaving the empty voxel. If the interval is not correctly adapted no errors are made but too many operations are performed. Figure 2.21 lists all cases and how they should be handled. Obviously cases where only one child would have been visited but was skipped due to being an empty voxel are correctly handled by default since those cases do not update the interval.

[16] This figure obviously depends on the packet size since rendering with larger packets visits fewer nodes in total which changes the ratio. Therefore with 64 rays per packet "only" 76% of all visited voxels are empty.

Due to the alignment properties of kd-trees (see Section 2.1.3) the memory requirements of the kd-tree are not reduced by handling empty voxels using flags in the parent node. This is due to the fact that always records of left and right children are stored together and if only one child is an empty voxel then the size of the record does not change. If both children are empty voxels then one could omit the record and actually save space in the kd-tree but then already the parent node would have been a waste.

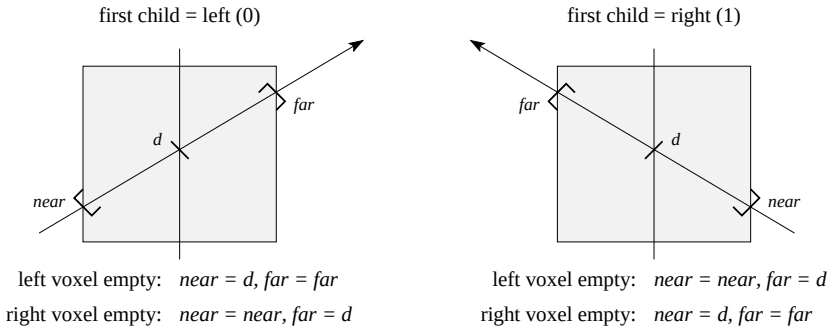


Figure 2.21: All cases of empty voxels that update the interval of [*near*,*far*]. Since without empty voxel handling only the assignments *near*=*far* and *far*=*d* need to be handled extending the hardware to also allow for setting *near*=*d* requires new logic, additional data paths, and ports on the register files. This is the reason why the SaarCOR prototype (see Chapter 7) has only efficient support for empty voxels on the *far*-side which can be handled without new data paths and write ports.

2.5 Profiling kd-Trees And Packets of Rays

So far this chapter presented algorithms for ray tracing in great detail. But before these algorithms can be implemented in hardware it is important to determine some of their characteristics. For example when designing an architecture it is vital to have some figures on how many traversal and ray-triangle intersection operations need to be performed to render benchmark scenes. Additionally, having estimates for the requirements of the memory interface allows for speeding up the development cycle.

Therefore this section presents an implementation independent evaluation of the ray tracing algorithms and shows how to profile the algorithms and the quality of the spatial index structures. These evaluations lead to guidelines on how to build a hardware architecture for ray tracing, which will be summarized in the last paragraph of this section.

Overview

For the following evaluations only a single representative benchmark scene was selected, which has average complexity and was taken from a current computer game. Additionally, two standard scenes of lower and higher complexity will be evaluated in selected measurements to give a wider overview. Details on those and all of the 20 benchmark scenes can be found in Chapter 8.

For the sake of simplicity only static scenes are evaluated since dynamic scenes are handled fully analogous but offer more parameters. All renderings are in 1024×768 pixels using various packet sizes of powers of two.

The scene examined in detail is *UTID* taken from *Unreal Tournament 2004* [Dig04] consisting of 465 000 triangles in a single rigid object with a good kd-tree (see Section 2.1.1) and rendered using primary rays only (Figure 2.22). The other two scenes additionally have secondary rays and use standard kd-trees.

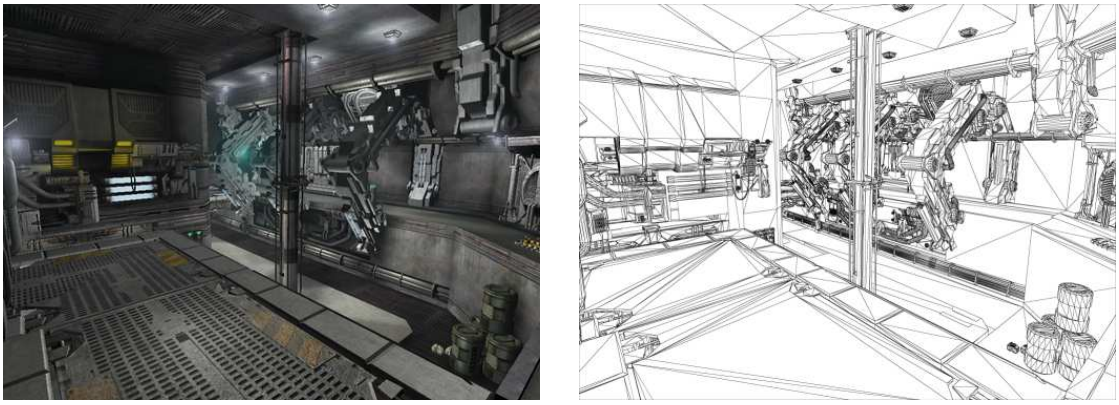


Figure 2.22: Original in-game screenshot (OpenGL) and wire-frame rendering of the *UTID* benchmark scene rendered in 1024×768 pixels.

Full Frame Statistics

Like any rendering system also the performance of a ray tracing system is measured in *frames per second* (*fps*). But since performance scales linearly with the number of rays shot the *rays per second* (*rps*) ratio gives a much more detailed measurement[17].

However, *fps* and *rps* are only the results of the interaction of many parameters. Thus more details on the computational complexity can be gained by counting the number of *traversal operations* (*t-ops*) and *intersection operations* (*i-ops*) performed during rendering of an image. Obviously *t-ops* and *i-ops* are independent on the number of rays in a packet since using an active-vector allows for avoiding computational overhead.

The memory accesses performed during rendering of a frame can be determined by counting the number of operations performed by packets since for a packet of rays every node respectively triangle is only fetched once. These figures combined with the size of the corresponding data structures gives the raw bandwidth requirements.

Per Pixel Statistics

While full frame statistics give a nice overview of the complexity of a benchmark view for detailed analysis these measurements are to coarse. A typical way to quickly get a visual judgement on the complexity of a scene is performing a wire frame rendering (Figure 2.22) and checking how many pixels are covered by large respectively small triangles. This is

[17] This is analog to rasterization based graphics cards which performance is measured in *triangles per second* and *pixels per second* (aka. the *fill-rate*).

equal to counting the number of different triangles that can be seen divided by the number of pixel of the image but additionally gives an impression which regions have the highest complexity.

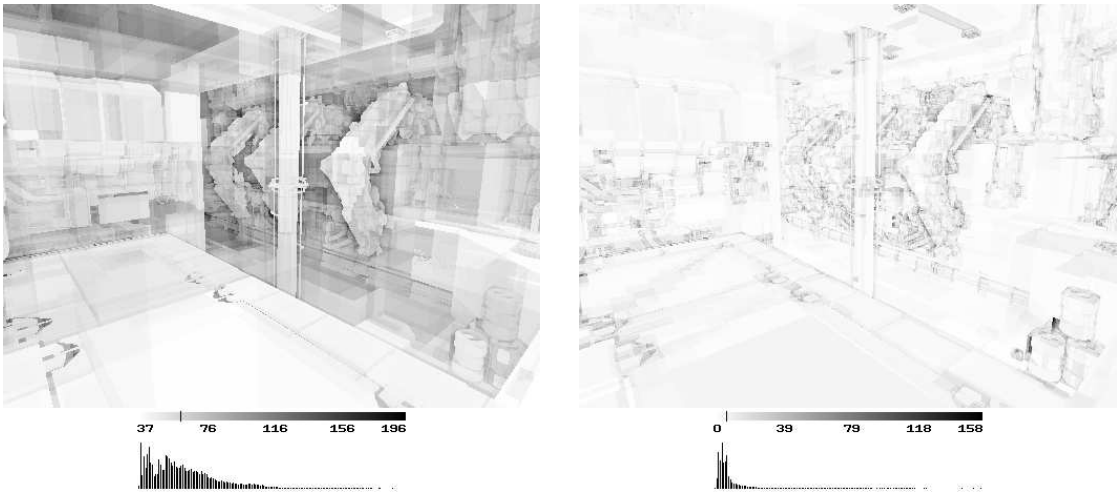


Figure 2.23: Cost images of a good kd-tree with low cost for traversal (left image) and intersections (right image). The darker a pixel is colored the more operations had to be performed to compute it. The histograms give an overview of the distribution of costs (measured in number of operations) while the bar on the scale marks the average workload.

More accurate judgements can be derived by using *cost images* representing for every pixel the number of *t-ops* respectively *i-ops* performed during rendering (Figure 2.23). These cost images can be used to locate areas causing performance deterioration during the design phase of the scene. Furthermore they allow to check the quality of the algorithm used to build the spatial index structure and to debug the traversal algorithm with all its special cases especially when using packets of rays. In particular for complex scenes with many possible secondary rays per pixel cost images are a nice tool to efficiently debug the shader programs.

Memory Bandwidth

When using packets of n rays the required bandwidth is reduced by a factor of n in the best case, but coherence decreases with increasing size of a packet (if the image resolution is not adapted accordingly). Therefore in the *UTID* scene packets of four rays yield a reduction of factor 3.6 to 28% of the bandwidth required when tracing single rays, but using packets of 64 rays only reduces by a factor of 31 which is 3% of the single ray bandwidth (see Table 2.5). Figure 2.24 shows graphically for all three scenes the bandwidth depending on the size of the packet. The right image additionally shows the average amount of memory requested by a packet which gives a rough estimate of the minimum size a cache should have.

Number n of Rays per Packet	Bandwidth [in MB]	Compared to bandwidth for ray tracing of single rays		Compared to bandwidth for packets with $n/4$ rays	
		Percentage	Reduction	Reduction	Red. [in %]
1	602.3	100.0%	–	–	–
4	168.2	27.9%	3.58	3.58	89.5%
16	52.1	8.7%	11.56	3.23	80.8%
64	19.2	3.2%	31.34	2.71	67.8%
256	9.1	1.5%	66.12	2.11	52.8%
1024	5.7	0.9%	105.79	1.60	40.0%
4096	4.4	0.7%	137.53	1.30	32.5%

Table 2.5: This table lists depending on the size of the packet the bandwidth required and the corresponding reduction. While the bandwidth decreases continually with increasing size of the packet the relative reduction achieved by using four times larger packets decreases drastically.

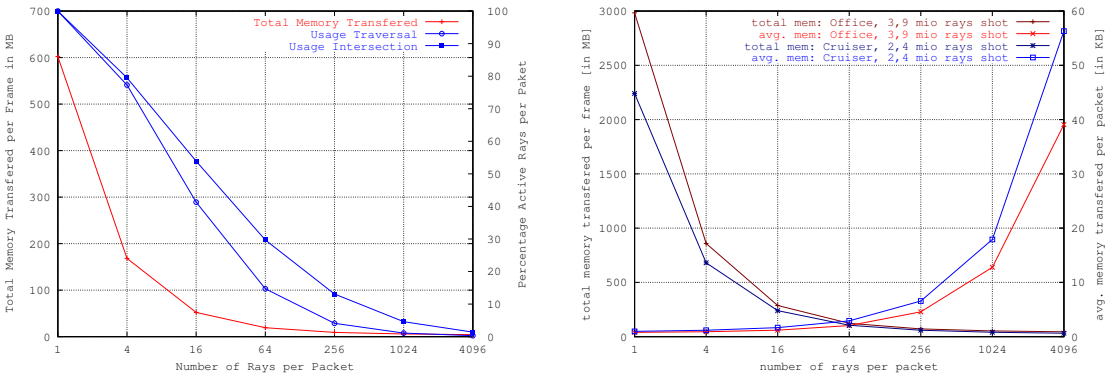


Figure 2.24: All three scenes (*UTID* left, *Office*, and *Cruiser* on the right) and their required memory bandwidth depending on the size of the packet. Additionally, the left image shows the activity of the packet split into *t-ops* and *i-ops* while the right image shows the average amount of memory requested per packet giving a rough estimate for the minimum size of the cache.

Coherence and Usage

The fact that coherence decreases with increasing size of the packets does not only limit the achievable reduction of bandwidth. It also influences the *activity*, i.e. the number of rays per packet that want to take part in the current operation of the packet (left image of Figure 2.24).

On systems that are built like SIMD machines and operate on all rays of a packet in parallel results of inactive rays have to be masked and thrown away. Thus a low activity results in a low usage and a waste of system resources on those machines.

For example when rendering the *UTID* scene with packets of 64 rays on average 54 rays take part in a traversal step of the packet but only 20 rays want to intersect the same triangle (Table 2.6). Thus the average activity becomes rather low especially for intersection operations on larger packets. Using *traversal depth statistics* (TDS) gives a better insight why this happens by plotting the *t-ops* and *i-ops* performed depending on the depth of the traversal in the kd-tree (Figures 2.25 and 2.26).

Number of Rays per Packet	Activity of Traversal [in %]	Traversal [in rays]	Intersection [in %]	Intersection [in rays]
1	100.0	1.0	100.0	1.0
4	97.5	3.9	79.5	3.2
16	92.6	14.8	53.9	8.6
64	84.0	53.8	29.7	19.0
256	69.0	176.6	13.1	33.5
1024	46.7	478.2	4.6	47.1
4096	24.2	991.2	1.4	57.3

Table 2.6: Depending on the number of rays per packet this table lists for the *UTID* scene the activity of the packet split into operations for traversal and intersection. The activity is given in percent of the packet and also as absolute figures of how many rays want to visit the same node respectively triangle on average.

Profiling the Spatial Index Structure

Traversal depth statistics allow to evaluate and debug algorithms that build kd-trees. Furthermore they can be used to adapt the parameters of the kd-tree to specific needs. For example on a hardware architecture there might be a limit in the maximum depth for kd-trees that can be handled due to a fixed amount of memory for stacks. Simply cutting the tree at the maximum depth results in voxels containing too many triangles which can be seen in TDS as a spike in the *i-ops* graph.

Any set of parameters for the kd-tree results in a specific *ratio of traversal to intersection operations* (*ti-ratio*) that need to be performed for rendering a benchmark view. As a simple example Figure 2.26 shows how that ratio changes depending on the maximum allowed depth of the kd-tree. This illustrates how the *ti-ratio* can be used to gain the best performance on a system that can perform traversal operations n-times faster than intersections simply by adjusting the *ti-ratio* to $n : 1$.

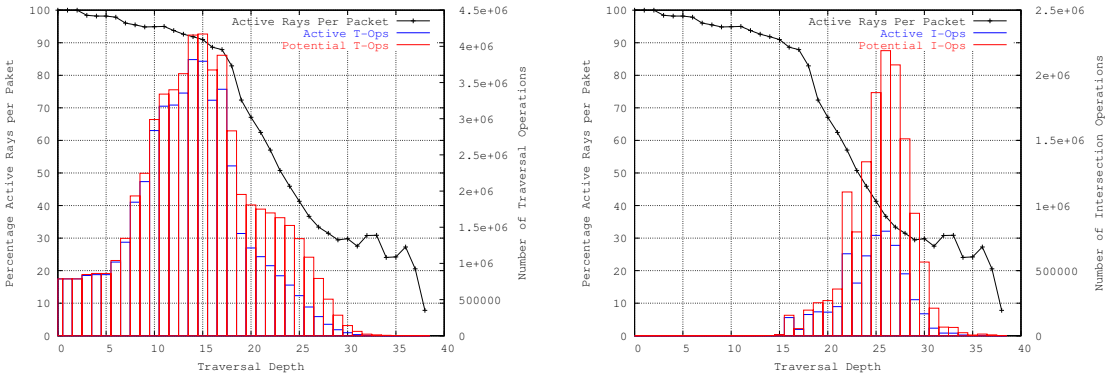


Figure 2.25: Traversal depth statistics for packets of 64 rays in the *UTID* scene. It shows that especially for intersections the actual number of operations performed (the active operations) is much smaller than the number of operations that could be performed (the potential operations) if all rays of the packet would take part in the same computations. This yields a low activity ratio and therefore a low usage of SIMD-like units (also depicted in the graphs). Please note that the number of traversal operations performed per depth is not strictly declining although all rays are active in the root node (depth 0) and are masked out when descending the tree. However, a typical traversal strictly descends for several steps till the first leaf is reached and then continues more or less horizontally from leaf to leaf. Additionally, note that the kd-trees are typically not balanced.

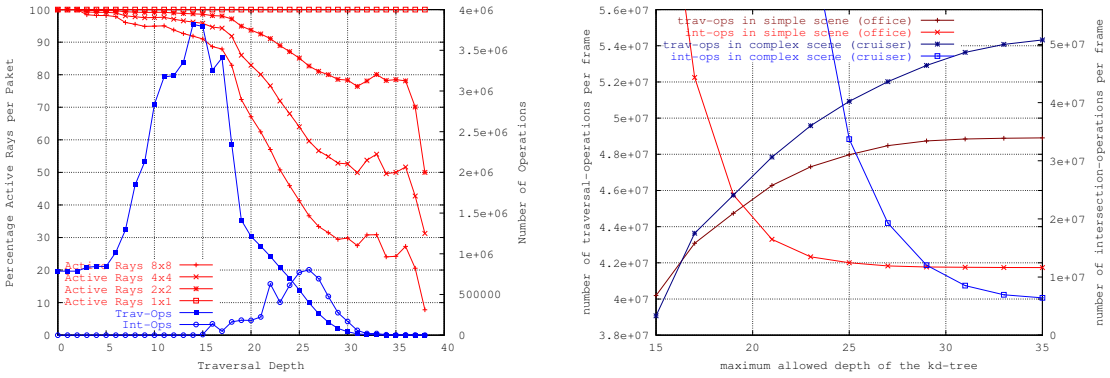


Figure 2.26: The left image shows the traversal depth statistics in the *UTID* scene for several sizes of packets and the actual number of operations performed per depth. The right image shows for the *Office* and *Cruiser* scenes how the ratio of traversal to intersection operations performed changes depending on the maximum depth of the kd-tree. Although the maximum depth is only one of the many parameters of a kd-tree this simple example shows how the *ti-ratio* can be tuned by adjusting the parameters of the kd-tree.

Mailboxing

The measurements presented so far did not include the effect of mailboxing which can reduce the number of *i-ops* by avoiding multiple intersections of the same triangle with a ray. The amount of savings achieved by mailboxing depends on the kd-tree. Our measurements use two different heuristics for building kd-trees called *standard* and *good* (see Section 2.1.1), that were optimized to achieve the best frame rate and the lowest number of *t-ops* and *i-ops*.

For all of our benchmark scenes (see Chapter 8) using standard kd-trees mailboxing can save up to 40% of the ray-triangle intersections, but when using good kd-trees only up to 5% can be saved. The reason for the low percentage of *i-ops* that can be saved in a good kd-tree is that there the splitting planes are positioned such that only few triangles overlap several voxels. This positioning also reduces the total number of ray-triangle intersections regardless of mailboxing and therefore gives a good kd-tree.

Using mailboxing obviously changes the number of *i-ops* presented in the figures and graphs above and further also leads to a change in the *ti-ratio*. However, the *t-ops* stay unchanged and empirically also the *activity* is unaltered since in most cases a triangle is intersected either by none of the rays of a packet or by all of them simultaneously (see Figure 2.27).

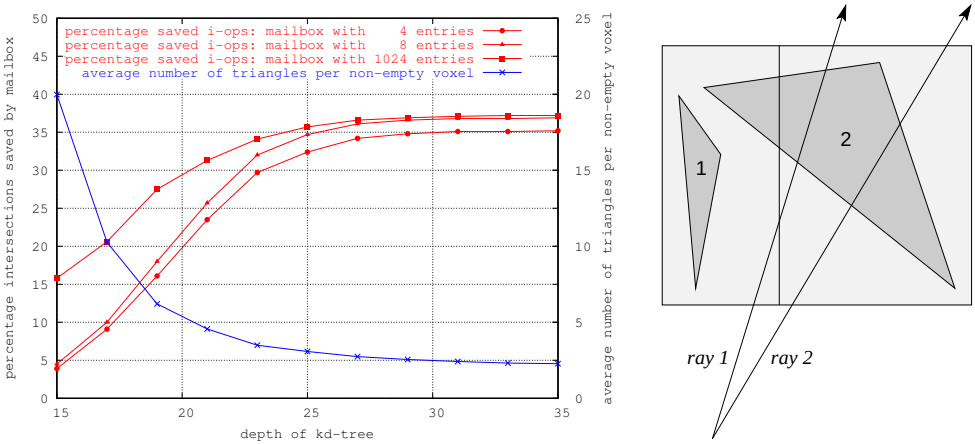


Figure 2.27: The graph on the left side shows how the number of triangles per non-empty voxel depends on the depth of the kd-tree for the office scene using a standard kd-tree. This figure can be used as a guideline on how many items a mailbox should store to be useful. Additionally, the graph depicts the percentage of ray-triangle intersections that could be saved by using a mailbox with 4, 8, and 1024 items respectively. The figure on the right illustrates why the *activity* of a packet is typically unchanged by using mailboxing. Only if there are triangles that overlap several voxels (which a good kd-tree tries to avoid) and a packet visits both voxels the activity may change. In the left voxel only ray 1 is active, giving an activity of 50%. Using mailboxing in the right voxel only ray 2 is active giving again an activity of 50%, but without mailboxing both rays are active yielding 100% activity. However, although the activity might be lower using mailboxing it is always a good choice not to perform unnecessary operations since this can shorten the latency and save power by switching off idle units.

However, supporting mailboxing is useful even if the kd-trees of all geometric objects are good. When ray tracing dynamic scenes typically the kd-trees of meta objects have to be changed in every frame. This puts hard constraints on the algorithm that builds kd-trees and does not allow for complex optimizations. So typically there are rigid objects overlapping several voxels and thus mailboxing is very important especially since intersecting an object is much more costly than ray-triangle intersections.

Additional Advantages of Packets of Rays

Using packets of rays not only reduces the bandwidth to memory but also allows to reduce the size of the stack for traversal since the address of the node has to be stored only once per packet. Thus when using 32 bits for addresses and floating-point values then for each packet of n rays every entry on the stack requires $32 \cdot (n+1) + 2 \cdot (n-1)$ bits. The extra two bits per ray are used for the *active* and *both vectors* required on packets with more than a single ray.

Thus when storing the same number of rays on a stack the costs are reduced to 76% for packets with 4 rays and to only 54% for packets of 64 rays compared to the requirements for single rays. This is important since besides the cache the stack accounts for the majority of on-chip memory requirements (see Section 8.1).

Summary and Guidelines

This section has shown that exploiting the *coherence between rays* by using packets of rays allows for drastic reductions of the required bandwidth. For example, if the *UTID* scene should be rendered with 25*fps* and three light sources using four rays per paket requires a bandwidth of 16 GB/s. While this is rather low compared to 59 GB/s required for tracing single rays, when using packets of 64 rays less then 2 GB/s are required which greatly simplifies the memory interface.

However, these bandwidth estimates did not take caching into account, which is likely to decrease the required bandwidth even further. The cost images have shown that typically only few nodes and triangles are fetched by a packet and therefore even a small cache should allow to exploit the *coherence between packets*.

The many *parameters of kd-trees* allow for optimizations in two ways: first by looking on the costs and figures of traversal steps and ray-triangle intersections a compromise for the number of dedicated functional unit for these operations can be found. Then when building a kd-tree for a specific hardware architecture the *ti-ratio* can be adjusted to suit this architecture best.

Using *mailboxing* may yield significant savings if only standard optimizations for kd-trees can be used. However, for good optimized kd-trees mailboxing hardly pays off. Thus, for geometric objects the hardware and memory for mailboxing should be saved and invested on the implementation of mailboxing only for meta objects.

Unfortunately, using larger packets also has negative effects as it decreases the activity, resulting in a low usage of dedicated functional units. Therefore the number of dedicated units should be at most equal to the *activity of a packet* measured in rays (see Table 2.6). This typically leads to having *less functional units than rays per packet* and therefore requires *sequential execution* of the rays of a packet on the functional units. Therefore an

efficient *scheme to distribute only the active rays* of a packet to dedicated units is needed (see Section 4.2.1).

Using large packets for rendering incoherent scenes can cause another bottleneck. Since in those scenes typically only few rays are active per packet there might not be enough work to hide the computational and memory access latencies. Using the same number of rays in the system distributed in smaller packets allows for more threads and therefore to hide more of the latencies. This results in a higher usage of the functional units and – most important for incoherent scenes – also a higher (although less optimal) usage of the memory interface.

However, since smaller packets do not allow for great reductions in bandwidth a compromise has to be chosen. Alternatively the ray tracing algorithms can be extended by using *flexible packets of rays* which allows to adjust the number of rays per packet on-the-fly (see Section 2.6).

Finally, packets of rays also allow to *lower the requirements of on-chip memory* since the traversal stack can be shared between all rays. The pros and contras of packets of rays are summarized in Table 2.7.

Packets of rays ...

+	reduce the number of memory requests and thus the internal bandwidth.
+	reduce the external bandwidth to memory (details in Chapter 4).
+	allow for efficient hiding of various and long latencies.
+	allow for having multiple functional units sharing the connection to memory due to low bandwidth requirements and efficient latency hiding (Chapter 3).
+	reduce the internal storage requirements for stacks.
+	with valid rays do not cause any arithmetical overhead.
+	including their management are easy to implement.
+	allow for using super-scalar functional units (details in Chapter 3).
+	allow for simple connection schemes and narrow busses (Chapter 3).
–	can yield a low usage in incoherent scenes.
–	can reduce the performance in bandwidth limited scenes.
–	can cause arithmetic overhead for invalid packets.

Table 2.7: The pros and contras of packets of rays

2.6 Future Work: Flexible Packets

Packets with many rays allow for great savings in memory bandwidth but with increasing size of the packet also the overhead increases when rendering less coherent scenes. Another interesting fact is that the coherence of a packet with 2×2 rays rendering images of 512×384 pixels is roughly the same as rendering images of 1024×768 pixels using packets with 4×4 rays.

Therefore rather than building a hardware that supports only a fixed number of rays per packet this section presents a method called *flexible packets* that allows for adjusting the number of rays per packet on-the-fly. This adjustment can be based on any heuristic, e.g. the current position in the kd-tree or the number of currently active rays, or simply

depending on the image resolution or scene complexity.

Since large packets are beneficial but the limited resources of chips allow only for few functional units, typically packets of x rays are computed sequentially on n functional units in x/n steps (with $x > n$). However, the *flexible packets* technique can be used on any set of functional unit regardless of the number of parallel sub-units (as long as the smallest supported packet size is not less than n) and whether they are fixed function or fully programmable.

For a packet of rays as well as when tracing single rays the operations are always performed individually per ray. Therefore the use of packets of rays has only an impact on the bandwidth to memory. This makes clear why the size of the packet influences the algorithm only at two positions: when calculating the group's traversal decision and when performing memory accesses.

Data Structures

Therefore in principle it does not matter whether a system is designed to allow for splitting up large packets into small packets or vice versa. They both only differ in the data structures used (see Figure 2.28). However, due to some implementation details of the traversal stack it is generally easier to group small packets into larger ones.

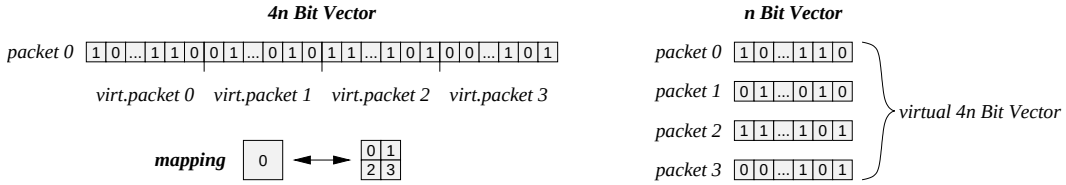


Figure 2.28: Data structures for managing the splitting of large packets into smaller packets respectively grouping of small packets into larger packets.

Obviously it is only feasible to support a discreet set of packet sizes. For example such a set could contain packets with 4, 16, 64 rays. In that case all sizes for packets of great relevance would be supported (see Section 2.5). Such a discreet set has requirements of internal memory of stacks equal to the smallest packet size supported and allows in the best case for bandwidth reductions equal to the largest supported packet (see Table 2.8). In the same spirit as flexible packets Section 4.1 presents a scheme to dynamically manage the memory of stacks.

	small packets	large packets	flexible packets
reduction of bandwidth		×	×
activity of packet	×		×
reduction of stack size		×	

Table 2.8: This chart presents a comparison of the features for the various types of packets. It shows that flexible packets are a good compromise yielding a good reduction of the bandwidth and a high activity of the packet.

Furthermore the design of the architecture is greatly simplified if grouping and splitting

is only supported for whole packets of rays. That means either none or all four small packets are grouped into a larger packet but never only 1, 2, or 3 packets are grouped. Supporting packets sizes 4, 16, and 64 additionally requires two bits[18] per thread ID to denote the level of grouping (where level 0 specifies no grouping using packets with 4 rays).

Communication Scheme

In general all operations are performed as when using static packet sizes. However, if a packet is grouped from several smaller packets between units only thread IDs of grouped packets are transfered. The following describes this communication scheme in more details using circuits and terms explained in Chapter 4. However, the general idea can be understood without exact knowledge of these circuits.

When the traversal unit receives a grouped ID scheduling the work to the traversal sub-units (the slices) requires sequential *decompression* of the single grouped ID into the corresponding IDs of the smaller packets. The results of all packets are then gathered by the circuit Global, which calculates the traversal decision for the grouped packet. Other units like the ray-triangle intersection unit perform grouping and regrouping similar to the traversal slices. In contrast, units like the Traversal Memory Interface always work only on the grouped IDs.

Thus in summary the implementation of flexible packets is rather trivial but nevertheless allows for significant improvements in bandwidth reduction and activity. Therefore *flexible packets of rays* allow to overcome the problem of either being able to efficiently render incoherent scenes or to allow for a low bandwidth to memory.

[18] Independent on the sizes of packets that are grouped, this requirement can be reduced down to a single bit denoting whether grouping is used at all. If grouping is used the lowest bits of the thread ID are not used and therefore can be used to store the additional bits denoting the level of grouping.

3 Overview of the Architecture

The previous chapter presented algorithms for ray tracing of static and dynamic scenes with rigid objects. Besides detailed discussions of hardware oriented implementations also careful investigations about the requirements have been made. These measurements have led to estimates for the number of operations that need to be performed by an hardware architecture for realtime ray tracing.

Each of these operations can be implemented either using dedicated hardware or using a more general programmable unit. While multi-purpose units are more flexible and allow for easier load-balancing to achieve the same performance in general they also require more area on the chip than a dedicated hardware unit.

Therefore in the SaarCOR project several variants of the architecture have been developed and evaluated: the *fixed function SaarCOR* variant which gives the best performance, the *programmable shading SaarCOR* variant which uses fixed function ray tracing units and programmable shading for highest image quality, and the *fully programmable SaarCOR* variant on which the whole ray tracing as well as the shading algorithms are executed on general purpose processors.

The next chapter explains in great detail the fixed function components for ray tracing before in Chapter 5 the fixed function and the programmable variants of shading are presented. In that chapter also a new processor is described and extended to support traversal and intersection operations.

But first this chapter discusses the design decisions that led from the algorithms to the SaarCOR hardware architecture. Then the key features of the architecture are summarized and an overview of the architecture and its components is given.

3.1 Design Decisions

In this section the design decisions used in the SaarCOR architecture are presented which are based on the guidelines developed in Section 2.5.

Functional Units

Turning descriptions of algorithms or programs into hardware units allows for many options and optimizations but generally it can be handled in an easy way. For sequential parts of the code the implementation is trivial which also holds for conditional assignments. However, when the algorithm diverges, e.g. on jumps and loops, it should be split and implemented in independent units.

Thus, units with complex control flow, e.g. the traversal unit, are split into many rather small components. In contrast units like ray-triangle intersection and transformation

can be implemented straight forward using a single large functional unit with few smaller units to manage the control flow.

In the measurements of Section 2.5 the *ti-ratio* for several benchmark scenes was presented. This shows that there should be several times more units for traversal than for ray-triangle intersection. However, implementing the *ti-ratio* $m : n$ in a naive way using m traversal and n intersection units requires a complex connection network (e.g. crossbar, butterfly network) to efficiently schedule the workload between these units. Compared to simple point-to-point connections in strict pipelines this network is more complex and typically requires longer wires which reduces the clock frequency.

Therefore all functional units are implemented in strict pipelines with only short and local connections. This requires to implement the *ti-ratio* $x : 1$ with $x = \frac{m}{n}$ using a single super-scalar traversal unit with x sub-units connected to a single intersection unit. These sub-units are called *traversal slices* and calculate in parallel the traversal decision of different rays. The concept of having a single super-scalar unit allows for sharing large portions of logic for control flow and especially the connection to other units.

However, implementing the *ti-ratio* $x : 1$ has an additional advantage as it allows for implementations where full implementations of all m traversal and n intersection units are not feasible (e.g. due to constraints in chip area).

Packets of Rays

The previous chapter has shown that using packets of rays allows for great savings in the bandwidth requirements. Although larger packets may yield better reductions a good compromise between overhead and savings is achieved for packets with 4 to 64 rays.

The k rays of a packet are assigned statically in *chunks* of $\frac{k}{x}$ rays to the x traversal slices. This obviously requires $k \geq x$ since otherwise some traversal slices are rendered useless or alternatively more complex scheduling is necessary. An analysis of this assignment including measurements of the quality of the load-balancing is given in the next chapter.

For most configurations especially with larger packets there are fewer functional units than rays in a packet. Therefore each functional unit performs its computations sequentially on all rays of a packet respectively all rays of a chunk. Due to this sequential execution every unit performs a memory request only every k-th cycle in the best case. Therefore this execution scheme is very beneficial as it allows for hiding long memory latencies and sharing of a single connection to memory by several units.

Pipelining

The ray tracing algorithm was broken down into several independent functional units. None of these units has loops or branches and therefore can be implemented trivially as a computational pipeline.

When designing a chip for high clock frequencies only few gates are allowed per pipeline stage. This results in deep pipelines for computationally complex operations. However, in every cycle new inputs must be ready to achieve a good usage of the pipeline.

Therefore current processors use great amounts of logic to deliver new inputs to the pipelines. For example *branch prediction* is used to predict which instruction is likely to

be executed next and several heuristics are used to speculatively fetch data from memory before it is requested. Additionally, *speculative execution* performs operations without knowing whether their results will be of any use.

Current processors use these techniques since if running a single sequential program in general it is more efficient to perform operations which only speed up the execution in some cases rather than rendering the pipeline idle by not having new inputs.

However, although *branch prediction*, *prefetching*, and *speculative execution* could also be applied to ray tracing – maybe giving even higher gain than on general purpose processors – they are not used in the SaarCOR architecture.

A major goal in the SaarCOR design was to keep the logic simple and the hardware complexity as low as possible. Fortunately, in contrast to the execution of single sequential programs ray tracing offers huge amounts of small program fragments that can be executed independently and thus allows for *multi-threading*.

Multi-Threading

A *thread* is a set of data (e.g. ray data and scene data) that requires for its computation a sequence of operations. If different operations need to be performed during the computation of a thread it is possible to use several dedicated units for the operations. Achieving a good usage of the various functional units can be done using multiple threads instead of a single thread. Then the units perform the necessary computations sequentially on each thread using a *time slicing* scheme.

Multi-threading could be implemented by identifying a ray as a thread. However, as shown earlier great amounts of bandwidth can be saved if rays are grouped into packets of rays and the same operation is performed on all rays of the packet. Therefore in the SaarCOR architecture each packet of rays forms a thread.

Thus *implicitly* a packet of rays contains several independent threads, which can be executed sequentially in a time slicing scheme. However, although it would be possible to simultaneously execute rays of the same packet on different functional units this is not done except for super-scalar units (e.g. the traversal slices and programmable shading units). While this greatly simplifies the assignment of rays to functional units it also puts a lower bound on the number of threads required to keep the functional units busy and this bound is independent on the number of rays per packet (see Section 8.2).

Connection Schemes

The costs of a chip measured in area is the sum of three components: the transistors used for logic functions, registers and memory blocks, and the connections between these components. Especially the connections can account for a large portion of the costs if complex connection schemes like crossbar switches and butterfly networks are used. Therefore one of the goals in the SaarCOR project was to *keep connections short, narrow, and simple*.

This is goal is achieved by storing local copies of the static data on each unit. For example ray data is generated once and sent via a broadcast bus to the traversal and the ray-triangle intersection unit. This allows for reducing the communication between units to only transferring a single ID specifying the packet instead of sending the data of all

rays. Besides this ID additionally only few control bits and the active-vector need to be transfered.

In general when implementing how commands are transmitted between units there is a trade-off on costs for connections and logic, as illustrated in Figure 3.1. *Dependent transactions* require the receiving unit to perform requests to other units before it can actually start the operation (e.g. look up of data in other units). In contrast *self-contained transactions* simplify the connection scheme since all required data is already contained in the transaction. This allows for turning bi-directional busses into uni-directional connections and further reduces the required logic for state control. Therefore in the SaarCOR architecture self-contained transactions are prefered.

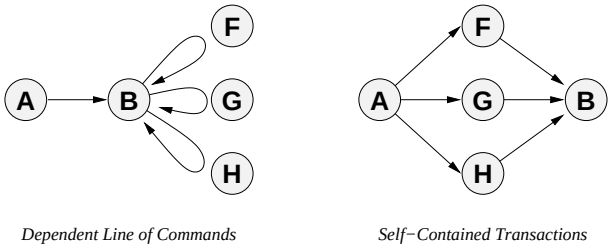


Figure 3.1: This example shows two variants for sending a command from unit A to unit B. In both cases the units F, G, and H store the data required by unit B to perform the operation requested by unit A. However, while a *dependent line of commands* requires unit B to perform several lookups using self-contained transactions all data is delivered to unit B without any requests. Thus self-contained transactions allow for simplifying the connection scheme and also reduce the amount of logic for state control required to perform an operation. It also turns the corresponding block of functional units into a pipeline without loops and dependencies. However, dependent on the layout of the chip dependent transactions might be cheaper.

Sending commands between units can require handshaking and acknowledging the commands as units can be busy or require several cycles to perform an operation. However, protocols and handshaking requires additional wires on the busses and logic, and can cause latencies and idle cycle.

Therefore in the SaarCOR architecture self-contained transactions are implemented using a fire-and-forget scheme: Every unit has FIFOs to store as many jobs as there can be threads and thus commands can be sent independent of the state the corresponding unit is currently in. Since typically only few threads are used these FIFOs are rather small although they fully remove the requirement for any handshaking or busy signals.

Additionally, the implementation of a *ti-ratio* of $x : 1$ also allows for keeping the connections short since always independent sets of a super-scalar traversal unit, a ray-triangle intersection, unit and a transformation unit are grouped together. These sets are called *pipelines* and although there can be several pipelines on a single chip no connection between these pipelines is necessary. They only share the connection to a unit that schedules the work and the memory interface.

Memory Interface

Using packets of rays allows for cutting down the memory bandwidth required for real-time ray tracing from several GB/s to only few MB/s. This removes the requirement for complex memory interfaces. Therefore the memory interface on the SaarCOR architecture simply uses a round robin scheme that allows a different pipeline to perform a memory request in every cycle. The results delivered from memory are broadcasted to all units which identify their own requests by an additional label sent with each request consisting of two IDs: one for the pipeline and one for the unit in the pipeline.

If caches are used then these two labels additionally allow to increase the bandwidth between the caches and the functional units by adding *filtering* as illustrated in Figure 3.2. This shared memory interface performs very well even for complex scenes, low external memory bandwidth, and several pipelines on the same chip (see Chapter 8).

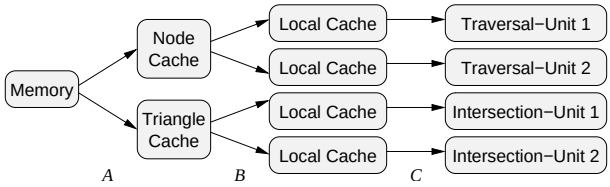


Figure 3.2: Using a single broadcast bus to transfer data from memory to the functional units puts an upper bound on the available bandwidth which can not be raised by using caches since the bus can only transfer a single item per cycle. While this is true for the shared connections to memory (labeled A) items can be *filtered* on the connections B and C by simply not broadcasting data labeled with ID $x = (pipeline, unit)$ to a unit that has ID y. For example this method can filter triangle data on the connections to traversal units (B) and filter either type of data requested by a different pipeline (C). Please note that the *local caches* are not mandatory but only shown to illustrate filtering between pipelines while the node and the triangle caches illustrate filtering for unit IDs.

3.2 Key Features

This thesis presents the SaarCOR architecture including variants with fixed function and programmable components and several alternatives for their implementation. The architecture is built in a modular way to allow for mixing variants to suit application specific requirements best. However, to great extends all components share the same key features:

1. All algorithms are broken down into small independent tasks without loops and branches and each task is implemented in a separate unit in hardware.
2. Each unit is fully pipelined and never stalls. This simplifies clock distribution as all registers are always clocked.
3. Connections between units are only point-to-point or labeled broadcasts with a unique recipient ID which avoids complex routing schemes.

4. Ray data is only transfered once per ray via a broadcast bus and afterwards only the index of a ray is transfered between units. This allows for keeping connections narrow and transaction times short.

5. Communication between units uses only self-contained transactions and the control flow is rewritten such that it only consists of local decisions. Thus given its own history and its current input a unit can always perform the next operation without performing additional requests on other units.

6. All units are implemented in a fully decoupled way using FIFOs for jobs with as many entries as there are threads. This removes the requirement for any handshaking or busy signals.

7. All operations are performed on packets of several individual rays. This lowers the bandwidth requirements, the internal storage for the stack and allows for hiding smaller latencies.

8. All units use multi-threading on packets of rays to hide larger latencies caused by waiting for other units or memory accesses.

3.3 The SaarCOR Hardware Architecture

The SaarCOR hardware architecture consists basically of three main parts: the *ray tracing core* (RTC) which traverses and intersects rays, the *ray generation and shading* (RGS) unit, and a *memory interface* (MI) to manage all external connections.

Since ray tracing scales trivially as every ray or packet can be computed independent of any other ray respectively packet, the architecture supports scaling by having several *ray tracing pipelines* (RTP), consisting of a RGS and a RTC, in parallel. The work performed on all pipelines is managed by the *ray generation controller* (RGC), which simply schedules the pixels to be rendered to the pipelines. Figure 3.3 gives an overview of the SaarCOR architecture and its functional units, which are summarized in the following paragraphs and details can be found in Chapters 4 (RTC), 5 (RGS), and 6 (MI).

3.3.1 Ray Tracing Core (RTC)

The RTC implements the core of the ray tracing algorithm that traverses and intersects top- and bottom-level kd-trees. Section 2.3.3 presented the pseudo-code for traversing packets of rays through a static scene. In this code three different stages, each with separate memory accesses could be determined: loading a node and traversing it, fetching IDs of the triangles contained in a voxel, and finally loading the triangle and intersecting it. In case of dynamic scenes there is the need of a fourth memory access for the matrix used when transforming rays into the coordinate system of a specific object.

For the hardware implementation of the SaarCOR architecture these stages are implemented as dedicated units called *traversal* (TRV), *list* (LST), *ray-triangle intersection* (RTI), and *transformation* (TFM), respectively[1]. Since there is no functional difference

[1]While the names for these units are mostly taken for obvious reasons, the name of the list unit comes from the fact that every leaf node of the kd-tree points to a list of IDs of the objects respectively triangles contained in that voxel.

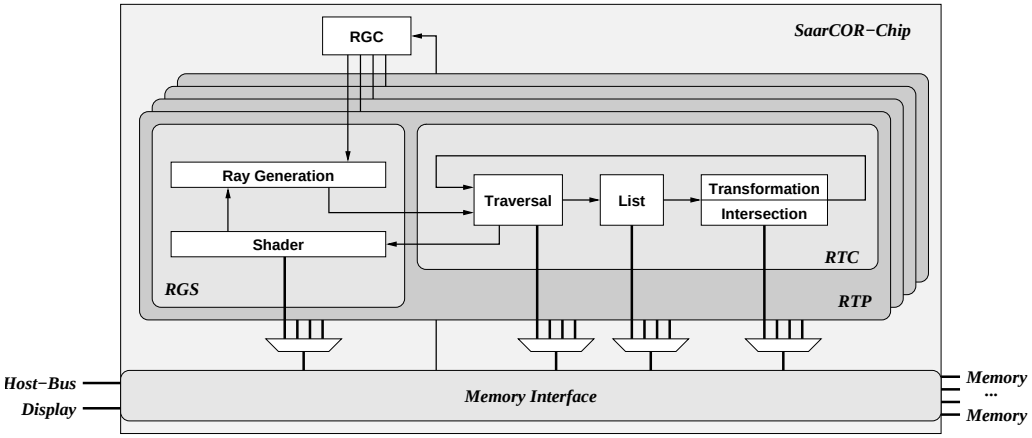


Figure 3.3: Overview of the SaarCOR architecture that is embedded into a single chip. It supports for scaling by having several pipelines (RTP) on a single chip, several chips on a single graphics card, or even by having several graphics cards in a single computer. Furthermore, as long as network bandwidth and latency do not become an issue also scaling using multiple computers is possible.

in traversing kd-trees of meta objects and traversing those of geometric objects this is performed using the same traversal and list units.

Of all modules of the architecture, the ray-triangle intersection unit and the transformation unit require the most floating-point operations. Fortunately, it can be shown (see Section 4.5) that the transformation unit can be used to perform ray-triangle intersections. Using mailboxing and adjusting the *ti-ratio* accordingly the workload on the ray-triangle intersection unit as well as on the transformation unit can be reduced. This allows for implementing a single unit that performs both operations without introducing a bottleneck.

The adjustment of the *ti-ratio* is also guided by the costs measured in floating-point operations and memory bandwidth required by traversal respectively ray-triangle intersection operations. It shows that even highly optimized ray-triangle intersections have more than 4-times higher costs than traversal operations (see Appendix D). Therefore it pays off to implement a super-scalar traversal unit with several traversal slices and adjust the *ti-ratio* accordingly.

3.3.2 Ray Generation and Shading (RGS)

As the name RGS implies this unit provides the functions to generate and shade rays. During shading additional secondary rays may be spawned recursively.

In general both functional parts of the RGS unit could be either fixed function or programmable. If shading is programmable and fast enough it might take over ray generation as well. Alternatively, the transformation unit of the RTC can be used to generate rays, which allows for very elegant ray generation (see Section 5.4).

Since programmable shading usually involves shading programs with a run-time of many cycles, several processors for shading are required. These processors need to access the same memory and therefore are arranged as a shared-memory parallel computer, which

leads to the same properties and problems known from these machines.

Given the right framework any general purpose processor could be used for shading. Chapter 5 presents a suitable framework and additionally a simple processor optimized for shading. This processor follows the *push-model*, i.e. an external controller (called *master*) pushes the content of registers for the program to be executed next into the CPU. This speeds up switching of programs compared to a polling or interrupt-based approach.

Furthermore, the CPU uses multi-threading and has additional instructions to support efficient communication and resource management. Therefore this processor is not limited to shading rays but also allows for advanced image space operations, building kd-trees, and possibly even non-rendering operations like calculations for physics-engines of computer games.

3.3.3 Memory Interface (MI)

The memory interface provides more functionality than the name suggests since besides managing the connection between ray tracer and memory it also handles all external communications. Again for efficiency reasons all connections use only point-to-point communication and self-contained transactions.

For all types of data delivered by the memory interface there is a separate item-based cache, i.e. rather than caching single bytes, it manages triangles (or any other geometric primitive) or kd-tree nodes. However, since there is no fixed data structure for shading data and lists of IDs have variable lengths they are cached using standard techniques with cache lines of suitable lengths. These caches are shared by all units of all pipelines using labeled broadcasts and shared busses.

The *memory controller* handles the connection to various memory chips using address hashing and FIFOs to reorder requests for best-case transaction times. Since displaying an image requires a guaranteed bandwidth the *display controller* is tightly coupled with the memory interface.

The communication with the application uses the *host-bus*, which might be of arbitrary type (PCI, PCI-X, AGP, etc.). However, Section 8.3 shows that even for fully automatic *virtual memory management* a slow system-bus is sufficient. Besides support for virtual memory, scenes can be managed very efficiently on an object level.

For shared-memory communication a *memory processor* might be required to efficiently support *multi prefix operations* and memory management. An additional processor can be integrated into the data paths used to transfer geometric primitives for intersections. This *geometry processor* can modify the position and shape of geometric primitives on-the-fly when they are loaded from memory and stores the results in the standard cache for primitives avoiding multiple processing of the same geometry.

3.3.4 Scalability of the Architecture

Ray tracing offers a huge amount of independent threads (e.g. pixels or packets to render), which are trivial to execute in parallel. This property greatly simplifies *scaling* the performance of a ray tracing system. The SaarCOR architecture is a modular design

that supports scaling in several ways by having super-scalar units per pipeline, multiple pipelines per chip, multiple chips per board, multiple boards in a single computer, and multiple computers connected via network.

However, as every component has a limiting factor, scaling has its limitations. In general, scaling the performance of a ray tracing system leads to the following issues:

- *Scheduling:* jobs have to be scheduled load balanced to all parallel units
- *Bandwidth:* all necessary data has to be delivered to the units
- *Combining:* partial results of units have to be combined into final result

The following discusses these issues in the context of the SaarCOR architecture.

Scheduling and Load Balancing

The only information required to be sent individually to each unit is which pixels on the screen to render. Fortunately, this scheduling can be implemented without any communication. If every chip knows its number m and the total number n of SaarCOR chips in parallel, then all packets each chip should render are given by equation $m = (x + y)\ MOD\ n$, with x, y the coordinates of a packet in screen-space. This scheme already balances the workload quite efficiently, but Section 4.2.1 presents an optimization in more detail.

Inside a single chip, load balancing can be performed dynamically by the RGC which schedules only the coordinates in screen space for whole packets of rays to the RTPs[2]. Every packet of rays stays on the same RTP until its computations, including any secondary rays are finished. Therefore a single coordinate with two integer values specifies enough work to keep the RTP busy for a long time. Thus the RGC never becomes a bottleneck.

Bandwidth

The bandwidth required per ray tracing component (i.e. pipeline, chip, or board) splits into two parts. The first issue is the transfer of information common to all units, e.g. settings of the camera and lighting conditions. This can be solved by using broadcasts to all units, which then store local copies of these settings.

The more demanding issue is delivering data requested during ray tracing, e.g. for scene data and virtual memory accesses. This issue again splits into three parts: the connections between ray tracing pipelines and local memory, local memory and ray tracing chip, and between ray tracing boards and host computer.

Inside a single chip scaling of the ray tracing performance is possible as long as the bandwidth does not become the limiting factor (see Chapter 8 for details). Further scaling can be achieved easily using multiple ray tracing chips and replicating the local memory for each chip.

[2] To increase the efficiency of the cache scheduling can be performed using space-filling Hilbert curves instead of scanline order. An hardware efficient variant of this curve can be found in [Woo04].

This scaling reaches its limitations if virtual memory techniques are used, since then the connection between the ray tracing boards and the host might become a bottleneck. However, Chapter 8 shows that for fairly complex scenes even slow PCI busses are capable of delivering the required bandwidth.

The limitations of the host system can again be overcome by replication using independent clients each consisting of a host computer together with ray tracing boards where all clients are interconnected via network. This setup is similar to the OpenRT software rendering system [] where it was shown that almost linear scaling can be achieved as long as the network provides enough bandwidth. However, in this setup the network connection also puts an upper bound on the amount of changes that can be made to the scene data base as these changes need to be broadcasted for each frame.

Combining

One of the major issues for any distributed rendering system is combining the partial results (e.g. image tiles or pixels) into a final image. Since the framebuffer needs to be stored in the memory of a single unit to allow for displaying, all requests to the framebuffer have to be serialized[3].

Thus when scaling the performance of a ray tracing system using parallelization this connection to the framebuffer might become a bottleneck. However, scaling the quality of the rendered image is much easier than scaling the size of the image as the bandwidth used for updates to the framebuffer is not affected by the number of rays used to calculate the color of a pixel (see [WSB01, WKB$^+$02]).

On current graphics boards based on rasterization technology in many cases the frame buffer is the limiting factor since several reads and writes on the color data and the corresponding z-values are required. However, in standard ray tracing every pixel is written only once and never read back which greatly simplifies the connection to the frame buffer especially if multiple boards need to access it.

Anyhow for advanced image filter operations a read back and maybe even an update is required. In general this can limit the performance gained by scaling but if each unit uses packets of rays or tiles of pixels then the required bandwidth to the frame buffer can be reduced. Here filtering can be applied to all pixels for which the current packet or tile contains the required data and only the missing data needs to be read. However, if a computational overhead can be tolerated then even advanced image filtering can be performed without increasing the bandwidth to the shared frame buffer. This is achieved by having overlapping packets of rays and this way all information necessary for image filtering is already available locally (see [WKB$^+$02, BWS03]).

[3] In general this is not true as shown in the *SB-PRAM* [PBB$^+$02]. However, it is true for all standard architectures used in computer graphics.

4 Ray Tracing Core

The previous chapter gave an overview of the SaarCOR architecture and Section 2.3.3 presented the detailed pseudo-code for traversing packets of rays. This chapter will explain how the core algorithms of tracing rays are embedded into an efficient hardware architecture that follow the guidelines specified in the previous chapter.

The *ray tracing core* (RTC) basically consists of a *traversal* unit (TRV) for traversal of kd-trees, a *list* unit (LST) to enumerate the lists of references to all objects respectively triangles contained in a leaf node, a *transformation* unit (TFM) to transform rays from the current coordinate system into the coordinate system of an object, and a *ray-triangle intersection* unit (RTI) to intersect rays with triangles.

The central unit to organize the control flow in the RTC is the traversal unit deciding which operation a packet performs next. All other units are simple pipelines that execute a command and return the results to the traversal unit.

In a ray tracing system most operations are performed on rays. Therefore the description of the RTC starts with the data paths and register files required to handle rays. Then the traversal, list, transformation, and ray-triangle intersection units are explained in detail.

4.1 Data Paths and Storage for Ray Data

The SaarCOR ray tracing system traverses rays through dynamic scenes consisting of a hierarchy of rigid objects. This requires several functional units to perform different operations on rays and to store ray and meta data in various formats. In the following first the various levels of hierarchically modeled dynamic scenes are summarized before the data structures and their storage requirements are discussed. Then some issues are explained and the data paths for handling rays are presented.

Levels of Hierarchically Modeled Dynamic Scenes

The left image in Figure 4.1 shows an example for dynamic scenes using rigid objects (labeled A, B1, $\cdots$, C3) which are organized hierarchically. Here the labels A, B, and C correspond to the levels 0, 1, and 2 of the hierarchy respectively. The corresponding tree to this hierarchy is shown in the middle. Please note that this tree represents only a logical ordering and the data structure used in ray traversal to find objects in scene space is the kd-tree.

When traversing a ray through a scene, the ray may enter some of the objects in which case the ray is transformed into the local coordinate system of the object. For the ray shown in the example the tree in the middle illustrates which objects are visited and in which order. The kd-tree on the right is an example for a rigid object which is traversed and triangles or objects found in its leaves are intersected.

These examples show the similarities between the hierarchical tree of the logical ordering and the kd-tree of rigid objects. In both cases traversal of the tree goes down but sometimes needs to go back to follow a different link down. So there are two different positions in the architecture which require stacks.

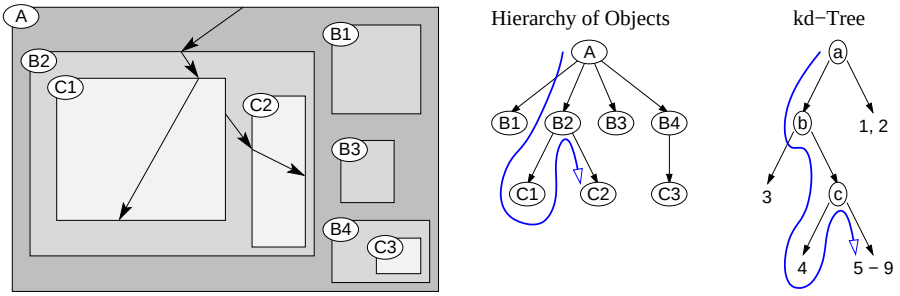


Figure 4.1: Example for the various levels of hierarchically modeled dynamic scenes.

Storage Requirements for Ray Data

Traversing n levels of a hierarchy of rigid objects requires to store ray data for every level. But traversal and intersection perform only operations on the current level. This allows for reducing the storage requirements by storing the ray data of all non-current levels (at most $n-1$) only in the transformation unit. Then when descending the hierarchy the transformation unit calculates the ray data for the current level and broadcasts the data to traversal and intersection. When going back again in the hierarchy the ray data stored in the transformation unit becomes current and thus is broadcasted to both units.

Table 4.1 lists the amount of storage required per ray when using single precision IEEE floating-point numbers and a single 32-bit value to store the object and triangle-ID of the *hit-information*.

Examples, Issues, and Discussion on Stacks

For dynamic scenes with two levels of hierarchy, kd-trees of depth 32 and mailboxes with 8 entries this sums up to 382.25 bytes per ray. Using a standard configuration (see Chapter 8) with 64 rays per packet and 16 packets per pipeline requires 382 KB storage for ray data and additionally 320.25 bytes for packet data in each pipeline. In contrast for static scenes only 213.5 KB for ray data are required per pipe.

Thus the memory requirements per RTC are rather high. However, the main part of the memory is used for stacks (71% in dynamic and 64% in static scenes). This motivates a discussion on the options that allow for reducing the memory requirements of stacks. Additional implementation details can be found in Appendix C.

Fitting kd-Trees to Hardware Resources A simple option is to build a hardware that only provides a small number of entries on the stacks. Then the kd-trees of the scene to be rendered are built such that they fit the hardware resources available. While this trivially reduces the memory requirements it may also limit the performance of the ray tracing system as subdivisions in scene space might become too coarse ending up with many triangles that need to be intersected per ray.

Parameters	Storage per Ray	Quantity	Functional Unit
origin, *direction*, *d* *active*, *traversal decision*	28 bytes 4 bits	1	Traversal (current)
near, *far*, *hit-information* *not-terminated*, *validity*	24 bytes 2 bits	n	Traversal (per object-level)
far respectively *near* *active*, *both*, *validity*	4 bytes 3 bits	$n \cdot (m-1)$	Traversal (stack)
active	1 bit	$n \cdot b$	List (per object-level)
origin, *direction* *active*	24 bytes 1 bit	1	Intersection (current)
origin, *direction* *active*	24 bytes 1 bit	$n-1$	Transformation (stack)

Table 4.1: Amount of storage for ray data required in the units of the RTC. Here n specifies the number of nested object hierarchies and m the maximum depth of the kd-trees. The list unit uses mailboxing with lists of b items. Although not listed per packet of rays each functional unit additionally requires a register of $LOG(n)$ bits to store the current level of the packet. Please note, that memory for registers independent of rays is not counted. Thus for example temporary and pipeline registers as well as registers storing the IDs in the mailbox are not listed.

Swapping Stacks to Off-Chip Memory Alternatively a swapping mechanism could be implemented, which transfers data to off-chip memory and back again when needed. However, this requires additional data paths and a rather high peak bandwidth making this technique too costly especially when implementing several pipelines on a single chip.

Dynamic Allocation of Stack Memory on Per Frame Basis In the same spirit as *flexible packets* (see Section 2.6) also stacks can be build easily to support a discreet set of different numbers of rays per packets and threads per pipeline. Thus each functional unit which implements this support can manage its local memory for stacks dynamically.

This capability makes it interesting to look again at the amount memory required for stacks: $\textit{memory required} = \textit{memory per ray} * \textit{rays per packet} * \textit{packets per pipeline}$. Here *memory per ray* depends on the maximum depth of the kd-trees in the scene to be rendered. Reformulating this equation yields:

$$\textit{rays per packet} * \textit{packets per pipeline} = \frac{\textit{memory required}}{\textit{memory per ray}}$$

If we label *memory required* as *memory available* it suggests to choose a suitable amount of memory and then to calculate the number of packets per pipeline or rays per packet that are supported. This suitable amount of memory could be derived from technical feasibility but also from typical scenes the chip is designed for.

This support for dynamical allocation of stack memory allows for a new trade-off between three options which can be chosen dynamically on a per frame basis. However, all options also potentially lead to new issues that limit the performance.

The first option is to *fit the kd-trees to the hardware resources* available and thus potentially increase the number of objects required to be intersected due to a coarse scene space subdivision. The second option is to *reduce the number of rays per packet* which

potentially increases the memory bandwidth (internally and externally). The third option is to *reduce the number of threads per pipeline* which reduces the workload on the functional units and thus potentially renders units idle while waiting for other units or memory requests.

The first option is generally the worst to do since it not only increases the workload for object intersections but also increases the memory bandwidth since data for more objects needs to be loaded. However, there is no simple way to tell which of the remaining options is best as this clearly depends on the scene and other components of the architecture.

Dynamic Allocation of Stack Memory During Rendering In contrast to selecting a suitable set of rays per packet and threads per packet which stays constant during rendering of a frame it is also possible to manage the available memory fully dynamic and demand driven.

Hardware support for dynamical allocation of memory segments with variable sizes (e.g. depending on the number of currently active rays) is not trivial. Additionally, support for variable sizes requires some sort of garbage collection to recluster the memory segments when they are not used anymore. Fortunately, support for dynamic allocation of fixed size memory segments is rather simple and using the *hw-malloc* scheme (see Section 5.2.7) requires only an additional standard FIFO.

However, regardless of the size of the memory segments it might happen that the local memory is exhausted, which could lead to deadlocks that can be resolved only by swapping out content to off-chip memory or by *killing packets*. This "killing" simply frees all memory occupied by a packet and initializes the packet to start over from scratch again when enough resources are available (for details see Section 5.1).

The problem of potential deadlocks can be reduced by using only a single stack for all hierarchies (with stack-pointers for each level) instead of independent stacks per level. While does not solve the issue of deadlocks at least it allows for using the available memory fully without wasting resources.

Approximations Typically at the same time not all rays require the maximum number of entries on the stack. This observation suggests to share the memory of stacks between different rays and packets handled by the same functional unit.

Without implementing swapping or killing mechanisms the stack memory might become exhausted. Then still traversal can continue but no further content can be written to the stack. As only nodes describing objects and triangles further away are written to the stack this may work well in many cases. However, it may also lead to "overlooking" triangles and objects further away if no hit-point has been found in the close range.

Summary There are many options on how to reduce the memory requirements for stacks. However, all methods that produce correct results are in some sense expensive – either due to memory requirements, bandwidth issues, or computational overhead.

Thus the only practical alternative is approximation which takes advantage of the fact that typically between 15% and 25% of all nodes visited put the other child to the stack. Therefore on average kd-trees of depth m can be rendered using a hardware with stacks of size $\frac{m}{4}$. A more conservative variant with $\frac{m}{2}$ was chosen for the SaarCOR prototype (Chapter 7) where visible artifacts due to the size of the stacks have never been encountered.

Initializing Ray Data

Before the data paths for transferring ray data are presented this paragraph describes when and how ray data is initialized. The initialization is separated into *init once* and *init per level*. Init once is simple and described directly in the data paths below. Per level *near*, *far*, and the stacks have to be initialized. Here the issue is not how but when to initialize: Initialization must take place when entering a higher level or when entering a new object on the same level, i.e. when accessing a list of objects.

There are three units – traversal, list, and transformation – that could manage initialization. The traversal unit has only a local view of the current level thus it can only manage its local stacks and keep a local state while being inside a single object. The purpose of the transformation unit does not require any storage of ray data as all requests are finished after delivering the transformed rays. Although the list unit knows whether there are any further items in the list this gives only indirectly the information required. Thus there is no obvious decision which unit to extend for managing the hierarchical transition between objects.

We choose the transformation unit as it directly receives the flag *last-item-in-list* from the list unit. This selection allows for a simple communication scheme between traversal and transformation unit using a single command "send an object for traversal", which might either send ray data in a new object from the list or the ray data from the previous level. Only in the first case the transformation unit also issues a flag *init-data-structures* to signal that the traversal unit should init *near*, *far*, and the stacks.

Data Paths for Ray Data

The previous sections have discussed the storage requirements and initialization of ray and packet data. This section describes the data paths to transfer this data between the functional units (Figure 4.2).

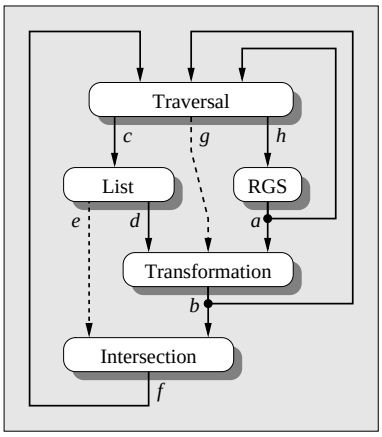


Figure 4.2: Unoptimized data paths for ray data and control flow. The stacks in the traversal slices of the traversal unit and in the transformation unit are not shown for the sake of simplicity.

Initially rays are sent by the *ray generation* unit (via path a) to the transformation unit which broadcasts the current ray data (via path b) to the traversal and the intersection

unit. The transformation unit either broadcasts the transformed rays to traversal and intersection (when descending) or the stored ray data when going back together with the flag *init-data-structures* set accordingly.

The initial *not-terminated vector* is sent to the traversal unit also using path a. When receiving the *not-terminated vector* the current level is set to 0 and also the *hit-information*, the *validity vector*, and the stack-pointers on level 0 are initialized.

Traversal continues until a leaf node is reached which is then handled by the list unit (path c). If this voxel contains objects the list unit orders the transformation unit (path d) to transform the packet into the coordinate system of the object. Again the new rays are broadcasted to traversal and intersection and traversal starts over.

If the voxel contains triangles the list unit orders the intersection unit to intersect the packet (path e). The results of the intersections are sent to the traversal unit (via path f), which collects the hit-information.

The traversal unit keeps track of the state of all rays and checks their termination. After all rays are terminated the packet is handed over for shading via path h.

Optimizations of the Data Paths

The data paths e and g shown with dashed lines can be removed to simplify the communication scheme. Path e is not required if intersections are performed using the *unit triangle intersection method* (see Section 4.5) since it uses the transformation unit to preprocess ray data before performing the intersection computation.

Using this method the intersection unit does not need to store any ray data. This turns path b from a broadcast bus into two point-to-point links. One link is used to transfer ray data from the transformation unit to the traversal unit. The other link is smaller and transfers partial intersection results from the traversal unit to the intersection unit.

Path g was introduced to order the transformation unit to send a next object to continue traversal. However, by extending paths c and d by a single bit this command can still be sent but does not require an additional data path. Additionally, the complexity of the control logic in the transformation unit is lowered since the number of commands that can be received simultaneously is reduced.

These optimizations would introduce only a bottleneck if having independent paths would allow for higher usage of the functional units. But when using the unit triangle intersection method obviously there is no problem since either the list unit is idle and passing the command is for free or the list unit is active and then the command sent by the traversal unit has to be stalled anyway.

Ray Generation Using the Transformation Unit

Initially the rays are sent by the ray generation unit to the transformation unit which stores the ray data on all levels. Since the transformation unit touches the rays anyway it can also be used to generate the rays using matrix multiplications. Section 5.4 will show that this can be done in a very elegant way and that using the transformation unit many floating-point operations in the ray generation unit are saved.

However, although ray generation using the transformation unit sounds like a trivial,

elegant, and cheap solution it also introduces new issues. For ray generation the transformation matrix is not loaded from memory but sent via an additional bus by the RGS. This bus is rather wide since a matrix of 12 floating-point values has to be transfered in a single cycle requiring a bus with at least 384 wires.

Furthermore the RGS requires knowledge of the transformed rays to perform shading operations and generation of secondary rays. Therefore rays have to be sent back again via an additional data path. This data path is rather wide since a ray with origin and direction contains 6 floating-point which has to be transfered in a single cycle and thus requiring the bus to have at least 192 bits.

Nevertheless, the SaarCOR prototype presented in Chapter 7.3 uses the transformation unit for ray generation. This pays off since the reduction in floating-point requirements overweights the disadvantages at least on a FPGA-based architecture.

4.2 Traversal Unit

In Section 3.1 it was shown that the traversal unit should be built in a super-scalar way by having several sub-units in parallel to perform the necessary operations. Therefore the traversal unit splits into the *traversal logic* (TL) for management and scheduling and the *traversal slices* (TS) for computing the traversal decision. But before the data paths of the traversal unit are presented the next section discusses how rays of a packet are distributed among several traversal slices. This distribution has consequences for the storage of ray data and how the stacks are implemented.

4.2.1 Balancing the Workload

This section evaluates how ray data is managed and distributed between the traversal slices. Additionally, some alternatives for scheduling jobs to the traversal slices are discussed.

Scheduling Jobs to Traversal Slices

The traversal slices can be arranged similar to a SIMD machine with packets of rays split statically between the slices and with the same operation executed on all slices simultaneously. Splitting the rays statically allows for storing the ray data in local register files on each slice and therefore results in very simple and efficient data management.

This strict SIMD-like management requires masking of inactive rays and therefore achieves only a good usage for scenes with high coherence. However, even in coherent scenes with increasing size of the packet the overhead increases (see Table 2.6). This overhead can be reduced by decoupling the slices.

A simple variant is using *synchronous decoupling* in which a thread is scheduled for execution to all slices in parallel. But each slice only executes operations on the active rays that are assigned to the corresponding slice. The execution of the operation is synchronized by waiting for the slice that had the most rays being active.

This simple variant leaves room for improvements by using *asynchronous decoupling*. Here work is also scheduled to all slices simultaneously but each slice has a dedicated

FIFO to store jobs and switches to the next job as soon as the current one is finished. By keeping several jobs ready for execution in the FIFOs the slices can be kept busy even in incoherent scenes.

The following presents three methods of how to distribute the rays statically among several traversal slices. These methods will be evaluated and compared to a dynamical assignment, which is impractical due to its costs but represents the best case. From these measurements consequences for the architecture will be derived.

Distributing Rays to Traversal Slices

Method "A" balances the workload between the slices by distributing the rays in the following way: Let $r = (2^m)^2 = 2^{2m}$ be the number of rays per packet and $s = 2^n \leq \frac{r}{2}$ be the number of slices per traversal unit with $n, m \in \mathbb{N}^+$. A ray with packet-coordinates (x, y) with $0 \leq x, y < 2^m$ is handled by slice i with $0 \leq i < s$ if $i = (x + \frac{s}{2} \cdot y)$ *MOD* s.

This method distributes the rays of a packet such that any neighborhood of $r' = 2^{2m'}$ rays with $m' \leq m$ has the fewest possible number of rays assigned to the same slice[1]. Figure 4.3 illustrates this property[2].

For comparison of the efficiency of method "A" additionally a rather worst case assignment "B" that tiles the packet as well as a simple variant "C" that distributes the packet using the number of the column is evaluated (see Figure 4.3).

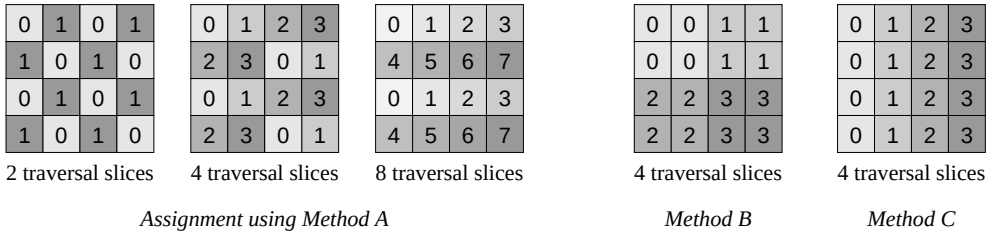


Figure 4.3: Examples of assigning a packet of 16 rays to slices. Each box denotes a pixel in screen-space with its corresponding ray(s) and the number in the box specifies the slice this pixel is assigned to.

Comparison of the Efficiency

In this section comparisons of the efficiency for several scheduling methods are given. In particular these are SIMD-like scheduling and three methods of statical assignment using synchronously decoupled slices. Additionally, the *synchronous overhead* is listed specifying the overhead that occurs by scheduling jobs to all slices simultaneously and synchronizing to (i.e. waiting for) the slice on which the most rays are active. Slices which wait are idle and therefore lower the efficiency of the scheduling method.

The efficiency is calculated by measuring the number of cycles (i.e. the execution time) required to perform a traversal operation on a packet of rays. The synchronous overhead is the ratio between the number of cycles in which all potentially working units of a

[1] Obviously there are several possible solutions yielding the same quality.

[2] All possible cases for packets with 16 rays except for the trivial ones with $s = 1$ and $s = r$ are shown and verification for other sizes of packets is straight forward. Obviously this scheme does not work for $s \geq r$.

hardware are occupied for the execution of an operation (which is the product of the number of traversal slices times the execution time) versus the minimum number of cycles required for that operation (which can be measured using a single traversal slice). This leads to the following definitions:

$$\textit{scheduling efficiency} = \frac{\#\textit{cycles with dyn.scheduling}}{\#\textit{cycles with stat.scheduling}} \cdot 100\%$$

$$\begin{matrix}\textit{synchronous}\\ \textit{overhead}\end{matrix} = \frac{\#\textit{cycles with dyn.scheduling} \times \#\textit{trav.slices}}{\#\textit{cycles on hw. with single trav.slice}} \cdot 100\% - 100\%$$

Thus the best case *scheduling efficiency* (e) is 100% and typically lower for any statical scheduling. The *synchronous overhead* (o) is 0% in the best case but can go up drastically for large packets in incoherent scenes. Furthermore both measurements together allow to calculate the *usage of the traversal unit* $= \frac{e}{(100\%+o)/100\%}$. Using asynchronous scheduling the synchronous overhead can be reduced (Appendix C shows how this overhead can be removed completely).

The corresponding measurements are listed in the Tables 4.2 and 4.3. Here Table 4.2 lists measurements[3] for the *UTID* scene, which was taken from a game and as such was designed to achieve a good performance[4]. Therefore the *UTID* scene represents an example of a coherent scene. Additional measurements including those of incoherent scenes are presented in Table 4.3.

Number of Rays per Packet	Efficiency of Scheduling / Synchronous Overhead [in %]								
		4 Traversal Slices				16 Traversal Slices			
	SIMD	A	B	C	o	A	B	C	o
4	97.3	—	—	—	2.8	—	—	—	—
16	93.0	99.2	96.0	97.5	1.1	—	—	—	—
64	83.3	99.0	91.7	97.3	0.7	94.8	89.2	—	4.2
256	67.3	99.1	85.4	97.5	0.4	94.3	77.0	87.8	2.1
1024	44.5	99.2	77.1	97.7	0.2	94.6	62.1	88.8	1.2
4096	22.5	99.3	68.5	98.1	0.1	95.3	47.1	90.3	0.7

Table 4.2: Depending on the number of rays per packet this table lists the efficiency of four scheduling methods on a traversal unit with 4 respectively 16 traversal slices when rendering the *UTID* scene in 1024×768. These measurements compare SIMD and the three statical synchronously decoupled scheduling methods to an ideal dynamical scheduling. Additional, the *synchronous overhead* (o) specifies the overhead that occurs since jobs are scheduled to all slices simultaneously, e.g. having 7 active rays scheduled to 4 traversal slices gives 14% overhead. Thus having the efficiency of the scheduling method and the synchronous overhead allows for calculating the usage of the traversal unit. Obviously the assignment methods only make sense when there are more rays in the packet than traversal slices and method "C" only works for $s \leq 2^m$ since otherwise no work is assigned to some slices.

[3] The efficiency of the SIMD variant is of course equal to the activity given in Table 2.6. However, the values in Table 4.2 differ sightly since the measurement does not include the traversal of the clipping kd-tree.

[4] Although this game was designed for rasterization hardware in general scenes from computer games also perform well for ray tracing (see Chapter 8) since they are designed to have roughly a uniform distribution of the triangles in the scene to provide a constant frame rate during game-play.

Scene Type, RPP	Efficiency of Scheduling / Synchronous Overhead [Range in %]					
	4 Traversal Slices			16 Traversal Slices		
	SIMD	A	o	SIMD	A	o
c,16	88.8 - 97.3	98.9 - 99.7	0.5 - 2.4	—	—	—
i,16	25.7 - 63.3	97.4 - 97.5	13.3 - 200	—	—	—
c,64	70.7 - 92.8	98.6 - 99.7	0.3 - 1.6	75.8 - 94.0	92.0 - 98.0	1.6 - 9.0
i,64	6.6 - 26.8	96.2 - 96.8	11.3 - 189	25.0 - 40.3	87.6 - 94.5	67 - 1007
c,256	49.9 - 84.3	98.6 - 99.7	0.2 - 1.1	46.4 - 84.9	91.2 - 97.9	0.8 - 6.0
i,256	1.7 - 8.6	95.8 - 96.5	9.9 - 183	6.3 - 12.4	86.1 - 93.8	59 - 971

Table 4.3: This table lists the efficiency of two statical assignment methods and additionally the synchronous overhead depending on the number of traversal slices and *rays per packet* (RPP) used to render benchmark scenes in 1024×768. The benchmark scenes are separated by their *scene type* which is either *coherent* (c) or *incoherent* (i) (see Chapter 8 for details on the benchmark scenes). It shows that rather independent of the scene type scheduling method A always achieves a good efficiency.

Summary and Consequences

These results show that it is more important to work only on the active rays of a packet and do not use a strict SIMD-like scheduling than to use any specific method of assignment. Furthermore using a tile based method is always the worst to do and method "A" yields the best results which are so good that for four traversal slices even in incoherent scenes further improvement is hard to achieve. However, additional to method "A" an asynchronous decoupling should be used as it has moderate costs (see Section 4.2.4) and provides the best possible load-balancing allowing to achieve a high usage and performance.

Using this statical assignment allows for splitting the register files between the traversal slices. This works out very well since there is no need for transferring ray data between the traversal slices as all operations performed on the slices always access only the local register file. This splitting simplifies the implementation since it lowers the number of ports on the register files and gives short connections.

This statical separation of ray data additionally allows for performing the operations on the stacks in parallel on all traversal slices. Following this concept of separation the part of the stack that stores the addresses of nodes is not part of any traversal slice. Since all computations of addresses are performed in the traversal memory interface the stack for addresses is implemented directly in the TMI.

4.2.2 Data Paths of the Traversal Unit

In this section the data paths that connect the four functional units of the traversal unit are described. The following gives an overview of these functional units, which are described in detail in Section 4.2.3.

All memory requests are issued by the *Traversal Memory Interface* (TMI), which schedules memory data read either to the traversal slices or the list unit for processing. Each

Traversal Slice (TS) calculates the local traversal decisions for all rays assigned to the slice and handles the corresponding parts of the stack. The local decisions sent by the traversal slices are collected by the circuit *Global* (GL) that calculates packet traversal decisions and manages the stack push operation. During the intersection of a packet of rays the *CollectHits* (CH) unit collects and checks the hit-information sent by the intersection unit and manages popping from stack.

Although the functionality of these units is independent on the scheduling mechanism the interconnection between these units varies. Thus before the data paths are described the influence of scheduling is discussed.

Scheduling

In the previous section the two alternative scheduling methods synchronous and asynchronous have been presented. But although both variants use different strategies for job management the corresponding top-level data paths of the traversal unit are the same and differences can only be found inside some of the units.

An obvious difference is the position and number of FIFOs used to store jobs in the various units. However, the scheduling method affects only some of the functional units (shown in Table 4.4) since others always require FIFOs to handle either multiple commands received simultaneously from several units or to allow for stalling in case the memory system is busy[5].

Command	Source → Destination	Synchronous	Asynchronous
Traversal Step	TMI → TS	FIFO in TMI	FIFO per TS
Push to Stack	GL → TS/TMI*	FIFO in GL	FIFO per TS
Pop from Stack	CH → TS/TMI*	FIFO in CH	FIFO per TS

Table 4.4: Position and number of FIFOs for job-management in the traversal unit. Please note that the TMI (marked with *) does not require any FIFOs to execute a stack operation since it can be performed in a single cycle. Furthermore independent on the scheduling mechanism TMI and CollectHits use additional FIFOs since they must handle multiple commands received simultaneously from several units and the TMI must be able to stall if the memory interface is busy.

Besides this obvious difference only gathering the partial traversal decisions depends on the scheduling mechanism since in asynchronous decoupled slices results are calculated by the traversal slices out-of-order. However, there is an easy solution that is presented along with further details on the FIFOs for jobs in Section 4.2.3.

Data Paths

The traversal unit is implemented as a super-scalar functional unit with several sub-units. This requires that besides point-to-point connections also broadcast and partial result busses are used (see Figure 4.4). Furthermore since most functional units are specialized

[5] This stalling is no stalling in the common sense that the clock signal for a whole pipeline is turned off but rather a stalling by not issuing new commands into the pipeline.

to perform only a single operation typically there is no need to transfer a command about what operation to perform but only on which data the operation needs to be performed. Therefore transfers typically consist only of a *valid* flag denoting that a command is being transfered and the *thread-ID* specifying on which packet the command should be executed. This allows for an implementation using narrow busses only.

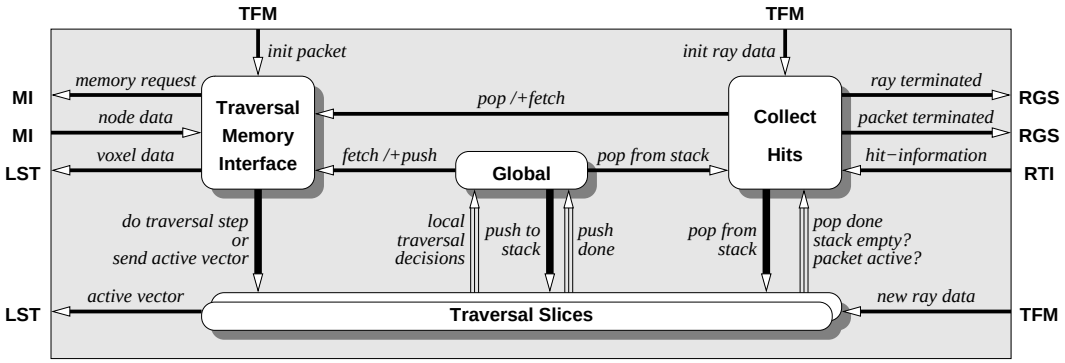


Figure 4.4: Data paths of the traversal unit with descriptions of the commands transfered via the various paths. Since the traversal unit is a super-scalar unit besides point-to-point connections there are also broadcast busses (to the traversal slices) and partial result busses (from the slices). The diagram also lists for outgoing connections the corresponding destination unit, i.e. memory interface (MI), list unit (LST), transformation unit (TFM), shader (RGS), or intersection unit (RTI). Please note, that the initialization of TMI and CH sent by TFM can be implemented as a single bus since the information required by TMI is a subset of the data sent to CH and thus allowing for synchronous initialization of both units.

Rendering starts with a new packet of rays sent by the transformation unit to the traversal slices, which also initializes the ray data in CollectHits and the packet data in the traversal memory interface. After initialization the TMI starts fetching the root-node for the new packet.

Depending on the type of node returned by the memory interface the TMI either sends the data to the traversal slices (inner nodes) or the list unit (leaf nodes) for processing. In the latter case a command is issued to the traversal slices to send the active vector stored in parts on the slices to the list unit. This command is executed simultaneously on the slices such that the active vector and the voxel data are transfered in the same cycle to the list unit.

If an inner node has been fetched the traversal slices compute the corresponding local traversal decisions and send them to the unit Global. Global collects these partial results and computes the packet traversal decision.

If both children need to be visited the far-side has to be pushed to the stack. The corresponding command is sent to the traversal slices and when they signal completion the command push far-side to stack and fetch near-side is sent to TMI.

If only one child is to be visited the fetch command can be issued directly to the TMI. But when encountering empty voxels it might also be the case that no child is to be visited and then a signal is issued to CollectHits to pop ray data from stack. Popping

from stack is managed by CollectHits as this unit already handles popping from stack after intersection.

Since the traversal stack is split between the traversal slices and the TMI the command to pop from stack is sent simultaneously to these units by CollectHits. Again the TMI executes the stack operation in one cycle while the duration on the traversal slices depends on the number of active rays. Therefore the slices have to signal completion before CollectHits can continue processing the packet.

The decision how processing of the packet continues depends on the result of the stack pop operation, i.e. if there are any rays active in the popped packet, and whether the stack is empty. Thus if only rays have been popped which are terminated already then again ray data has to be popped from stack. But if there are any active rays then traversal continues and TMI is ordered to fetch the next node.

If the stack is empty and there are no active rays left or if all rays are terminated then the termination of the packet is signaled to the RGS. In a standard implementation after packet termination the hit-information of all rays of the packet has to be transfered to the RGS in a costly block transfer. This also introduces a bottleneck since while one block transfer is performed the next packet could terminate which would require additional FIFOs to support stalling.

Therefore whenever CollectHits receives hit-information from the intersection unit it checks its validity, i.e. whether there is a hit which is closer than any previous hit, and every valid hit is send directly to the RGS. This way there might be multiple updates for any ray but if the hit-information in the RGS was correctly initialized when packet termination is received no further data has to be transfered for that packet.

4.2.3 Details

The previous section explained the high-level data paths of the traversal unit. In the following details are presented on the four functional units that form the traversal unit and the next section presents further optimizations.

Traversal Slices

Each traversal slice splits into three independent functional pipelines for calculating the traversal decision, and handling the stack operations push and pop (see Figure 4.5). The implementation is straight forward as there is no complex control flow for these operations. Simply all active rays assigned to the corresponding slice are enumerated (using *ENAC*) and fed into the computational pipeline.

The only interesting circuit is Collect Local Decisions which gathers the local decisions of all rays of the same packet assigned to that slice and computes the partial packet decision. Since within the pipeline it is easy to provide knowledge about the current ray and packet implementing this circuit directly in the traversal slice can simplify the design over an implementation in Global. This fact becomes clear when examining asynchronously decoupled slices which can compute partial results of different threads out-of-order and therefore require more complex data management to allow for read-modify-write cycles during computation of the partial results (see Appendix C for details).

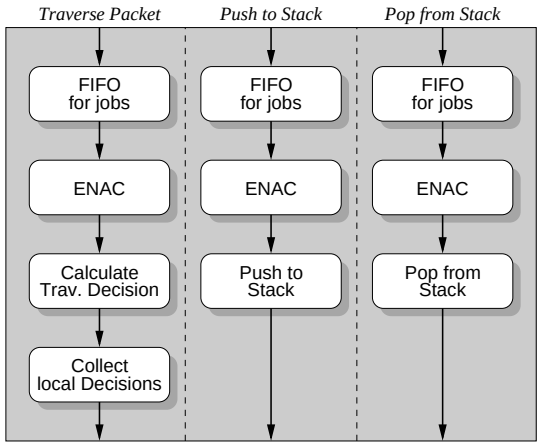


Figure 4.5: Data paths of a traversal slice with its three functional units. Each unit is implemented as a simple static pipeline without conditionals or loops.

Global

Since the tricky part of Global is implemented in the traversal slices the control flow of Global is rather simple which can be seen in Figure 4.6. Further implementation details on Global can be found in Appendix C.

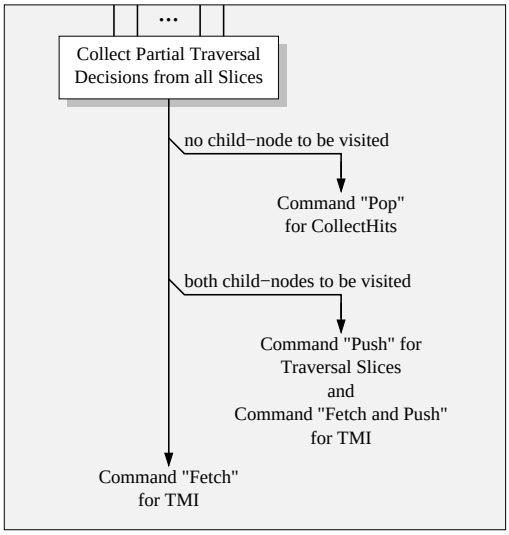


Figure 4.6: Control flow of Global. The tricky part of collecting partial traversal decisions computed out-of-order is greatly simplified as this is mostly done directly in the corresponding traversal slice. This way in Global there is no read-modify-write processing but only storing of the partial result sent by each traversal slice and therefore Global is easy to implement.

Collect Hits

The main purpose of Collect Hits is to collect the hit-information sent sequentially and out-of-order for all rays. Every result received from the intersection unit is compared to the best previous result and the closest valid hit is kept. If the current result is marked *last-triangle-in-voxel* termination is checked. Since it is needed to perform the comparison the previously best result and the current hit-information are both available simultaneously allowing to send the final result to the RGS without additional read cycles on the register file.

Therefore the implementation of Collect Hits is straight forward as can be seen in Figure 4.7. The most interesting parts are the computation of the stack-pop command which can be issued by three different sources and the packet finished command which has two different sources. These conflicts of several sources issuing commands simultaneously are solved by adding FIFOs to all but one path and connecting the FIFOs and the remaining path to a *multiplexer with prioritized port*. This multiplexer always selects the prioritized port if it provides valid data and otherwise chooses round-robin from one of the other ports with valid data.

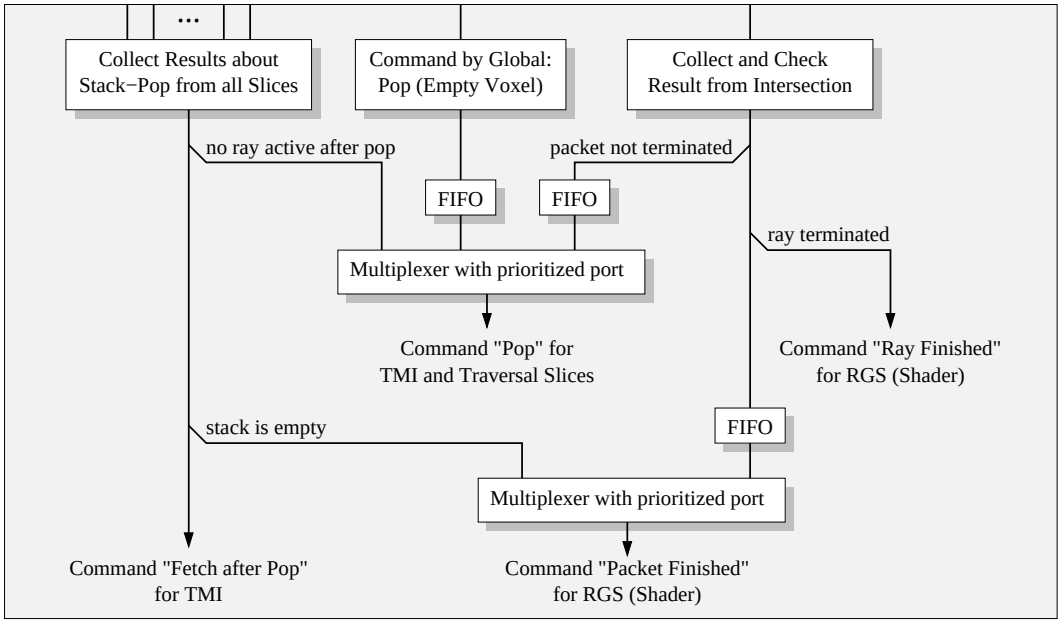


Figure 4.7: Control flow of Collect Hits. It can be seen that the most interesting component is only a simple management for several sources that can issue the same command simultaneously.

Traversal Memory Interface

The traversal memory interface (Figure 4.8) manages all memory requests and the stack for node addresses. Besides stack management requests it can receive commands to fetch the left or the right child of a node. In this case it reads the *offset* stored in the node

data structure from its local register file and adds it to the current address of the node[6]. Due to the alignment property of kd-tree nodes (which have the lowest four bits set to zero) calculating the address of the right node is simply done by setting bit three of the address to one.

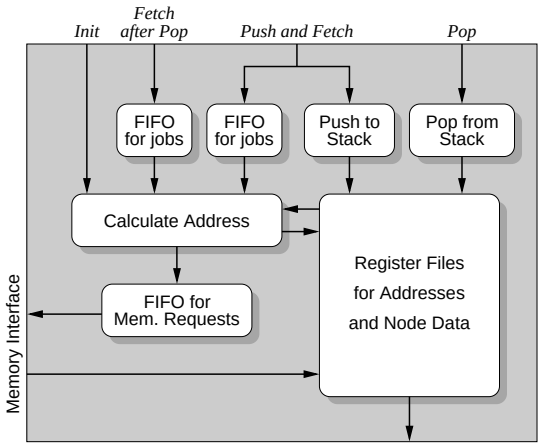


Figure 4.8: Data paths of the traversal memory interface which manages the stack for node addresses and performs memory requests for node data.

Stacks

The stacks form a large portion of the on-chip memory requirements. Building the stacks as a large block of memory accessed simultaneously by several units is hardly feasible. Fortunately, in the SaarCOR architecture the stacks are split into small and independent blocks of memory implemented in the TMI (storing node addresses) and the traversal slices (storing ray data). This splitting allows for an easy implementation with short connections since only the local unit needs to access the stack.

The implementation of the stacks is further simplified since at any time only a single unit processes a packet of rays. Thus although several units (e.g. the push and pop operations in a traversal slice) might read and write stack-pointers simultaneously the stack-pointers for any packet of rays are only accessed by a single unit at any time. This allows for building stacks using simple dual-port memory as Appendix C shows.

4.2.4 Optimizations

This section discusses some issues and derives optimizations on the functional units and data paths presented above.

[6] This adder can be saved by using only absolute addresses in the node data structure instead of relative addressing. However, relative addresses allow for easy and efficient memory management by relocating the kd-tree where it fits best in memory.

Synchronous vs. Asynchronous Decoupling

Using asynchronously decoupled slices sounds rather expensive as for each slice three additional FIFOs for job-management are required (see Figure 4.5). But since each entry in the FIFO consists of a single thread-ID and each FIFO needs to store only as many items as there are threads the memory requirements for these FIFOs are rather small. Let t be the number of threads per pipeline then the memory of each FIFO is $mem_{fif}(t) = LOG(t) \cdot t$ bits. Thus for a typical case (see Chapter 8) of 16 threads and 64 rays per packet this sums up to only 12 bytes per FIFO.

The design can be simplified by using a single FIFO for jobs per traversal slice. This FIFO then stores the commands for all three operations but its size does not need to be three times larger since at any time a thread can only be scheduled for at most one operation. This unification also allows for sharing the job-management circuit *ENAC*.

This sharing reduces the frequency in which jobs can be scheduled to the functional units of a traversal slice to one third (worst case compared to perfect usage). Fortunately for coherent scenes where each job provides work for several cycles (in the example above: 16 cycles if four traversal slices are used) it has no significant impact on the performance.

Traversal Memory Interface

When rendering incoherent scenes only few rays are active per packet, which does not provide enough tasks to fill the deep pipelines resulting in large latencies and a low frequency of memory requests. Therefore if using only few but large packets the available bandwidth might not be used optimally and the usage of the functional units becomes rather low.

At least partly this issue of a low frequency for memory requests can be solved by exploiting the knowledge that every node put to stack is likely to be visited later (typically with over 95% propability). Thus in those cases fetching not only the near-side but also the far-side and using a small FILO[7] to store the last n far-side nodes the latency of a memory request on the far-side node is greatly reduced.

For $n = 4$ already 10%–20% of all memory requests can be served using the prefetched data (depending on the scene and measured with 64 rays per packet). This is very much since only 15%–25% of all traversal steps want to visit both children. However, by increasing $n = 16$ only a moderate improvement of additionally 1%–2% can be achieved.

In a similar way also a prefetching mechanism can be used that reads both children for every node visited before it has been calculated, which child to visit. Both children are aligned and therefore highly suitable for a block transfer. This obviously causes an overhead in the memory bandwidth as always 50% more nodes are fetched than currently needed. But although a 128 bit wide memory interface delivers both nodes (each 64 bits and aligned) for the price of one, since typically only 15%–25% of all nodes want to visit both children 38%–42% of the caches are wasted using this technique as they store data which is not needed by any unit. However, the latency of the memory accesses is greatly reduced except for requests after a stack pop, which already have been solved by the first approach.

[7] FILO = First In, Last Out. See Appendix C for details.

4.3 List Unit

Traversal of a kd-tree continues in the traversal unit until a voxel is reached. Then the objects and triangles contained in those voxels need to be processed either by a transformation or an intersection unit. Since voxels only store references a unit is needed that reads these references and sequentially hands them over for processing at the corresponding unit. This enumeration and management is done by the *list unit*.

The list unit is a rather simple unit that always works on packets of rays in total and not on individual rays. Since its main purpose is fetching data from memory it heavily relies on multi-threading to hide memory latencies. When fetching lists of IDs prefetching of the next ID starts as soon as the current ID has been transfered to the transformation or intersection unit, respectively.

Mailboxing

Processing the same object multiple times is avoided by implementing a *mailboxing* algorithm (see Section 2.4.1). This algorithm takes as input an ID that has been fetched and then decides whether the corresponding object needs to be processed.

This technique reduces the workload on the transformation respectively ray-triangle intersection unit. However, it may also reduce the usage of these units since in case mailboxing returns that the current ID does not require further processing there is no alternative ID available that can be scheduled for processing instead.

In cases where the hit-rate of the mailbox becomes too high (e.g. when rendering standard kd-trees) adding a small FIFO between the list unit and the transformation respectively the intersection unit helps to average the workload over time. Additionally, if the list unit becomes a bottleneck it can be built in a super-scalar way by exploiting that on standard memory busses several IDs can be transfered in parallel.

The SaarCOR hardware architecture implements mailboxing as a small parallel list with typically 4 or 8 entries. The main advantage of this implementation is that initialization and checking for an ID can be performed in a single cycle. Due to the memory interface which has variable latencies and returns requests out-of-order FIFOs are needed in the transformation and intersection unit. As as positive side effect these FIFOs average the workload over time keeping the functional units busy even when the list unit does not provide new IDs.

Optimizing the Enumeration

Each command sent by the traversal unit contains the address of a list of IDs and the number of items contained in that list. Therefore two adders are required for fetching lists: one to increment the address and one to decrement the number of items.

The latter adder could be saved by adding a flag to each ID denoting the end of the list. This would additionally reduce the width of the data path from the traversal to the list unit since then the number of items does not need to be transfered.

However, using an end-of-list flag does not allow for storing IDs in a packed format as shown in Figure 4.9. This packed format allows for great savings in memory storage

(see [Wal04]) and also can increase the cache hit-rate. Therefore in general the advantages of packed lists outweigh the savings of a single integer adder of typically less than 16 bits width.

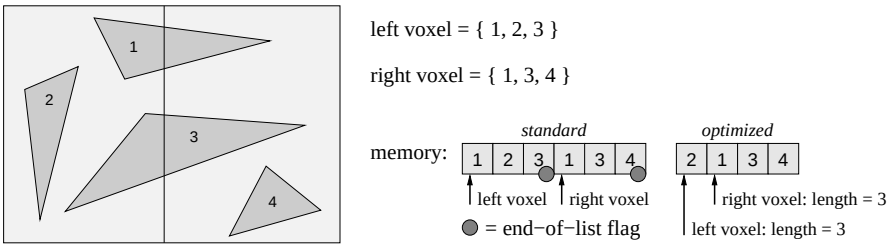


Figure 4.9: Example for optimized storage of triangle-ID lists. With an end-of-list flag only the standard variant of storing lists of IDs can be used. With an additional adder the optimized variant of storing *packed lists* of IDs can be used which allows for great savings in memory storage.

4.4 Transformation Unit

As presented in Section 2.2.1 a hardware that supports dynamic scenes using rigid objects needs to be able to transform a ray $R_a = (O_a, D_a)$ from one coordinate system into ray $R_b = (O_b, D_b)$ in another coordinate system.

This transformation uses the affine transformation matrix T= (M, N) with $M \in \Re^{3,3}$ and $N \in \Re^3$. Since a ray consists of two components two transformations have do be performed: $O_b = M \cdot O_a + N$ and $D_b = M \cdot D_a$.

The implementation of the transformation unit is straight forward and can be pipelined trivially. Again FIFOs for jobs and multi-threading are used to efficiently schedule new work to the transformation unit without introducing idle cycles.

Arithmetic Complexity and Optimizations

Implementing this unit requires many floating-point operations and therefore is rather costly. Unfortunately in most applications the transformation unit is hardly used since transforming rays is only required after many traversal steps. Therefore it pays out to reduce the cost of the transformation unit by reducing its performance.

Instead of the full operation only a circuit to calculate $X_b = M \cdot X_a + N$ with $X_a, X_b \in \Re^3$ is implemented. This circuit directly calculates the transformation of the origin but can also be used to transform the direction when setting $N = (0, 0, 0)^T$.

With this optimization transforming a ray requires two steps (which is half the performance of a full implementation) but the cost of the transformation unit has roughly been cut in half. The standard implementation requires 18 floating-point multiplications and 15 floating-point additions while the reduced version requires only 9 multiplications and 9 additions.

For packets of n rays that share the same origin (like primary or shadow rays) only $n+1$ steps are required instead of $2n$ steps since the origin needs only to be transformed

once per packet. As a result in a standard configuration with packets of 64 rays (see Chapter 8) only less than 2% of the performance is lost at the transformation unit (since 65 instead of 64 operations need to be performed) for packets sharing the same origin but the costs have been reduced drastically.

4.5 Intersection Unit

The core of any ray tracing algorithm is formed by ray traversal and the intersection of a ray with a geometric primitive. But while ray traversal requires only few operations the intersection of a ray with a triangle is a compute intensive task. Therefore it has been investigated by many researchers and led to several different algorithms, see [MT97, Eri97, Wal04]. But although the hardware complexity of these algorithms varies (see Chapter 7.3.1) their pipelined implementation in an *intersection unit* is always straight forward and similar to the transformation unit.

Unit Triangle Intersection Method

Besides these general ray triangle intersection algorithms Arenberg [Are88] has presented a variant that uses affine transformations to preprocess ray data. After this preprocessing the actual intersection computation is trivial. Therefore this method becomes very interesting in the context of a hardware architecture that supports dynamic scenes and for this purpose already contains a dedicated transformation unit.

The following presents a slightly extended version of Arenberg's algorithm developed by Sven Woop [Woo04] and that additionally computes the dot product between ray direction and the normal of the triangle for free. Furthermore this method fits nicely into the architecture as such an intersection unit does not require a memory interface. The only data required is an affine transformation, which is fetched by the transformation unit.

Figure 4.10 illustrates the *unit triangle intersection method* consisting 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.

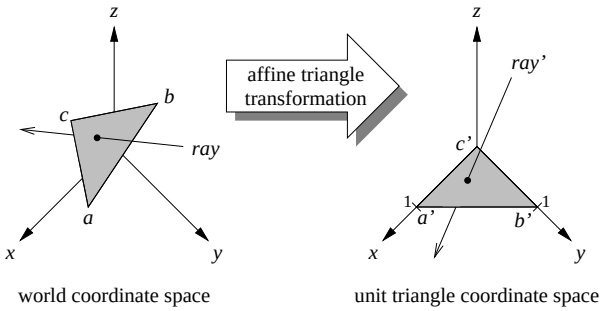


Figure 4.10: The unit triangle intersection method consists of two stages: First the ray is transformed, using a triangle specific affine triangle transformation. In the second stage, a simple intersection test of the transformed ray with the unit triangle is performed.

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. 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 bijective affine transformation.

Testing the 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 (see Section 2.1.3) of P within Δ do not change under a bijective 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:

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

Dot Product Preservation

An additional analysis of Arenberg's algorithm [Are88] allows for computing 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 $D' \cdot N_{unit} = D'_z$ (see [Woo04] for a detailed proof). This property can be exploited when designing shading units, as shaders typically require the

cosine between the direction of the ray and the geometric normal for color calculation and ray generation (see next chapter).

This concept of first transforming a ray to a canonical coordinate space before intersecting it can also be applied to many other types of geometric primitives, such as quads, discs, boxes, ellipsoids, spheres, cylinders, pyramids, etc.

Another advantage is that only a single representation (the transformation) needs to be stored together with a flag indicating the type of primitive. Only a much simpler and smaller primitive-specific second stage intersection unit must be added.

There is also a drawback since the intersection test is not performed using the original ray and primitive. Since every calculation has only a limited precision, the transformation from world coordinate space into unit coordinate space, with possibly several orders of magnitude difference, might lead to a loss of precision. This loss actually occurs twice since both the primitive and the ray are transformed before the intersection is computed. This can cause visible errors for some scenes when floating-point numbers with a low precision are used, e.g. in the SaarCOR prototype (Chapter 7).

5 Shading

"A supercomputer is a device for turning compute-bound problems into I/O-bound problems."
Ken Batcher [Bat]

The previous chapter presented the core of a ray tracing system, i.e. the components for tracing rays and intersecting objects. For image synthesis the color that is contributed by a ray needs to be calculated. However, the focus of this chapter is not the shading computation itself but a framework that connects RTC and memory interface to one or more components that perform the shading computations.

For the purpose of this chapter it is assumed that there is a dedicated piece of hardware that receives rays and the corresponding hit-information, and performs shading calculations which may generate additional rays. On return from the RTC the results of these rays are again handed over to the black box. Therefore the actual implementation of the shading unit is irrelevant and only its I/O characteristics are important, i.e. the way data is exchanged, the frequency in which new rays can be given as input, its latency, and its memory access behavior.

The outline of this chapter is as follows: In the first part some general issues and characteristics of hardware for shading operations are discussed. Then a *general architecture for shading* is presented, which allows for plugging in arbitrary shading units.

An example of a *fixed function shading unit* was developed for the SaarCOR prototype (see Section 7.3.2). However, since fixed function shading can be treated as a special case only the more general approach to shading in a ray tracing context is presented in Section 5.3.

There the *SCPU* is described, a *general purpose processor* that was designed to have very low costs and uses a minimalistic instruction set with several extensions to support ray tracing. Since these extensions are not specific to any CPU they can be used to improve the efficiency for ray tracing on any other processor architecture.

5.1 General Characteristics and Issues of Shading

There are many different ways how shading operations can be implemented using various sets of parameters, material settings, textures, and secondary rays for evaluation. However, all possible implementations have many characteristics in common which allows for building an infrastructure capable of hosting any kind and number of *shading processing element* (SPE).

Input Frequency and Latency

Two of the core characteristics are the frequency in which new rays can be sent as input and the latency it takes to compute an output. However, neither the frequency nor the output need to be constant but can vary depending on the shading performed and the characteristics of the memory interface. But at least the average frequency and latency can be measured for a given set of scenes or derived from the implementation of the SPE.

These average values are important to evaluate how many SPEs are necessary to avoid making shading the bottleneck. For example Chapter 8 shows that typically every 10 to 20 cycles a ray is sent from the RTC to the RGS for shading. Thus if a single SPE can not provide a corresponding input frequency several SPEs have to be used in parallel.

Exchanging Data

In principle the parallelization of SPEs is trivial since there are no data dependencies between rays but issues like load-balancing and data management should be taken into account. For load-balancing it is important to equally distribute the rays of a packet across several SPEs (which can be done similarly to load-balancing on traversal slices, see Section 4.2.1).

More complicated is the management of data itself. Any mechanism using *polling* or *interrupts* is too slow and wastes many cycles on waiting and protocols. Therefore the SPEs are implemented using the *push-model*. Here a dedicated managing unit – the *master* – keeps track of the states all SPEs are in (e.g. busy or having idle threads) and pushes the content of a new job directly into the register file of the corresponding SPE.

However, the push-model only works out well if data can be written by the master simultaneously to the execution of the standard operations on the SPE. Therefore similar to a streaming computer model, the register file is split into three parts: *input*, *scratch*, and *output* registers. The master can only write to the input and read from the output registers. In contrast a SPE can only read from input and write to output registers. The scratch registers can not be accessed by the master but are fully readable and writable by the SPE to allow for general purpose computations during shading (see Section 5.3.2).

Clustering Rays Into Packets

A thread on a SPE handles exactly one pixel of the image. The computation starts with the generation of the primary ray and terminates after the last recursively generated ray is fully shaded and the final color of the pixel is calculated. For standard rendering this pixel and all corresponding rays can be calculated independently of any other pixel or ray. Therefore no communication between threads or SPEs is necessary.

However, since the RTC works on packets of rays only when (primary or secondary) rays are generated they have to be clustered into packets. The following lists three different options of how to cluster rays into packets. Please note, that although it is possible to cluster secondary rays of different primary packets into new packets in the SaarCOR architecture only rays generated by a single packet are clustered as this greatly simplifies clustering of rays.

Fire-And-Forget A brute force solution to this problem uses *self contained rays* only: Each shader simply spawns all secondary rays it requires and terminates without waiting for these rays to return. Instead it adds all required information to each ray allowing to continue shading when this ray returns. This solution results in very efficient load-balancing but requires some effort for clustering the rays.

The main reason not to use this concept is the large amounts of memory used to store the rays and the problem of potential deadlocks which always arises with limited amounts of storage when there is no limit in the storage requirements. However, the on-chip storage requirements can be lowered by adding a *swapping mechanism*, but this requires additional data paths with a rather high peak bandwidth and complicate ray management and scheduling.

An alternative method to costly swapping is *killing of packets*. While swapping saves the current immediate results and later continues the work killing packets simply throws away all partial results of a packet of primary rays (including all secondary rays). Then when enough resources are available the calculation is started from scratch again.

Thus there is a trade off between the costs for saving intermediate results and the costs of recalculating the intermediate results after throwing them away. Nevertheless in [Dre05a] it was shown that if hardware parameters are chosen reasonable for the desired scene complexity typically only few packets have to be killed and therefore killing has only a minor impact on the performance.

Sequential Ray Generation However, there is a more cost-efficient solution to the problem of clustering rays. Instead of spawning all rays at once, simply only one type of ray is generated at any time. This requires a first iteration over all rays and shaders in a packet to collect which ray each shader wants to spawn next. Then for each type of ray sequentially all corresponding shaders are called and a paket of rays of only this type is generated. While this makes clustering of rays trivial it is very inefficient to iterate over all shaders several times.

Coordinated Ray Generation There is another method which is similar to the sequential ray generation and has roughly equal costs but far less overhead in management. This method is suitable for any type of SPE and was used in the SaarCOR prototype (see Chapter 7). Since this method is implemented in the general architecture for shading its details are described there (see Section 5.2.3).

5.2 General Architecture for Shading

The *General Architecture for Shading* (GAS) is a framework to connect several *shading processing elements* (SPE) to the ray tracing core (RTC). Therefore it provides the infrastructure to manage memory accesses, transfers ray data, starts jobs on the processing elements, and collects their results. The GAS together with the SPEs provide the functions to generate and shade rays and thus form the RGS. The data paths of the GAS and the embedding of any SPE (fixed function or programmable) are shown in Figure 5.1.

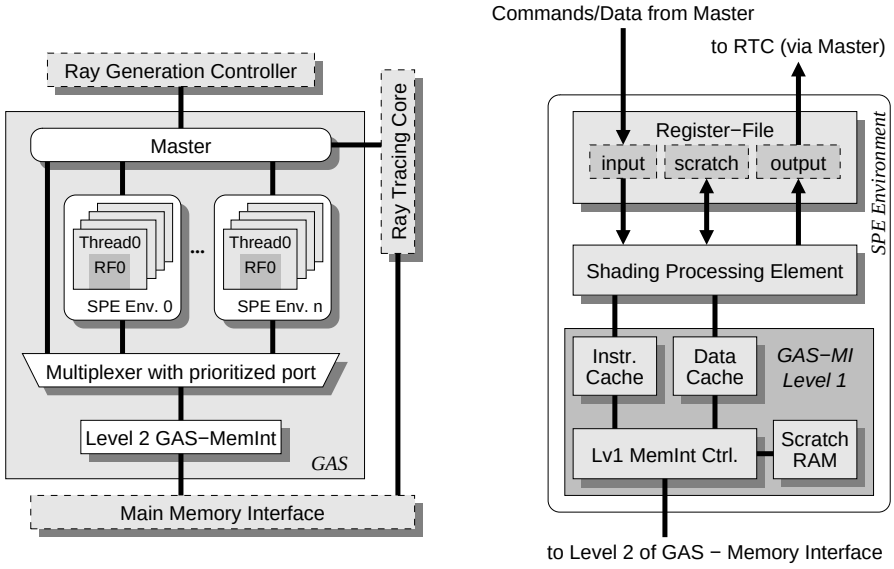


Figure 5.1: Data paths of the *general architecture for shading* (GAS) and a suitable embedding of an arbitrary *shading processing element* (SPE). The GAS provides the infrastructure to manage memory accesses, transfers ray data, starts jobs on the processing elements, and collects their results. The GAS together with the SPEs provide the functions to generate and shade rays and thus form the RGS.

5.2.1 Master

Since the RTC works on packets of rays to reduce the memory bandwidth the GAS also has to support packets of rays. However, shading can be performed in various different ways and therefore managing packets of rays in the RGS is more complex than in the RTC. This management is performed by the *master*, a special functional unit, which implements several efficient schemes for data management described in the following paragraphs.

Initially, the master receives the coordinates of the pixels to render from the *ray generation controller* (RGC). Each pixel is assigned to a different thread on one of the SPEs such that all rays of a packet are distributed evenly across all SPEs for load balancing.

All operations necessary for that pixel ranging from primary ray generation over shading including the secondary rays but except for post rendering effects are handled by this thread. Thus secondary rays are not moved to other threads for load-balancing since this would require data transfers between SPEs.

The master transfers all input data into the register files of the SPE and initiates the corresponding process on the SPE. For example these processes can be primary ray generation, any kind of shading, or image filtering. On fixed function SPEs the operation is specified using an ID while on programmable SPEs simply the starting address of the corresponding program is pushed into the *program counter* (PC) register of the corresponding thread. After all data has been transfered the master sets the corresponding *valid* flag and the SPE begins with the execution.

When the processing terminates a flag signals the master to check the corresponding

output. If any rays have been generated they are transfered to the ray tracing core for processing. Finally, the results of the traversal are transfered back into the SPEs where processing continues.

5.2.2 Packet Shading

Similar to traversing packets of rays where a single item is fetched from memory and used for all rays of the packet the concept of packet shading tries to reduce the bandwidth requirements. However, since shading has much more parameters than traversal or intersection computations exploiting coherence by packet shading is much more complicated.

In a simple version of packet shading the master checks for every ray in the current packet which triangle was hit, and builds the set of triangles hit by this packet. The data needed for processing each triangle of the set is fetched by the master and then broadcasted to all threads processing corresponding the rays. While this first version looks up only the triangle that was hit a straight forward extension performs an additional look up of the material of the triangles and fetches and broadcasts the common material data, too.

However, although performing memory operations only once per packet can greatly reduce the bandwidth requirements it clearly has its limitations. Obviously it has to be decided which parameters are common to all triangles or materials such that fetching by a master is useful and no overhead occurs due to a master loading data not needed by specific shaders.

An additional example for limitations are texture lookups which first require the calculation of the texture address. However, that address can depend on arbitrary parameters defined in the corresponding shaders and therefore in general its calculation can not be performed efficiently by a master.

Thus additionally to packet shading caches should be used. On the design of caches for textures in a rasterization based environment several papers have been presented [IEP98, IEH99, Blo98, HG97, SKS96]. However, our measurements (see Chapter 8) have shown that standard schemes for caching already work out well also for shading data.

5.2.3 Coordinated Ray Generation

During shading several secondary rays might be spawn, which have to be managed and clustered into packets again. Since Section 2.3 has shown that the RTC can be built to handle even invalid packets no special care must be taken to avoid errors. However, the efficiency degrades also for valid packets if incoherent rays are clustered in the same packet.

The following system allows for coordinating the generation of secondary rays without introducing an overhead for communication between different shaders. When designing the shaders for a scene all possible types of rays are enumerated and these numbers are used as an ID when generating the corresponding type of ray. An example of an enumeration is shown in Table 5.1.

Every shader is written such that of all secondary rays a shader wants to spawn always the one with the smallest ID is generated first. When shading a new packet of rays all shaders halt after spawning their first rays. Then the master sends to the RTC a packet

ID of type		Type of Ray
$r \cdot e +$	0	Primary ray
$r \cdot e +$	$1, \ldots, l$	Shadow rays for l light sources
$r \cdot e +$	$l+1$	Reflection ray
$r \cdot e +$	$l+2$	Transparency ray
$r \cdot e +$	$l+3$	Refraction ray
$e =$	$l+4$	

Table 5.1: Example for a classification of ray types using IDs. Here r denotes the level of recursion, l the number of light sources, and e specifies the total number of different ray types.

with the smallest type of all generated rays where all rays which have a different type are set to inactive. When the packet returns only the shaders of the active rays are continued. An example of this process is shown in Figure 5.2.

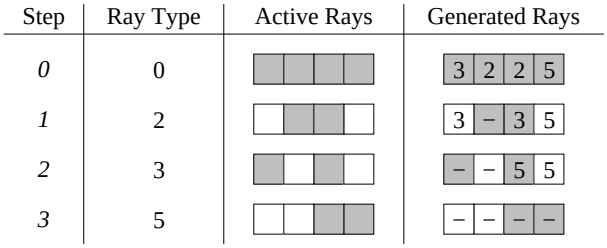


Figure 5.2: Example of Coordinated Ray Generation. In the first step a primary packet (ray type 0) with four active rays is shaded by the corresponding shaders. For every active ray (denoted by the grey box) the secondary ray to be traced next is generated (the type of this secondary ray is denoted in the boxes in the right most column). The type of ray with the smallest ID (here: 2) is traced in the next step. All rays of higher type are kept but marked inactive (denoted by white boxes). This process is iterated until all secondary rays have been traced. Since always the the ray with the smallest type is traced next no deadlocks can occur and no rays can be lost even if shaders do not generate secondary rays in ascending order of the ray type.

It is important to note that no deadlocks or errors can occur even if a shader does not generate rays in ascending order of the ray type. Furthermore this concept can be evaluated very efficiently in hardware and does not require multiple iterations over shader programs or communication between the shaders. These features of coordinated ray generation allowed for an efficient implementation in the SaarCOR prototype (see Chapter 7).

5.2.4 Communication Schemes

For standard rendering every pixel and all corresponding rays can be calculated independently of any other pixel or ray. Therefore no communication between threads or SPEs is necessary. All management of clustering rays into packets and combining memory

accesses is done transparently by the GAS.

However, for advanced shading effects and image filtering techniques exchanging data between SPEs can be useful. Therefore similar to the *SB-PRAM* [PBB+02] a memory based communication scheme is implemented using dedicated memory processors. This scheme is highly efficient since communication via memory only means that the memory interface is used for communication and not that every data transfered has to be stored in slow external memory and read back.

Similar to all other functional units also the SPEs and the master are connected to a shared memory using a simple multiplexed bus with labeled broadcasts for the data returned (see Chapter 6 for general details on the memory interface). However, the use of the two level *GAS memory interface* is optional but can allows for increased performance e.g. in conjunction with hierarchical caches and shading effects with high bandwidth requirements. Obviously the instruction cache is only relevant for programmable SPEs.

5.2.5 Ray Mapping

The RTC uses multi-threading where each thread corresponds to a packet of rays. In contrast the SPEs use multi-threading where each thread is a single ray. Therefore exchanging data between the RTC and the GAS requires some sort of *ray mapping*.

The following description of ray mapping uses the term *ray-slot* for a thread of a SPE regardless whether is actually contains a ray under computation or is idle. In the same spirit the term ray-slot is also used for the data structures of the RTC which could store an individual ray although the RTC actually works on full packets and not on single rays.

Given p the number of packets per RTC, r the number of rays per packet, c the number of SPEs per GAS, and t the number of threads per SPE. Then in the simplest version the number of ray-slots in the RTC is $p \cdot r$ which is equal to the number of ray-slots $c \cdot t$ in the GAS.

Besides this classical one-to-one mapping of ray-slots one could also have more ray-slots in the GAS than in the RTC. This is an option for complex shading where the number c of processors is increased or when the latency of the RTC is rather long compared to shading and therefore more threads t are required to keep the SPEs busy.

Having twice more ray-slots in the GAS than in the RTC can be realized trivially by mapping two ray-slots of the GAS to a single ray-slot in the RTC in a round robin fashion. This mapping is trivial since for each packet of rays the RTC does not store any data besides temporary intermediate results.

The other case of having more ray-slots in the RTC than in the GAS is of interest if the RTC has very long latencies for memory accesses or computations. Therefore two ray-slots of the RTC share the same ray-slot in the GAS. This is not trivial and requires careful management since the memory resource and the register file of the thread on the SPE has to be shared between two independent rays.

Fortunately the latter case plays hardly any role since typically more processing power is needed for shading than for the RTC and memory latencies affect shading and tracing rays equally and therefore require the same amount of threads for latency hiding.

In all cases the distribution of rays to threads in the SPEs can be made similar to the distribution of rays to traversal slices (see Section 4.2.1). This allows for a statical

distribution that achieves a good load-balancing for typical cases.

5.2.6 Managing Threads on the SPEs

Shading processing elements can be arbitrary circuits that are either fixed function or programmable. Since typically multiple elements are integrated on the same chip and accessing a shared memory they are a highly attractive target to perform also other computations besides shading. For example SPEs in a ray tracing system are suitable for image post processing, calculation of kd-trees, and even simulations for physics.

Although the GAS puts hardly any constraints on the SPEs regarding memory accesses and program execution the SPEs follow the push-model, which does not allow for standard job management techniques like forking processes[1]. Therefore this section discusses how the various tasks are started on SPEs.

Obviously there is a trivial solution if the SPEs are programmable since then a program once started can run forever and simply execute arbitrary tasks sequentially using polling and memory based communication schemes for job distribution, e.g. *parallel job queues* [Röh99]. But this trivial solution is rather ineffective when it comes to synchronization to specific events, e.g. building kd-trees prior to start rendering or filtering the image after the last ray has been shaded.

In addition, during ray tracing there is the need to efficiently start a shader not only depending on the material of the object but also on the type of the ray. For example if an object is hit by a primary or reflection ray then the corresponding thread should be frozen and a *surface shader* corresponding to the material should be started to process the ray. But if the ray does not pierce an object then an *environmental shader* that calculates for example the sky should be started. However, for standard shadow rays or test rays of a physics engine typically not the color of the ray but only the information whether any object has been hit is of interest for further computations.

The simple solution for these mappings is the use of a table that maps the type of the ray and the information whether an object has been hit to the ID of an operation to perform (respectively the address of a program to execute on programmable SPEs).

This trivial table based approach can be extended easily to also support synchronized events, e.g. prior to rendering on all threads of the SPEs an *init frame operation* can be started. Further interesting events to synchronize to are when there are no more primary rays to generate and the image is not yet finished (which allows for using processing power of idle threads), and whenever a packet respectively image is finished.

5.2.7 Managing Temporary Storage

Sometimes shader require memory to store intermediate data or to swap out registers. Since the GAS supports the execution of multiple shaders which can access the same memory simultaneously, the memory a thread can use for private purposes has to be managed somehow. Costly communication for locking, synchronizing and management of global data structures is avoided by using a simple unit called *hw-malloc*[2].

[1] Forking a process means that after forking there are two independent threads running (virtually) in parallel on the same processing element.

[2] `malloc` is a standard command of the language *C* and its name comes from memory allocation.

This unit consists of a standard FIFO initially filled with all available memory pages. Whenever a thread needs a new page a simple `load` command on a special address is performed and the return value is the base address of the malloced page. Freeing a page is performed similarly by executing a `store` command to the same address and with data of the page's base address, which is then simply put again into the FIFO. Since operations on the FIFO are performed atomic no race conditions can occur[3].

The concept of hw-malloc can be extended easily to support different FIFOs for various sizes of pages and different types of memory. For example the memory interface can be implemented hierarchically with each level having a separate cache, hw-malloc and some scratch RAM for temporary storage.

Then performing a malloc operation is also performed hierarchically and if there are no free pages on the first level the request is forwarded to the next level. Since the scratch RAM is memory mapped there is no need for the shader to distinguish where the page is actually located.

5.3 SCPU

In the last decades quite an enormous amount of different processors have been presented and built[4]. But the reason why this thesis presents yet another processor is simple: Almost all processor designs deal with speeding up the computation of a single sequential program and in many cases expensive hardware is used to achieve this goal.

In the context of a ray tracer this is the wrong concept since instead of a single sequential program there are rather arbitrary numbers of small and independent program fragments. Furthermore these fragments do not have to be computed as fast as possible and therefore allow for optimizing the CPU for efficiency rather than for execution time.

The *SCPU* is a minimalistic, general purpose, multi-threaded CPU following the push-model and designed for a multi-processor environment with memory based communication schemes. It uses a RISC-like load/store architecture with all operations performed on registers and only few instructions supporting immediate constants.

It is designed for efficiency and not to speed up execution of a single sequential program and therefore has no hardware support for *speculative execution*, *branch prediction* or *forwarding*. Instead all of these issues are simply solved by multi-threading, which does not only hide all latencies from structural, data, or control hazards but also from memory accesses and while rays are traced by the RTC.

[3] Obviously freeing pages not owned by any or a different thread can lead to errors. But since every `load` and `store` request is tagged with the IDs of the CPU and the thread a mechanism for automatical checking can be implemented easily in hardware. This furthermore allows for restricting the number of pages per thread and to implement various forms of resource managements.

[4] The best known commercial processors are made by *Intel* [www.intel.com], *AMD* [www.amd.com], *Sun* [www.sun.com] and MIPS [www.mips.com]. Important textbooks on processor design (including the advanced techniques mentioned later in this section) are especially [MP00] which provides precise definitions and formalisms but also [Fly95, Hwa93, Joh91, Car03, HP96] of which the latter one is for computer architecture what [FvDFH97] is for computer graphics.

Processor Design

It is generally believed that special purpose hardware is best suited for compute intensive tasks that can be pipelined easily and general purpose processors are ideal for arbitrary computations with case switchings and data dependencies. But in fact the best efficiency on general purpose processors is achieved by compute intensive tasks that can be pipelined easily and degrades heavily when case switching and data dependencies occur.

The reason is simply that modern general purpose processors are deeply pipelined and even branch prediction, speculative execution, and multi-issue on super-scalar ALUs can only minimize the hazards but never achieve the efficiency of a compute intensive program, which does not have any hazards.

Therefore the design of the SCPU is quite different in many ways from those of current desktop CPUs. First of all it uses multi-threading with many threads to allow for efficient latency hiding[5].

This multi-threading allows for a strict sequential execution of all programs without forwarding or instruction reordering since in case of any hazard the thread is simply switched. Thus each program is still written like any other sequential program but hardware efficiency is achieved by always switching to a task that can be processed without wait-states or dependencies (see Section 5.3.2).

Additionally, the ALU of the SCPU has hardware support for some specific operations, which can be realized easily using only a few gates but require several standard instructions if implemented in software (see Section 5.3.1).

Parallelization

From *Amdahl's Law* [Amd67] it can be derived that when designing hardware to perform a variety of arbitrary operations the average case of a balanced mixture of all operations does not profit from speeding up only some operations. Therefore the SCPU is optimized for efficiency and to have low costs allowing for having many SCPUs in parallel on the same chip thus speeding up every instruction equally through parallelization.

Nevertheless, the design of the SCPU still allows for optimizations specific to the benchmark the chip is designed for. For example many interesting shader programs can be written to perform mostly vector operations. In this case a SCPU with a super-scalar ALU and functional units for parallel vector operations can pay off.

Current CPUs try to execute a sequential program in parallel on super-scalar architectures by using multi-issue of several instructions fetched in the same cache-line or using VLIW or EPIC[6]. The SCPU uses only in-order instruction issue of parallel threads and therefore does not require any of the complex mechanisms to parallelize instructions of sequential programs for multi-issue architectures (see Section 5.3.3).

Besides multi-issue on parallel threads a mechanism called *background hardware program*

[5] Multi-threading should not be confused with Intel's *Hyper-Threading* [Int02] which uses only very few threads (on *Intel Xeon* processors it is two threads) and therefore does not provide all features of multi-threading.

[6] *Very long instruction words* (VLIW) consist of multiple structural hazard free instructions combined to simultaneously issued instruction groups. A recent example of such an architecture is Intel's Itanium CPU which belongs to *explicit parallel instruction set computing* (EPIC) [SR00].

(BHP) is supported, which has extra dedicated hardware resources to allow for efficient execution of complex programs like DMA-like memory transfers, kd-tree traversals, and ray-triangle intersections (see Section 5.3.4).

5.3.1 Arithmetic and Logic Unit (ALU)

The SCPU does not require a special ALU but can use an arbitrary ALU with standard instructions for floating-point and integer operations, and logic functions. But the ALU of a CPU designed for shading in a ray tracing context allows for several optimizations of which some examples are shortly presented in this section.

In the spirit of minimalistic extensions that allow for performing several standard instructions using a single operation texture and frame-buffer color format conversions can be supported. Similarly calculating the traversal decision uses only 13 gates (see Chapter 2.1.3) but removes several cycles used for case switching and data management with standard instructions. The same holds for tests and case switchings in ray-triangle intersection tests. Finally, also the calculation of the texture address can be supported trivially for texture sizes that are powers of two (see Chapter 7).

The most flexible way to achieve an application specific speed up is to attach a small FPGA to each ALU. These hybrid CPUs have been shown to work out very well for special operations [ANA04, Hau00, YSB00],[`www.stretchinc.com`] and would be of great benefit for computations in a ray tracing environment. Since ray tracing allows for many balancing options reconfigurable architectures like those hybrid CPUs or FPGAs in general allow for choosing the balancing dynamically based on current measurements resulting in best case frame rates. A similar approach for rasterization based hardware has been shown recently [HL03].

Finally, in the spirit of hardware efficiency some functional units can be recycled for other computations. For example every floating-point adder contains an integer adder for the mantissa and every floating-point multiplier contains an integer multiplier for the mantissa and an integer adder for the exponent.

Therefore if no full precision for integer operations is required (e.g. shading is performed using floating-point values and integers are only used for address calculations) those parts of the floating-point circuits can be recycled for integer operations. This allows not only for savings in hardware but also simplifies keeping the functional units of the ALU busy. Thus a floating-point ALU for single precision IEEE numbers can compute at least 23 bit integer multiplications and 31 bit integer additions with almost no additional hardware.

5.3.2 Register Files (RF)

Similar to the RTC each SCPU supports multi-threading natively with separate contexts using several sets of registers where each set corresponds to a different thread. This technique has been proven to perform very well on the *SB-PRAM* [PBB+02], but was further optimized to even perform well when there are only a few threads active (see Section 5.3.3).

Thus threads do not share resources of the register file and therefore do not require complex management and deadlock prevention. This allows for many optimizations since registers of various threads can be addressed independently removing bottlenecks of mul-

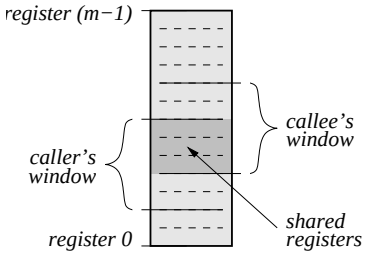
tiple reads and write backs (see Section 5.3.4).

A thread manages all computations of a pixel starting with the primary ray(s) and lasting till the last secondary ray has been shaded. Since all data required for future processing stays on the CPU during that period a mechanism for efficient management of the registers is required.

Register Windows

Therefore instead of providing only n registers for each thread which might need to be swapped out to memory during function calls there are $m > n$ registers for each thread ($n, m \in \mathbb{N}$). A simple windowing mechanism (similar to [SI91]) is used that allows for accessing a subsequent range of n registers from all m registers. The location p of that window ($p \in \mathbb{N}$, $0 \leq p < m - n$) can be specified using a special instruction[7].

This allows for easy communication between a child process and its parent on a function call (see Figure 5.3). Since the number of registers per thread is fixed swapping register content to memory might still be required. But each implementation of the architecture can be designed to support a typical set of secondary rays and function calls without swapping.



exchanging data on function calls

	master r	master w	program r	program w	memory r	memory w
PC	–	(x)	x	x	–	–
Std. Regs.	–	–	2x	x	–	–
Load Reg.	–	–	x	–	–	x
Constants	–	x	x	–	–	–
Ray Input	–	x	x	–	–	–
Ray Output	x	–	x	x	–	–

assignment of read and write ports

Figure 5.3: On the left there is an example for register file separation and the use of register windows for exchanging data on function calls. The table on the right lists the operations performed on the register file for all units. The PC is a special register that is additionally read during the instruction fetch. Furthermore when starting a new program the address is not taken from the PC but from an additional register-file that can be written by the master.

Further improvements of this scheme are possible by providing multiple windows, e.g. splitting n into k parts n_i (with $0 \leq i < k, \sum_{i=0}^{k-1} n_i = n$) and allowing for setting each n_i independently. The efficiency of the hardware implementations can be increased if n, m, n_i as well as the starting positions of the window(s) are powers of two.

[7] Instead of using an additional instruction the pointer can be modified alternatively by using the standard `store` command to special address.

Multiple Simultaneous Accesses

Similar to this windowing mechanism there is a standard problem on push-model based CPUs: while one thread is running data for another thread is pushed into the register file by the master or the memory interface which requires two independent write-ports to avoid stalling. Since threads are independent this could be solved by having separate register files for each thread. But a register file architecture separated by threads requires complex routing schemes and therefore is very costly in terms of routing resources.

Therefore a compromise is being used in the architecture of the SCPU. Instead of having many separated register files it is split similar to a streaming computing model. There the register file contains three parts: the input, scratch and output registers. However, since only the number of write-ports per register files is an issue greater flexibility is achieved by only splitting the register file into two parts: input registers and combined scratch and output registers.

Thus there are only two register files and every thread has a separate window in each register file (see Figures 5.1 and 5.3). One of the register files is a standard general purpose read/write register file (RWRF) while the other is read-only (RORF) by the SPE but can be written by the master (while the master can not write to the RWRF).

Example for Register Assignment

The RWRF is used during computations, for exchanging data on subroutine and function calls, and for return values which are collected by the master. These return values are for example the ray data of the next ray and the data for the frame buffer (if not transfered by the software).

While all registers i (with $0 < i < n$) are mapped to registers $p + i$ to simplify address computations (see Section 5.3.5) register 0 is always mapped to the *program counter* (PC) register independent of p. However, to avoid introducing a severe bottleneck the PC is not part of the register file but implemented as separate registers.

The RORF contains all inputs to a shading program, which are most importantly the hit-information including the triangle's material data if fetched by the master (see Section 5.2.2). Furthermore, the RORF contains the *previous PC* register, which always contains the PC of the previous instruction to support function calls and the *load register*, which contains the return value of memory requests (see Section 5.3.5).

There are many values, which are common to most programs, e.g. typical constants like 0 and 1, scene specific information like the number of light sources, or the current seed for a noise function specified on a per-frame basis. Since transferring immediate values to registers always requires an additional instruction and loading data from memory takes several cycles computations are speed up by having k *special values* in the RORF.

These special values can be constants as described above or standard masks for swizzle and bit operations. Since floating-point operations require different bit strings than integer operations to denote the same number every register actually stores two values and the corresponding value is selected by the type of the instruction (see Section 5.3.5).

Obviously these special values are the same for all threads and are independent of the

position of the thread's window[8]. Therefore these $2 \times k$ registers are very cheap to implement even if they are not hard-wired but writable by the application (via the master). Table 5.4 in Section 5.3.5 shows that with $k = 16$ for typical cases all important constants can be implemented.

5.3.3 Optimizations for Multi-Threading

In the *SB-PRAM* project [PBB+02] multi-threading is used in a fixed round-robin fashion where every thread is executed for one cycle before switching to the next thread. It has been shown that this concept performs very well if all threads contain active programs. Furthermore the constant number of cycles between the execution of two consecutive instructions of the same thread allows for achieving perfect usage of the functional units without costly forwarding mechanisms.

The only drawback of this mechanism is that every idle thread causes a wasted cycle since it is called in this round-robin scheme regardless of its state. Since in a ray tracing context a shader program is idle while a ray is traced by the RTC a strict round-robin execution scheme would result in many wasted cycles even if there are threads that could be executed.

Therefore the SCPU uses a different scheme that switches only to active threads and continues the execution of a thread sequentially as long as no data hazards[9], jumps, or memory requests with a return value are encountered. Since instructions requiring to switch the thread can be detected statically this is done by the compiler, which computes an additional flag *stnt* (switch to next thread) for each instruction.

This flag is part of the opcode and when an instruction with an active *stnt* flag is encountered the thread is put asleep after scheduling the execution of the current instruction. For every thread there are two status bits denoting whether it waits for a memory request or the write back to the register file. The corresponding bit is set on *stnt* and cleared when a memory request returns respectively when data computed by the instruction with the *stnt* flag set has been written to the register file[10].

A thread is scheduled for execution only if both status bits are clear and the thread's active bit is valid. The implementation of this scheduling is trivial if using string encoded bit-vectors.

Since the flag *stnt* is under full control of the compiler several optimizations can be performed. For example if the compiler knows the number of cycles it takes until the completion of write-back it can try to avoid the hazard by rearranging instructions and sets *stnt* only if this fails.

However, the *stnt* flag simply provides a *barrier* and thus allows for compile-time opti-

[8] This actually splits the RORF into two separate windows. The window to the special values is constant and the window to ray data and hit-information can be moved.

[9] Data hazards describe computational dependencies, which can be solved by forwarding. Structural hazards can only occur when multiple instructions are issued simultaneously and two instructions require the same resources (see [MP00]). Control hazards describe the problem of changes in the control flow of a program after jumps and branches, which typically require the pipeline to be flushed if no speculative or delayed execution is used.

[10] No structural hazards can occur by setting or clearing these bits using string encoded bit-vectors. Additionally, multiple simultaneous write-backs are avoided by padding all data paths to have equal length. Due to multi-threading the longer latencies caused by this padding are typically no issue.

mizations for shortening the latencies of memory accesses. Here simply a load instruction is issued without *stnt* flag and the flag is set on the instruction before the loaded data is needed.

This directly suggests the efficient implementation of *lock-step synchronizations* by simply adding another flag that synchronizes between threads. If threads can be synchronized easily the efficient combining mechanism used in the *SB-PRAM* [PBB+02] allows for packet-like bandwidth reductions for arbitrary tasks without using packets.

5.3.4 Strategies to Increase Hardware Efficiency

CPUs require to support a variety of different operations such as addition, multiplication, and logic functions. But a pipelined CPU that fetches and executes a single instruction per cycle can only feed a small fraction of these units which renders most of the hardware resources idle most of the time. Therefore modern CPUs try to parallelize the execution of a sequential program by scheduling multiple instructions fetched in the same cache-line or with explicit parallelism using VLIW or EPIC instructions.

The SCPU supports both techniques: the *parallel execution of different threads* (PEDT) and explicit parallelism for special instruction groups like in EPIC. But since the SCPU is optimized for cost efficiency this support is implemented by adding as few hardware as possible and thus leaving costly optimizations aside.

Parallel execution of different threads uses the already available next thread prefetching mechanism. This mechanism switches to the next thread in case of *stnt* without wasting any cycles and only schedules an instruction if enough resources are available.

Typical bottlenecks of multi-issue are the register file, which is not only required for fetching operands when issuing but also on completion for write back of the result. For PEDT these bottlenecks can not be resolved without adding costly hardware and therefore cheap multi-issue can only be performed for a few combinations of instructions.

Background Hardware Programs (BHP)

In contrast to PEDT it is possible to support a variant of EPIC efficiently by adding only a bit of dedicated hardware. The technique proposed here is called *background hardware programs* (BHP) and allows for implementing DMA-like memory transfers, kd-tree traversal, ray-triangle intersections, cross- and dot-products, square-root computations, texture RGBA to floating-point conversion, standard shading techniques, ray generation and similar algorithms to be executed in parallel to the thread currently running on the SCPU at hardly any costs.

Basically BHPs are pre-compiled micro-instructions, which are not restricted to the instruction set of the CPU. The instruction fetch of these programs can be realized with a cheap on-chip ROM table (but an extension to support application specific BHPs is straight forward). The software starts a BHP with a special function call and synchronization between the software and the hardware program is achieved using the *stnt* mechanism and corresponding flags.

Since the major bottleneck is the register file for BHPs there are shadow registers which

allow for fetching operands sequentially[11] and to store immediate results outside of the register file. These extra registers for BHPs allow for executing a great variety of programs in background of the software program at hardly any cost. However, transferring the results of the BHP into the RWRF is still a bottleneck. Therefore it is of great benefit to add for each thread at least one separate register for immediate results which is not part of the register file allowing simultaneous writes to each *immediate result register* (IRR). If compilers generate programs using these IRRs instead of RWRF registers BHPs like DMA-transfers for writing and reading can be executed with great efficiency.

Obviously BHPs can be used with any kind of ALU regardless of its granularity (single operation or parallel SIMD) and design. Nevertheless a straightforward extension is to support several BHPs with separate dedicated resources in parallel. This allows for using even more of the CPU's idle resource but the gain in performance degrades with increasing number of BHPs as the I/O of the register file is typically the limiting factor.

5.3.5 Minimalistic Instruction Set (MIS)

Every turing complete instruction set allows for writing arbitrary programs but the number of instructions required to perform an operation greatly varies between various instruction sets. Therefore this section presents a *minimalistic instruction set* (MIS) consisting of only 32 instructions designed to be a good compromise between usability and efficiency.

The SCPU was primarily designed for shading in a ray tracing context. Therefore an instructions set is used that allows for performing the most important operations of shading using as few instructions as possible. At the same time only those instructions have been used that can be implemented without increasing the hardware costs by relatively great amounts.

For example if building a fully programmable SaarCOR without a traversal unit rays need to be traversed in software. Therefore adding support to calculate the traversal decision is always a good choice as this avoids costly case switchings and compares and only requires few additional gates (see Section 2.1.3). In contrast adding dedicated hardware to calculate dot-products is only of interest for some cost performance ratios[12].

Table 5.2 shows a basic variant of the MIS and Table 5.3 presents a corresponding instruction set encoding[13]. The remaining part of this section explains the MIS in detail, discusses alternatives, and presents application specific adaptations and further improvements. Appendix F shows how instructions missing in the MIS can be simulated efficiently.

[11] This means that even if a BHP instruction requires two operands but in the current cycle there are only resources to fetch one operand then one operand is fetched now and in the next cycle with free resources to fetch at least one operand the execution of the BHP instruction can be scheduled. Please note, that due to the construction of the RWRF and RORF it is very common that even instructions requiring two operands fetch only a single operand from each register file leaving enough resources for the BHP, which additionally has its private immediate registers.

[12] In computer architecture everything is about cost performance ratios. With unlimited resources rather arbitrary fast circuits can be build but since typically especially money is limited the key to efficient computer architecture is to design circuits that perform the most important operations fast enough while not requiring too many resources. For a discussion about cost performance ratios see [MP95] and the corresponding lecture by Wolfgang J. Paul. Additional [Gus91] is a quite amusing reading.

[13] Detailed information on encodings and construction of corresponding control automata can be found in [MP00].

Mnemonic	Effect	Description
Arithmetic Operations		
`fadd`	T := S1 + S2	float add
`fsub`	T := S1 − S2	float subtract
`fmul`	T := S1 · S2	float multiply
`finv`	T := 1.0 / S1	float invert
`iadd`	T := S1 + S2	int add
`isub`	T := S1 − S2	int subtract
`imul`	T := S1 · S2	int multiply
Type and Immediate Conversion		
`f2i`	T := (int)S1	float to int
`i2f`	T := (float)S1	int to float
`f2r`	T := (float)*imm*	fp-immediate to register
`i2r`	T := (int)*imm*	int-immediate to register
Logic and Shift Operations		
`and`	T := S1 ∧ S2	*AND*
`or`	T := S1 ∨ S2	*OR*
`xor`	T := S1 ⊗ S2	*XOR*
`shl`	T := S1 ≪ S2	logic shift left
`shr`	T := S1 ≫ S2	logic shift right
`swiz`	T := swizzle(S1,S2)	exchange / mask bytes
Conditionals		
`sip`	if S1 > $+\epsilon$: T := S2	set if fp-value is positive
`sin`	if S1 < $-\epsilon$: T := S2	set if fp-value is negative
`siz`	if S1 ∈ $[-\epsilon, +\epsilon]$: T := S2	set if fp-value is zero
`snz`	if S1 ∉ $[-\epsilon, +\epsilon]$: T := S2	set if fp-value is not zero
`sep`	EP := S2	set ϵ for comparisons
Test and Set Operations		
`tbz`	T := $\overline{1}$, if S1[BM(S2)]=0, else $\overline{0}$	set if bit / byte is zero
`tbo`	T := $\overline{1}$, if S1[BM(S2)]=1, else $\overline{0}$	set if bit / byte is one
Memory and Special Operations		
`ld`	LR := M(S1)	load
`ldi`	LR := M(S1); T := S1 + S2	load and integer add
`st`	M(S1) := S2	store
`lfa`	LR := M(S1); M(S1) := M(S1) + S2	load and fp-accumulate
`sfa`	M(S1) := M(S1) + S2	store fp-accumulate
`lio`	LR := M(S1); M(S1)=M(S1) ∨ S2	load and integer-*OR*
`scp`	CP := S2	set cache-policy
`sfp`	FP := (S2, S1)	set rf-frame pointer
`bhp`	BHP(S2, T, S1)	start BHP

Table 5.2: A minimalistic instruction set for an efficient ray tracing shading processor. Here $\overline{X}$ with $X \in \{0, 1\}$ denotes the bit string $< X \dots X >$ and M(A) refers to the content of the memory at address A. ϵ denotes a floating-point constant positive and close to zero. The special registers (EP, LR, CP and FP), the bit-mask function (BM) and detailed descriptions of the instructions can be found in the text. Please note, that this table is a bit informal as S1 actually denotes the content of the register specified by the bit-string S1 (S2 and T are used in an analog way). A precise and detailed discussion on formalisms and semantics can be found in the textbook [MP00].

R-Type	1 bit	6 bits	6 bits	6 bits	5 bits
int	*stnt*	00 *opcode*	S1	S2	T
float	*stnt*	01 *opcode*	S1	S2	T

I-Type	1 bit	2 bits	16 bits	5 bits
int	*stnt*	10	*immediate*	T
float	*stnt*	11	*immediate*	T

Table 5.3: The basic MIS encoding requires 24 bits per instruction and allows for trivial distinction between floating-point and integer operations as well as between operations on immediates (*I-Type*) and registers (*R-Type*). Here S1 and S2 specify the source registers and T denotes the target register.

Number Formats

The MIS distinguishes between operations on floating-point and integer numbers while the latter one also includes operations on bit-strings. As mentioned in Section 5.3.2 the RORF contains several constants. To save register addresses every constant register address actually addresses two values, a floating-point constant and an integer value, and the value selected depends on the type of the instruction accessing the register file. Table 5.4 shows an example of special values in the RORF.

This distinction is important for the extension of the immediate constants (16 bit in the example above) to the width of the register file (on most of today's architectures: 32 bit). For many instructions it is obvious to which type they belong to but for some the type can be chosen arbitrarily. For example the store instruction `st` has no obvious type but it needs to be defined whether a store of the special value register(0) writes the floating-point or the integer value of 0 to memory.

Without going into detail of reasoning about the decision the following set of instructions belongs to the floating-point type: `fadd`, `fsub`, `fmul`, `finv`, `f2i`, `f2r`, `sip`, `sin`, `siz`, `snz`, `lfa`, `sfa`, `ld`, `st`, `bhp`, and `sep`. All remaining instructions have integer type.

Register	FP	Int	Register	FP	Integer
0	0.0	0	6	+inf	bit mask: <0111...1111>
1	1.0	1	7	-inf	bit mask: <1111...1111>
2	-1.0	-1	8	0.5	swizzle mask: get byte 0
3	2.0	2	9	0.25	swizzle mask: get byte 1
4	-2.0	-2	10	$\frac{1.0}{255.0}$	swizzle mask: get byte 2
5	4.0	4	11	255.0	swizzle mask: get byte 3

Table 5.4: Example for a mapping of $k = 16$ special values in the RORF. Floating-point instructions use the value listed in the columns labeled FP while integer and swizzle operations use the constants of the columns labeled Int(eger). There are 4 registers that are not listed above which could contain a random number generator, a timer, or simply more constants specified by the application on the host (or the compiler).

The cost of the ALU can be greatly reduced by dropping the support of fully IEEE com-

pliant operations and skipping the various rounding modes, the special cases (NaN, ±inf, and denormalized numbers) and interrupts based on arithmetic inaccuracies (see [MP00] for a detailed implementation and analysis of IEEE compliant FPUs). Again the implementation of the ALU depends on the application and the desired performance cost ratio[14].

Potential Savings on the ALU

Since the SaarCOR architecture is highly scalable and can be balanced in many ways it is very likely that savings in hardware at some functional units can be used to improve other functional units resulting in a better overall performance. This motivates further investigations on how to save hardware and reduce the requirements of each SCPU.

An optimization which can be found already in the MIS as presented above is to support only `finv` and no floating-point division. Furthermore since integer division can not be implemented by reusing parts of floating-point units the basic MIS does not support integer division. Nevertheless integer divisions by powers of two can be realized using logical (i.e. non-cyclical) right shifts. In the same spirit `shl` could be omitted if `imul` operates on full precision.

If arithmetical integer operations are only used for address calculations `isub` can be omitted. Additional savings result from limiting the shift distance to a single bit (which reduces the barrel-shifter to a single mux).

Logical Operations

An interesting fact is that the hardware cost of logical operations is dominated by circuits to select the result rather than to calculate the logical function (if not realized by tri-state busses). Therefore if `and`, `or`, and `xor` are replaced by a single `nand` function still all logical operations can be calculated but the cost is reduced to far less than a third.

Typically the logic functions are used to select and mask bytes of a word, e.g. when working with textures. Especially for this purpose a powerful instruction called *swizzle* (`swiz`) is used. A similar instruction is implemented in today's GPUs, which allows for selecting and masking components of a vector register.

In the SCPU with 32 bit wide registers swizzle allows for selecting and masking bytes in a word and uses 12 bits of the content of S2 to choose how S1 is modified. These 12 bits contain 4×2 bits to select the corresponding bytes of S1 and 4×1 bit to mask the swizzled bytes in the result (see Figure 5.4).

Instruction Set Encoding

The MIS contains many cases which allow to choose whether an additional parameter is given using the content of a register or by an immediate constant. This is especially

[14] For example single precision IEEE floating-point numbers are 32 bits wide, but GPUs of Nvidia and ATI in 2004 supported only 16 bit respectively 24 bit floating-point numbers without performance penalties. Similarly on the SaarCOR prototype (see Chapter 7) based on FPGA technology also 24 bit floating-point numbers have been used. These implementations are great examples that even at reduced precision (and reduced cost) many interesting applications can be realized.

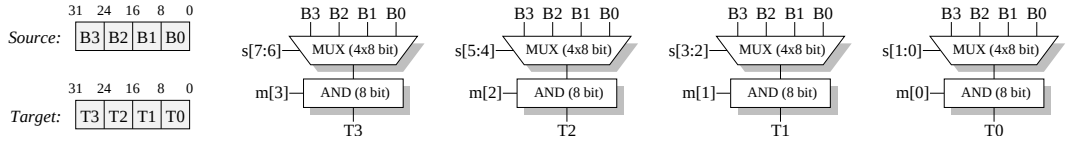


Figure 5.4: Implementation of the swizzle instruction on a SCPU with 32 bit wide registers. The content of the second source register is used as parameters to specify which bytes to select (using s[7:0]) and which bytes to mask (using m[3:0]).

true for instructions which only require a few bits as immediate constant and therefore can be encoded using standard R-Type instructions. Therefore the 6 bits used to specify S2 can be used as immediate constants in the following instructions: `shl`, `shr`, `tbz`, `tbo`, `sep`, `scp`, `bhp`, and `sfp`.

If the instruction set encoding becomes a bottleneck `lfa`, `sfa`, `ldi`, `lio` and `scp` can be realized via `ld` and `st` using the highest bits of the address as opcode. Although less obvious also `sep`, `sfp`, and `bhp` can be realized in the same way.

Register Files

In this instruction set there are no explicit instructions for jumps and branches. Therefore those instructions are realized by direct manipulation of the *program counter* (PC) register which is virtually mapped to the RWRF. In combination with the *previous PC* (PPC) register which is part of the RORF this also allows for subroutine calls. Termination of programs (for shooting rays and when shading finishes) can be realized by setting the PC to special addresses (e.g. 0 and 1) or by jumping into the operating system for further processing before exit.

As presented in Section 5.3.2 to allow for efficient transfer of input and output parameters between caller and callee of on a subroutine or function call the register file contains many registers of which only a window of several registers is visible to the current process. These windows can be set using the `sfp` instruction where S2 specifies the window that is to be changed and S1 defines how the window pointer is changed. Therefore S1 can specify absolute and relative changes as well as the position respectively the distance. However, if the windows can be set only to discrete positions (e.g. with a stepping of 8) some hardware is saved.

Background Hardware Programs

The key idea behind adding special support for operations like evaluation of the traversal decision or performing intersections is to provide a flexible system with many options for extensions. For example in the *OpenRT* ray tracing system (see Section 1.2.3), many new features (like global illumination and volume ray tracing) have been added simply by writing the corresponding shading programs and there was no need of changing any part of the core ray tracing system.

One of the goals of the SCPU design was to provide this flexibility and additionally keeping the high degree of efficiency. This has lead to the development of the *background*

hardware program (BHP) mechanism (see Section 5.3.4). This technique generally does not provide new arithmetic operations but typically requires only a few additional gates to evaluate complex case-switchings and conditionals, which would require several standard instructions to perform in software.

In addition, some extra immediate registers are provided to decouple the execution of the BHP from normal program execution. In many cases this allows for simultaneous execution of a standard program and a BHP.

The instruction `bhp` is used to start those programs using S2 to specify the program, S1 as the input parameter and T as the output parameter. If the program does not use either parameter it is ignored and if the program uses several input and/or output parameters S1 respectively T denote the start address in the register file of the corresponding parameters.

The BHP can run in parallel to the thread that started it using barriers (see Section 5.3.3) to synchronize the execution. Alternatively the thread starting the BHP can simply sleep during the execution of the BHP similar to performing a memory access.

Conditionals

A very important part of every instruction set are conditionals. In the basic variant of the MIS there are four *conditional set operations*, which only assign a value to a register if the condition is fulfilled and two *test and set operations* which write the result of a test to a register.

The test and set operations are `tbz` and `tbo`. If registers are 32 bit wide these instructions use a 6 bit wide parameter of which one bit selects whether a single bit is tested or if one or multiple bytes are tested. In the first case the remaining 5 bits specify the bit to test. In the other case additionally only 4×1 bit are used to specify the bytes which are to be tested (similar to the specification of the mask in Figure 5.4).

The conditional set operations are `siz`, `snz`, `sip`, and `sin`. These operations perform comparisons of floating-point numbers against zero. In many algorithms and especially in computer graphics typically comparisons are not performed hard against a number but use a small ϵ-environment around the value in which all numbers are considered to be equal. This allows for avoiding visible artifacts resulting from computational inaccuracies.

Since these ϵ values typically are not set by precise arithmetic considerations but guessed or found by experiments usually it is not crucial to use an exact value. This allows for adding hardware efficient support for ϵ-based comparisons with very low costs by restricting ϵ to a fixed set of values. Table 5.5 shows the implementation of the conditions for the conditional set operations with and without ϵ-environments using typical values as used in our ray tracers[15].

The value of ϵ for the conditional set operations depends on the application and maybe even on the data currently processed. For example when tracing rays for traversal and intersection calculations the value of ϵ depends on the bounds of the current object. Therefore the value of ϵ can be set using the `sep` instruction. This instruction takes as

[15] A standard implementation of a test for an ϵ-environment first subtracts the ϵ and the tests the result of the subtraction. This either produces a data hazard or increases the length of the data path since the test can only be performed after the subtraction was executed. Furthermore this standard test uses the floating-point subtraction unit while a hardware supported ϵ-test leaves the floating-point units unused allowing BHPs to be executed in parallel.

a parameter the bit-mask that is used in the comparisons (see Table 5.5).

Since setting the value of ϵ influences all comparisons there are two choices to specify this value: either it is used only for the next instruction and then reset to 0 (i.e. no ϵ-environment) or it is persistent until it is overwritten by a different value.

S1 > 0.0	S1 < 0.0	S1 = 0.0	S1 ≠ 0.0
/s[31]∧/(ez∧mz)	s[31]∧/(ez∧mz)	ez ∧ mz	/ez ∨ /mz
S1 > $+\epsilon$	S1 < $-\epsilon$	S1 ∈ $[-\epsilon, +\epsilon]$	S1 ∉ $[-\epsilon, +\epsilon]$
/s[31] ∧ agz	s[31] ∧ agz	/agz	agz
"exp=0": ez	$/OR[i$=0:7](s[i+23])		
"mant=0": mz	$/OR[i$=0:22](s[i])		
"|s|>0": agz	s[30] ∨ s[29] ∧ s[28] ∧ (s[27] ∨ $OR[i$=0:3](s[i+23]∧e[i]))		

Table 5.5: Let s be the content of the 32 bit wide register S1 and represent a standard normalized single precision IEEE floating-point number with s[31] specifying its sign, s[30:23] and s[22:0] denoting the biased exponent respectively the mantissa without the leading one (for a precise and detailed specification see [MP00]). Further the following notation is used: $OP[i$=a:$b]$(v[i]):= $v[a]$ OP $v[a+1]$ OP ... OP $v[b-1]$ OP $v[b]$. The conditions presented above allow to choose from several values in the typical range from 10^{-10} to 10^{-5} with $\epsilon = 2^{-30}$, 2^{-29}, 2^{-27}, 2^{-23},or 2^{-15} by setting $e[3:0]$ =<1111>,<1110>,<1100>,<1000>,or <0000> respectively. It is interesting to note that in many cases the test for the ϵ-environment is cheaper than the test for the exact value. Adaptations to double precision IEEE numbers, reduced representations as for example in the prototype and other values of ϵ are straight forward. Using representations where the exponent is in two's complement might simplify the conditions. This is especially true for hardware supported evaluation of shadow rays using S1 < $(1.0 - \epsilon')$.

Memory Interface

One important issue in processor design is the memory interface. This is especially true for the SaarCOR architecture, which is a multi-threaded multi-processor environment and has a memory interface with variable latencies and out-of-order[16] service.

Since memory requests for any thread return asynchronous to the program currently being executed on the SCPU storing the data from memory in the RWRF would either require to stall the program execution or to add another write-port to the register file. Since neither variant is an option for efficient processor design data loaded from memory cannot be stored in an arbitrary register of the RWRF but is always stored in a special *load register* (LR) of the RORF.

This solves the issue of returning memory requests of one thread simultaneous to the execution of a different thread. But on some applications that require to copy several words from memory into the register file before calculations can be started this introduces

[16] Out-of-order service of memory requests take only place for requests to different addresses. Requests to the same address are always in order.

the problem that every load operation requires two instructions (one for the load and one to transfer the data from the RORF to the RWRF).

However, many of the important applications like traversal, ray-triangle intersection and texture-filtering, can be written to load some data and process it before the next data is fetched. Nevertheless, it would be also possible to implement several load registers in the RORF. Furthermore, with its dedicated resources BHPs are ideally suited to exploit the burst capabilities of today's memory chips by using DMA-like copies of several words of data from memory into the RWRF.

The use of the load register also allows for implementing `ldi` at almost no additional cost since the data path for the addition and the write back port on the register file are not used for the load instruction[17].

Memory Based Communication Schemes

In a multi-processor as well as in a multi-threaded environment the communication between different processes is an important topic. Although this is not necessary for standard shading applications, advanced effects (e.g. global illumination and image filtering) and other applications greatly benefit from efficient communication schemes. Therefore some *multi prefix operations* (MPO) like in the *SB-PRAM* [PBB+02] are supported.

The name *multi prefix operation* is used as a generalization of the *parallel prefix operation* (PPO), which is not executed on a single processor but on a *multi* processor machine. Therefore the MPO can compute any associative function like *SUM*, *AND*, or *MAX* over a list of values in logarithmic time [Bla04]. Furthermore, efficient communication schemes can be implemented using MPOs [Röh99].

The MIS supports the three MPOs `lfa`, `sfa`, and `lio`, but these operations are actually performed by the memory processor (see Chapter 6) and the SCPU only forwards these operations using the standard memory data paths and additional control bits. Here it is important that the value stored in memory before the thread's operation is performed is returned to the corresponding thread. This allows for efficient locking mechanisms and especially the *OR* operation speeds up computations on parallel task queues [Röh99].

Interrupts

Besides multi prefix operations the memory processor is used to send interrupts to the host. Although this interrupt mechanism was introduced for efficient virtual memory management (see Chapter 6), it also allows for software generated interrupts.

However, these interrupts are drastically different from those on standard CPUs where interrupts are mainly used by the operating system for management of external peripheral devices and to handle page-faults in the virtual memory system. Therefore on standard processors interrupts are highly critical and have very strict constraints (see [MP00]).

On the SaarCOR architecture SCPUs are not connected to external I/O and also the critical case of page-faults can not occur as the whole memory management is performed

[17] Obviously instead of performing an post increment it is also possible to first perform the addition and then to start the memory access using the result of the addition. But this would increase the memory access latency.

fully transparent to the SCPUs, the GAS, and the RTC (see Chapter 6). Thus for shading processors there are only two reasons to generate an interrupt.

One is for debugging purposes if an error occurs that should be examined by the application on the host system. This can be realized using an `lio` instruction to a specific address with a data word containing some information for the application. The additional information which thread on which SCPU generated that interrupt is already part of the memory transfer since a labeled memory interface is used.

The memory processor does not serve this `lio` request (except for generating an interrupt on the host) and therefore the thread is frozen it its current state. However, the host can release the thread again by telling the memory processor to return an arbitrary value as the memory's response to the `lio` command.

The other case is if any information needs to be sent to the application, e.g. to return some special results, which should be returned already during rendering of the current frame. If program execution can continue normally after initiating the interrupt the standard store instruction to a specific address can be used. If the corresponding thread should be frozen until service of the interrupt as in the previous case the `lio` instruction can be used.

Controlling the Behavior of the Memory System

In Section 5.2 the hierarchical memory interface of the GAS was shown. Since such a memory interface has several options, which depend on the application it can be configured using the `scp` instruction. Therefore the memory bus is extended by some bits to allow for influencing and manipulating the caching strategies of the various caches and the behavior of the memory management units.

But care must be taken as every level one memory interface is shared by all threads on the corresponding processor and higher order interfaces might even be shared by independent pipelines. Similar to `sep` the memory policy can be set for the next memory operation only or permanently until it is overwritten.

One of the main purposes of `scp` is to increase the efficiency of the caches by giving hints. For example a program can initiate a virtual memory prefetch to shorten memory access latencies. Another example are values that are written to the frame buffer (e.g. color data, depth values, or even the hit-information for some applications) but not likely to be read back. Therefore this data should not be stored in any cache but written directly to external memory.

In a similar way read requests for frame buffer data and data that has been swapped out previously also should not be stored in any cache but passed directly to the only thread that requires that data. On the other hand data that could be of use for more than a single CPU (e.g. texture and shading data) should be stored at least in the highest cache level.

If there are several tasks using different data structures running simultaneously on a SCPU (e.g. ray traversal and triangle intersection), this might increase trashing of the caches. Therefore the `scp` instruction can be used to give hints in which way of the multi-way cache to store the data depending on the type of the data.

On multi-processors systems with a hierarchy of caches *cache consistency* is very impor-

tant. Since MPOs can only be performed by the memory processor those requests are not stored in any caches except for the one on the highest level (see Chapter 6). Besides for MPOs there is no hardware support for cache consistency for general data since every support would be too costly[18].

Extensions to the Minimalistic Instruction Set

The MIS as presented above allows for being encoded using only 24 bits per instruction including the required flags. This reduces the size of the instruction cache and allows for fetching 5 instructions simultaneously on a 128 bit memory interface. On the other hand this encoding leaves much room for further extensions to the instruction set since typically the width of registers is 32 bits.

Table 5.6 presents some useful extensions to the minimalistic instruction set. Further extensions support the use of *single instruction multiple data* (SIMD) operations. These operations can be implemented like on standard processors or as sequentially executed operations using an *ENAC* function. In both cases additional instructions for efficient masking and gathering are required similar to the *Test and Set* respectively the *Arithmetic Operations* of Table 5.6.

The instruction set presented in this section does not allow for register indirections for purpose. Although for a programmer it is nice to specify a register by the content of another register supporting those instructions would greatly increase the cost of the CPU and therefore violate the maxim of the design of the SCPU. Therefore only BHPs can do some kind of register indirection for example to allow for efficient SIMD operations using the *ENAC* function.

5.4 Shading using the Transformation Unit

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 but also simplifies computations of the first part as shown in the next paragraphs (see [Woo04] for details).

Cosine between Normal and Ray Direction

As shown in Section 4.5 the unit triangle intersection directly calculates the cosine between the direction of a ray and the geometry normal of a triangle for free if the ray was normalized. Since this cosine is very often used for shading this reduces the computational complexity of shading.

[18] On multi-processor systems typically consistency is ensured by using write-through caches and bus-snooping. But as these techniques produce many transactions and require additional data paths and logic for snooping they are far to expensive and simply will not fit the philosophy of the SaarCOR architecture. On the other hand for shading and ray tracing there is hardly any useful application that uses simultaneous writes and reads on the same memory cells from various processors without using any locking mechanisms.

Mnemonic	Effect	Description
Arithmetic Operations		
`imin`	T := *MIN*(S1, S2)	compute int minimum
`imax`	T := *MAX*(S1, S2)	compute int maximum
`fmin`	T := *MIN*(S1, S2)	compute fp minimum
`fmax`	T := *MAX*(S1, S2)	compute fp maximum
Type and Immediate Conversion		
`l2r`	T := <T[31:16]*imm*[15:0]>	immediate to low-reg.
`h2r`	T := <*imm*[15:0]T[15:0]>	immediate to high-reg.
Conditionals		
`szp`	if S1 $\geq -\epsilon$: T := S2	set if fp value zero or pos.
`szn`	if S1 $\leq +\epsilon$: T := S2	set if fp value zero or neg.
Test and Set Operations		
`tip`	T[<S2>] := (S1 $> -\epsilon$)	set if fp value is positive
`tin`	T[<S2>] := (S1 $< +\epsilon$)	set if fp value is negative
`tiz`	T[<S2>] := (S1 $\in [-\epsilon, +\epsilon]$)	set if fp value is zero
`tnz`	T[<S2>] := (S1 $\notin [-\epsilon, +\epsilon]$)	set if fp value is not zero
`tzp`	T[<S2>] := (S1 $\geq -\epsilon$)	set if fp value is zero or pos.
`tzn`	T[<S2>] := (S1 $\leq +\epsilon$)	set if fp value is zero or neg.
Memory and Special Operations		
`sca`	$M(S1)_{r(n)} := M(S1)_{r(n)} + S2_{r(n)}$	store color-accumulate

Table 5.6: Some examples for useful extensions of the minimalistic instruction set. The multi-prefix operation `sca` allows for performing atomic read-modify-write operations on textures and framebuffers stored in 4×8 bit (e.g. RGBA) format. Thus $r(n)$ specifies the range $[a : (a + 7)]$ with $a = (n \cdot 8)$ and $n \in \{0, 1, 2, 3\}$. Obviously this instruction can be adapted to any color format (e.g. 4×floating-point values on vector machines) and could also support masking. The addition of the extended test and set operations completes the highly efficient support for masking and selective operations in standard programs and SIMD-like operations. Please note that <S2> denotes an integer value that is used as a bit-address rather than the content of a register interpreted as a bit-string.

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.

Ray Generation

In Chapter 4 it was shown that every ray is sent from the RGS to the transformation unit which stores and distributes the ray data. This suggests to use the transformation unit directly for ray generation which greatly simplifies floating-point requirements in the shader.

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.

Issues with Ray Generation

Although the use of the transformation unit greatly simplifies the calculations of ray generation there are some issues involved.

Section 4.4 has shown that the transformation unit can be built in a cheaper way which requires two steps to transform a ray. For rays sharing the same origin the cost of transforming a packet of n rays can be reduced from $2n$ to only $n + 1$ steps. But obviously this can not be done for all types of rays and therefore only works for primary and shadow rays.

For reflection and transparency rays this could also be achieved by modifying the transformations given above and additionally specifying the *near* value for each ray (which is simply t in the formula above).

However, clustering a packet of rays and taking care of whether a packet can be calculated in $n+1$ steps or not is costly. Furthermore since each transformation requires 12 floating-point values which have to be specified for each ray[19] additionally to the ray data of 6 values this drastically increases the bandwidth between RGS and the transformation unit.

Nevertheless, for the SaarCOR prototype (see Chapter 7) still the savings outweigh the costs. The prototype uses FPGA technology on which routing is comparably cheap but logic and memory resources are fairly limited. Therefore using the transformation unit also for shading was the key to allow for an implementation using the FPGA technology available in 2003.

[19] For primary rays and a planar perspective camera transformation as given above a single transformation for all packets of rays suffices and thus allows for simplifying the connection.

6 Memory Interface

The memory interface presented here is designed to support a full featured ray tracing graphics card with dedicated on-board memory, virtual memory management using the host's memory, and an interface to directly output the rendered images to a displaying device.

Therefore the memory interface not only handles the various kinds of memories but also all other types of external I/O including the interface to the host and it is also coupled tightly with the display controller since this device requires a guaranteed bandwidth to avoid artifacts.

Thus the memory interface (shown in Figure 6.1) contains a *bus controller* for the system bus of the host (e.g. PCI, PCI-X, AGP), various *caches* (separated by data type), a *memory controller* (Section 6.1), units for *virtual memory* and *object management* (Section 6.2), and *memory processors* (Section 6.3).

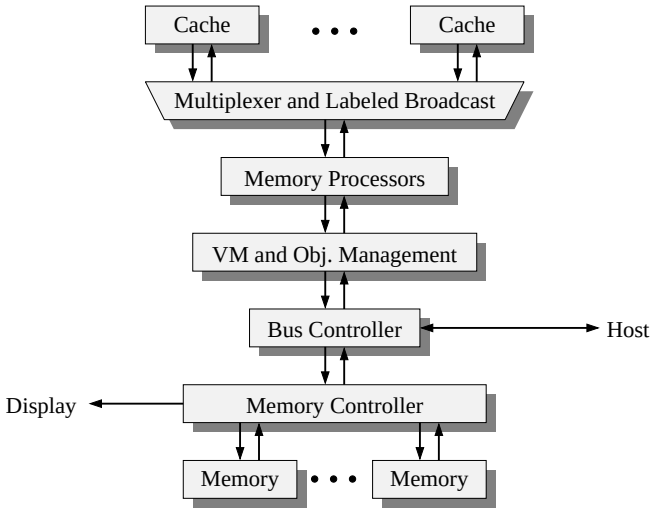


Figure 6.1: Data paths of the memory interface. All units of the same type (e.g. traversal units from all pipelines) share the connection to an *item-based cache* using a simple round-robin multiplexed bus for requests and a labeled broadcast for data sent by the memory.

Item-Based Caches

For all types of data delivered by the memory interface there is a separate item-based cache, i.e. rather than caching single bytes it caches triangles respectively matrices (or any other geometric primitive) and kd-tree nodes. For lists of triangle-IDs and shading data a more general caching in words is used.

Therefore depending on the size of the data structure a cache miss on an item-based cache might require several sequential memory request. Since the memory controller serves requests out-of-order (see Section 6.1) the caches need to be able to reorganize data received from memory.

The design of the caches for traversal, list, transformation respectively intersection are trivial since all data is read-only. However, if using programmable shading a more advanced cache is required as data can also be written (see Chapter 5.3.5).

Shared Busses

All units of the same type (e.g. all traversal units) from all pipelines share the connection to the corresponding item-cache by using *simple round-robin multiplexing* for requests and a labeled broadcast for answers from the memory. Therefore this communication scheme is rather trivial to implement and allows for simple and cost-efficient point-to-point connections.

Every memory request contains the address, a flag denoting the validity of the transaction, and the ID of the sender. This ID contains the ID of the unit (either traversal, list, transformation respectively intersection, or shading), the number of the thread on this unit, and the number of the pipeline the request comes from. If using multiple SPEs for shading, the ID is extended accordingly.

On return data is sent using a labeled broadcast with the sender's ID and a flag denoting the validity of the transaction. However, since this data is only of interest for a single unit all other units can ignore the data by using *filters*. These filters additionally allow to transfer data from the traversal cache while triangle data is broadcasted.

Simple Round-Robin Multiplexing (SRRM)

The data paths of the *simple round-robin multiplexing* (SRRM) connection scheme and a general variant of the memory interface are shown in Figure 6.2. The scheduling algorithm used in the SRRM is rather trivial and allows for scheduling exactly one request every cycle if there is at least one valid request pending.

Let n be the number of channels on the SRRM and let $state_j$ with $j \in$[0,n-1] denote the state of the SRRM with $state_k$=1 if channel k is the channel who's request is scheduled if it is valid. Thus it holds for any time t that only $state_k$=1 with $k = t \ MOD \ n$ and $state_j$=0 $\forall j \neq k$. Further let $\ominus$ denote the subtraction modulo n, i.e. $x \ominus y = (x - y) \ MOD \ n$. Then the acknowledgement signal ack_i for channel i is calculated as:

$\text{ack}_i = \text{valid}_i \wedge (\ \text{state}_i \vee \text{allowed}(i, j) \wedge /\text{valid}_j\ )$ with $j = i \ominus 1$

using

$$\text{allowed}(f, g) = \begin{cases} \text{state}_g \text{ , if } f = g \ominus 1 \\ \text{state}_g \vee \text{allowed}(f, h) \wedge /\text{valid}_h, \text{ with } h = g \ominus 1 \text{ , else} \end{cases}$$

Since this formula is not very intuitive, here is an example for ack_1 and $n = 4$:

$\text{ack}_1 = \text{valid}_1 \wedge (\text{state}_1 \vee (\text{state}_0 \vee (\text{state}_3 \vee \text{state}_2 \wedge /\text{valid}_2) \wedge /\text{valid}_3) \wedge /\text{valid}_0)$

rewritten in an equivalent, but more readable way:

$$
\begin{aligned}
\mathrm{ack}_1 \quad = \quad & \mathrm{valid}_1 \wedge \mathrm{state}_1 \\
\vee \quad & \mathrm{valid}_1 \wedge \mathrm{state}_0 \wedge /\mathrm{valid}_0 \\
\vee \quad & \mathrm{valid}_1 \wedge \mathrm{state}_3 \wedge /\mathrm{valid}_3 \wedge /\mathrm{valid}_0 \\
\vee \quad & \mathrm{valid}_1 \wedge \mathrm{state}_2 \wedge /\mathrm{valid}_2 \wedge /\mathrm{valid}_3 \wedge /\mathrm{valid}_0
\end{aligned}
$$

SRRMs are widely used throughout the design of the SaarCOR architecture not only in the memory interfaces but also inside many units to manage simultaneous requests from several subunits. At some places a special variant of the SRRM is used which has priorities attached to each channel such that a channel with the highest priority can always schedule a request and only between channels of equal priority the round-robin scheme is used.

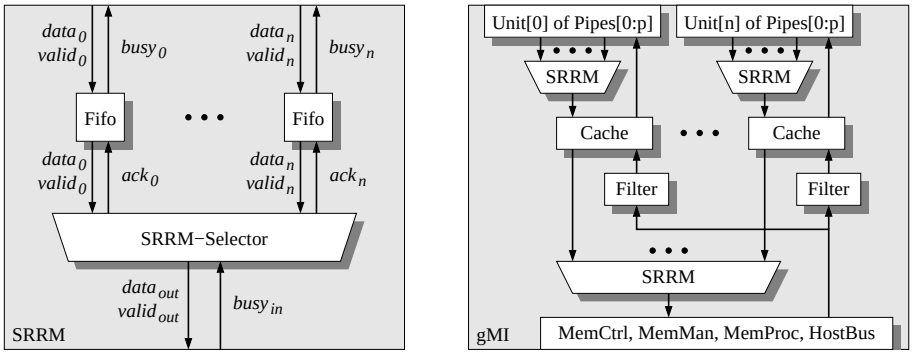


Figure 6.2: Data paths of the *simple round-robin multiplexing* (SRRM) interconnection scheme and a general version of the memory interface (gMI). Here the simple point-to-point connection scheme and the shared caches are clearly visible.

6.1 Memory Controller

The memory controller manages the connection of the memory interface to the external memory chips. These chips can be any kind of memory including mixtures of various types such as fast SRAM (e.g. to allow for efficient swapping) and large portions of DRAM (e.g. for storage of scene data).

Current advances in memory technology allow to deliver gigabytes of data per second from huge amounts of memory at rather cheap prices. Unfortunately this performance is only achieved for sequential memory accesses while random reads and writes directly cut down the available bandwidth to a small fraction.

Therefore a solution typically used is to perform bursts of sequential reads although only a single word is requested and to store the data in caches with large cache-lines. Then with some luck the cache already contains the data requested on the next memory access.

With this kind of memory this solution is the best one can do for single sequential programs that stall on every memory request until it is served. However, for the SaarCOR architecture there is a different solution available which does not require any luck.

Fortunately the SaarCOR architecture does not contain a single sequential ray tracer but many ray tracers executed in parallel whose requests do not need to be served in order. This allows for the following scheme (with its data paths shown in Figure 6.3).

High Speed Random Memory Accesses

Let there be m identical memory chips each with a data bus of d bits and a burst length of b words required to operate at full performance. Then instead of having several memory chips with a common address bus arranged to deliver $b \times (m \cdot d)$ bits wide words from memory every chip is connected independently with its own busses for the address, data, and control signals.

This allows for performing burst individually per memory chip where memory words of $b \cdot d$ bits are stored sequentially. The advantage of this concept becomes clear when typical values for b, d, and m are inserted.

Let b=8 words, d=8 bits, and m=8 chips and let the size of a word requested from memory be 64 bits (e.g. a node of a kd-tree). Then instead of always fetching 8×64 bits ending up with 512 bits of which only 12% can be used for sure it is more efficient to fetch the 64 bit wide word sequentially from a single memory chip.

This increases the memory access latency for a single word by b-1 cycles but allows for achieving optimal performance if there are enough requests to keep all memory chips busy. To ensure that the memory requests are equally spread across all memory chips *address hashing* is used. The concept of address hashing has been shown to perform very well on the *SB-PRAM* [PBB+02].

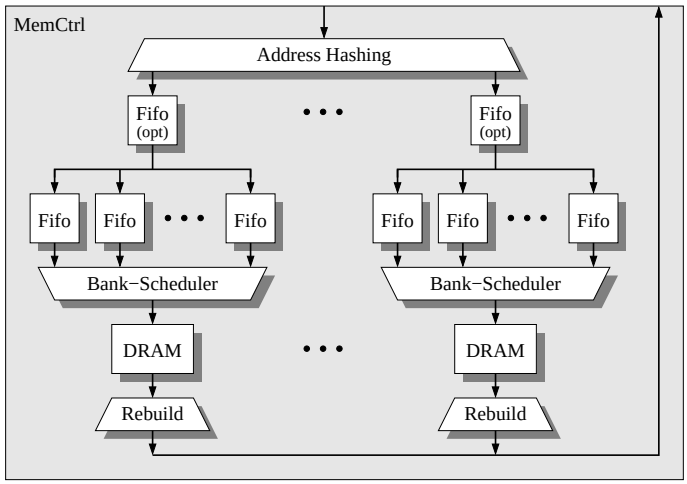


Figure 6.3: Data paths of the memory controller that allows for performing random memory accesses at a bandwidth close to the theoretical maximum. It utilizes for each internal memory bank a small FIFO and optionally an additional FIFO for each DRAM to further decouple the banks. Each word is fetched sequentially from memory using burst transactions from a single memory chip and then the unit *rebuild* arranges the final data word.

Further Issues

There is another issue on DRAM memory not yet covered (see [Win02] for a detailed description of DRAM memory technologies). As DRAM memory is arranged internally in banks of memory only on the bank currently opened an access can be performed.

However, it is possible to open a bank while accessing a different bank. Thus if the memory accesses are arranged such that the accesses are performed in a *hazard free order* optimal performance is achieved. For this purpose the memory controller contains several FIFOs with one for each bank. This sorting leads to an out-of-order service of the memory requests.

Thus given enough memory requests this scheme guarantees to deliver almost the theoretical bandwidth unless there are some hazards requiring internal bank switching of the memory chips which can not be hidden. However, this connection scheme obviously requires more pins on the chip and therefore again there is a trade-off on how many chips are clustered to share the same busses for address and control signals. Furthermore sometimes it is even cheaper to buy faster RAM and to use only half of the best case performance than to buy slower RAM and to add special circuits to achieve the optimal performance.

6.2 Memory Management

The level of realism of virtual worlds is steadily increasing and with it the detail in geometry and textures grows. This requires more memory to store the virtual world and at the same time more memory is needed during rendering.

With current rasterization based graphics boards efficient memory management is hardly possible simply by the architecture of the rasterization pipeline (see Chapter 1). Here an application guesses which parts of the scene are visible and uploads all data to graphics memory that could be needed to correctly render the image. Therefore automatic virtual memory management is only available for textures [3DL99].

In contrast to rasterization during rendering the ray tracer itself detects which parts of the scene are required for rendering. This allows for automatical demand driven memory management of all items of the virtual world including geometry, textures, and shading data without any interaction with the application on the host.

Memory management can be performed on page level similar to virtual memory management of current operating systems[1] and on object level similar to level of detail mechanisms. The next sections present both techniques in detail.

6.2.1 Virtual Memory Management

The following assumes that the entire scene data is stored in main memory of the host system and that the graphics subsystem can fetch this data independently of the application (using DMA). Then the next section presents a mechanism that allows for reducing the memory requirements of the host system to the currently visible subset.

The *virtual memory architecture* (VMA) [SLS03] is built into the bus managing circuit of the memory interface allowing for transparent address translations of memory requests from the ray tracer and automatic loading of missing scene data from host memory. This allows for realtime rendering of complex scenes using only a small memory on the

[1] Although the level of this management is similar the techniques used in modern operating systems (e.g. the TLB, see [Car03, Hwa93]), those techniques are quite different from the method described here. [Lei04] discusses these differences in detail.

graphics board to cache scene data.

Caching Memory Pages

It has been shown that caching of single triangles and kd-tree nodes works out very well for on-chip caches (see Chapter 8). Therefore a similarly simple caching scheme with a standard n-way set-associative cache is used to manage the transfer of memory pages in the virtual memory.

Since memory pages are much larger than standard cache lines the penalty of transferring a missing page is much higher. Therefore n-way caches are used to reduce the probability of collisions, i.e. of different addresses mapping to the same cache entry and this way overwriting data still in use. Additionally, the number of collisions can be reduced by using address hashing suitable to the size of the data structures used.

Experiments with both strategies has shown that the combination of a cache with 4-way set-associativity and simple address hashing works extremely well over a wide variety of benchmark scenes (see Section 8.3). In standard caches, the lower bits of the memory address are used as the address into the cache memory. The hashing function used extents this scheme by simply adding the upper bits of the memory address to the lower ones, which skews regular access patterns.

Storage Requirements of VMA

The on-board memory is divided 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. 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 non-bijective address hashing is used, 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*.

For 16 MB of cache memory with cache lines of 128 bytes this already requires 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, a line size of 128 bytes performed best on all measurements over various benchmarks.

With the significant size of the meta data also ways to reduce the on-chip memory requirements are explored 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. This reduces the latency and the external memory bandwidth due to meta data lookup (much like a TLB in CPUs). Further details and the impact of both variants on the performance are discussed in Section 8.3.

6.2.2 Management on Object Level

The virtual memory management allows for realtime rendering of complex scenes using only a small memory on the graphics board to cache scene data. However, the entire scene has to be stored in the memory of the host to allow for demand driven transfers of scene data to the graphics board fully transparently to the application on the host.

In this section some techniques to manage scene data on object level are presented. This allows for exchanging parts of the scene depending on their visibility. For example it is possible to unload objects from memory if they are no longer visible. However, then a placeholder (e.g. the bounding box of the object) is required to detect when the object may become visible again. This process is illustrated in Figure 6.4.

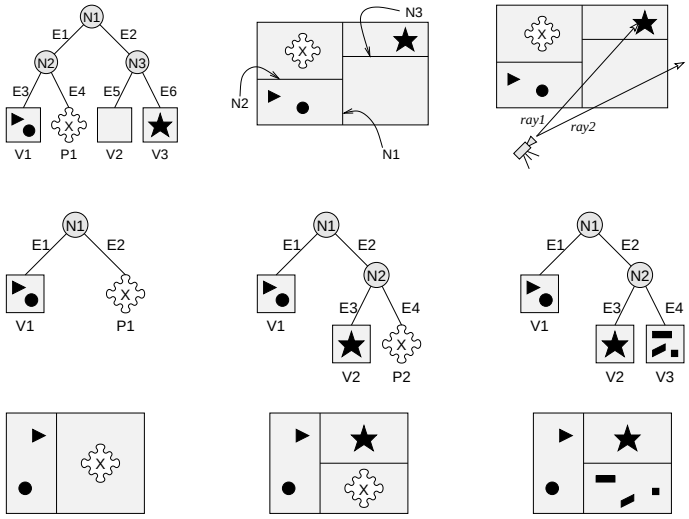


Figure 6.4: The first row shows an example of a kd-tree with a placeholder object P1. This object can be a voxel containing a few triangles as imposters. When rendering the view depicted in the right most image the placeholder object is not touched and thus does not need to be replaced. In the second row it is shown how placeholder objects are replaced recursively from one frame to the next frame.

The technique presented here is called *Occlusion Query for Ray Tracing* and performs some measurements during rendering yielding detailed information about the visibility of an object split into *the distance* to and *the importance* of an object. Here importance is equal to the number of pixels covered by the object, which also includes the indirect effects (e.g. objects seen through a mirror).

Since complex shading operations (e.g. many secondary rays) also have an impact on the performance, this scheme can be used to guarantee a minimal frame rate for example by limiting the recursion depth of ray refractions in complex glass materials if the viewer walks too close to the object. Therefore this technique could be called *inverse level of detail* since in contrast to standard level of detail techniques here with decreasing distance the objects are replaced by less detailed versions. In this context the term “detail” is used for geometric complexity as well as for shading complexity and refers to exchanging either combination of geometry, textures, shader parameters, or shader programs.

Occlusion Query for Ray Tracing

Current rasterization based graphics cards support a mechanism called *occlusion query*, which returns the number of pixels rendered during a period of time. This mechanism allows the application to evaluate whether and how many pixels of an object are visible.

For ray tracing a similar approach can be used but in a different way. With rasterization the objects being measured are defined implicitly by the triangles that are sent during the occlusion query. For ray tracing a set of objects to be measured are defined before rendering starts and the measurements are only available for read back after the frame is fully rendered.

Since in ray tracing every ray is shaded exactly once the measurement can be performed when shading starts. Thus, the occlusion query for ray tracing basically counts the number of shading operations per object where an object can be anything from a single triangle to a set of complex geometry. Clustering of objects can be done by either tagging the desired objects (useful if the object is a compound object) or by a comparision of the object's ID to a given list (only useful for single objects).

A nice property of this mechanism is that it can be implemented efficiently using shader programs only (especially by using MPOs) and without the need for any additional support in hardware. Nevertheless, hardware support can be used to save cycles on the SPEs. Preferably this support is added to the master which already has all required information since it schedules the hit-information for shading.

6.3 Future Work: Memory Processors

The memory interface can perform some processing on the data read from and transfered to memory. For this purpose special processing elements can be integrated into the memory interface. However, since their purpose is rather fixed several optimizations are possible.

The first processing element is used for *shared memory communication schemes* and since those schemes are only used by programmable SPEs of the GAS it is built into the shading cache. This processor can perform read-modify-write operations for *multi-prefix operations* (MPO) in an atomic way (see Section 5.3.5 and the *SB-PRAM* [Lic00, Lic96]). Additional to MPOs also *hw-malloc* (see Section 5.2.7) is supported.

Geometry Processor

The other processing element is a *geometry processor* similar to a *vertex processor* on current rasterization based GPUs. It is built into the cache of the transformation respectively intersection unit and every item fetched from memory can be processed before it is stored in the cache. Since only processed elements are cached the cache also allows for reducing the work-load on the geometry processor.

Processing of an item fetched can be done in various ways. For example the memory can contain geometric primitives in a different format than required for intersection computations and the geometry processor performs the conversion. An example is the conversion of triangle coordinates into Plücker coordinates [Eri97] or into coordinates for the fast

ray-triangle intersection test by Wald [Wal04].

Furthermore this technique can also be used to efficiently model animations of objects e.g. for water surfaces and character animation. But since a kd-tree is used to find the geometric objects which are likely to be pierced by a ray, the kd-tree must contain references to all geometric objects that could *potentially* be inside the voxel of the kd-tree (see Figure 6.5). However, similar to key-frame animation using several poses it is possible to have several kd-trees each built for a different range of the movement. This reduces the overhead and still allows for a fluent animation not restricted to a discreet stepping.

The geometry processor can be extended by an additional data path specifying the number of geometric primitives generated. This allows for on-the-fly computation of detailed geometry and to convert implicitly defined surfaces (such as splines) to triangle meshes. But since this extension requires to intersect with all generated primitives its application is limited. Especially for free-form surfaces there exists a much more efficient solution for architectures with programmable intersection units [BWS04].

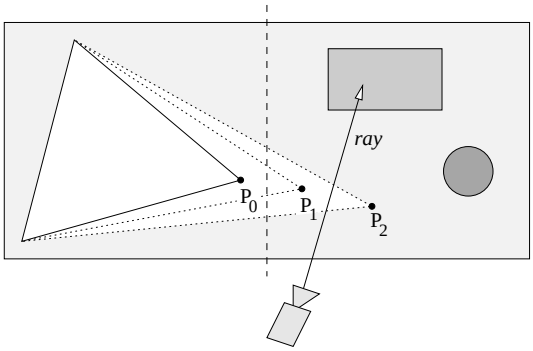


Figure 6.5: The geometry processor can modify any geometric primitive when fetched from memory. In the example above the vertex P_i of the triangle moves depending on the current time between position 0 and 2. However, since the kd-tree is independent of the time both voxels must contain a reference to the triangle to avoid missing an intersection (e.g. when the vertex is at P_2 and the right voxel would not contain the reference to the triangle).

7 Implementation

The previous chapters have shown the general design of the SaarCOR architecture. However, there is a lot of work involved in turning a general design into a working implementation that fits a given chip technology and serves application specific needs efficiently.

Therefore the next section discusses mechanisms to evaluate how the general architecture can be adapted to given requirements in scene complexity, image quality, frame rate, and chip resources. This adaption starts with the conceptual issue of how to start when too many architectural parameters are unknown.

Then a new system for cycle accurate simulations is presented that has two main features: being very fast and allowing for decoupling of gate-level hardware design and development of algorithms.

In Section 7.2 the architectural variants that have been implemented for evaluation are presented before Section 7.3 describes the SaarCOR prototype including its fixed function shaders in more detail.

7.1 Conceptual Issues

In the presentation of the general architecture of SaarCOR it has been shown several times that a ray tracing architecture is highly scalable and can be balanced and optimized in many ways. But although this allows for many application specific adaptations it also becomes one of the biggest issues on the design of a ray tracing architecture. At the beginning of the design phase simply too many parameters are unknown, which makes it hard to make proper decisions about the various options.

Thus, it is important to gather some data about the various requirements of the ray tracing algorithms. But before any measurements can be taken, first it has to be analyzed *what* can be measured. This first step is discussed in Section 7.1.1.

After the algorithms have been separated and broken into small pieces the requirements of all parts of the ray tracing system can be analyzed. Since the decisions about various options are made using these measurements the analysis has to be as accurate as possible.

However, most accurate results can only be achieved after a variant has been implemented on gate-level. Unfortunately, currently available hardware description languages do not allow for highly parameterized designs, which requires separate and very time consuming implementations for every set of parameters.

Similar to many other projects the algorithms used in the SaarCOR hardware architecture have been evaluated using software programs on standard PCs. Thus time could be saved and errors could be avoided if these programs could also be used for the evaluation of alternative implementations of the hardware architecture.

This reuse of the same programs is especially important as results are likely to change

not only some parameters but also the design of the architecture. As changes on the hardware architecture typically require new variants of the algorithms to be evaluated the development process is tightened if only a single system has to be adapted.

These considerations have led to the development of a new kind of simulator, which allows for evaluating algorithms at a cycle accurate level without re-implementation. This simulator is presented in Section 7.1.2 and forms the basis for all simulation based results of Chapter 8.

With the results of these simulations finally a synthesizable design on gate-level can be implemented. For this purpose a medium level hardware description language was used that at least allows for some parametrizations and a modular design (see Section 7.1.3).

7.1.1 Finding Estimates for Hardware Parameters

The first step of the design phase starts with a careful analysis of the software implementation of the algorithms. Here it is useful to cluster operations by functions (e.g. calculation of the traversal step or the ray-triangle intersection) and split functional units at memory accesses, jumps, and branches.

However, flow control in a hardware unit does not necessary follow the same rules and restrictions as a sequential flow control in a software program. Therefore, when splitting a program into segments it should be considered that a branch or conditional execution does not necessary require to split the operations into independent hardware units. Therefore splitting and clustering of a software program should be done carefully and with a hardware implementation in mind.

After the program has been split into segments, counters for every segment are added. These counters measure how many times a segment is executed during various benchmark scenes. Furthermore it is useful to also count the memory accesses and additionally write the addresses of the accesses to a file.

However, especially when object oriented programming methods are used it might not be obvious where memory accesses are performed or where only a variable stored in registers is accessed. This complicates this analysis and requires deep understanding of the algorithms.

Estimates for Memory Bandwidth

The addresses of the memory accesses logged in a file can be used to perform some rough estimates on the caching behavior of the algorithm. If for every access also the unit performing the access is logged hierarchies of caches with separate caches per functional group can also be evaluated.

But since the memory accesses are logged during the execution of a sequential program their order differs from the memory accesses of a highly parallel multi-threaded architecture. Thus in general only a rough estimate for the caching behavior and the external memory bandwidth can be achieved.

Nevertheless the bandwidth requirements to the caches (or to external memory if no caches are used) are exact since they are independent of the order of the memory accesses. This is especially useful when evaluating packets of rays.

Example for the Process of Estimation

The process of finding estimates is illustrated in the following example from the SaarCOR architecture. Here some first estimates are gathered by looking at the number of traversal, list, transformation, and intersection operations performed during various benchmarks.

The traversal operations are further split into calculation of the traversal decision, stack push and pop operations, and collect hits operations. Additionally, the communication between these groups has been measured and all memory accesses have been logged.

This allowed for estimates on how many floating-point operations and memory accesses are necessary to calculate various benchmark scenes. All measurements presented in Section 2.5 are results of this technique.

However, since these estimates do not take into account that the various functional units might become idle the resulting figures do only present a lower bound on how many floating-point units and how much bandwidth is needed in hardware. Still these figures allow to analyze the quality of the kd-tree and to evaluate the balancing between traversal and intersection operations.

In general there is no guarantee that these estimates are in the same order of magnitude as the measurements on the real system. Therefore it is interesting to note that for the SaarCOR architecture even the caching statistics were very close to the measurements on the real system.

7.1.2 High Level Hardware Simulation

The previous section has shown how some estimates on the hardware requirements can be gathered. Although these figures are only estimates for the hardware requirements of the whole system they represent exact measurements of the individual functional units if the temporal behavior is ignored.

For example, a process performed by the software program executes the same operations in the same order as the hardware architecture. The only information missing in the estimates derived from the software version is the latency of the function units on computations, memory accesses, or from hazards when two units access the same resources.

This is the key to an efficient high level hardware simulator that uses the measurements of the software version and only adds the temporal behavior of the functional units. This simulator uses the topology of the hardware architecture and feeds the results of the software implementation where needed.

Basically every functional unit can be emulated as a black box that simply looks up for every input the corresponding output in a file. If the box outputs the result at a delay equal to the delay the functional unit would have on a specific chip technology from an external point of view the emulation of the box cannot be distinguished from a simulation of a gate-level implementation of the corresponding circuit.

Thus the key idea is not to simulate re-implementations of the algorithms but to emulate the logic of every box by simply using the results of the existing implementation. For this kind of box-level simulation the software version has to be instrumented to write the required results to files.

Higher High Level Hardware Simulation

This concept can be taken even to a more abstract point of view. As the boxes do not compute anything there is actually no need to feed any real values into the boxes. For example instead of sending a complete triangle and a ray to a box that only emulates ray-triangle intersections simply an unique identifier for the triangle and the ray can be sent.

This allows for narrowing the simulated busses of the architecture and reduces the whole simulation to a message passing system between various black boxes. The main effort is to not to calculate any results but to simulate the decisions derived from the results. This also and most importantly includes the simulation of the temporal behavior of units.

Therefore most of the work spend on the simulator has to be done to implement the various data paths, the protocols, and control structures (such as FIFOs, MUXes with priorities and SRRMs). Furthermore, care must be taken to figure out where results have to be loaded from file and where simply passing of a unique identifier suffices.

Since besides the control structures the simulator does not calculate anything there is no direct way to ensure that the simulation actually simulates the desired hardware architecture. Therefore it is important to implement additional sanity checks to verify that the simulation is correct.

Implementation Details

The simulation of a circuit using a discreet timing model is much simpler than if using a continuous timing model. Thus if ignoring timing issues of registers (e.g. setup and hold times) a synchronously clocked circuit is most easy to simulate.

For the sake of simplicity we restrict the simulation to synchronously clocked boxes that have registers at the inputs and outputs (see Figure 7.1). However, inside a box also asynchronous circuits can be simulated.

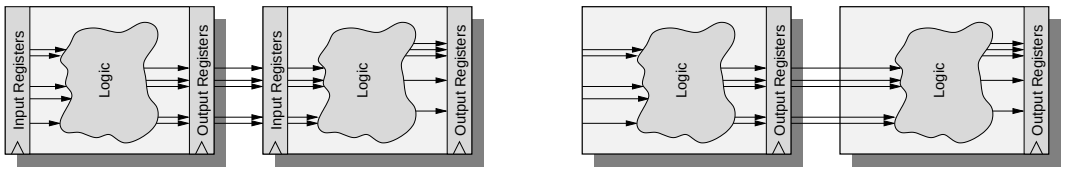


Figure 7.1: The inputs and outputs of each black box come from registers. The implementation uses only output registers and therefore input registers are only pointers to output registers (image b). However, wire delay between boxes can be simulated by increasing the delay of the corresponding black box.

The implementation of the architecture in the simulator uses a strict separation between functional units that actually compute things and general boxes to implement communication schemes. This distinction is artificial but allows for implementing a box with logic independently of the delay this box would have if implemented in hardware. Then the delay for the outputs is added by a secondary box directly connected to the outputs of the logic box.

In general the execution of real hardware is parallel and a software program simulating any hardware is executed strictly sequential. Therefore a mechanism is needed to ensure that the order in which the boxes are executed has no influence on the simulation. The mechanism used in the SaarCOR simulator realizes the connections between boxes with input and output registers implemented as arrays that can store two values per register. Then in each simulated cycle, boxes read only from index t of the corresponding input registers and write only to index $1-t$ of output registers with $t \in \{0, 1\}$ and t alternating in every cycle[1].

Since the simulator does not use any gate-level implementation the delay of the boxes has to be specified manually. Here for every box the depth of the longest path through the box has been used (taken from existing implementations or by careful investigations). Depending on the chip technology and clock frequency of the simulated architecture it was assumed that per cycle between 4 (for ASICs with 533 MHz) and 20 (for FPGAs with 100 MHz) gates can be computed. For example the ray-triangle intersection circuit was simulated using 20 to 100 pipeline stages.

Building Netlists

Before the simulation can be started the hierarchical and parameterized description of the architecture needs to be *flatted*. This means that *meta modules* describing the instantiations of *functional modules* are executed and then removed resulting in a flat hierarchy of interconnected boxes. Flatting is done fully automatically in less than a second on standard PCs although it additionally checks the wiring for open connections and shortcuts.

The full fixed function architecture with four pipelines uses a hierarchical design with 23 meta modules that describe how the 24 general purpose modules (e.g. FIFOs) and the 36 special purpose modules (e.g. calculate traversal decision) are instantiated and connected. The flatted netlist contains 510 individual black boxes. These figures give an idea of the complexity and the granularity in which the communication scheme has to be broken down.

Limitations of Behavioral Dumps

This basic version of the simulator is ideally suited to perform quick and highly efficient simulations of architectures with a rather fixed flow control. Especially since the files generated by the instrumented software are independent on the actual hardware architecture they need to be calculated only once but can be used in many simulations of various architectural variants.

However, the emulation of the more complex behavior of a processor like the SCPU shows the limitations of the concept of reading the results from file. Here simply too many pairs of inputs and outputs for various situations have to be stored. Thus in this case it is more efficient to read the programs and input data from file and actually perform the simulation of the CPU than to use files with results only.

This sounds more expensive than it is since the simulation of the CPUs does not require

[1] If c is a integer value specifying the number of the current cycle then $t =< c > [0]$, i.e. t is the least significant bit of the binary representation of c.

a gate-level implementation and can be mixed efficiently with black boxes that read their results from files. This trade-off between simulations and emulations using files might also be useful for other parts.

Nevertheless, all fixed function variants of the SaarCOR architecture have been evaluated using simulations of the communication schemes and black box emulations with results stored in files. This allowed for very efficient and fast cycle accurate evaluations. Therefore the cycle accurate simulation of rendering a full screen image (1024×768 pixels) took roughly only one hour per simulated pipeline on a standard PC with 2 GHz.

7.1.3 Medium Level Hardware Development

The implementation of highly parametized architectures with many options for balancing and parallelization would greatly benefit from a high level hardware description language that allows for specifying the design using only a few lines of code. Especially the support for parameterized pipelining and parallelization which could be compiled automatically and correctly into a synthesizable low level hardware description language would greatly promote hardware development.

Unfortunately, there is no such language and compiler. Today the industry's standard for hardware development is still low level entry software such as VHDL, Verilog, or Schematic Entry. Besides these low level tools there are some converters for higher level languages, e.g. *System-C* [Sys03] and *Handle-C* [Cel03].

However, although converters for higher level languages perform fairly well for turning an existing program, e.g. written in C, into hardware they do not support parameterized designs. Furthermore, since these tools are designed to convert standard high level language programs into hardware there is no direct way to describe how the data paths of the hardware should look like. Thus for experienced hardware designers who know how to implement a function efficiently it is hard to write a C program such that the compiler outputs what the designer had in mind.

In contrast to these commercial systems the Brigham Young University offers an Open-Source *Java Hardware Description Language* (JHDL) [Bri03]. This system is built as a library that can be linked to any program written in Java and allows for parameterized designs at a medium level. For example the floating-point units can be designed such that their precision and the number of bits used for exponent and mantissa can be specified by parameters.

An important advantage of JHDL is that it allows for rapid prototyping in a very easy and effective way. For example synthesizable implementations of circuits can be connected directly to standard Java programs which allows for fast and easy verifications of the inputs and outputs of the circuit under test. These features of JHDL are the reasons for choosing it as the main tool for the development of a prototype of the SaarCOR-architecture (see Section 7.3).

7.2 Implemented Architectures

The previous chapters presented a general architecture for ray tracing with many options and parameters. However, similar to other projects of that size not all parts were devel-

oped and implemented at once but incrementally and every step used the results of the previous step for improvements.

Since the implementation of an ASIC was beyond the capabilities of a small University project most variants of the SaarCOR architecture were only simulated using the cycle-accurate simulator presented in Section 7.1.2. However, optimizations on a variant of the SaarCOR architecture for dynamic scenes allowed for a prototype implementation using FPGAs (see next section).

In general the variants of the SaarCOR architecture can be split into the following classes: fixed function hardware for ray tracing of static scenes (*static SaarCOR*), fixed function hardware with support for dynamic scenes (*dynamic SaarCOR*), and variants with support for programmable components (*programmable SaarCOR*).

Static SaarCOR

Guided by the estimates presented in Chapter 2.5 several variants of the fixed function SaarCOR architecture for static scenes have been simulated and implemented. *SaarCOR-A* was the first variant that was implemented and did not include shading. Together with a *virtual memory interface* shading was added in *SaarCOR-C*. The variants *SaarCOR-B* and *SaarCOR-D* are similar to *SaarCOR-A* respectively *SaarCOR-C*, but differ in some internal parameters (e.g. number of rays per packet and cache size) and the benchmarks used.

Architectural details on these variants and the exact parameters are given in the Chapter 8 were also the results of various measurements are presented. Figure 7.2 presents the general architecture of a fixed function hardware for ray tracing of static scenes *SaarCOR-C*.

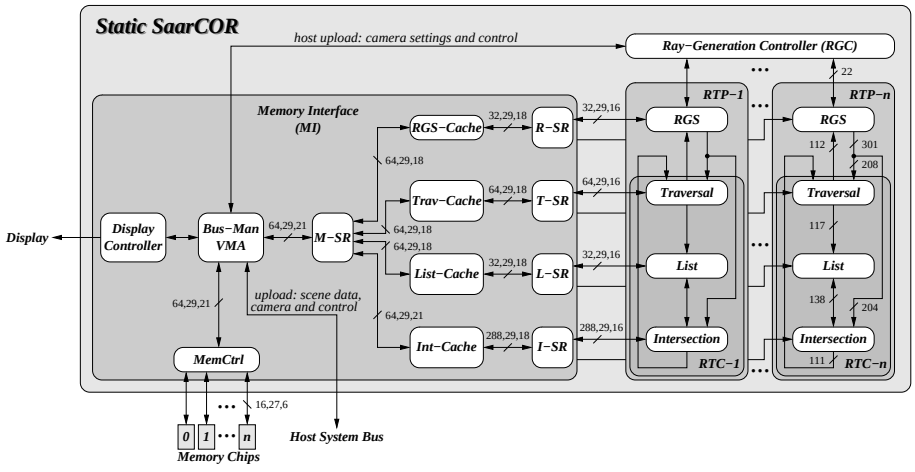


Figure 7.2: Data paths of a generic variant of a fixed function SaarCOR architecture for static scenes. The memory interface supports virtual memory, uses only one level of caches and has no support for advanced operations (like MPO and adjustable cache policies). Additionally, the width of the busses is shown.

Dynamic SaarCOR

The variants of static SaarCOR have been extended to support ray tracing of dynamic scenes. But as the type of scene does not change the behavior of ray tracing fundamentally simulations of dynamic SaarCOR have shown no unexpected results. However, careful analysis of the operations performed during ray tracing of dynamic scenes allowed for reducing the hardware requirements even below that of static SaarCOR.

These optimizations allowed for a prototype implementation using FPGA technology. Figure 7.3 depicts the general dynamic SaarCOR architecture while the prototype and the optimizations that were used are presented in the next section.

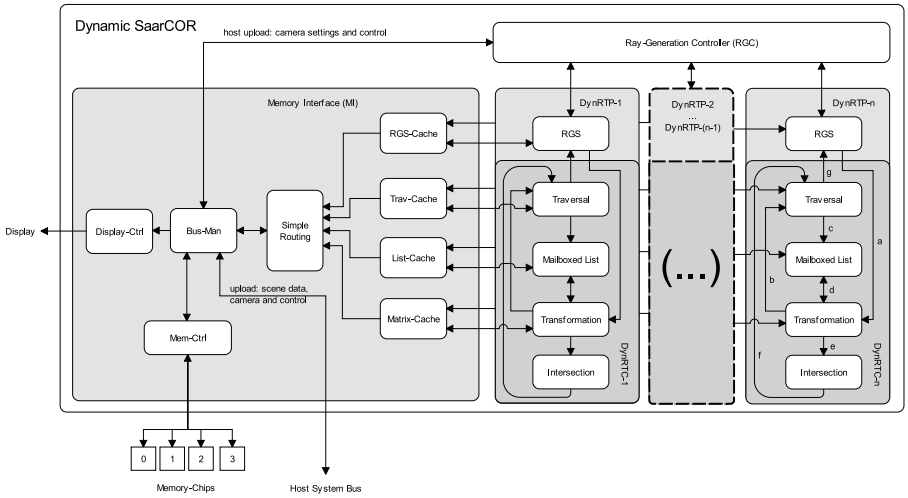


Figure 7.3: Data paths of a generic variant of a fixed function SaarCOR architecture for dynamic scenes. The SaarCOR prototype implements this architecture but uses only a single pipeline and has no cache for shading data. A fairly detailed description of the prototype can be found in Section 7.3.

The development of the SaarCOR prototype was done mainly using JHDL for all parts of the ray tracing pipeline. Additionally, Schematic Entry was used to glue several JHDL-implemented units together. Using these tools a team of three students implemented a fully working prototype withing three months from scratch. This time also includes writing a device driver, the software applications, and soldering a DA-converter that connects the FPGA to a standard monitor. Figure 7.4 shows the first images rendered on the prototype for dynamic SaarCOR.

Programmable SaarCOR

Some amount of work has been spend on the development of programmable shading and a fully programmable variant of SaarCOR. The master thesis of Patrick Dreker [Dre05b] presents a first implementation of a multi-core SCPU system. The same FPGA that was used for the prototype (see Section 7.3) can alternatively hold 12 identical SCPUs. Here similar to the SaarCOR prototype the limiting factor was the available on-chip memory. But although this system has been evaluated using several benchmarks this is still work in progress.

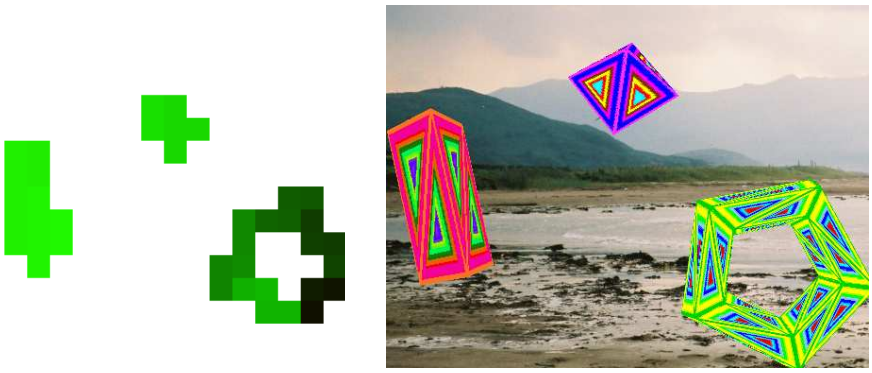


Figure 7.4: On the left is the first image rendered by the SaarCOR prototype on November, 13th 2003. The resolution was limited to 16×16 pixels since the frame buffer had to fit into internal RAM. One week later the external memory (with a frame buffer of 512×384 pixels and textures for shading) was running as can be seen on the right. This image shows the same scene but already includes texturing using the barycentric coordinates directly as address (also the background is a texture).

7.3 Prototype Architecture

When it comes to building the first prototype of a new architecture this always becomes a challenging task. Now it has to be proven that results gathered by simulations can be verified using a real world prototype.

As it was mentioned earlier without having a high level hardware description language allowing for parameterized designs there is no fast and easy way revising decisions once made. Therefore the first prototype of SaarCOR was developed using very conservative features and constraints.

However, using only a single FPGA with a rather low clock frequency (expected to be below 100 MHz) it was clear that the resulting performance in the best case would not be too impressive compared to the performance achieved on standard processors with several GHz. Especially when comparing the expected frame-rates to those typically achieved for games on standard graphics cards it is likely that the first ray tracing prototype looks rather uninteresting.

Therefore it does not suffice to build a proof of concept that computes bundles of rays quickly and it is necessary to design a full featured prototype for ray tracing based graphics. This led to some effort spent on the development of a shading system that is simple but at the same time allows for secondary rays and interesting special effects (see Section 7.3.2). Additionally, applications have been written that show nice demos using moving objects, animated textures, and even have 3D sound effects.

Design Decisions

A major goal in the development of the prototype was to keep the design as simple as possible. Thus the basis for the prototype is given by a commercial FPGA prototyping board (see Section 7.3.3). This board is equipped with SRAM and DDR memory, but

since the implementation of the SRAM interface is much simpler and guarantees fixed latencies it has been prefered over the much larger DDR memory.

The six banks each consisting of 4 MB SRAM memory were split statically between the various units. Two of the banks are used for a double buffered frame buffer (supporting up to 1024×768 pixels in true color 24 bit RGB), a single bank stores shading data and textures, and three banks contain the kd-trees and triangles.

This prototype board is a PCI card and therefore a PCI bus interface has to be implemented for communications to the host. Again for the sake of simplicity only a simple design has been chosen that does not allow for DMA transfers and therefore limits updates to the scene to 15 MB/s and downloads (e.g. for framebuffer read backs) to only 1 MB/s.

This low bandwidth for the PCI bus does not allow for transferring the ray traced image to a standard graphics card for displaying. Therefore a tiny self-made board with D/A converters that connects the FPGA directly to a standard VGA monitor was developed.

The next section will show that an optimized variant of a fixed function SaarCOR architecture for dynamic scenes requires even less resources than any variant for static scenes. Therefore the prototype implements this optimized variant with support for ray tracing dynamic scenes with two levels of hierarchy. Supporting dynamic scenes allows for object instantiation and thus helps to overcome the restrictions of the memory size which allows for only roughly 50 000 unique triangles (depending on the size of the corresponding kd-tree).

Choosing The Number of Rays Per Packet

All measurements presented so far where performed for variants of the SaarCOR architecture implemented in ASIC technology supporting multiple ray tracing pipelines on a single chip using only a relatively small bandwidth to external memory (see Chapter 8 for details). Therefore these variants required large packets of rays for latency hiding and bandwidth reduction.

However, the memory system of the prototype has a quite different compute-to-bandwidth ratio. Here the memory is clocked at the same frequency as the ray tracer and the 128 bit wide connection serves random memory accesses with a guaranteed latency of only a few cycles. Furthermore the FPGA only provides enough resources for a single pipeline with four traversal slices.

Since a relatively high bandwidth to external memory is available there is no need for large packets which allowed for using packets with four rays only. As even simple SIMD-like scheduling performs well for small packets (see Section 2.5) the amount of logic required for managing packets could be reduced. Nevertheless, packets of four rays were not sufficient for the required bandwidth reduction (see Section 7.3.3) and therefore some small caches (of only 12 KB in total) had to be used for geometry while shading could be performed uncached.

7.3.1 Optimizations

The most important optimizations (shown in Figure 7.5) on the architecture for the implementation of the prototype were the use of the *unit triangle intersection* (Section 4.5) and to perform *shading using the transformation unit* (Section 5.4).

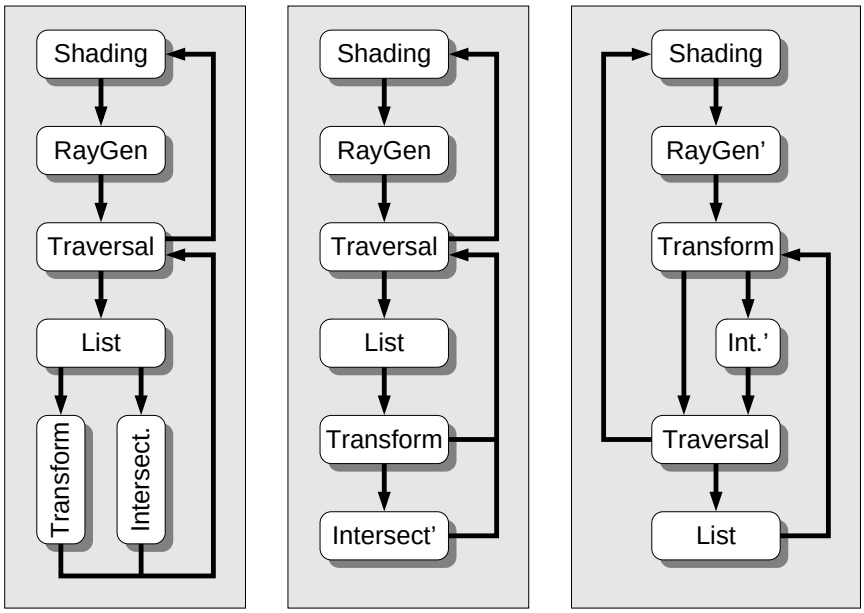


Figure 7.5: The images from left to right illustrate how great amounts of hardware can be saved when reusing the transformation unit for ray-triangle intersection computations (the image in the middle) and additionally for shading (the right most image). Compared to a standard implementation using dedicated hardware units (the image on the left) the architecture on the right uses 68% less hardware and is therefore ideally suited for a prototype implementation. Please note that the optimized versions use reduced versions of the Intersection respectively the RayGen units.

The following evaluates the savings in hardware gained by these optimizations. Here the costs have been measured by counting the number of floating-point operations of four different implementations (see Appendix D for details). All variants shown here have the traversal and the list unit in common. However, of these two only the traversal unit requires floating-point operations.

The first variant RT_{static} consists of a static ray tracing pipeline using a ray-triangle intersection algorithm I_w based on Wald [Wal04]. Additionally, this variant uses a simple shading unit S that performs the geometric calculations on rays with floating-point operations but all other computations using integer arithmetic only.

The second variant RT_{Dyn1} is a dynamic ray tracer using the optimizations described above. Therefore additionally to the traversal unit only the transformation unit T and a small primitive intersection unit require floating-point operations.

Finally, 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 coordinates [Eri97], which is a bit cheaper than Moeller-Trumbore's intersection test [MT97].

Table 7.1 shows that with the optimizations presented above, 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 functionality. Furthermore, re-using the transformation unit for other purposes reduces the number of floating-point units at least by a factor of two.

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 7.1: Comparisons of the cost of four different ray tracers measured in floating-point units. It shows that the optimizations presented in this section allow to significantly reduce the hardware costs.

Besides these architectural optimizations also the arithmetic can be optimized to the rather limited resources of the prototype. Therefore the floating-point units are streamlined using the options presented in Sections 5.3.1 and 5.3.5. The format of the floating-point numbers is similar to the 24 bit format on current GPUs by ATI and uses a 7 bit large exponent represented in two's complement presentation and does not support special cases. In several benchmark scenes this floating-point format has been proven to be a good compromise between accuracy and hardware cost.

7.3.2 Shading

Besides yielding a high performance one of the major goals in the development of the SaarCOR prototype was to design a system that can present ray tracing as a clear alternative to rasterization based graphics. Therefore in addition to the optimizations on the ray tracing core that allow for good performance reasonable efforts have been spend on the development of a shading system that also allows for special effects. However, this goal had to be achieved using only rather limited resource that are neglectable compared to those of the RTC.

Thus a fixed function shading system consisting of six different shaders has been developed. All operations on colors are performed using integer arithmetic only with colors in 3×8 bit RGB format. Similarly, the operations in texture space are computed using integers only and care has been taken that address computations for textures can be implemented with rather low effort.

The barycentric coordinates (see Section 2.1.3) and the absolute value of the cosine between direction of the ray and normal of the triangle are required for shading. Fortunately the RTC calculates these values for free using floating-point arithmetic (see Section 5.4). However, since for valid hits these values are guaranteed to be in the range of $[0, 1]$ they can be stored easily using 10 bit integers only which allows for direct use in the integer-based shaders and additionally saves memory for storage in the RTC.

The following paragraphs present an overview of the shading architecture, the formulas how rays are shaded, the arithmetics used for texturing, the fixed function shading pipeline, and the special effects system.

Overview

The shading architecture uses *coordinated ray generation* (see Section 5.2.3) and the formulas presented in Section 5.4 to calculate packets of rays. At the beginning of a new frame *primary ray generation* is initialized which starts to generate as many packets of new primary rays as there can be threads in the ray tracing core (RTC). Sending each

packet of four rays to the RTC takes at least 10 cycles: four cycles for the rays, four cycles for the matrix and at least two additional cycles due to communication overhead.

When traversal of a packet is finished the RTC returns the packet together with the hit-information to the *shader pipeline*. This pipeline is 29 stages deep and depending on the shading performed it issues up to four memory requests and at most one frame buffer write (see Table 7.2). If no secondary rays are required for shading, a command is sent to the primary ray generation to generate a new packet of primary rays.

If secondary rays are required for shading a new packet of secondary rays is sent to the ray tracing core. This transfer takes at least 27 cycles due to the matrices required for each ray (see Section 5.4) and additionally a minimum of 7 cycles due to communication overhead.

Since there are two sources of new packets of rays there is a *select next packet* unit that switches between both sources. Figure 7.6 gives an overview of the ray generation and shading in the SaarCOR prototype.

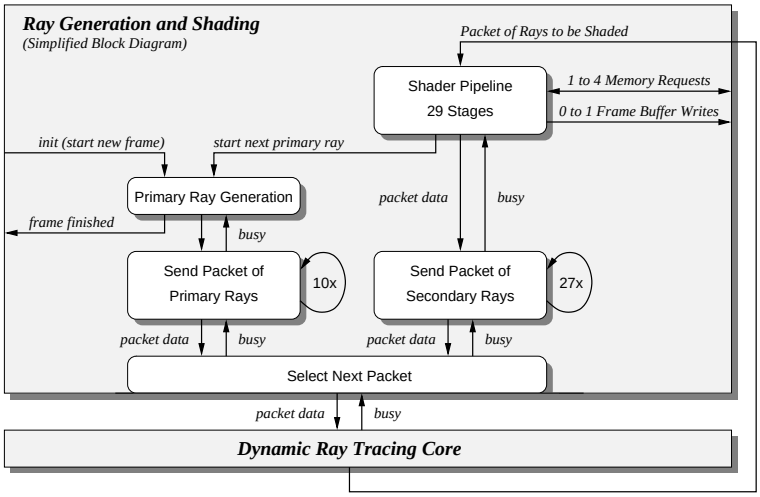


Figure 7.6: Simplified block diagram of the ray generation and shading unit.

Calculation of the Color

Shading is performed using the following formula on 8 bit integers:

$$\begin{matrix} new \\ color_c \end{matrix} = \begin{matrix} weight \\ of\ ray \end{matrix} \cdot \begin{matrix} env. \\ factor \end{matrix} \cdot \begin{matrix} color_c \\ of\ light \end{matrix} \cdot \begin{matrix} color_c \\ of\ triangle \end{matrix} + \begin{matrix} previous \\ color_c \end{matrix}$$

with $c \in \{$ *red*, *green*, *blue* $\}$, and

weight of ray	depending on recursion depth of ray
env. factor	cosine between normal and viewer or light intensity
color of light	color of ambience light or direct light
color of triangle	material color or texture color of triangle
previous color	current color of ray or special effect color

The multiplication with the *weight of ray* is removed by allowing either a reduction of the ray's influence by 50% on each recursion or to keep it at 100%. This allows for implementing the multiplication using a simple right shift with the distance set to the recursion depth.

The addition is implemented using 8 bit wide adders with saturation[2]. Together with the 8 bit wide integer adder used to generate shadow rays from (at most) 256 light sources the arithmetic requirements of the color calculation are only 6 multipliers and 4 adders working on 8 bit wide unsigned integers.

Texture Calculations

In the context of rasterization based graphics many work has been spend on the implementation of texturing hardware. However, since these methods deal with incrementally texturing whole triangles here a different approach is used. The barycentric coordinates of a triangle are used to interpolate between the texture coordinates at the vertices yielding exactly the texel required to color the hit-point.

The formulas presented below use integer arithmetic with varying precisions that only output the p most significant bits as denoted by the following notation:

`a` $*_p$ `b` outputs `o[n+m:n+m-p]`, *with* `o[n+m:0] = a[n:0] * b[m:0]`
`a` $+_p$ `b` outputs `o[p-1:0]`, *with* `o[1+`*max*`(n,m):0] = a[n:0] + b[m:0]`

The 20 bit wide memory address `ad[19:0]` of the texel depending on the resolution of the texture (between 32×32 and 256×256 texels) is calculated as:

$$\texttt{as[14:0]} = \texttt{<s0[9:0],}0^5\texttt{>} +_{15} \texttt{s1[9:0]} *_{15} \texttt{u[9:0]} +_{15} \texttt{s2[9:0]} *_{15} \texttt{v[9:0]}$$
$$\texttt{at[14:0]} = \texttt{<t0[9:0],}0^5\texttt{>} +_{15} \texttt{t1[9:0]} *_{15} \texttt{u[9:0]} +_{15} \texttt{t2[9:0]} *_{15} \texttt{v[9:0]}$$

$$\texttt{ad[19:0]} = \begin{cases} < \texttt{base}[19:10], \texttt{at}[\ 9:5], \texttt{as}[\ 9:5] >, \text{if texture} = & 32 \times 32 \\ < \texttt{base}[19:12], \texttt{at}[10:5], \texttt{as}[10:5] >, \text{if texture} = & 64 \times 64 \\ < \texttt{base}[19:14], \texttt{at}[11:5], \texttt{as}[11:5] >, \text{if texture} = & 128 \times 128 \\ < \texttt{base}[19:16], \texttt{at}[12:5], \texttt{as}[12:5] >, \text{if texture} = & 256 \times 256 \end{cases}$$

Here `s`i, `t`i, with $i = \{0, 1, 2\}$ specifies the texture coordinate (s, t) of vertex i and `u, v` denotes the barycentric coordinates of the hit-point. The start address of the texture in memory is given by `base[19:0]`, and `as, at` are the interpolated texture coordinates. Here 15 bits of precision are used to allow for replicated textures on triangles (4 replications for 256×256 and 32 replications for 32×32).

Beside *sample nearest filtering* as presented above also *bilinear texture filtering* is supported. Here the four texels `t00`, `t01`, `t10`, and `t11` are read from memory using almost the same formulas as above. In the memory access `M[ad(as,at)]` only the interpolated texture coordinates vary:

`t00 = M(as,at)`, `t01 = M(as,at1)`, `t10 = M(as1,at)`, `t11 = M(as1,at1)`
with `as1[12:5] = as[12:5]` $+_8$ `<`0^7`,1>` and `at1[12:5] = at[12:5]` $+_8$ `<`0^7`,1>`

Then the interpolated color `col[7:0] = bst[8:1]` is calculated using:

`bs0[8:0] = t00[7:0]` $*_9$ `as[4:0]` $+_9$ `t01[7:0]` $*_9$ `/as[4:0]`
`bs1[8:0] = t10[7:0]` $*_9$ `as[4:0]` $+_9$ `t11[7:0]` $*_9$ `/as[4:0]`
`bst[8:0] = bs0[8:0]` $*_9$ `at[4:0]` $+_9$ `bs1[8:0]` $*_9$ `/at[4:0]`

[2]Saturation is simply implemented with an 8 bit wide *OR* of the result and the carry out signal of the adder.

For the bilinear interpolation a fractional part of 5 bits is used (i.e. `as[4:0]` respectively `at[4:0]`). This allows for 32 shades between two texels which has been shown to be sufficient even for close ups in higher resolutions.

Comparing the number operations spent on texturing to the calculations performed when shading a ray shows interesting figures. For sample nearest texturing only 4 multiplications and 4 additions are necessary which is roughly equal to the complexity of the shading operation. However, for bilinear texture filtering 22 multiplications and 15 additions are required which is roughly four times higher than for shading itself.

Fixed Function Shading Pipeline

The fixed function shading pipeline implements the six shaders: *material color surface shader*, *sample nearest textured surface shader*, *bilinear filtered textured surface shader*, *light shader*, *effect shader*, and *ray loss shader*. The ray loss shader handles every ray that did not pierce any geometry and applies any of the three surface shaders to it using the pixel's coordinates as barycentric coordinates to allow for a background texture (see Figure 7.4 for an example).

The light shader calculates the contribution of up to 256 light sources and generates the corresponding shadow rays. Besides shading the current ray the surface shaders additionally generate reflection or transparency rays according to the material properties of the triangle. Details about the effect shader that reads data from a special purpose *BG Texture* using the pixel's coordinates as index are given in the last paragraph of this section.

However, the shading pipeline supports three different modes *s1*, *s4*, and *s8*, but only mode *s8* allows for choosing from all shaders. In mode *s4* no bilinear texture filtering and no effects are supported. The most restricted mode is *s1* which supports only the material surface and ray loss shader, and no secondary rays.

The names of the modes have a special meaning and denote the frequency in which new rays can be given as input which is equal to the number of memory accesses the longest supported shader performs. However, since all shaders share many functional units they are implemented as only a single shading pipeline. Therefore shading has always the same latency of 29 cycles (see Table 7.2) and only the frequency with which new rays can be sent into the pipeline varies depending on the mode.

Not only due to the varying frequencies the three shading modes can influence the performance of the system. Especially the number of cycles it takes until a new packet of rays is sent to the RTC dominates the performance. Since in mode *s1* secondary rays are not supported as soon as a packet is received for shading in parallel to shading the packet already a new primary packet can be sent to the RTC[3]. Therefore *s1* delivers the highest possible performance.

In the other modes the input frequency is lower and a new packet can be sent only after shading is finished. The total frequency of shading can be calculated as:

number of rays per packet × *input frequency* + *latency of shading pipeline*.

Therefore a new packet can be sent only after 48 cycles in mode *s4* and after 64 cycles in mode *s8*. However, sending a packet of rays itself takes some amount of time, which

[3] For this purpose there is an additional shortcut not shown in Figure 7.6 that issues *start next primary packet* immediately and shortcuts the standard latency of 29 cycles.

is 10 cycles for packets of primary rays and at least 27 cycles for packets with secondary rays. Nevertheless, unless very simple scenes are rendered even *s8* shading does not influence the performance (see Section 8.5).

Depending on the mode of the shading pipeline data is read from memory regardless whether the shader actually requires it. In mode *s1* only the material data (color and properties) are read (stage 2). In mode *s4* additionally the texture coordinates, texture base address and the normal of the triangle are read (stages 3 and 4). For sample nearest filtering stage 13 performs the texture read. In mode *s8* additionally bilinear filtering is supported and the corresponding texels are read in stages 14 to 16. The special effect texture is read in stage 17. Every texel additional includes material properties, e.g. specularity, opacity and flags for special effects.

The generation of secondary rays is managed using *coordinated ray generation* (see Section 5.2.3). However, there is a restriction on secondary rays to save on-chip memory which prohibits shadow rays from spawning additional rays (i.e. light does not shine through glass) and allowing a surface shader to spawn only either a reflection or a refraction ray. Nevertheless, there is no restriction on the number of rays that can be spawned recursively due to tail recursion which has constant memory requirements.

Special Effects

There are two sets of effects supported on the fixed function shading system. The first group consists of special purpose material flags which allow to skip shadow rays (e.g. to guarantee that no shadows are casted on sky boxes and lights) and to ignore the cosine (e.g. to avoid that faked lights included in textures are darkened by the angle to the viewer).

The second group handles 2D overlays (Figure 7.7). In contrast to standard implementations here the overlays are implemented in reverse order. Thus instead of adding the overlay to the final image the starting color of each pixel is set according to the pixel in the overlay. This removes the read-modify-write cycle and allows for a very cheap implementation since the adder for the *previous color* already exists (to add the contribution of secondary rays) and is unused otherwise during initialization.

However, instead of using a texture for the overlay this adder can also simply add the same value to all pixels of the image. This allows for effects like dazzling by the sunlight, flashes (e.g. when shooting), or “seeing red” (e.g. like the “berserk” mode of a very famous game by Id-Software [IS04]).

Figure 7.7: Without any special effects the left image looks rather simple. However, much nicer presentations are possible by using 2D overlay effects (the image in the middle). For comparison the image on the right shows only the overlay. (All images are rendered in 1024×768.)

Stage	Activity 1	4m	4t	8	Action
0	■	■	■	■	Ray and hit data ready from RTC
1	■	■	■	■	Handle ray losses, Modify environmental factor
2	■	■	■	■	Memory read: Material Data
3	□	■	■	■	Memory read: Extended Material 0
4	□	■	■	■	Memory read: Extended Material 1
5	□	□	□	□	(waiting for memory)
6	□	□	□	□	(waiting for memory)
7	□	□	□	□	(waiting for memory)
8	□	■	■	■	Mem.ready: Material Data
9	□	■	■	■	Mem.ready: Extended Material (0)
10	□	■	■	■	Mem.ready: Extended Material (1)
11	□	□	■	■	Calculate Texture Address (0)
12	□	□	■	■	Calculate Texture Address (1)
13	□	□	■	■	Memory read: Texture (00)
14	□	□	□	■	Memory read: Texture (01)
15	□	□	□	■	Memory read: Texture (10)
16	□	□	□	■	Memory read: Texture (11)
17	□	□	□	■	Memory read: BG-Texture
18	□	□	□	□	(waiting for memory)
19	□	□	■	■	Mem.ready: Texture (00)
20	□	□	□	■	Mem.ready: Texture (01)
21	□	□	□	■	Mem.ready: Texture (10), Filter texture bi-lin: 00-01
22	□	□	□	■	Mem.ready: Texture (11)
23	□	□	□	■	Mem.ready: BG-Texture, Filter texture bi-lin: 10-11
24	□	□	□	■	Modify BG-Texture, Filter texture bi-lin.: final
	■	■	■	■	Select color of light, Select environmental factor
25	■	■	■	■	Select color of triangle, Store ray data
	■	■	■	■	Calc.: *environmental factor* · *color of light*$_c$
26	■	■	■	■	Calc.: *result*$_c$*(stage 25)* · *color of triangle*$_c$
27	■	■	■	■	Calc.: *result*$_c$*(stage 26)* · *weight of ray* + *prev.color*$_c$
	□	■	■	■	Calc.: Secondary rays and active vectors
28	□	■	■	■	Store: *prev.color*$_c$ = *result*$_c$*(stage 27)*, active vectors
	□	■	■	■	Calculate number of light source

Table 7.2: The fixed function shading pipeline implementing six shaders in a single pipeline. In each stage of the pipeline calculations and memory accesses are performed (marked with ■) or skipped (marked as a wait-state □) depending on flags denoting the activity. Each of the four columns describes the activity of a different surface shader. The first column "1" describes mode *s1* and uses material colors only without secondary rays. Column "4m" is the material shader in mode *s4* and "4t" for the same mode is the sample nearest texture filter. The last column "8" is used by the bilinear interpolated texture filter in mode *s8*. Due to the use of SRAM all memory latencies are fixed. Please note that depending on the mode even in a single stage the activity of functional units may vary since pipeline stages are not made up by grouping functional units logically but by grouping units depending on the timing to achieve the highest clock rate and shortest pipeline depth.

7.3.3 Hardware Complexity

The SaarCOR prototype is built on hardware made in 2003 and uses 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 (see Figure 7.8).

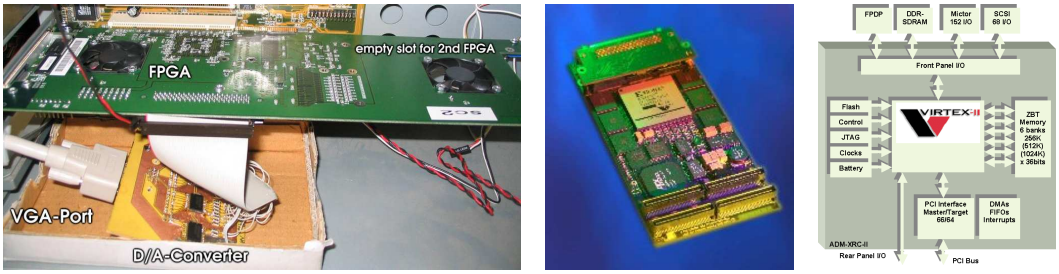


Figure 7.8: The left image shows the inside of the PC which is equipped with the prototyping board. Each PCI-board can hold up to two mezzanine cards (the image in the middle) containing an FPGA, 6 banks of SRAM and (optionally) DDR-RAM. The right image shows the data paths of this card. Additionally, the left image shows the self-made DA-converter interfacing the prototyping board to a standard VGA-monitor. (Both images on the right are courtesy of Alpha-Data [Alp03].)

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 which nicely allowed to use the PC containing the prototyping board by several people on separate PCs simultaneously and even remotely from home.

The memory bandwidth for shading is rather low because only a small amount of shading data is required (28 bytes 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 the next Chapter) that it even did not require caches (which nonetheless would be fairly effective as shown by our simulations).

Section 3.3.1 has shown that for memory and arithmetic reasons a ray tracing architecture should support at least four times more traversal operations than ray-triangle intersections. Furthermore for the prototype it is desired to use as few rays per packet as possible to keep the hardware requirements low. Therefore 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 since they could perform a memory request in every cycle. Due to small direct mapped caches (only 12 KB in total, see Section 8.5) that already yield good hit-rates (typically 70-95%) and the non-perfect utilization (typically 20-80%) we easily reduce the bandwidth to a small fraction of the 1 GB/s available to the prototype (for

details on the figures see Section 8.5).

Table 7.3 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. Additionally, the rightmost column provides the amount of internal memory required per unit. 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	–	–
RTC-Cache	–	–	–	–	15.6 KB
Shader	–	–	–	–	4.8 KB
Total	16	11	5	13	75.0 KB

Table 7.3: 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 pipeline 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.

Comparisons

Even at the beginning of 2004 much larger FPGAs were available 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 high end rasterization hardware of 2004. For instance, Nvidia's GeForce FX-5900 contains 125 million transistors [Nvi04] (3-times more than Intel's Pentium-4 at that time). 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.

In fact the SaarCOR prototype has roughly the same floating-point performance as Nvidia's Riva 128 [Ber02] built in 1998! The next chapter will shown that the memory bandwidth of the prototype 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 graphics boards already in 2004 offer more than 30 GB/s external memory bandwidth, this would support more than 100 independent ray tracing pipelines.

Thus despite that a direct comparison is not really possible, the resources available on current graphics cards would allow for quite a large number of parallel ray tracing

pipelines. This indicates that this prototype could be scaled by one to two orders of magnitude going from the current FPGA technology to one used by rasterization engines in 2004.

Furthermore Section 8.5 compares the performance of the prototype with the highly optimized SSE-variant of the OpenRT software raytracer on a Pentium-4 and 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 and uses its floating-point resources 7 to 8 times more efficiently.

For comparison the fastest published ray tracer on GPUs delivers 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.

8 Results

"Absolutely nothing should be concluded from these figures except that no conclusion can be drawn from them."
Joseph L. Brothers [BSD94]

The SaarCOR architecture presented in this thesis was not developed in a single step but incrementally designed and evaluated over three years. Therefore later variants contain improvements derived from previous evaluations and some of the extensions currently known are not covered by measurements in this chapter.

As the measurements performed were always designed to evaluate specific properties of the architecture also the parameters for each measurement are slightly different. Nevertheless since the architecture was only extended but never any general changes had to be made all results are still valid.

Additionally to the results previously published there are some new measurements filling the gap between the various sets of parameters. Therefore as an easy reference Table 8.1 presents an overview of the measurements and Table 8.2 shows a short summary about the corresponding results.

Details on the measurements can be found in the corresponding sections of this chapter. Finally, Section 8.6 summarizes all measurements and results, and draws conclusions on the hardware architecture.

Section	Architecture	Publication
8.1	SC1, no shading, large packets and large caches	[SWS02]
8.2	SC1, with shading, various sizes for packets and caches	[new]
8.3	SC1, with shading, VMA, large packets and large caches	[SLS03]
8.4	SC1, like 8.3 but smaller caches, improved kd-tree	[WPS+03]
8.5	fixed function prototype, dynamic scenes (two levels), several variants of shading, small packets and small caches	[SWW+04] +[new]

Table 8.1: Overview of the various variants of the fixed function SaarCOR architecture and in which section they are evaluated. The architecture SC1 is a variant for ray tracing of static scenes that has been simulated only.

Parameters of Benchmarks

The various variants of the SaarCOR architecture have been evaluated using a large number of different benchmark scenes. However, every scene allows for several benchmarks by changing the point of view, the image resolution, the number of secondary rays (by using different light sources and material properties), the object management (i.e. static with a single object or dynamic using several objects), and the algorithm used to build

Section	Results
8.1	performance similar to rasterization for same costs, architecture is highly scalable and guarantees high usage, low external memory bandwidth and high cache hit-rates performance independent on type of ray
8.2	relation between size of packets to cache size and bandwidth
8.3	automatic virtual memory management performs very well
8.4	improvements in performance and issues with VMA
8.5	prototype verifies simulation based results prototype verifies estimates of the chip complexity ray tracing of dynamic scenes performs very well ray tracing on prototype more efficient than on GPU and CPU

Table 8.2: This overview presents the most important results and in which section details can be found. However, typically later measurements also showed and verified previous results using a newer architectural variant.

the kd-trees.

These parameters of the benchmark scenes have been changed according to the key parameters under evaluation to stress the corresponding variant of the architecture. Therefore the geometry of a scene (referenced by the name of the scene) does not suffice to specify the benchmark performed. This is the reason why in every section the benchmarks are listed explicitly although it would be more convenient to list the parameters together with the corresponding images that are shown in Figure 8.1.

Benchmark Scenes

A wide range of 3D applications is covered by using a variety of scenes ranging from simple scenes, such as a single room with table and chairs as in *Scene6* (original untextured scene courtesy of Pete Shirley) 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, courtesy of Oliver Deussen).

The full details of all scenes are rendered with no level-of-detail mechanisms since 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 examples are taken from computer games, such as *Castle* [Act02], *Quake3* [IS04], *DMID*, and *UT2003* (both [Dig04]). *Rabbit* is a variant of *UT2003* with nice lighting effects and a madly bouncing rabbit that has to be killed in the first game running on the SaarCOR prototype. The *Castle* scene also shows some nice ray tracing effects including multiple reflections.

The *Quake3* scene is used in several variants with additional details in *pQuake3* and 16 moving players and monsters in *jQuake3*. The *BQD* scene places the whole *Quake3* scene into a valley of a huge terrain. While all of these variants use a highly tessellated version of the *Quake3* scene the benchmarks in Section 8.1 use only a standard version (for historic reasons).

Other scenes like *Office* and *Conference* (both originally untextured models are courtesy of Greg Ward) provide additional examples of indoor scenes showing an office and a conference room with many light sources. The *Cruiser* scene models the lower deck of a navy battle cruiser in very fine detail and was created by Saba Rofchaei and Greg Ward.

The *Sodahall* scene was designed by Philippe Bekaert and is a model of a real seven-stories building at the Berkeley University completely modeled in high detail and fully furnished with chairs, books, plants, and even pencils on the desks. This model is highly occluded, where at each location only a small part of the scene is actually visible. This is where the built-in occlusion-culling of ray tracing shows its strengths.

Furthermore *Island* and *Terrain* provide examples of large outdoor scenes with many plants. Especially in *Terrain* highly complex trees and an Eiffel tower are used which cast very detailed shadows.

The plug-and-play concept of ray tracing allows for quickly and easily creating such benchmark scenes: every object is described by its own self-contained shaders 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 the fully automatic generation of 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.

General Remarks on the Measurements

All measurements are performed for single frames only. This means that all time and all computations required to start from scratch until the final image is ready is taken into account. Thus although it has been shown [WBWS01] that there is a significant overhead involved in the start-up phase of a rendering and for waiting until the last pixel is completed no interleaving of frames was used. This simplifies the hardware but also leaves room for easy improvements on the performance.

Furthermore Wald [Wal04] has shown that the quality of kd-trees has significant influence on the performance of the system. Nevertheless, for the first simulations only standard kd-trees were available while later measurements using the improved kd-trees by Wald show performance improvements of up to three times.

However, the focus of this thesis is not on benchmarking kd-trees but evaluating hardware-architectures for ray tracing. Therefore detailed statistics of standard and good kd-trees are presented. These measurements illustrate that good kd-trees not only improve the performance but also reduce the memory requirements for caches due to a lower working set and increased coherence.

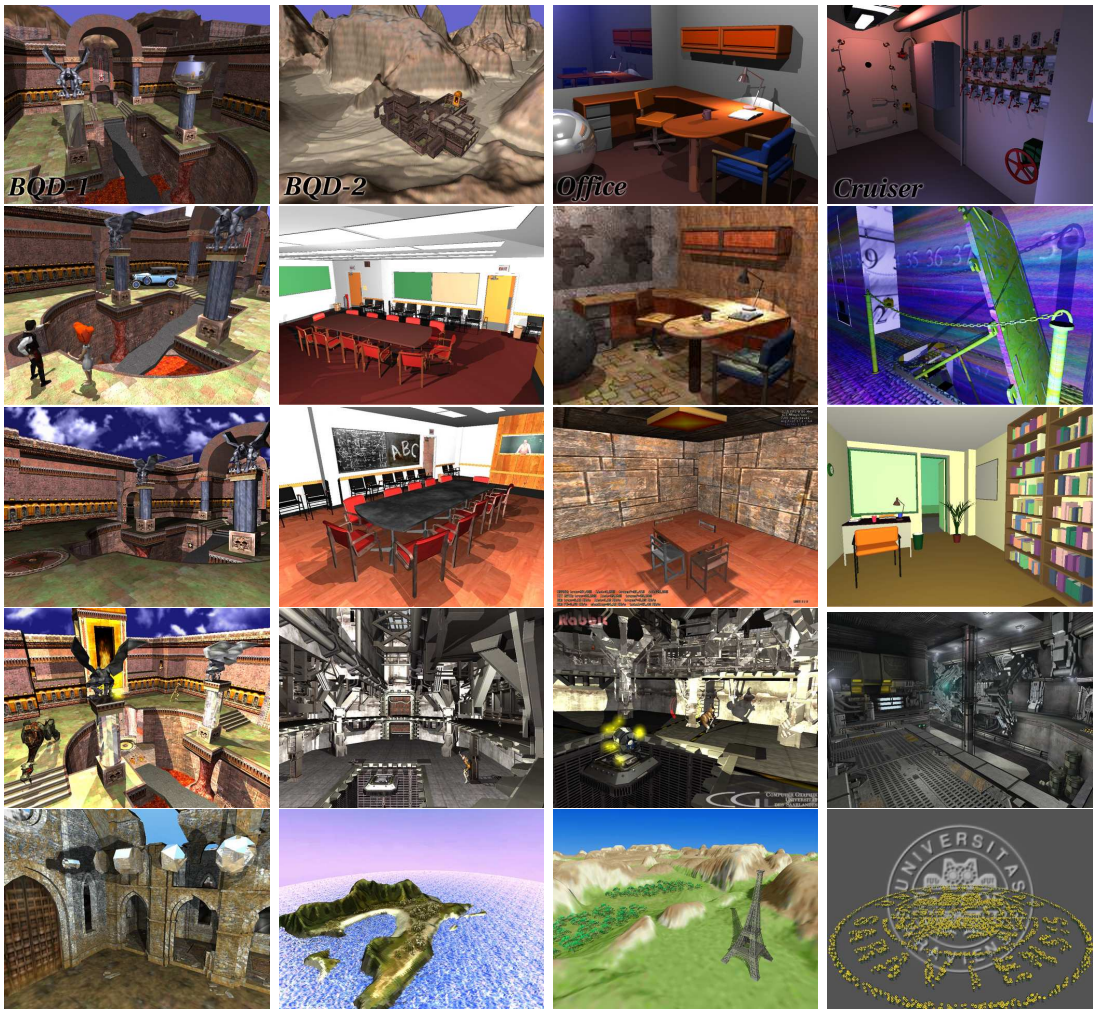


Figure 8.1: Overview of the scenes used for benchmarking: the first row shows two views of the *BQD* scene, *Office* and *Cruiser*. The second row contains *pQuake3*, *Conference*, *jOffice*, *jCruiser* followed by the third row: *Quake3*, *jConference*, *jScene6* and *Sodahall*. The fourth row shows *jQuake3*, *UT2003*, *Rabbit* and *UTID* while the last row presents *Castle*, *Island*, *Terrain* and *SunCOR*. The views shown for *Castle* and *jQuake3* are only to give an impression of the scene and are not identical to the views used in the measurements. *Rabbit* is the first game running on the SaarCOR prototype and is not listed in the measurements of Section 8.5 but videos showing the rabbit in action can be found on the project's homepage `http://www.SaarCOR.de`.

8.1 Static SaarCOR Parameter Set A

The first parameter set of the SaarCOR architecture for ray tracing of static scenes aims at an implementation on ASIC. Therefore the parameters are chosen from typical parameters for ASIC implemented architectures of 2002. At that time especially external memory was limited in speed and thus external memory bandwidth was rather expensive.

A major goal of this parameter set was to show that ray tracing is feasible using moderate hardware resources. Additionally, it should be shown that a reasonable amount of scaling of the performance is easily supported by having multiple ray tracing pipelines on the same chip. However, this support for scaling required large packets of rays for the necessary reduction in bandwidth from each pipeline to the caches (see Section 2.5 and the next section for a detailed analysis).

Therefore the standard SaarCOR-A system (Figure 8.2) for ray tracing of static scenes consists of four RTCs each using 16 threads of packets with 64 rays and four asynchronously decoupled traversal slices and a single intersection unit. Due to historical reasons only standard kd-trees are available for benchmarks and the rays are statically assigned to the slices using the sub-optimal scheduling method C^* shown in Figure 8.3.

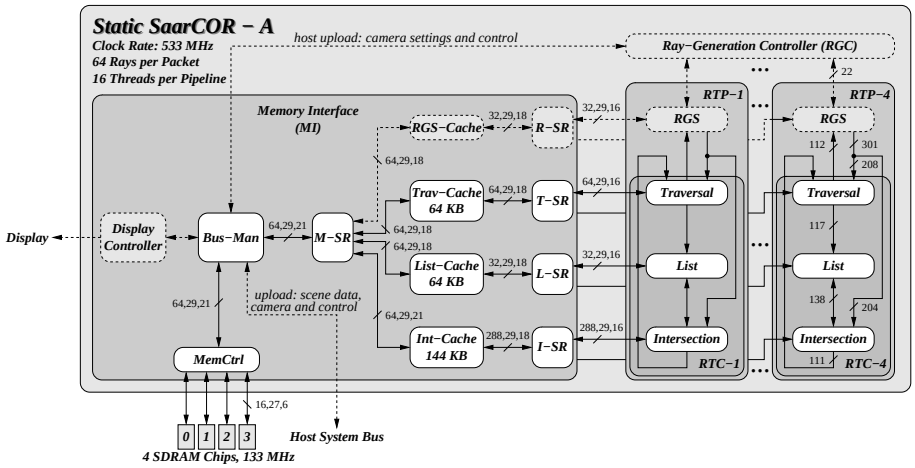


Figure 8.2: Data paths of Static SaarCOR Variant-A. Functional units shown in white are simulated using a cycle accurate model while grey boxes use a behavioral model only.

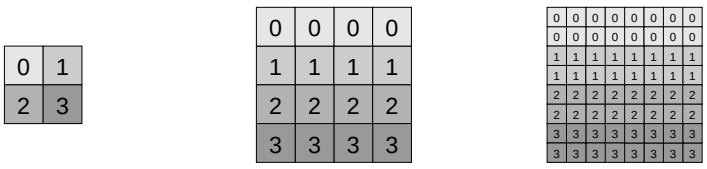


Figure 8.3: Assignment of rays to traversal slices using the sub-optimal method C^* which is almost identical to method C presented in Chapter 4.2.1. Here for packets with 2^n rays, a ray with packet-coordinates (x, y), with $0 \leq x, y < n$ is assigned to traversal slice $t = \frac{x+y \cdot n}{\text{number of slices}}$.

Hardware Complexity

We assume that the chip runs at a clock rate of 533 MHz and is connected to four 16 bit wide SDRAM chips running at 133 MHz (yielding 1 GB/s best-case bandwidth) via three 4-way set associative caches of 272 KB total. This cache is split into 64 KB for the traversal cache, 64 KB for the list cache, and 144 KB for the intersection cache. The simulations of SaarCOR-A do not account for the bandwidth required for shading and assume that the on-board memory is large enough to store the entire scene.

With the standard configuration a SaarCOR-A chip without shading requires a total of 192 streamlined single-function floating-point units, 822 KB for registers-files, and 272 KB for cache, adding up to 1094 KB total on-chip memory. All on-chip memory is split into small local pieces of memory, allowing for simple connections, and a feasible chip design.

For comparison, consumer graphic cards of 2002 required significant compute power and memory bandwidth to achieve their level of performance. For example Nvidia's GeForce3 [Nvi02] offers 76 GFlops at a clock rate of 200 MHz and has a 256 bit wide memory interface running at 230 MHz. These results require at least 380 parallel floating point units and offer a memory bandwidth of 7.2 GB/s, which is several times more than the requirements of SaarCOR-A.

Performance Measurements

Table 8.4 presents the performance of standard SaarCOR-A rendering benchmark scenes using standard kd-trees only. Sections 8.2 and 8.4 will show that improved kd-trees can speed up rendering up to three times while reducing the memory requirements of the caches.

Scene	triangles	lights	recursions	rays shot
Quake3	34 772	0	0	786 432
Sodahall	1 510 322	0	0	786 432
Cruiser-nl	3 637 101	0	0	786 432
Conference	282 000	2	0	2 359 296
Cruiser	3 637 101	2	0	2 359 296
Office	33 952	3	3	3 863 846
BQD-1	2 133 537	1	3	1 583 402
BQD-2	2 133 537	1	3	1 548 632

Table 8.3: The scenes used for benchmarking SaarCOR-A. For every scene the number of triangles, shadow casting light sources, and recursion depth for reflection rays is given. The right most column shows the total number of rays (including shadow and reflection rays) required to render the image in 1024×768. The *Cruiser-nl* scene uses the same geometry as the *Cruiser* scene but no secondary rays for lights.

The performance measurements of Table 8.4 show several interesting points: The performance scales almost linearly with the number of RTCs used and with the number of rays used to calculate the image (see Table 8.3). In comparison to rasterization, where performance degrades linearly with the number of triangles rendered [WBWS01], this

Scene	1 RTC	2 RTCs	4 RTCs	
Quake3	27.20	54.45	111.12	fps
Sodahall	28.88	56.71	113.22	fps
Cruiser-nl	28.58	52.04	65.86	fps
Conference	8.91	16.77	31.56	fps
Cruiser	9.82	17.38	20.05	fps
Office	7.52	14.34	28.56	fps
BQD-1	11.74	23.12	45.90	fps
BQD-2	7.55	12.98	17.43	fps

Table 8.4: Absolute performance measurements for the SaarCOR-A chip with 1, 2 and 4 RTCs, 272 KB cache, and 1 GB/s memory bandwidth. 4 RTCs have only half the floating-point performance of a GeForce3 and there is an almost linear relation between performance and the number of RTCs.

number has only a small impact on the performance for our architecture. However, some figures are not as expected. In particular both *Cruiser* scenes and *BQD-2* show that there must be a bottleneck limiting the performance of the system.

A closer analysis shows, that the *Cruiser* scene with 3.5 million triangles is limited by the memory bandwidth for triangle fetching caused by the standard kd-tree in a complex scene. Table 8.5 gives performance measurements of the *Cruiser-nl* scene for different sized caches for triangles in combination with 1 and 2 GB/s bandwidth to main-memory. This shows that with a bandwidth of 2 GB/s and an cache of 288 KB the performance again scales linearly in the number of RTCs. Achieving linear speed-up with 4 RTCs in *BQD-2* scene is harder: we need to enlarge all caches four times to roughly 1 MB together with a 2 GB/s bandwidth to main-memory.

size of cache for triangles	144 KB	288 KB	576 KB
1 GB/s (4 SDRAMs)	65.86 fps	77.54 fps	86.36 fps
2 GB/s (8 SDRAMs)	87.24 fps	103.62 fps	113.89 fps

Table 8.5: Influence of memory bandwidth and size of the cache for triangles on the scene Cruiser-nl with 4 RTCs. This shows again a linear speed-up with the number of RTCs.

In contrast to these complex models, the *Quake3* scene (where the standard kd-tree performs well) shows perfect linear scaling. Using the standard cache and a bandwidth of only 250 MB/s linear scaling is achieved even up to 16 RTCs. The floating-point performance of the GeForce3 equals the floating-point performance of a full SaarCOR-A chip with 8 RTCs and full shading. Rendering the *Quake3* scene with 8 RTCs achieves 235 fps.

Ideal and Measured Performance

The performance of a chip can be measured in two ways: the absolute and the relative performance. Table 8.4 lists the absolute performance, while Table 8.6 shows the relative performance. The relative performance is defined as the percentage of absolute performance versus ideally achievable performance. The ideally achievable performance is defined as

$$\text{fps}_{ideal} = \frac{\text{chip-speed in cycles per second}}{max\{\frac{\#\text{trav-ops}}{\#\text{RTCs}\times\#\text{trav-sub-units}}, \frac{\#\text{int-ops}}{\#\text{RTCs}}\}}$$

Simply speaking: if there is no overhead at all, we need at least one cycle for every operation we have to perform. If we divide the number of operations by the number of units we obtain the theoretical achievable minimal number of cycles needed.

Table 8.6 shows that even the simple architecture of the standard SaarCOR-A chip already achieves 70%–80% of the ideal performance. Using 32 threads instead of 16 threads per RTC, we increase these results achieving 80%–90% of the ideal performance. On the other hand, using 32 threads instead of 16 increases the on-chip memory from 822 KB to 1050 KB (not counting the caches which keep their size). If we increase the size of the cache, the memory-bandwidth or the number of threads per RTC, these figures can be improved even further. This shows the flexibility of ray tracing and our hardware architecture, which can be scaled over a wide performance range.

Scene	1 RTC	2 RTCs	4 RTCs
Quake3	76%	76%	78%
Sodahall	80%	79%	79%
Cruiser-nl	71%	65%	41%
Conference	67%	63%	59%
Cruiser	72%	63%	37%
Office	71%	68%	68%
BQD-1	73%	72%	71%
BQD-2	54%	46%	31%

Table 8.6: Relative performance: percentage of the theoretically ideal performance achieved with a standard SaarCOR-A chip.

Details on Usage and Hit-Rates

The relative performance, as listed in Table 8.6, equals roughly the usage of the traversal and intersection units. Let c be the number of clock-cycles for rendering an image and w the number of cycles a unit or a bus was busy. We then define the usage as w/c. Figure 8.4 shows several characteristic measurements for a standard SaarCOR-A chip rendering the *BQD-1* scene.

The hit-rates that can be achieved for caching depend to great extends on the size of the working set required to render a frame (see Table 8.7). However, typically much smaller caches already achieve good results since when rendering the lower part of an image the geometry covered by pixels on the top is usually not needed anymore.

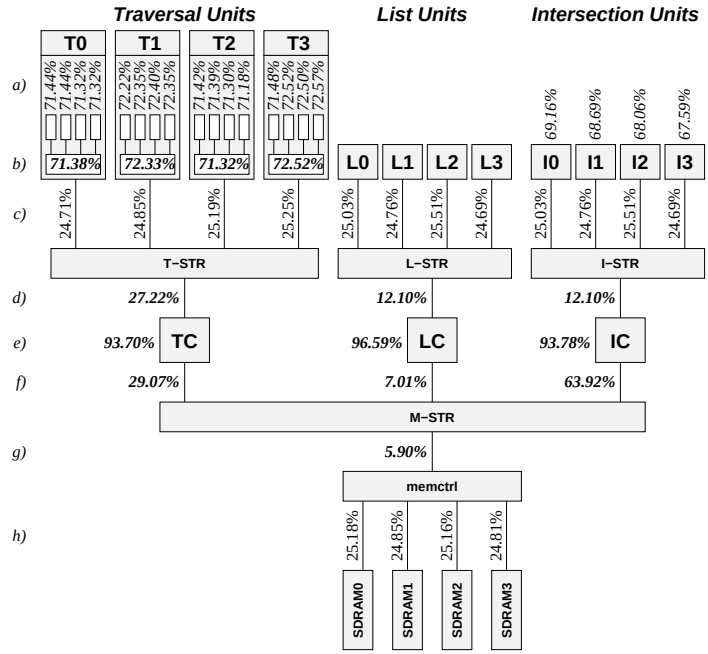


Figure 8.4: Usage and hit-rates of a standard SaarCOR-A chip running the *BQD-1* benchmark: this shows in detail, that trivial static load-balancing works out very well. The characteristics measured are: a) The usage of each of the traversal sub units, b) the usage of each of the traversal and intersection units, c) the percentage a unit contributed to all accesses to the common bus, d) the usage of the bus to the caches, e) the hit-rate of the caches, f) the percentage a cache contributed to all accesses to memory, g) the usage of the bus to the memory controller, and h) the percentage of all accesses to the memory each SDRAM handles. The high amount of traffic to main-memory contributed by the cache for triangles is due to the fact that all accesses to main-memory are only 64 bits wide and therefore each triangle requires 5 consecutive accesses.

Since small caches already perform very well it is not surprising that when rendering the same view multiple times the performance increases only by less than 1% due to data already stored in the caches even when large caches are used. Therefore hardly any performance is lost when initializing the caches in every frame to guarantee that updates to the scene (by the host) are not shadowed by old values stored in the on-chip caches.

	working set for		triangles	
scene	nodes	triangles	working set	% of total
BQD-2	663 KB	1.3 MB	37 558	1.8 %
Cruiser	315 KB	3.9 MB	112 359	3.1 %
SodaHall	230 KB	269 KB	7 651	0.5 %
Quake3	60 KB	122 KB	3 457	10.0 %

Table 8.7: This table lists the working set for rendering a frame in 1024×768 pixels. For optimal caching the working set must fit into the caches without any collisions.

Influence of the Spatial Index Structure

As mentioned in Section 2.5, the depth of the kd-tree can be used to adjust the hardware architecture to any scene and vice versa. Changing the number of scene-space-subdivisions influences the architecture in three ways: As shown in Figure **??**, the number of traversal and intersection operations required to calculate a frame changes, resulting in different frame rates. As the number of subdivisions increases, the memory needed to store all items of the kd-tree grows exponentially. Since the iterative traversal of a kd-tree requires a stack of the size of the maximum depth of the tree, the required on-chip memory increases linearly with the depth of the kd-tree.

The following formula calculates the on-chip memory of a standard SaarCOR-A chip depending on the maximal supported depth d of the kd-tree:

$$\textit{on-chip-memory} = \textit{cache} + 287.6\ \text{KB} + d \cdot 17.25\ \text{KB}$$

Lights, Reflections, and Anti-Aliasing

One of the main advantages of ray tracing is its ability to render physically-correct shadows, reflections, and refractions. The following analyzes the impact of these different types of rays on the overall performance by rendering the *Office* scene in different conditions, as listed in Table 8.8 and shown in Figure 8.5: (a) eye rays (er) only, (b) er and reflections up to 3 levels (r3), (c) er and 3 lights (3l), (d) er, reflections and 3 lights, (e) er with a simple four times oversampling (4×os), i.e. for each pixel, 4 rays are shot and their contribution is averaged to calculate the color of the pixel. Please note that in (b) 20% of all rays are reflected.

Table 8.8 shows that the performance degrades linearly with the number of rays shot, independently of the type of rays. This is also true for refracted rays used to simulate glass-effects (not shown here). Case (e) shows that oversampling is slightly cheaper than linear: 4 times more rays cost only 3.6 times more, due to a better cache hit-rate and increased coherence in each packet of rays. See Table 8.9 for a detailed look on the caches.

It is widely assumed that secondary rays are more costly than primary rays. However, this is not true in general as the cost of rays only depends on the location of the ray in the scene. Therefore secondary rays might be more expensive as well as even cheaper than primary rays. Nevertheless, if rays of various types are traced simultaneously they are more likely to thrash the caches due to the different regions of the scene they are accessing.

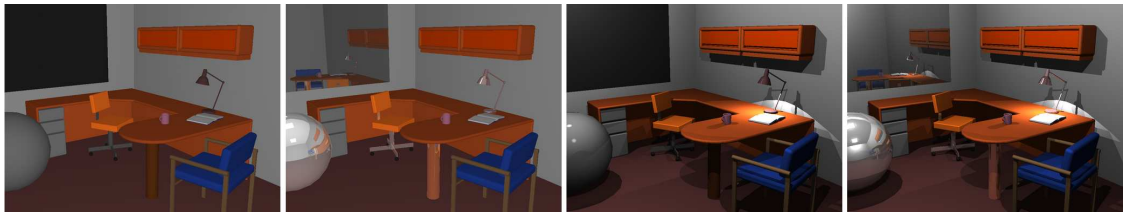


Figure 8.5: The *Office* scene with (from left to right): eye rays only (er); er and reflections; er and three point lights; er, reflections, and three point lights.

	#rays	$\frac{\#rays(er)}{\#rays}$	FPS	$\frac{fps}{fps(er)}$
(a) er	786 432	100%	127.75	100%
(b) er,r3	966 275	81%	99.23	78%
(c) er,3l	3 145 728	25%	36.67	29%
(d) er,r3,3l	3 863 846	20%	28.56	22%
(e) er,4×os.	3 145 728	25%	35.06	27%

Table 8.8: The *Office* scene rendered with different types of rays. This shows that the performance is very close to linear in the number of rays shot and almost independent of the type of the ray.

hit-rate of	oversampling	
cache for	none	4-times
nodes	89.9%	96.8%
lists	89.7%	95.7%
triangles	97.1%	98.8%

Table 8.9: The cache hit-rate increases for four-times oversampling giving a 10% performance improvement over the expected cost of anti-aliasing.

8.2 Static SaarCOR Parameter Set B

This section contains new measurements using a parameter set that fills the gap between SaarCOR-A, which was designed for a scalable ASIC implementation with low external memory bandwidth and the SaarCOR prototype (see Section 8.5), which implements a single ray tracing pipeline on a FPGA where a relatively high external memory bandwidth is available.

The SaarCOR-A variant uses large packets of 64 rays and large shared caches to allow for linear scaling of the performance by having up to 16 pipelines on the same chip even when using only standard kd-trees (Section 8.1). In contrast the SaarCOR prototype (Section 8.5) achieves very good results when rendering scenes with good kd-trees using only small packets of four rays together with small caches that are replicated per pipeline.

In principle, the performance of a ray tracing chip can be scaled by adding more ray tracing pipelines as long as the memory interface can provide the required bandwidth. Then further scaling is only possible by reducing the bandwidth requirements using dedicated caches on each pipeline or with large enough packets.

With standard kd-trees (which where the only kd-trees available for most benchmarks) caches have to be rather large (see the previous section) and therefore replication of caches for each pipeline is very costly. This leaves the use of large packets as the only option for scaling as large packets allow for sharing a single cache between multiple pipelines. This is the reason why all variants of SaarCOR that have been simulated with ASIC-like constraints and support for scaling inside a single chip used an architectecture of shared caches and large packets of rays.

Parameter Set B

The influence of a good kd-tree is evaluated in this section on an architecture identical to SaarCOR-A except that also shading is simulated. However, the results do not contain the bandwidth used for shading since the focus of this section is on the RTC. See Figure 8.6 for the data paths of parameter set B.

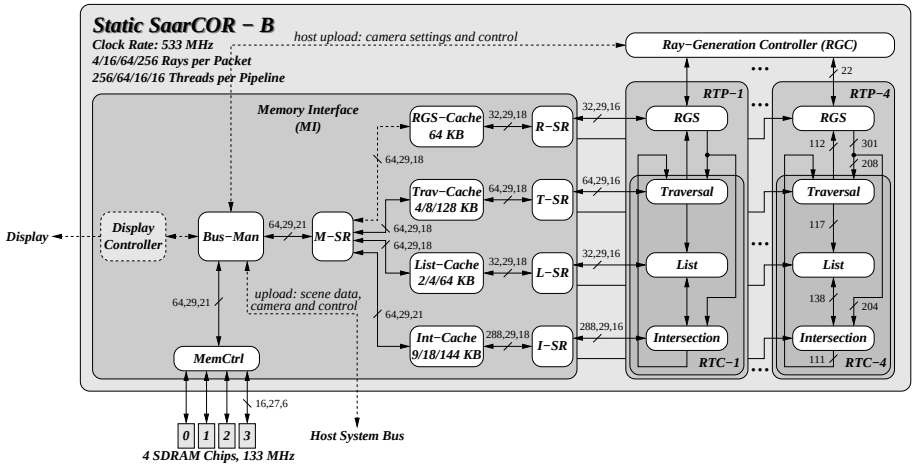


Figure 8.6: Data paths of Static SaarCOR Variant B. Functional units shown in white are simulated using a cycle accurate model while grey boxes (the Display Controller and the SDRAM chips) use a behavioral model only.

In all measurements per pipeline there were always 1024 rays in flight simultaneously. Thus if r is the number of rays per packets and t the number of threads per pipeline than $r \cdot t = 1024$. The only exception is that 16 threads per pipeline have been used for packets of 256 rays. This was necessary since in the current architecture at any time a packet of rays can only be processed by a single functional unit regardless of the size of the packet and using only four threads per pipeline does not suffice to keep all functional units busy.

Performance Measurements

Table 8.10 shows for several configurations of rays per packet and sizes of the caches the relative speed-up gained by scaling the number of ray tracing pipelines. For a single pipeline the absolute performance achieved is roughly the same for all configurations and is within a 1% window around $37fps$[1]. The reason is simple as even for the worst cases the bandwidth to memory is less than 1 GB/s and therefore even bad caching has no influence on the performance as multi-threading suffices to efficiently hide also the larger memory access latencies.

[1] The current implementation of the management of the asynchronously running traversal slices causes an overhead for equal numbers of traversal slices and rays per packet. Thus although this overhead actually reduces the absolute performance it virtually reduces the latency for memory accesses of the traversal unit and therefore speeds up the relative performance for increasing numbers of pipelines sharing the same cache. Nevertheless this overhead does not affect the measurement for bandwidth per frame and cache hit-rates and could be removed easily by using synchronous SIMD-like traversal slices.

The absolute performance is roughly 93% of the ideal performance and corresponds approximately to the usage of the traversal unit. The usage of the intersection unit is only 41% since the the hardware is build using a ti-ratio of 4:1 but the specific kd-tree uses a ratio of 9:1.

Size of Packets	Relative Speed-Up (128,64,144) 1→2	(128,64,144) 1→4	(8,4,18) 1→2	(8,4,18) 1→4	(4,2,9) 1→2	(4,2,9) 1→4	(Trav,List,Int) Cache in KB # Pipes
4 rays*	2.00	3.87	2.00	2.99	1.98	2.72	
16 rays	2.00	3.81	1.68	2.02	1.56	1.80	
64 rays	1.99	3.98	1.97	3.49	1.95	3.18	
256 rays	1.99	3.98	1.99	3.88	1.99	3.75	

Table 8.10: The relative speed-up gained by increasing the number of ray tracing pipelines depending on the number of rays per packet and the size of the cache is listed for scene *UTID* (1024×768, primary rays only). Please note that the relative performance for four rays per packet scales too good[1].

Hit-Rates and Bandwidth

A look at the measurements on the bandwidth (Table 8.11) shows that using 4 rays per packet and a minimalistic cache requires roughly the same bandwidth as using 64 rays per packet and no cache at all. Nevertheless, using 64 rays per packet and a minimalistic cache reduces the bandwidth to only a fourth.

This is somewhat confusing since Table 8.12 shows that the hit-rate goes down with increasing numbers of rays per packet. But the reason can be found in the definition of "hit-rate", which simply specifies the *ratio* between the number of memory accesses that could be served using data stored in the cache and the total number of memory accesses.

Using larger packets might reduce the number of cache hits (since the working set of the packet increases and the coherence between packets decreases) but it definitively reduces the total number of accesses by great amounts (see Section 2.5) allowing for simple connection sharing and low external bandwidth requirements. Thus it is no contradiction that higher hit-rates might also have higher bandwidth requirements[2].

Table 8.12 shows another interesting fact: Since the working set for triangles is much larger than for nodes a cache for triangles has to be 36-times larger than the cache for nodes to achieve roughly equal hit-rates. This is very interesting since the data structure for triangles is only 5-times larger and the bandwidth to the cache is almost equal (per frame 7.4 MB for nodes and 10.6 MB for triangles). Therefore if only little memory can be spent on caches it always pays out to trade intersection operations for more traversal operations as this reduces the memory bandwidth.

The measurements presented here did not directly compare the caching strategies used (e.g. 4-way set associativity used in SaarCOR-A to D and direct mapped caches used in the SaarCOR prototype). However, for completeness Appendix E presents additional and fairly detailed measurements on caching and memory bandwidth. It shows that

[2]Simply speaking (and exaggerating): a few percent of very much is still much but a high percentage of nothing stays nothing.

in general a direct mapped cache achieves roughly the same hit-rates as a four-way set associative cache. This is simply due the fact that ray tracing has an almost random memory access behavior and therefore collisions in the cache are neglectible.

Size of Packets	Bandwidth per Frame			
	to cache	to memory after cache with size (trav,list,int)		
		(4,2,9 KB)	(8,4,18 KB)	(128,64,144 KB)
4 rays	191.32 MB	24.10 MB	17.45 MB	3.73 MB
16 rays	59.55 MB	12.70 MB	8.72 MB	2.71 MB
64 rays	22.12 MB	5.90 MB	4.63 MB	2.32 MB
256 rays	10.53 MB	4.61 MB	3.75 MB	2.18 MB

Table 8.11: This table lists the bandwidth per frame for different sizes of packets and caches on a single pipeline in scene *UTID* (1024×768, primary rays only). Please note that these measurements differ from those given in Table 2.5 since the simulation also includes the clipping kd-tree. Furthermore memory accesses on a 64 bit wide memory bus are correctly simulated which leads to an overhead for triangles that actually contain 36 bytes but are mapped to 5 sequential accesses and thus 40 bytes per triangle.

Size of Packets	Hit-rate of the caches depending on their sizes								
	trav	list	int	trav	list	int	trav	list	int
	4 KB	2 KB	9 KB	8 KB	4 KB	18 KB	128 KB	64 KB	144 KB
4 rays	96.1%	95.2%	72.8%	97.8%	97.3%	79.3%	99.7%	98.9%	95.3%
16 rays	92.4%	92.7%	61.1%	95.5%	95.0%	72.5%	99.0%	97.3%	91.2%
64 rays	88.7%	89.8%	59.7%	91.4%	91.4%	68.3%	96.7%	94.3%	83.7%
256 rays	77.0%	86.0%	56.5%	80.8%	86.8%	62.7%	89.8%	90.1%	73.0%

Table 8.12: The hit-rates of the caches of the RTC depending on their sizes and the number of rays per packet on a single pipeline in scene *UTID* (1024×768, primary rays only). It shows that already rather small caches achieve very good hit-rates.

8.3 Static SaarCOR Parameter Set C

The measurements of the previous sections presented detailed analyzes of ray tracing of static scenes but did not include statistics for shading and memory management. This section presents measurements of SaarCOR parameter set C, which includes all details of ray tracing of static scenes with fixed function shading and support for virtual memory management.

The architectural parameters of SaarCOR-C (see Figure 8.7) are almost identical to the previous variants. The main differences are that also a fixed function shading model is fully simulated including bilinear texture-filtering and a virtual memory architecture is added reading on-demand pages from host's memory when required during rendering.

To account for the bandwidth requirements of shading also the available bandwidth was increased.

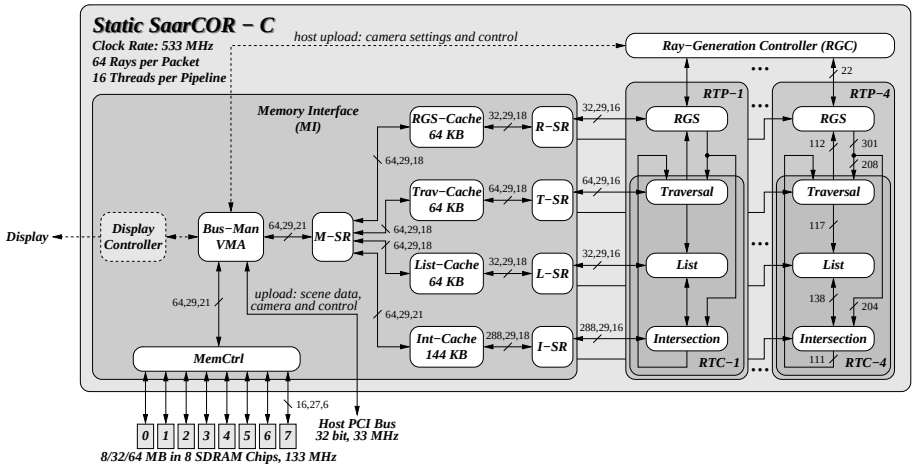


Figure 8.7: Data paths of Static SaarCOR Variant C+D. Functional units shown in white are simulated using a cycle accurate model while grey boxes use a behavioral model only.

Thus SaarCOR-C is a scalable ASIC parameter set, uses 4 pipelines with 16 threads each and a core frequency of 533 MHz. It is connected to 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.

Section 6.2.1 described in detail the virtual memory architecture. There it was shown that the meta data required to manage the virtual memory becomes rather large even for small amounts of cache memory and a large page size. Therefore besides the standard *version A* that stores the whole set of meta data in on-chip memory a second *version B* was implemented that uses only a small cache of 64 KB to cache the meta data stored in external DRAM memory.

Similar to the measurements in the previous sections the simulations of SaarCOR-C are performed on the register transfer level (see Section 7.1.2) and 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 long walk-throughs of a several benchmark scenes of which images and videos can be found at

`http://www.SaarCOR.de/VMA4RT`

Two Step Approach

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 8.8). In the second step we used the cycle accurate simulator to simulate in detail how the SaarCOR-C 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.

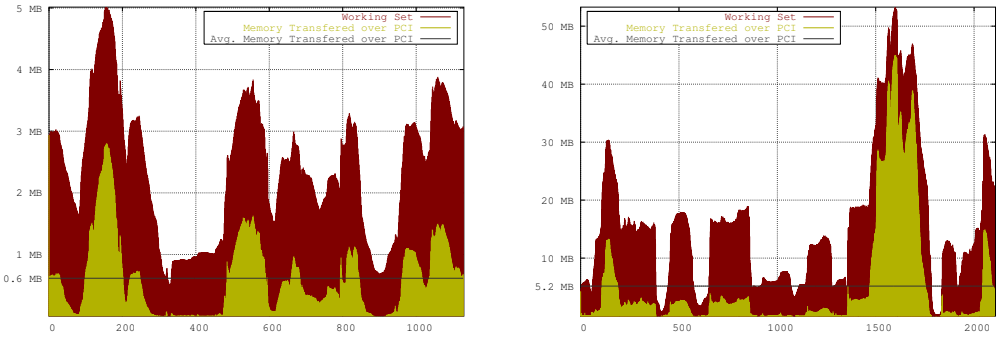


Figure 8.8: Results of the first simulation step for the *pQuake3* scene (left, with textures and light) using 8 MB card memory and the cruiser scene (right, with textures and light) with 64 MB. For each frame of the sequence it plots the size of the working set and the amount of memory transfered.

Performance Measurements

Most of our benchmark scenes (Table 8.13) are similar to the ones used in Section 8.1 where it was shown that 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.

Figure 8.8 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, 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 8.14 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 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-C architecture contains several independent threads running

Scene	triangles	lights	rays shot	size on hard-disk geometry	textures
pQuake3-nlnt	46 356	0	786 432	16 MB	—
Conference-nl	282 000	0	786 432	88 MB	—
Sodahall	1 510 322	0	786 432	429 MB	—
pQuake3-nt	46 356	1	1 572 864	16 MB	—
Conference	282 000	2	2 359 296	88 MB	—
Conference-8l	282 000	8	7 077 888	88 MB	—
pQuake3-nl	46 356	0	786 432	16 MB	28 MB
pQuake3	46 356	1	1 572 864	16 MB	28 MB
Cruiser	3 637 101	1	1 572 864	540 MB	340 MB

Table 8.13: The scenes and their parameters as used for benchmarking (suffix *nl*: without lighting, suffix *nt*: without textures). All scenes were rendered in 1024×768 using the given number of shadow casting light sources and thus requiring the listed numbers of rays to be shot. The names of the scenes are abbreviated using pQ for *pQuake3*, Conf for *Conference*, Soha for *Sodahall* and Cru for the *Cruiser* scene.

in parallel some small, non-obvious variations can be seen in the results due to scheduling issues (e.g. *pQuake3-nlnt*, frame 1046).

Results

It 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 900 MB of data due to highly localized complex geometry and huge textures and bump-maps[3]. Section 8.1 has 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 8.14 were taken for the worst-case hot-spots of Figure 8.8. It showed that the performance of the cruiser scene is limited by the bandwidth to local memory (the L2-cache) due to the large working sets for triangles.

[3]The textures and bump-maps are not included in the original model and were added for testing purposes only.

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 were not yet implemented for the hardware simulations in this section. However, the next section shows measurements of good kd-trees including full simulation of the VMA.

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

Table 8.14: 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-C and the on-board memory are listed. All memory transfers are measured in MB per frame. All measurements are performed with a standard SaarCOR-C-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.

8.4 Static SaarCOR Parameter Set D

The previous section has shown that fully automatic virtual memory management performs very well when using standard kd-trees for benchmark scenes. However, while using the new algorithms for building kd-trees allows for reducing the required number of traversal and intersection steps and thus improving the performance it also changes the memory access behavior. This section evaluates the influence of good kd-trees on the performance with and without support for virtual memory.

Thus SaarCOR parameter set D is identical to SaarCOR parameter set C and includes support for static scenes, fixed function shading and virtual memory management. The only difference is in the benchmark scenes used, which only use good kd-tree (for details on the kd-trees see Section 2.1.1). Table 8.15 provides data about the scenes used to benchmark the SaarCOR-D.

Scene	triangles	lights	recursions	rays shot	textures
Cruiser	3 637 101	0	0	786 432	–
BQD-1	2 133 537	0	0	786 432	–
BQD-2	2 133 537	0	0	786 432	–
Quake3	39 424	0	0	786 432	bilinear
Conference	282 000	2	0	2 359 296	–
Office	33 952	3	3	3 863 846	–

Table 8.15: The scenes used for benchmarking SaarCOR-D. For every scene the number of triangles, shadow casting light sources and recursion depth for reflection rays is given. Additionally, the total number of rays required to calculate the image and the type of textures (if any) used for shading is shown. The image resolution was 1024×768.

Performance Estimates

This selection of benchmark scenes is motivated by the following results: For a fixed scene the performance of a ray tracing system scales linearly with the number of rays used, but is mostly independent on the type of the ray (see Section 8.1). Thus given the performance p of a system and a scene S with no light and no reflections it is possible to estimate the performance p' for the same scene with n lights and where r % of all primary rays are reflected as

$$p'(S) = \frac{p(S)}{\frac{100+r}{100} \cdot (1+n)}$$

The benchmark scenes listed in Table 8.15 require up to 500 MB of storage on hard-disk. These requirements are far less than for the same scenes using a standard kd-tree. A standard kd-tree simply performs worse on splitting the volume of a scene and therefore requires much more nodes and replicated lists of triangles. Thus the improved version does not only speed-up ray tracing by reducing the number of traversal and intersection operations but also improves the caching behavior due to a smaller working set and reduces the storage on hard-disk.

But despite these storage requirements the scenes can be rendered with only 64 MB on-board memory. For most scenes even as little as 8 MB are sufficient. However, in order to simplify measurements, all scenes were rendered using 64 MB of local memory.

Performance Measurements

Table 8.16 presents the performance measurements of SaarCOR parameter set D. Comparing these results to those of SaarCOR-A in Table 8.4 shows that although SaarCOR-D also includes full simulation of shading and memory management a speed-up of two to three times is gained when using good kd-trees instead of standard kd-trees.

These results are already rather good although the kd-trees used were optimized for the software ray tracer on a Intel Pentium-IV (which has quite different hardware parameters than the SaarCOR architecture). Thus it should be possible to increase the performance even more by building more suitable kd-trees (see Section 2.5).

The side effect of using kd-trees that were optimized for a different architecture can be seen by analyzing the results listed in Table 8.16. Unbalanced workloads are likely to show higher variations in performance if architectural parameters are changed. For instance the BQD-2 scene utilizes ten times as many traversal than intersection operations. However, the architecture is designed for a 4:1 ratio.

The improved kd-trees also reduce the working set and increase the hit-rate in the various caches — especially in more complex scenes. The performance measurements on the Cruiser scene in the previous sections showed that the working set on triangles was too large such that a triangle-cache with 576 KB became necessary. The improved kd-tree reduces the working set such that the scene can be efficiently rendered using only the same small cache (144 KB) as for all other scenes.

	std. SaarCOR-D		SaarCOR-D with VMA		
Scene	fps	mem	fps	mem	PCI
Cruiser	170	6.1	121	7.7	0.14
BQD-1	137	1.9	135	2.5	0.03
BQD-2	59	26.6	42	34.1	0.91
Quake3	129	9.4	126	11.4	0.01
Conference	77	8.5	68	10.8	0.09
Office	44	2.1	43	2.6	0.02

Table 8.16: Performance measurements of a standard SaarCOR-D chip with large enough on-board memory to store the entire scene and using only 64 MB on-board memory but also VMA. Columns labeled with *fps* state the performance measured in frames per second, while columns labeled with *mem* list the amount of off-chip memory transfers per frame in MB. The column labeled *PCI* shows the memory transfer over the PCI bus in MB.

8.5 Dynamic SaarCOR Prototype

The previous sections have shown detailed measurements on SaarCOR variants designed for an implementation on ASIC. In this section the first prototype implementation of the SaarCOR architecture is evaluated. In contrast to the variants presented in the previous sections the prototype is implemented using FPGA technology and therefore some of its characteristics in hardware differ.

The prototype (see Section 7.3) fully implements a whole dynamic ray tracing pipeline including fixed function shading on a single FPGA and supports both ray tracing of static and dynamic scenes. It is build using a commercial prototyping board (see Section 7.3) containing a FPGA running at 90 MHz and SRAM memory, which delivers 1 GB/s for geometry data and 340 MB/s for shading.

The FPGA runs at only a sixths of the clock speed as for the ASIC variants was assumed and its memory connection is only shared by the functional units of a single pipeline. However, the memory bandwidth available for geometry data is the same as for SaarCOR variants A to D.

Thus the prototype has a much higher bandwidth to memory available and therefore can tolerate lower reductions in the bandwidth requirements due to smaller packets of rays and worse caching. This is important since the most restricting resource in the prototype is the on-chip memory and therefore large packets of rays and large caches are not possible. Fortunately as shown in Section 8.2 when using good kd-trees even small caches suffice for rendering scenes using small packets of rays.

The SaarCOR prototype for ray tracing of dynamic scenes was described in Section 7.3 and its data path are shown in Figure 8.9. It uses very small packets of four rays and implements a single ray tracing pipeline with four traversal slices and one transformation unit running at 90 MHz using standard FPGA technology of 2003. The memory accesses of the RTC are cached using very small direct-mapped caches of 4, 2, and 6 KB for traversal, list and transformation matrixes respectively. For shading no caches are used since the bandwidth of 340 MB/s greatly suffices.

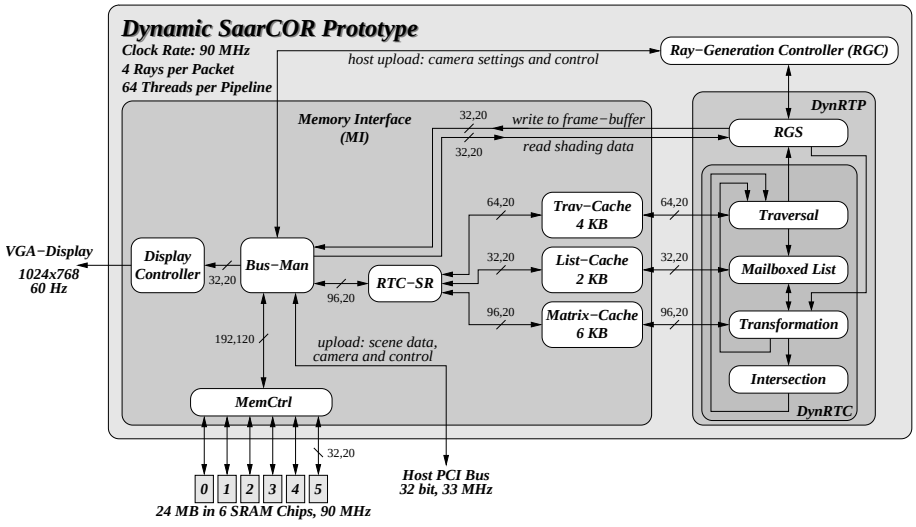


Figure 8.9: Data paths of Dynamic SaarCOR Prototype. In contrast to the *static SaarCOR* variants, the list unit implements mailboxing to avoid multiple intersection of objects and triangles.

Performance Measurements and Comparisons

Table 8.17 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 delivers 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
jScene6	806	1	60.8	12.9	4.7
Castle	20 891	8	23.8	9.2	2.6
jOffice	34 312	1	48.9	10.4	4.7
Quake3	39 424	1	33.6	11.1	3.0
jQuake3	52 790	17	26.7	7.9	3.4
UT2003	52 479	1	25.4	8.0	3.2
jConference	282 805	54	22.1	8.1	2.7
Island	1 409 338	621	15.4	4.5	3.4
Terrain	10 469 866	264	15.9	3.5	4.5
SunCOR	187 145 136	5622	35.8	7.5	4.8

Table 8.17: 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, *jScene6*, *jOffice*, and *SunCOR* were rendered with sample nearest texture filtering since otherwise the shading bandwidth would slightly limit the performance (for details see below: "*Impact of Shading on the Performance*").

Caching and Memory Bandwidth

Table 8.18 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. 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.

		usage rate [%]				hit-rate [%]			bandwidth [MB/s]		
Scene	fps	T	L	M	I	T	L	M	RTC	Shad.	Total
jScene6	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
jOffice	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
jQuake3	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
jConference	5.7	89	25	51	28	98	63	78	135	132	267
Island	4.2	80	31	42	22	96	38	49	290	59	349
Terrain	4.2	80	27	34	18	97	27	43	283	98	381
SunCOR	10.1	54	42	34	14	90	2	6	603	122	747

Table 8.18: 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 for the DynRTC and shading (excluding frame buffer readout for display). Here columns labeled T, L, M, and I belong to the traversal, list, transformation, and intersection unit respectively. It shows that multi-threading allows for efficiently keeping 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, *jScene6*, *jOffice* and *SunCOR* were rendered with sample nearest texture filtering since otherwise the shading bandwidth would slightly limit the performance (see below for details).

Impact of Shading on the Performance

Table 8.19 lists for all benchmark scenes the performance achieved depending on the shading frequency (see Section 7.3). Here the shading frequency denotes the number of cycles that have to pass before a new ray can be given as input to the shader. However, the latency before a thread is fully shaded and a new packet is sent to the RTC is much higher (without counting any wait-states or stalls the latency is up to 91 cycles depending on the shading mode).

Due to the input frequency and the various latencies it is possible that finished rays are waiting to be shaded. In this case the number of active threads decreases which can result in too few threads being active to achieve optimal performance. Examples are scenes which have very short computations in the RTC due to a low scene complexity (e.g. *jScene6*) or many rays that do not intersect any geometry and therefore are much cheaper to compute (e.g. *SunCOR*).

However, except for those extreme cases it shows that the performance of the system is not affected by shading. This is due to the fact that multi-threading works very well and efficiently hides all latencies resulting from memory accesses, computational dependencies in the RTC, and from shading calculations.

	Shading frequency [*cycles/ray*]			Loss in performance	
Scene	1	4	8	$1 \rightarrow 4$	$4 \rightarrow 8$
jScene6	15.31*fps*	15.31*fps*	13.54*fps*	0%	12%
Castle	5.95*fps*	5.94*fps*	5.94*fps*	0%	0%
jOffice	12.40*fps*	12.39*fps*	11.86*fps*	0%	4%
Quake3	8.54*fps*	8.54*fps*	8.49*fps*	0%	1%
jQuake3	6.78*fps*	6.78*fps*	6.77*fps*	0%	0%
UT2003	6.52*fps*	6.52*fps*	6.52*fps*	0%	0%
jConference	5.69*fps*	5.69*fps*	5.69*fps*	0%	0%
Terrain	4.20*fps*	4.20*fps*	4.19*fps*	0%	0%
SunCOR	10.28*fps*	10.14*fps*	8.69*fps*	1%	14%

Table 8.19: This table compares the impact of the shading performance on the performance of the SaarCOR prototype running at 90 MHz and with a resolution of 1024×768 pixels using primary rays only. It shows that multi-threading works very well and therefore shading has no influence on the performance. However, there are exceptions for scenes (e.g. *jScene6* and *jOffice* for their simplicity, and *SunCOR* for the number of ray that do not intersect any geometry), which require only rather few computations in the RTC. In these cases the number of active threads is reduced since compared to tracing rays, shading is too slow for these rays, which leads to wait-states for finished rays.

Analysis and Results

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 Section 8.1 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 some type of 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 is constant compared to rendering the scene without shadow rays.

Many images, videos and further measurements can be found on the project's web page
`http://www.SaarCOR.de/DynRT`

8.6 Summary

This chapter has presented several measurements and statistics and provided a deep insight into characteristics of the various variants of the SaarCOR architecture. As a quick reference Table 8.20 summarizes the results and in which section the corresponding details can be found.

Result	Section
high scalability of the architecture	8.1
low external bandwidth to memory	8.1, 8.3, 8.5
high cache hit-rates even for small caches	8.1, 8.5
performance independent on type of ray	8.1
anti-aliasing improves rays per second performance	8.1
comparable costs allow for same performance as rasterization	8.1
fully automatic virtual memory management performs very well	8.3, 8.4
simulations are verified by prototype	8.5
SaarCOR more efficient in ray tracing than GPU and CPU	8.5
multi-threading allows for very good usage of functional units	8.1, 8.5
multi-threading hides latencies of memory, ray tracing, and shading	8.1, 8.5
packets of rays drastically reduce the bandwidth to memory	8.1, 8.2
packets of rays allow for shared connections with many units	8.1
caching works very well even for large packets of rays	8.1, 8.2
direct mapped and four-ways associative caches have equal rates	8.2
frame-to-frame coherence is very low for standard caches	8.1
working set of scenes is rather low compared to size of scene	8.1
simple statical load-balancing performs very well	8.1 to 8.5

Table 8.20: The most important results derived from the measurements on the various architectural variants of SaarCOR.

These results clearly show the many advantages of ray tracing with its *high scalability*, *low bandwidth to memory*, and *very efficient implementation* that allows for a *good and output-sensitive performance*. However, even these very good results leave room for improvements since many extensions allowing for higher frame-rates and even more efficient designs have not yet been implemented in the SaarCOR architecture.

Nevertheless the SaarCOR prototype verified the simulations and showed that the hardware requirements of ray tracing are not higher than those of rasterization based graphics. Therefore it is technically feasible to build a ray tracing system capable of rendering complex sceneries in full screen resolution at real time performance.

9 Conclusion

"The difficulty lies, not in the new ideas, but in escaping the old ones, which ramify, for those brought up as most of us have been, into every corner of our minds."
John Maynard Keynes [Key35]

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.

This thesis has presented a full featured hardware architecture for realtime ray tracing. The architecture has been evaluated in great detail using simulations and in a prototype implementation.

The fast and cycle accurate simulations have been made possible by a new system for simulations of highly parameterized architectures that has been developed as part of this thesis. This system further reduces the efforts of gate-level implementations and re-implementations during the design process of a hardware architecture.

The results of these evaluations have been used to implement what we believe to be the first prototype for 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 (using technology of 2003) already achieves realtime performance for a wide variety of scenes and is several times faster than any CPU of 2004, although these CPUs are clocked more than 30-times faster.

Expoiting the Coherence

The hardware architecture proposed uses packets of rays to efficiently exploit coherence in ray tracing. It was shown that using packets of rays reduces the bandwidth to the caches allowing for sharing the memory interface between several ray tracing pipelines. Additionally, packets of rays reduce the external memory bandwidth and even the on-chip memory requirements.

However, packets of rays can also introduce several overheads and computational dependencies which need to be addressed carefully. Therefore algorithms have been evaluated and adapted to use even large packets of rays efficiently. These adaptations and improvements are especially important for shading packets of rays when using many different shaders like it is done in the movie industry [Bjo04].

Overcoming Bottlenecks

There are two main limitations of current rasterization hardware that ray tracing allows for overcoming: The external memory bandwidth requirements of ray tracing are only a

fraction compared to rasterization, where bandwidth has been a major limiting factor. Furthermore ray tracing offers efficient scalability over a wide range by simply 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 work performed to only the relevant parts of the data.

We have shown that this key feature of ray tracing allows for an fully automatic and highly efficient memory and scene management. Using these techniques scenes many times larger than the available on-board memory can be rendered at hardly any impact on the performance. These results invalidate the common assumption that ray tracing is impractical because it would need to store the entire scene in local memory.

Future Work

All of this has been shown using a rather simple approach with only static load balancing, trivial routing, low memory bandwidth, simple memory technology, and small caches. This is promising as it leaves many opportunities for later optimizations and extensions.

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 covers the majority of cases, and free-form surfaces and vertex shading add further flexibility still there is much room for improvements for dynamically changing scenes.

The Future is Ray Tracing

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.

In summary, it seems that the old dream of real-time ray tracing is finally realizable at hardware costs similar to existing graphics systems. This would enable the display of highly realistic, physically correct, and accurately lit interactive 3D environments. Because ray tracing is at the core of any algorithm computing light transport, fast ray tracing is likely to also enable real-time global illumination computations and other advanced optical effects.

A Notation

The notations used in this thesis follow general standards. Nevertheless this section presents a summary.

Variables are denoted by *emphasized* letters, where lower case letters specify variables of scalar type and capital letters denote matrices in $\Re^{n,m}$ with vectors and points in $\Re^3$ as a special case with $n = 3, m = 1$. Program code and pseudo code is written in `typeface` letters. Functions are presented in *CAPITAL SLANTED* letters. The most important functions used in this thesis are listed in Table A.1.

The data structure for every number stored in computer memory is a *bit string*. We denote the binary representations of integer and floating-point numbers x by a string $< x >$ starting with the *most significant bit* (MSB) on the left and ending with the *least significant bit* (LSB) on the right. The LSB has the index 0 while the MSB has index $n - 1$ in a bit-string with length $| < x > | = n$.

There is also a full set of functions to interpret the bit-strings in various number formats. For example if x is a floating-point number with $s =< x >$ then $FLOAT(0,s[n-2:0])$ gives the absolute value of x, that means with the sign bit set to positive.

$/$	boolean algebra: unary *NOT*
$\wedge$	boolean algebra: binary *AND*
$\vee$	boolean algebra: binary *OR*
$\otimes$	boolean algebra: binary *XOR*
$STRING(x)$	returns the bit-string $\{0\|1\}^+$ of the binary representation of x
$< x >$	abbreviation for $STRING(x)$
$LENGTH(s)$	returns the length of s, i.e. number of letters in string s
$\|s\|$	abbreviation for $LENGTH(s)$
$BIT(b,s)$	returns bit b of s, i.e. the letter at position b in string s
$s[b]$	abbreviation for $BIT(b,x)$
$BIT(a,b,s)$	returns the bit-string from bit a to b of s
$s[a{:}b]$	abbreviation for $BIT(a,b,s)$
$CONC(s,z)$	returns the concatenation of the strings s and z
$< s, z >$	abbreviation for $CONC(s,z)$
0^n, 1^n	abbreviation for string $0\cdots0$ respectively $1\cdots1$ with length n
$INT_{us}(s)$	returns unsigned integer value of s with $n = \|s\|$: $\sum_{i=0}^{n-1} 2^i \cdot s[i]$
$INT_{2C}(s)$	returns two's complement value of s: $-2^{n\text{-}1} \cdot s[n\text{-}1] + \sum_{i=0}^{n\text{-}2} 2^i \cdot s[i]$
$FLOAT(s)$	returns single precision IEEE floating-point number represented by s
$SIGN(x)$	returns the sign of x, with $\begin{cases} 0 \text{ if } x \geq 0 \\ 1 \text{ if } x < 0 \end{cases}$
	if x is a single precision IEEE fp-number then $SIGN(x) = INT_{us}(x[31])$

Table A.1: Notation of the functions used in this thesis. Here a, b, n denote integer numbers, x, y denote integer or floating-point numbers, and $s, z \subset \{0|1\}^*$ are bit strings. There is a common abbreviation that similar to the programming language C++ where functions can be overloaded allows for writing a number x where actually a string-type argument s is needed. In this case not x is used but $s =< x >$. Although this looks confusing typically it is clear whether x or $< x >$ is meant and thus helps to improve readability and shortens examples.

B Implementation Details on Bit-Vectors

The pseudo-code of the ray tracer presented in Section 2.3.3 uses bit-vectors to store the activity, termination and other states of packets of rays. Besides trivial operations such as bit-wise *AND*, also gather operations (e.g. *OR*`(active-vector[n:0])`) and enumerations of all active elements (e.g. `for i = all active rays in active-vector[n:0]`) are performed on bit-vectors. Therefore in the following two different implementations of bit-vectors and their costs regarding storage and arithmetic complexity for various operations are evaluated.

There are two common ways to store the content of a bit-vector. The obvious way is to use a string of zeros and ones, in the following called *string encoded bit-vectors*. The other way is to store only the ids of the active elements in a list and we will call these *list encoded bit-vectors*. Both encodings have rather different costs as Table B.1 summarizes and the following paragraphs explain in detail. Fortunately, conversion between both formats can be performed on-the-fly, which allows for mixing both techniques and using the best suited encoding for every task.

		Depth of circuit	
Operation	Effect	string encoding	list encoding
SET	setting a bit	O(1)	O(n)/O(1)
CLEAR	clearing a bit	O(1)	O(n)
CHECK	checking the state of a bit	O(1)	O(n)
ANAC	test if any element is active	O($LOG(n)$)	O(1)
ENAC	enumerate all active elements	O($LOG(n)$)	O(1)
	Storage (in bits):	n	$n \cdot \lceil LOG(n) \rceil$

Table B.1: String encoded and list encoded bit-vectors with n elements have rather different costs regarding the depth of the circuit and the storage requirements. For example setting and clearing bits in a string encoded bit-vector is trivial. On list encoded bit-vectors setting a previously inactive element can be performed in O(1), but if the state of the element is unknown first a linear search on the list has to be performed. The storage requirements also give constraints for the transfer time of an bit-vector. While string encoded bit-vectors can always be transfered in a single cycle using a bus of width n bits it takes $m = \lceil LOG(n) \rceil$ cycles in the worst case to transfer a list encoded bit-vector. Nevertheless, typically list encoded bit-vectors are transfered in n cycles using a bus of width m.

String Encoded Bit-Vectors

On string encoded bit-vectors the implementation of *SET* (*setting a bit*), *CLEAR* (*clearing a bit*), and *CHECK* (*checking the state of a bit*) is trivial and *ANAC* (*test if any element is active*) can be realized by simply *OR*-ing all bits of the vector. In contrast *ENAC*

(*enumerate all active elements*) is much harder as it actually requires two steps: first an active element has to be found which is erased in the second step before iteration starts over with step one until no more active elements are found. While the second step is trivial step one is as hard as counting the leading zeros in the binary representation of a number which is a problem known to be in $O(LOG(n))$ [MP00]. Appendix C presents a circuit that computes *ENAC* on string encoded bit-vectors which also computes whether there are any further active elements.

Depending on the number of gates that can be executed in one clock cycle finding the next active element may take several cycles. This leads to a problem since the two steps of *ENAC* are dependent and therefore it might not be possible to output a new active element in each cycle. However, there are two solutions to this problem: *divide and conquer* and *multi-threading*.

Using the divide-and-conquer principle shortens the latency by splitting up the bit-vector and working in parallel on the parts. If for example we split up into two parts which both have a latency of 2 cycles, then a new element can be found every cycle if both parts have an equal number of active elements. Obviously the worst case latency is the latency of the largest part.

The other alternative is to use multi-threading by working on several bit-vectors in parallel. If the circuit has a latency of m cycles, having at least m bit-vectors guarantees to output a new active element every cycle. If this guarantee is not necessary, the number of threads can be reduced by sequentially exploiting the divide-and-conquer principle. For example having k bit-vectors which are split into two parts suffices for a latency of $2k$ cycles.

List Encoded Bit-Vectors

List encoded bit-vectors can be implemented quite elegantly by using FIFOs (see Appendix C) to store only the active bits. This leads to completely different costs compared to string encoded bit-vectors.

If it is known that a bit is currently inactive appending it to the list is trivially performed in O(1). However, in the general case setting a bit first requires to check the state of the bit (in O(n) since it is a linear search) and only if it is not set, appending it to the list (in O(1)). These two steps are necessary to avoid having the same bit occurring several times in the list. Clearing a bit can be implemented by copying the list (maybe in-place) and omitting the corresponding bit (if it is contained in the list).

Another drawback is that combining two bit-vector (e.g. line 119 in the pseudo-code) is rather costly O(n). However, there is also a big advantage of list encoded bit-vectors as executing *ANAC* and *ENAC* is trivial since they require only checking the number of elements stored in the FIFO respectively reading the first element.

C Selected Circuits

A number of standard circuits including detailed analysis of their costs and depths is given in [MP00]. However, some selected circuits used or developed for this thesis are presented in this chapter.

FIFOs

FIFO is an abbreviation for First In, First Out. A FIFO for n elements contains $n+1$ memory cells and has two pointers r and w for reading respectively writing of the memory cells. Reads and writes are performed round robin and the FIFO contains no elements if $r = w$ and is full if $w = (r-1)\ MOD\ (n+1)$ with MOD denoting the *modulo* function. Alternatively a FIFO for n elements can be build using only n memory cells but with pointers r, w using $\lceil LOG(n)+1 \rceil$ bits instead of $\lceil LOG(n+1) \rceil$ bits.

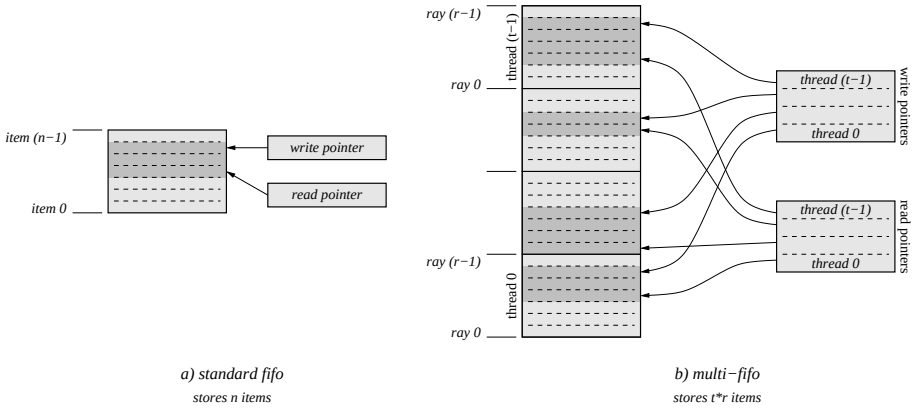


Figure C.1: A standard FIFO for n elements and a multi-FIFO which can store $t \times r$ elements. An implementation of these FIFOs which can insert and deliver an item in every cycle requires dual-port registers and register files for reading and writing in the same cycle. The blocks marked in grey are continuous regions of memory which shows the simplicity of this design.

FILOs

FILO is an abbreviation for First In, Last Out and implements a *stack*. A FILO for n elements contains n memory cells and has only a single pointer rw for reading and writing (which obviously can not be performed simultaneously). Additional there is a counter c which specifies how many entries are currently stored in the FILO.

Writing to the FILO increments the rw pointer and the counter c, and reading decrements both. All operations on rw wrap around, i.e. they are performed modulo n: $rw_{new} = (rw \circ 1)\ MOD\ n$, with $\circ = \{+|-\}$.

Stacks

Standard stacks can be implemented using FILOs as described above. However, if multi-threading is used but only a single thread accesses the stack at any time then there is a cheap way to build such a *multi-stack*. Multi-stack are almost identical to *multi-fifos* (see Figure C.1) with the only difference that instead of two register files for the read and the write pointers there is only one register file containing the read-write-pointers for all threads. This allows for an easy implementation of multi-stacks with only few and local blocks of memory and short connections.

ENAC on String Encoded Bit-Vectors

Figure C.2 shows the schematics of a circuit that calculates *ENAC* (see Appendix B) on string encoded bit-vectors. If required by the clock frequency the circuit can be pipelined. Then multi-threading can be used to hide the latency of the circuit and yield a new result in every cycle. Obviously then the registers need to be implemented in a dual ported register file (with simultaneous read and write capability).

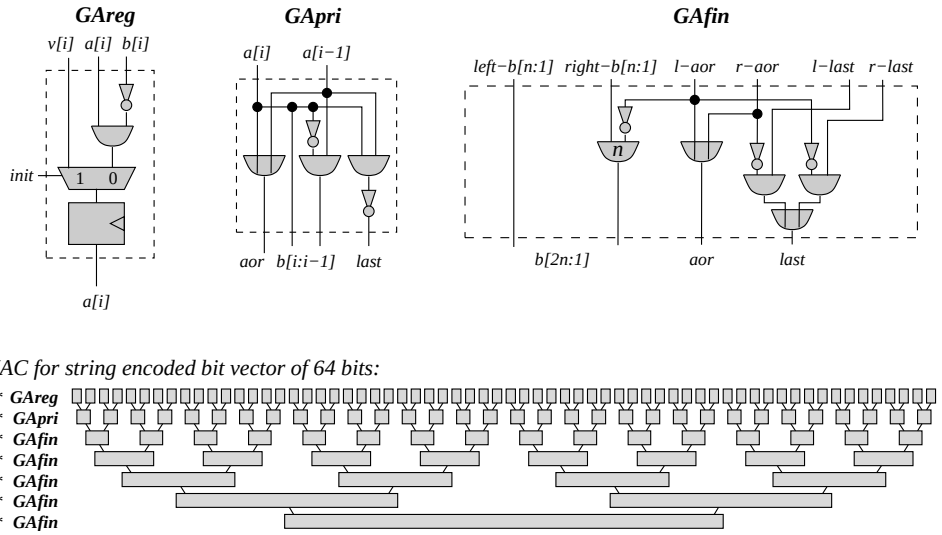


Figure C.2: A circuit to calculate *ENAC* on string encoded bit-vectors. On initialization the bit-vector *v[n:1]* is copied into the registers. Afterwards in every cycle the position *b[n:1]* of the first bit with value 1 (counted from left) is calculated in unary representation. Additionally, the signals *aor* and *last* specify whether any bit is active at all. If any bit is active it is additionally specified whether there are any further active bits or if this was the last active bit in the bit string. The cost of the circuit is $\frac{23}{2}n-8$ gates (not counting the registers) and it has a depth of $6+LOG(\frac{n}{2})$ (including the gates to subtract the previously enumerated element from the bit string).

Optimal Asynchronous Decoupling

If a job is scheduled and on the corresponding slice there are no active rays for this job then a cycle is wasted in which no useful work can be performed. However, this problem can occur especially in incoherent scenes. Fortunately there is a simple solution that avoids having any jobs for inactive rays in the job FIFO of a slice and therefore every job taken from the FIFO performs useful work.

This technique simple adds a delay queue with 3 stages in front of the FIFO. In stage one the active vector corresponding to the job is read. In stage two it is checked whether any ray is active. If no ray is active then in stage 3 the entry in not inserted into the FIFO. However, the corresponding ID has to be forwarded to Global signalling that this slice will not produce any results and therefore synchronization for this thread must only performed between the other slices.

Collecting Traversal Decisions Asynchronously

The traversal unit using asynchronously decoupled slices slightly increases the complexity of the Global circuit which collects the partial traversal decisions and computes the group's traversal decision. However, the following presents a cheap circuit that performs this task.

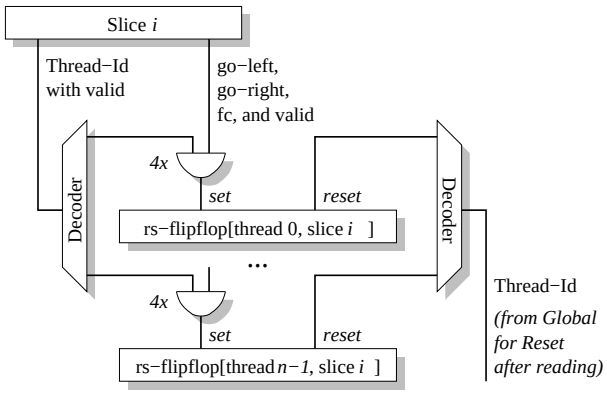


Figure C.3: Diagram of the communication between Global and the Traversal Slices.

Figure C.3 shows how the partial results of the traversal slices are stored using cheap *RS-Flip-Flops* (those flip-flops are one gate cheaper than standard d-flip-flops). The data stored in these flip-flops is combined into a string-encoded bit-vector for *packet-valid* and the packet decisions using the following formulas:

go-left[thread i]=*OR*(rsff.goleft[thread i, slice 0],$\cdots$,rsff.goleft[thread i, slice $n-1$])
go-right and *fc* are computed analog.
Here rsff.goleft denotes the go-left-component of a rs-flip-flop. Similarly
packet-valid[thread i]=*AND*(rsff.valid[thread i, slice 0],$\cdots$,rsff.valid[i, $n-1$])
t=*ENAC*(*packet-valid*) is used to schedule a finished packet t for further computation which also resets rsff[thread t,all slices] after scheduling.

D Comparisons of Costs

This chapter presents details on the comparisons shown for the optimizations in Section 7.3.1. The following comparisons take only into account the number of floating-point operations required. Table D.1 lists how costs are measured.

Symbol	Operation	Costs	Explanation
+	addition	1	1+
*	multiplication	1	1*
/	division	3	3*
v	R3 vector addition	3	3+
.	R3 dot-product	5	3*, 2+
x	R3 cross-product	9	6*, 3+

Table D.1: This table lists how costs are measured. Comparisons to 0.0 and 1.0 are not measured since they can be implemented using only a few gates (see Section 5.3). Subtractions are counted as additions.

Fast-Triangle (FT) [Wal04]
Test: $8+, 5*, 1/$
Calculate Barycentric Coordinates: $6+, 4*$
Total (without preprocessing) cost=26: $14+, 9*, 1/$

Moeller-Trumbore (MT) [MT97]
Test: 4x, 9., $9+, 2/, 1v, 3*$
PreProcessing: 2v
Total (without preprocessing) costs=102: $42+, 54*, 2/$

Pluecker (PL) [Eri97]
Pluecker-Test: 3+, 6.
Preprocessing for Barycentric Coordinates: 2v
Calculate Barycentric Coordinates: 2x, 4., $1/, 3*, 1+$, 1v
Calculate Pluecker-Coordinates: 4x, 3v
Total (without preprocessing) costs=81: $33+, 45*, 1/$
Total (with preprocessing) costs=87: $39+, 45*, 1/$

Simple-Intersection (SI) [Woo04]
Total costs=9: $4+, 2*, 1/$

Transformation-Unit (TU) [Woo04]
Total costs=18: $9*, 9+$

Fixed-Shader without Transformation Unit (FS)
Total costs=15: $7*, 8+$

Traversal Unit with 4 Slices (4TS)
Total costs=16: $4(1*, 3+)$

Comparison of the costs (without Traversal)
Statical SaarCOR:
[S1] FT + FS = 41
Dynamical SaarCOR:
[D1] TU + SI = 27
[D2] TU + FT + FS = 59
[D3] TU + PL + FS = 120
[D4] TU + MT + FS = 135

Comparison of the ratios (without Traversal)

D1 : S1 = 0.66		
D2 : S1 = 1.44	D2 : D1 = 2.19	
D3 : S1 = 2.93	D3 : D1 = 4.44	D3 : D2 = 2.03
D4 : S1 = 3.30	D4 : D1 = 5.00	D4 : D2 = 2.28

Comparison of the costs (with Traversal)
Statical SaarCOR:
[S1] 4TS + FT + FS = 57
Dynamical SaarCOR:
[D1] 4TS + TU + SI = 43
[D2] 4TS + TU + FT + FS = 75
[D3] 4TS + TU + PL + FS = 136
[D4] 4TS + TU + MT + FS = 151

Comparison of the ratios (with Traversal)

D1 : S1 = 0.75		
D2 : S1 = 1.32	D2 : D1 = 1.74	
D3 : S1 = 2.39	D3 : D1 = 3.16	D3 : D2 = 1.81
D4 : S1 = 2.65	D4 : D1 = 3.51	D4 : D2 = 2.01

E Additional Measurements

This section presents some additional measurements on the Conference scene (Figures E.1 to E.3) using a good kd-tree not covered in Chapter 8. They provide a deeper insight on the bandwidth requirements and hit-rates of various caching schemes.

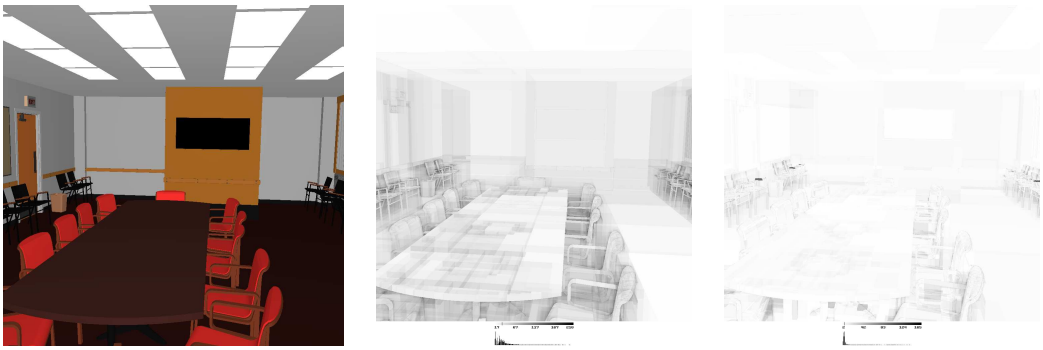


Figure E.1: These images show a rendering in 1024×1024 pixels of the Conference scene and the corresponding cost-images for traversal (middle) and intersection (right).

In Figure E.3 it can be seen that the costs of traversal are far less than the costs for performing the intersection calculations since for nodes not only the bandwidth to the caches decreases with increasing size of the packet but also the hit-rate for nodes is always above the hit-rates for triangles. Thus a better reduction of the bandwidth is achieved for all sizes of node caches than for any triangle cache of comparable size.

Looking at the costs measured in floating-point operations and size of the data structures then it shows that the ratio between intersection and traversal operations is 4:1 respectively 5:1 meaning that intersections are far more expensive. Taking the statistics about the caching behavior into account it becomes clear that the ratio is actually more in the order of 20:1.

However, all of these measurements did not use mailboxing which would further decrease the hit-rate of the triangle cache (since then triangles are read multiple times by the same packet) and therefore push the ratio even further in favor of traversal operations.

Thus the consequences derived from these measurements are: Trading intersection for traversal operations is a good choice for reducing the bandwidth and unless caches become rather tiny the caching scheme (direct mapped, 2-way or 4-way set associative) does hardly affect the hit-rate.

	Memory accesses			
Rays per Packet:	2x2 = 4 rays	4x4 = 16 rays	8x8 = 64 rays	Operations
Nodes (incl. Clipping Tree)	11'394'885	3'046'729	881'841	40'970'462
Tri-IDs (in Lists) Triangles	1'701'564	547'826	224'188	6'040'849

Figure E.2: Characteristics of the Conference scene when rendering the view shown in Figure E.1.

Rays	Cache	#Items, Size		Bandwidth to Cache	Cache Hits in % DM, 2Way, 4Way			Bandwidth to Memory	Architecture
2x2	Nodes	512	4 KB	87 MB	88.3 %	96.3 %	97.7 %	2.0 to 10.2 MB	SC-Proto
		4096	32 KB		98.8 %	99.3 %	99.4 %	0.5 to 1.1 MB	Medium
		8192	64 KB		99.1 %	99.4 %	99.5 %	0.5 to 0.8 MB	SC-1
	Lists	512	2 KB	6.5 MB	90.5 %	91.5 %	91.6 %	0.5 to 0.6 MB	SC-Proto
		1024	4 KB		92.3 %	93.5 %	94.0 %	0.4 to 0.5 MB	Medium
		16384	64 KB		95.7 %	95.9 %	95.9 %	0.3 MB	SC-1
	Tris	256	9 KB	58 MB	74.4 %	82.0 %	84.5 %	9.0 to 15.0 MB	SC-Proto
		1024	36 KB		89.3 %	92.2 %	92.8 %	4.2 to 6.2 MB	Medium
		4096	144 KB		94.6 %	95.5 %	95.6 %	2.6 to 3.2MB	SC-1
4x4	Nodes	512	4 KB	23 MB	84.8 %	92.8 %	94.4 %	1.3 to 3.5 MB	SC-Proto
		4096	32 KB		96.7 %	97.4 %	97.7 %	0.5 to 0.8 MB	Medium
		8192	64 KB		97.3 %	97.8 %	97.9 %	0.5 to 0.6 MB	SC-1
	Lists	512	2 KB	2.1 MB	78.0 %	79.2 %	79.3 %	0.4 to 0.5 MB	SC-Proto
		1024	4 KB		80.8 %	82.2 %	82.9 %	0.4 MB	Medium
		16384	64 KB		86.8 %	87.2 %	87.3 %	0.3 MB	SC-1
	Tris	256	9 KB	18.8 MB	66.5 %	72.8 %	74.9 %	4.7 to 6.3 MB	SC-Proto
		1024	36 KB		82.1 %	85.3 %	86.4 %	2.6 to 3.4 MB	Medium
		4096	144 KB		89.4 %	90.6 %	90.7 %	1.8 to 2.0 MB	SC-1
8x8	Nodes	512	4 KB	6.7 MB	76.1 %	83.9 %	85.9 %	1.0 to 1.6 MB	SC-Proto
		4096	32 KB		90.3 %	91.5 %	92.0 %	0.5 to 0.7 MB	Medium
		8192	64 KB		91.5 %	92.4 %	92.7 %	0.5 to 0.6 MB	SC-1
	Lists	512	2 KB	0.9 MB	55.8 %	57.1 %	57.2 %	0.4 MB	SC-Proto
		1024	4 KB		59.1 %	60.6 %	61.3 %	0.3 to 0.4 MB	Medium
		16384	64 KB		68.0 %	68.7 %	68.9 %	0.3 MB	SC-1
	Tris	256	9 KB	7.7 MB	56.2 %	61.0 %	62.5 %	2.9 to 3.4 MB	SC-Proto
		1024	36 KB		71.7 %	75.3 %	76.7 %	1.8 to 2.2 MB	Medium
		4096	144 KB		81.2 %	82.8 %	82.9 %	1.3 to 1.5MB	SC-1

Figure E.3: Detailed statistics about bandwidth requirements and caching behavior in the Conference scene (1024 × 1024 pixels, view of Figure E.1).

F Simulations of Missing Instruction for the SCPU

The instruction set of the SCPU has been reduced to a minimum. However, care has been taken that common instructions that are missing in the minimalistic instruction set can be simulated efficiently. Since for some cases it is not quite obvious how to simulate an instruction this section lists examples developed while writing benchmarks for the SCPU.

Notation: The instruction set is shown in Table 5.2. Additionally `adr(`x`)` denotes the physical address where x is stored in memory (required for jumps). A `!` at the beginning of a line denotes the *stnt* (switch to next thread) flag (see Section 5.3.3) and thus after its execution the control flow of the SCPU should switch to a different thread.

Integer Loops

The minimalistic instruction set does not provide any comparisons on integers. However, typically loops are implemented using integers. But although simply two different registers could be used – one storing the number of the iteration as integer and another as floating-point number to be used for comparisons – this is no elegant solution. The following presents two alternatives which additionally require only few instructions.

```
int   a = 17
float b = 0.0

      a = a or b
      c = adr(loop)
loop: ....
     !a = a int+ -1
      sip a pc=c
```

Alternative 1
using floating-point compares
on an integer value

```
int   a = 17

loop: ....
     !a = a int+ -1
     !tbz d=1, if all bits of a==0
      sip d pc=c
```

Alternative 2
using bit-wise compares
on an integer value

NOP

The instruction set does not provide any instruction to perform a *no operation* (NOP). However, there are cases in which NOPs are required. Since furthermore it is sometimes necessary to have a sequence of several NOP instructions it is important to simulate the NOP instruction such that no data hazards can occur.

Thus for example the NOP instruction can be simulated using

`sin` if `S1` $< (0.0 + \epsilon)$: `T := S2`

Here `S1` is specifies the integer constant 1 in the RORF and it does not matter which registers are specified as `T` or `S2` since the condition is always false and therefore the assignment is never executed.

Logical NOT

The instruction set does not provide any instruction to perform a *logical NOT operation*. However, a NOT(a) can be calculated trivially using XOR(a,a).

Bibliography

[3DL99] 3DLabs. Virtual Textures – a true demand-paged texture memory management system in silicon. `http://www.merl.com/hwws99/hot3d.html`, 1999.

[Act02] Activision. Return to Castle Wolfenstein. `http://games.activision.com/games/wolfenstein/`, 2002.

[Ake93] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (ACM Siggraph Proceedings)*, 1993.

[Alb25] Albrecht Dürer. Underweyssung der Messung, 1525.

[Alp03] Alpha-Data. ADM-XRC-II. `http://www.alphadata.uk.co`, 2003.

[Amd67] G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA*, pages 483–485. AFIPS Press, 1967.

[AMH02] Tomas Akenine-Möller and Eric Haines. *Realtime Rendering (2nd edition)*. A K Peters, July 2002.

[ANA04] David L. Andrews, Douglas Niehaus, and Peter J. Ashenden. Programming Models for Hybrid CPU/FPGA Chips. *IEEE Computer*, 37(1):118–120, 2004.

[App68] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. *SJCC*, pages 27–45, 1968.

[Are88] Jeff Arenberg. Ray/Triangle Intersection with Barycentric Coordinates. `http://www.acm.org/tog/resources/RTNews/html/rtnews5b.html`, 1988.

[AW87] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, pages 3–10. Eurographics Association, 1987.

[Bat] Ken Batcher. Goodyear Aerospace in Dayton, Ohio, USA. `http://www.worldhistory.com/wiki/K/Ken-Batcher.htm`.

[Ber02] Manfred Bertuch. Nvidia und die 100 Dinosaurier, 2002. `http://www.heise.de/newsticker/meldung/32189`.

[Bjo04] Kevin Bjorke. Cinematic Effects II: The Revenge, 2004. `http://developer.nvidia.com`.

[Bla04] Paul E. Black. `http://www.nist.gov/dads/HTML/parallprefix.html`, 2004.

[Blo98] Jonathan Blow. Implementing a Texture Caching System. Game Developer Conference, Vol. 5, No. 4, 1998.

[BP90] Didier Badouel and Thierry Priol. An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures. IRISA - Campus de Beaulieu - 35042 Rennes Cedex France, 1990.

[Bri03] Brigham Young University, USA. BYU JHDL. `http://www.jhdl.org`, 2003.

[BSD94] BSD Games. Fortune, 1994.

[Bur96] John M. Burwell. Redefining High Performance Computer Image Generation. In *Proceedings of the IMAGE Conference, Scottsdale, Arizona*, 1996.

[BWDS02] Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002.

[BWPP04] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[BWS03] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum (Proceedings of Eurographics)*, 22(3):621–630, 2003.

[BWS04] Carsten Benthin, Ingo Wald, and Philipp Slusallek. Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph*, pages 99–106, November 2004.

[Car03] Nicholas P. Carter. *Computerarchitektur*. mitp-Verlag, Bonn, 2003.

[Cel03] Celoxica Ltd. Handle-C, 2003. `http://www.celoxica.com`.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of Graphics Hardware*, pages 37–46. Eurographics Association, 2002.

[Cro77] F.C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH 77 Proceedings)*. ACM Press, July 1977.

[Dig04] Digital Extremes and Epic Games. Unreal Tournament 2004. `http://www.unrealtournament.com/ut2004/`, 2004.

[Dre05a] Patrick Dreker. Design und Simulation einer Hardware-Shading-Einheit für Ray-Tracing, 2005. Bachelor's Thesis, Computer Graphics Group, Saarland University, Germany.

[Dre05b] Patrick Dreker. Entwurf und Implementierung eines Shading-Prozessors für Echtzeit-Raytracing. Master's thesis, Computer Graphics Group, Saarland University, Germany, 2005.

[DWBS03] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, 2003.

[DWWS04] Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek. VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, pages 109–116, March 2004.

[EIH01] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of SIGGRAPH*, 2001.

[Eld01] Matthew Eldridge. *Designing Graphics Architectures around Scalability and Communication.* PhD thesis, Stanford University, 2001. `http://graphics.stanford.edu/papers/eldridge_thesis/`.

[EMP+97] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The Realization. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997.

[Eri97] Jeff Erickson. Pluecker coordinates. *Ray Tracing News*, 1997. `http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art11`.

[ESK97] J. Encarnacao, W. Straßer, and R. Klein. *Graphische Datenverarbeitung 1.* Oldenbourg Verlag München, 1997.

[Eve01] Cass Everitt. Interactive Order-Independent Transparency, 2001. `http://developer.nvidia.com`.

[Fly95] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, 1995.

[FPE+89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. In *Proceedings of SIGGRAPH*, 1989.

[FvDFH97] Foley, van Dam, Feiner, and Hughes. *Computer Graphics – Principles and Practice, 2nd edition.* Addison Wesley, 1997.

[Gai02] Neil Gaiman. *Don't Panic - Douglas Adams & The Hitchhiker's Guide to the Galaxy.* Titan Books, 2002. `http://www.csd.uwo.ca/Infocom/Articles/Douglas.html`.

[GH96] C. Scott Ananian Greg Humphreys. Tigershark: A hardware accelerated ray-tracing engine. Technical report, Princeton University, 1996.

[Gla89] Andrew Glassner. *An Introduction to Ray Tracing.* Morgan Kaufmann, 1989.

[GP90] Stuart A. Green and Derek J. Paddon. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer*, 6(2):62–73, 1990.

[Gre91] Stuart A. Green. Parallel Processing for Computer Graphics. *MIT Press*, pages 62–73, 1991.

[Gus91] John L. Gustafson. Twelve Ways to Fool the Masses When Giving Performance Results on Traditional Vector Computers, 1991. `http://www.scl.ameslab.gov/Publications/Gus/TwelveWays.html`.

[GWS04] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime Caustics using Distributed Photon Mapping. In *Rendering Techniques 2004*, pages 111–121, June 2004.

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

[Hau00] John Reid Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, University of California, Berkley, 2000.

[Hav01] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[HG97] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture (ISCA)*, 1997.

[HKR00] M. Porrmann H. Kalte and U. Rückert. Using a dynamically reconfigurable system to accelerate octree based 3d graphics. Technical report, System and Circuit Technology, University of Paderborn, 2000.

[HL03] Hans Holten-Lund. An Application Specific Reconfigurable Graphics Processor, 2003. Graphics Vision Day, IMM, DTU.

[Hof92] Georg Rainer Hofmann. Who invented ray tracing? a historical remark. *The Visual Computer*, 9(1):120–125, 1992.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach, 2nd edition*. Morgan Kaufmann, 1996.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[IEH99] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1999.

[IEP98] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998.

[Int02] Intel Cooperation. Introduction to Hyper-Threading Technology, 2002. `http://developer.intel.com/technology/hyperthread`.

[IS04] Id-Software. The Doom and Quake Series 1993 – 2004. `http://www.id-software.com/` , 2004.

[Joh91] William Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

[Kel98] Alexander Keller. *Quasi-Monte Carlo Methods for Realistic Image Synthesis.* PhD thesis, University of Kaiserslautern, 1998.

[Key35] John Maynard Keynes. The General Theory of Employment, Interest and Money, 1935. `http://en.wikiquote.org/wiki/John_Maynard_Keynes`.

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

[Kil97] Mark J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997.

[KSSO02] Hiroaki Kobayashi, Ken-Ichi Suzuki, Kentaro Sano, and Nobuyuki Oba. Interactive Ray-Tracing on the 3DCGiRAM Architecture. In *Proceedings of ACM/IEEE MICRO-35*, 2002.

[LAM00] Jonas Lext, Ulf Assarsson, and Tomas Möller. BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2000. Available at `http://www.ce.chalmers.se/BART/`.

[Lei04] Alexander Leidinger. A Virtual Memory Architecture for Ray Tracing Hardware. Master's thesis, Computer Graphics Group, Saarland University, Germany, 2004.

[Lic96] Cedric Lichtenau. Entwurf und Realisierung des Speicherboards der SB-PRAM. Master's thesis, Universität des Saarlandes, Saarbrücken, 1996.

[Lic00] Cedric Lichtenau. *Entwurf und Realisierung des Aufbaus und der Testumgebung der SB-PRAM.* PhD thesis, Universität des Saarlandes, Saarbrücken, 2000.

[LS91] Tony T.Y. Lin and Mel Slater. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, pages 187–199, 1991.

[MBDM97] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of SIGGRAPH*, 1997.

[MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH*, 1992.

[MFK+04] Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[MKS98] M. Meissner, U. Kanus, and W. Strasser. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Eurographics/Siggraph Workshop on Graphics Hardware*, 1998.

[MP95] Silvia M. Müller and Wolfgang J. Paul. *The Complexity of Simple Computer Architectures*. Springer Verlag, 1995.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer Verlag, 2000.

[MPJ+00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Recongurable Architecture. *IEEE International Symposium on Computer Architecture*, 2000.

[MT97] Tomas Moeller and Ben Trumbore. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[MT04] Tobias Martin and Tiow-Seng Tan. Anti-aliasing and Continuity with Trapezoidal Shadow Maps. In *Proceedings of Eurographics Symposium on Rendering*, 2004.

[Muu95] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[Neb97] Jean-Christophe Nebel. A Mixed Dataflow Algorithm for RayTracing on the CRAY T3E. In *Third European CRAY-SGI MPP Workshop*, September 1997.

[Nvi02] Nvidia Coperation. Press Release Nvidia GeForce 3 , 2002. `http://www.nvidia.com` .

[Nvi04] Nvidia Coperation. Nvidia GeForce FX 5900 , 2004. `http://www.nvidia.com/object/geforcefx_facts.html` .

[PBB+02] Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. *Real PRAM-Programming*. Proceedings of EuroPar, 2002. (Jörg Schmittler's name was formerly Fischer and this paper was published under his old name.).

[PHK+99] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time ray-casting system. *Computer Graphics*, 33, 1999.

[PHS04] Andreas Pomi, Simon Hoffmann, and Philipp Slusallek. Interactive In-Shader Image-Based Visual Hull Reconstruction and Compositing of Actors in a Distributed Ray Tracing Framework. In *1. Workshop VR/AR, Chemnitz, Germany*, September 2004.

[Ple05] Raoul Plettke. Interaktives Raytracing für komplexe Außenszenen, 2005. Bachelor's Thesis, Computer Graphics Group, University of Erlangen, Germany.

[PMWS03] Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek. Streaming video textures for mixed reality applications in interactive ray tracing environments. In *Proceedings of Virtual Reality, Modelling and Visualization (VMV)*, 2003.

[Poh04] Daniel Pohl. Anwendung von Strahlverfolgung für das Computerspiel Quake 3 (Applying Ray Tracing to the Quake 3 Computer Game), 2004. Bachelor's Thesis, Computer Graphics Group, University of Erlangen, Germany.

[PS04] Andreas Pomi and Philipp Slusallek. Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework. In *IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2004, Student Colloquium*, November 2004.

[PSL+99] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, April 1999.

[Pur01] Timothy Purcell. The SHARP Ray Tracing Architecture. SIGGRAPH course on Interactive Ray Tracing, 2001.

[Pur04] Timothy J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[Roc03] Rockstar Games. Grand Theft Auto: Vice City. `http://www.rockstargames.com/vicecity/pc/`, 2003.

[RSH00] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[Röh99] Jochen Röhrig. *Effiziente Interprozeßkommunikationsdatenstrukturen für die SB-PRAM und deren Anwendungen*. PhD thesis, Department of Computer Science, Saarland University, 1999.

[SBS04] D. Staneker, D. Bartz, and W. Strasser. Efficient multiple occlusion queries for scene graph systems. *WSI Report (WSI-2004-6)*, 2004.

[SD02] Marc Stamminger and George Drettakis. Perspective Shadow Maps. In *Proceedings of Siggraph*, pages 557–562, 2002.

[SDP+04] Jörg Schmittler, Tim Dahmen, Daniel Pohl, Christian Vogelgesang, and Philipp Slusallek. Ray Tracing for Current and Future Games. In *Proceedings of 34. Jahrestagung der Gesellschaft für Informatik*, 2004.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters, 2002.

[SI91] Inc. SPARC International. The sparc architecture manual, version 8, 1991.

[SKS96] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A Smart Memory for Texturing. In *Computer Graphics and Applications*, 1996.

[SLS03] Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek. A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Special Issue on Graphics Hardware*, 27:693–699, 2003.

[Son05] Sony. Cell Broadband Engine Architecture, 2005. `http://cell.scei.co.jp`.

[SR00] Michael S. Schlansker and B. Ramakrishna Rau. EPIC: Explicitly Parallel Instruction Computing, 2000. Hewlett-Packard Laboratories.

[SS92] Kelvin Sung and Peter Shirley. Ray Tracing with the BSP Tree. In David Kirk, editor, *Graphics Gems III*, pages 271—274. Academic Press, 1992.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert F. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys*, 6(1), 1974.

[Sub90] K. R. Subramanian. *A Search Structure based on kd-Trees for Efficient Ray Tracing*. PhD thesis, University of Texas at Austin, 1990.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*, 2004.

[Sys03] System-C, 2003. `http://www.systemc.org`.

[Wag02] Markus Wagner. Development of a Ray-Tracing-Based VRML Browser and Editor. Master's thesis, Computer Graphics Group, Saarland University, Saarbrücken, Germany, 2002.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wat00] Alan Watt. *3D Computer Graphics – Third Edition*. Addison-Wesley, 2000.

[WBDS03] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 2003.

[WBS02] Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[WBS03a] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[WBS03b] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive Global Illumination in Complex and Highly Occluded Environments. In Per H Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 74–81, Leuven, Belgium, 2003.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3), 2001.

[WDS04] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 81–92, 2004.

[WFM+05] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.

[WGS04] Ingo Wald, Johannes Günther, and Philipp Slusallek. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum*, 22(3):595–603, 2004. (Proceedings of Eurographics).

[Wil78] Lance Williams. Casting Curved Shadows on Curved Surfaces. In *Proceedings of Siggraph*, pages 270–274, 1978.

[Win02] Christof Windeck. *Neue Speicherstandards für den PC, Teil 1+2.* Heise Verlag, pages 262ff in c't 6/02 and 228ff in c't 8/02, 2002.

[WKB+02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusal lek. Interactive Global Illumination using Fast Ray Tracing. In Paul Debevec and Simon Gibson, editors, *Rendering Techniques 2002*, pages 15–24, Pisa, Italy, June 2002. Eurographics Association, Eurographics. (Proceedings of the 13th Eurographics Workshop on Rendering).

[Woo04] Sven Woop. A Ray Tracing Hardware Architecture for Dynamic Scenes. Master's thesis, Computer Graphics Group, Saarland University, Germany, 2004.

[WPS+03] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its Use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, pages 85–122, 2003.

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics (PGB)*, page to appear, 2005.

[WSB01] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[Xil03] Xilinx. Virtex-II. `http://www.xilinx.com`, 2003.

[YSB00] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A c compiler for a processor with a reconfigurable functional unit. In *FPGA*, pages 95–100, 2000.

[Zim03] Paul Zimmons. Cell Architecture, 2003. `http://www.ps3portal.com/downloads/cell.ppt`.