Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes

Sven Woop[†], Gerd Marmitt[‡], and Philipp Slusallek[§]

Computer Graphics Lab, Saarland University, Germany

Abstract

This paper introduces a new spatial index structure, called Bounded KD tree (B-KD tree), *for realtime ray tracing of dynamic scenes. By presenting hardware units of all time critical B-KD tree algorithms in the context of a custom realtime ray tracing chip we show that this spatial index structure is well suited for hardware implementation.*

B-KD trees are a hybrid spatial index structure that combine the advantages of KD trees and Bounding Volume Hierarchies into a single, simple to handle spatial index structure. Similar to KD trees, B-KD trees are binary trees where each node considers only a single spatial dimension. However, instead of a single splitting plane that divides *space into two disjoint sub-spaces, each node in B-KD trees contains two pairs of axis aligned planes that* bound *the geometry of its two child nodes. As a bounding volume approach B-KD trees allow for simple and efficient updates when changing geometry while maintaining the fast traversal operations and simple hardware implementation known from KD trees. This enables the support for dynamic scenes with constant mesh topology and coherent dynamic changes, like typical skinned meshes.*

Our hardware architecture contains several fixed-function units that completely handle skinning, updating, and ray tracing of dynamic scenes using B-KD trees. An FPGA prototype of this architecture already delivers realtime performance of up to 35 frames per second even when clocked at only 66 MHz.

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

1. Introduction

The state-of-the-art in realtime rendering is the *rasterization* algorithm [FvDFH97] mainly because low-cost and efficient hardware implementations are available that achieve remarkable levels of performance. Rasterization is, in particular, well suited for handling dynamic scenes as changes of geometry can be displayed directly without the need of maintaining any auxiliary data structures.

Conceptually, the basic operation of rasterization is to individually draw each triangle of a scene onto the screen by shading all covered pixels. Because triangles are treated individually and access to the rest of the scene is not available or at least highly restricted, the shading operations are inherently limited to depend only on *local data* provided with each triangle and some limited global state. Unfortunately, global effects such as shadows, reflection, refraction, or indirect illumination cannot be directly computed this way. While these effects can sometimes be approximated using multi-pass rendering techniques this often results in artifacts and is inefficient both with respect to computation and memory bandwidth.

In contrast, ray tracing [App68, Whi80] inherently requires global access to the entire scene, as its core operation of finding the first intersection with geometry along a ray is a global visibility query. It heavily relies on fine grained (hierarchical) spatial index structures to make these queries efficient. Consequently, ray tracing is well suited to access global information in a scene and thus forms the basis of almost all approaches that *simulate* the physics of light (e.g. global illumination) based on the rendering equation [Kaj86]. Such

† {woop}@cs.uni-sb.de
‡ {marmitt}@cs.uni-sb.de
§ {slusallek}@cs.uni-sb.de

global effects require frequent and detailed visibility information that can directly be computed by sampling the environment using ray tracing. While ray tracing has been accelerated to realtime performance in both software and hardware over recent years, support for dynamic scenes has been very limited due to the cost of rebuilding or updating the spatial index structures after every change.

The main contribution of this paper is a new hybrid spatial index structure, *Bounded KD (B-KD) tree*, as well as a complete hardware architecture that uses these B-KD trees for implementing ray tracing of dynamic scenes.

2. Previous Work

For a long time spatial index structures for ray tracing only considered static scenes as it took minutes to hours to render a single image. Several spatial index structures have been proposed including: Grids [CWBV83, AW87], Octrees [Gla84, Arv88], Bounding Volume Hierarchies [RW80], Binary Space Partitioning [FKN80], and KD trees [Jan86, SF90].

Interactive ray tracing applications have been mapped to many different kinds of supercomputers such as SGI shared memory machines [PSL*99]. More recently, interactive performance has been brought also to clusters of standard PCs [WBWS01, WPS*03] and single desktop PCs [RSH05]. Ray tracing has also been mapped to programmable GPUs [Pur04, FS05].

With realtime ray tracing it became also necessary to handle interactive changes in dynamic scenes. This is possible by using hierarchical grids as they allow to insert objects in constant time [RSH00]. Separation of the scene into objects with piece-wise rigid motion and separate static spatial indices has been suggested by [LAM01] and has been implemented for realtime use on a cluster of PCs [WBS03]. Bounding Volume hierarchies have successfully been used for rendering of dynamic scenes [TL03].

Multiple *custom hardware* architectures have been proposed, both for volume [MKS98, KR00] and surface models. A complete ray tracing hardware architecture has been simulated by [KiSSO02, SWS02] while the first fully functional *realtime ray tracing hardware* was presented in [SWW*04]. The fully programmable RPU hardware architecture for ray tracing was published in [WSS05]. All these hardware architectures are limited to scenes with static or piecewise rigid motion, thus none of them supports highly dynamic scenes.

3. B-KD Trees

The Bounded-KD (B-KD) tree is a new hybrid spatial index structure that combines in a single homogeneous data structure the advantages of Bounding Volume Hierarchies with those of KD trees. From Bounding Volumes it inherits the efficient support for dynamic scenes, while maintaining the simplicity and efficiency of KD tree traversal in particular with respect to hardware implementation (see Section 4).

Definition: *A B-KD tree is a binary tree, where each node recursively subdivides the geometry of the scene into two disjoint subsets represented by its two children. Each node stores the index of a coordinate axis and bounds on the geometric extent of its two children along this axis in form of two intervals often also referred to as slabs (see Figure 1). Each leaf node stores a reference to a single primitive of the scene.*

The use of disjoint subsets removes redundancy and storage overhead. For a scene with N primitives a B-KD tree has exactly N leaf nodes and $N-1$ inner nodes. This makes the size of the B-KD tree predictable which would be difficult for KD trees. Special handling of lists of primitives in the leaf nodes (as in conventional KD trees) is not required and simplifies hardware implementations significantly.

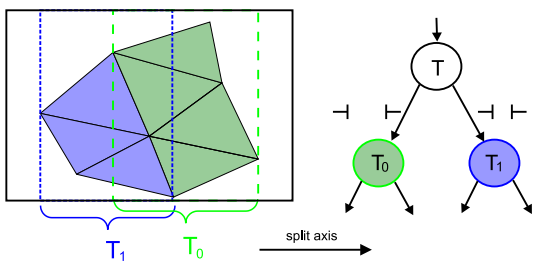

Figure 1: *B-KD tree: A B-KD tree node divides the geometry into two disjoint subsets, represented by the two children. The node stores the extend of the geometry for each child as two intervals along one splitting axis. The geometry is recursively subdivided until there is only a single primitive per node.*

Because of their bounding approach B-KD trees can be updated as efficiently as Bounding Volume Hierarchies, but in contrast their traversal operation is simplified because only a single axis is considered in each node. Instead of having to store complete bounding boxes in every node, B-KD trees can pick an optimal split axis in each step, thus reducing the size of the data structure by a factor of 3.

Furthermore, the traversal of B-KD trees is strictly ordered along the ray and the traversal computation can be terminated early if an intersection occurred in front of the next node. The traversal order does not influence the correctness of the algorithm but it greatly influences the traversal cost. The traversal order of the child nodes can be precomputed for the axis and depends only on the sign of the ray direction. Similar to KD trees, B-KD trees can also adapt well to the structure of the scene resulting in a compact representation and efficient handling of inhomogeneous distributions.

For many highly detailed scenes, the ability to instantiate multiple copies of an “object” is important in order to

minimize memory requirements. We support this through additional transformation nodes, that contain references to an affine transformation and to the root node of the object. By modifying the matrix one can transform the complete object, which may consist of thousands of triangles, while only changing the transformation node and possibly the bounds of its parents. This approach is well understood [LAM01, WBS03] so that we do not focus on it in this paper.

3.1. Updating B-KD Trees

For changed geometry the B-KD tree bounds can be updated by a simple bottom-up algorithm using only trivial min/max operations that do not touch the structure of the tree. This updating procedure merges the different bounds of the nodes from bottom-up through the tree and updates for each B-KD tree node the extend of the two children along the axis of the node. The order of both children is also precomputed for both possible signs of the ray direction along this axis. In Figure 2 the closer child is T_1 for a ray going from left to right, for instance.

For best results the overall structure of a B-KD tree should "match" the geometry and its dynamics. This means that geometry in a sub-tree should stay as close together as possible during the course of changes. A mismatch can result in significant overlap of the bounds of child nodes. This leads to redundant traversal and missed opportunities for early ray termination, as both child nodes *must* be traversed if a ray enters an overlap region. As a consequence only dynamic scenes that show some coherent motion can be handled effeciently with B-KD trees. Many typical motions, like skinned meshes, obey this restriction as will be shown in the result section. A random movement of triangles cannot be handled well as this would result in a traversal of many B-KD tree nodes, because of significant overlaps.

To optimize this random motion case, a non-optimal structure of the tree could in principle be detected by looking at the overlap of bounds. In the case of a mismatch it might be necessary to re-compute the structure of parts of the tree. Such computations can be performed by a software driver as they are usually rare and their cost can ideally be amortized over many frames, thus dedicated hardware would not required for this operation. We perform this reconstruction only for the higher nodes over the instantiated objects, as they might move around freely.

3.2. Traversal

The traversal algorithm for the B-KD tree is similar to that of standard KD trees [SF90]. The recursive traversal function traverses the scene in a traversal interval $I = [near, far]$ along the ray. We first test for early ray termination, with respect to the *near* distance. We then intersect the ray with the four bounding planes defined by the node giving the two intersection intervals $I_{\{0,1\}}$ for the two leaf nodes (see Figure 2). Before a child (for instance child 0) is traversed two comparisons determine if its intersection interval I_0 overlaps the current traversal interval I. We recursively traverse the child if this is the case with the traversal interval updated to the intersection of I and I_0, which requires two min/max operations. If the other child overlaps the traversal interval it is stored onto a stack together with the intersection of I and I_1 as its traversal interval.

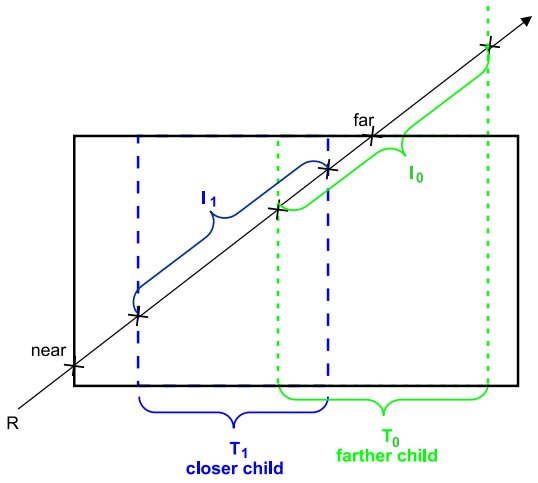

Figure 2: *Ray traversal: The ray is intersected with the four planes defined by the bounds of each child giving two intersection intervals $I_{\{0,1\}}$ along the ray. A child is traversed iff its intersection interval overlaps the traversal interval $I = [near, far]$ of the ray. The closer child is always traversed first to improve performance through early ray termination.*

3.3. Building B-KD Trees

As a B-KD tree is a Bounding Volume Hierarchy, similar construction algorithms can also be applied. Converting a binary Bounding Volume Hierarchy into a B-KD tree is easily possible by computing a splitting axis and bounding intervals for each node.

Bounding Volume Hierarchies can be created in a top-down or bottom-up fashion. Goldsmith and Salmon optimize the bottom-up construction using a cost model that minimizes an estimate for the traversal cost, called the Surface Area Heuristic [GS87]. For top-down approaches a median split of geometry [KK86] and a middle split of the volume [Smi98] have been analyzed.

We use a top-down approach similar to [MF99] that uses the Surface Area Heuristic to select an optimal partitioning of the triangles into two disjoint sets, similar to top-down construction algorithms for KD trees [Wal04]. Possible partitionings of the geometry are determined by looking at the

geometry sorted in all three dimensions according to the center of their bound. These sorted lists define possible partitionings $G = G_0 \cup G_1$ of the geometry into two disjoint sets ($G_0 \cap G_1 = \emptyset$), by splitting one of these lists at a cost optimal position. The list that is selected also determines the splitting dimension of the node. Sorting the geometry in the dimensions needs only to be performed once initially, which improves the runtime of the algorithm [Wal04].

The Surface Area Heuristic is used to select the partitioning with the smallest cost estimate. It is computed by a probabilistic model assuming uniformly distributed rays. The cost of a partitioning is an atomar traversal cost C_{trav} plus the intersection cost of each child (approximated by the number of primitives) multiplied with the probability $p(T_i)|T$ that a child T_i is traversed. These probabilities can be computed by the ratio between the surface areas $SA(T_i)$ of the child's bounding box and the surface area $SA(T)$ of the parent node.

$$SAH(T) = C_{trav} + \frac{SA(T_0)}{SA(T)} \cdot |G_0| + \frac{SA(T_1)}{SA(T)} \cdot |G_1|$$

A relative traversal cost of $T_{trav} = 1/3$ was experimentally showing to be a good choice for the hardware architecture described later.

4. Hardware Architecture

While software implementations of B-KD trees should be easily realizable and should achieve good performance, they suffer from the comparatively low computational power of CPUs and difficult low level programming in order to achieve high performance [RSH05]. Here we concentrate on efficient hardware implementations of B-KD trees.

Our hardware architecture named DynRT (dynamic ray tracing hardware) handles all aspects of dynamic scenes using B-KD trees. This includes a *Skinning Unit* to recompute a dynamic mesh using a very general skinning model, a dedicated *Update Processor* to maintain a valid B-KD tree after changes, and a fast ray casting engine, consisting of a *Traversal Unit* to traverses rays through the B-KD trees and a *Geometry Unit* to intersect them with triangles (or to transform them to the local coordinate space of an instantiated object).

The ray casting part of the architecture is similar to the hardware architecture described in [SWW*04] but with some significant changes. We also require a Traversal Unit to traverse rays through the spatial index, and a Geometry Unit to transform rays or to intersect them with triangles. A major advantage of our architecture is that neither a list nor mailbox unit is required, as the set of triangles maps one-to-one to the set of leaf nodes. We use a similar Traversal Unit that is modified for B-KD tree traversal, thus inherently supporting dynamic scenes without loss in performance. In contrast to [SWS02, SWW*04, WSS05] our Intersection/Geometry Unit operates directly on shared vertices (e.g. indexed face sets) instead of precomputing triangle data which makes our geometry cache (vertex cache) more efficient.

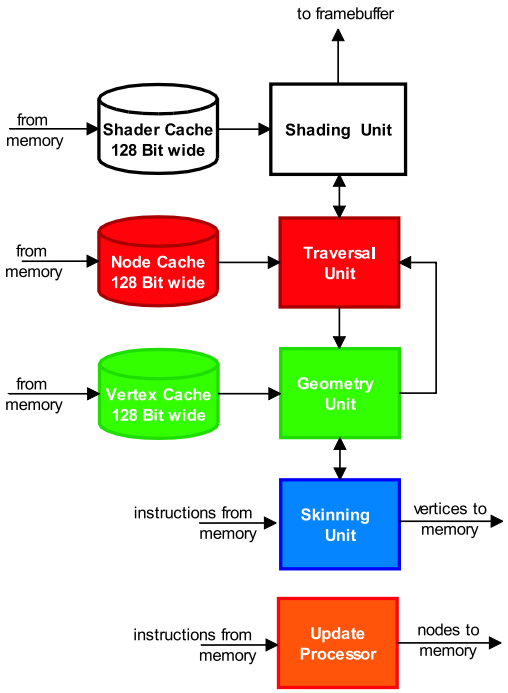

Figure 3: *DynRT Architecture: Vertices are skinned by a* Skinning Unit *that reuses some units of the geometry intersection, while B-KD trees are updated by the* Update Processor. *The rendering part consists of a* Shading Unit *to generate and shade rays and a fixed-function part for tracing rays through the scene. This fixed-function part contains of a high performance* Traversal Unit *to traverse the B-KD tree and a* Geometry Unit *to intersect rays with triangles or to transform them to the local coordinate space defined by a B-KD transformation node.*

The focus of this paper is not shading, but we are convinced that a similar shading unit as the RPU processor [WSS05] can easily be added to our architecture as well. With the addition of a shading processor, the proposed hardware architecture would have similar features as current rasterization engines, while also offering all the flexibility of ray tracing.

4.1. Skinning Unit

Recomputing the position of all vertices in a mesh for every frame is an expensive and time critical task that should not be performed by the application. The application should rather compute the motion at a higher level of abstraction thereby minimizing the computational cost and any communication overhead with the rendering engine.

Our hardware architecture contains a flexible fixed-function Skinning Unit that implements a general skinning model typically referred to as Skeleton Subspace Deformation

(SSD) [MTLT88, MTT91]. The position of a vertex is defined as the sum of vectors multiplied by some matrices:

Let $v_0, v_1, \cdots, v_{M-1} \in R^4$ be M vectors (premultiplied by some vertex weights) and $A_0, A_1, \cdots, A_{M-1} \in R^{3x4}$ be references to M matrices, then each vertex position $V \in R^3$ is defined as:

$$V = \sum_{j=0}^{M-1} A_j \, v_j$$

Each vertex of a mesh is defined by such a linear combination. Only the transformation matrices A_j are used to modify the mesh, while the vectors v_j stay constant.

The Skinning Unit is implemented as a vertex processor with a simple matrix instruction set working with 24 bit floating point accuracy in our implementation. The instructions form a simple instruction stream including instructions to set the row of a 3x4 matrix of one of the 32 matrix registers. Other instructions multiply 4 component vectors with a matrix from a matrix register and accumulate them. The accumulation is performed by a custom hardware unit while the matrix multiplication is performed by the the Geometry Unit that is also used for rendering, thus reusing an otherwise not utilized hardware resource. Accumulated results can then be stored to a vertex array in memory indexed by an immidiate constant. All vectors and matrix columns are encoded as immediate values in a 128 bit wide instruction format with 24 bit floating point accuracy.

This approach stores and encodes all the rules to compute all vertices in a mesh directly in a single sequential instruction stream with little overhead. With this approach only a single sequential instruction stream is read while a single sequential vertex stream is stored to memory. This simplifies the implementation and optimizes an already bandwidth limited computation by avoiding random memory accesses.

4.2. Data Layout

Each node of the B-KD tree is 16 bytes wide. Inner nodes store the 2 splitting intervals using four 22 bit floating point values, 2 bits to store the traversal order for the children, 2 bits for the splitting axis, and a relative pointer to the pair of child nodes. Pointing to the pair of children requires only one pointer (compared to two) but forces children to be stored next to each other in memory, which requires some special treatment during update (see next section). A leaf node stores pointers to each of the vertices in a vertex array and a pointer to the per-triangle shading data.

4.3. Update Processor

The updating of the B-KD tree of an object in the scene needs to be performed each time some geometry changes its position. In computer games this is typically the case in each frame, thus a high update performance is required. This

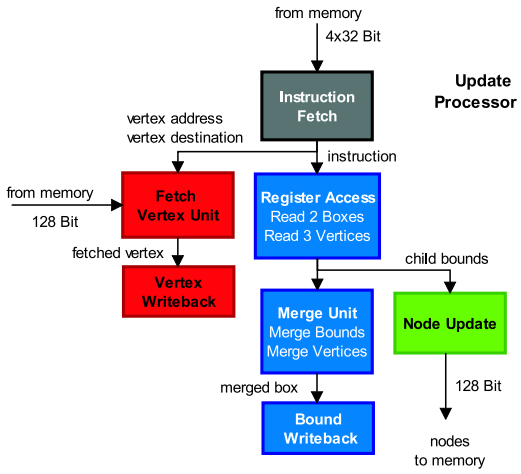

Figure 4: *Update Processor: The processor fetches update instructions (gray unit), that specify vertices to be read from memory (red units), leaf node bounds to be computed from 3 vertices (blue units), and inner-node bounds to be computed and updated from 2 child bounds (blue and green units).*

can be achieved by our special Update Processor, capable of updating one B-KD tree node per clock cycle.

The updating has to merge the axis-aligned bounds of the nodes from bottom-up through the tree and to update the bounds of the inner B-KD tree nodes. Principally this updating could be performed by recursively traversing the B-KD tree top-down in a special hardware unit. However, this would mean to cope with many data dependent memory fetches of B-KD tree nodes and the resulting latencies. To update a node such an approach would need to read the node, which produces much memory traffic (4 words per node). Furthermore, a cache would be required to cache the often shared vertices and a multithreading approach for high usage of the hardware unit at the cost of non-sequential memory requests.

This would result in a complicated unit, while the approach described in the following is more simple, programmable, and powerful. As it is programmable all the complexity is shifted to the compilation process of the B-KD trees in the driver, where more complex optimizations can be performed.

The *Update Processor* described here is optimized for triangle meshes where typically many vertices are shared. In regular meshes a vertex is shared by about 6 triangles. These shared vertices are stored to one of 64 internal *vertex registers*, where they can be used for computing the bound of several triangles. All partial results, such as computed node bounds are stored to one of 64 special *bound registers* to minimize the external memory requests to only the required updates of the nodes, vertex fetches, and additional instruction fetches. The architecture needs no caches, as no tempo-

rary values are stored in memory and the vertices are usually read only once and reused optimally from the register file. The nodes of the tree are processed in the reverse order as them are stored in memory such that the pointer to the next node to update can be computed by simple decrement operations in the hardware.

All data is arranged in such a way that it is processed mainly sequentially by the hardware. The instruction stream is read purely sequentially from memory, the nodes to update are processes sequentially, and the vertices are stored according to the order of their first access. All this together allows for an efficient usage of the external DRAM. One update operation of a leaf node and inner node can be computed with throughput 1 and a small latency of only a few cycles.

The Update Processor (see Figure 4) executes three different kinds of instructions for fetching vertices, merging 3 vertices to a bound, and merging two bounds to a new bound. These 32 bit wide instructions are fetched by the Instruction Fetch unit from the 128 bit wide memory interface. Vertices are fetches by the Fetch Vertex Unit from main memory, and then stored to one of the 64 vertex registers. If vertices are shared between multiple triangles, these registers can be reused to reduce the memory bandwidth.

4.4. Traversal Unit

The Traversal Unit traverses several packets of four rays in parallel through the B-KD tree similar to the algorithm from Section 3.2. In order to hide memory and computation latencies multiple packets of rays are being processed simultaneously using a wide multi-threading approach [SWS02]. The packets of four rays are used to reduce the memory bandwidth as the rays in the packet always perform the same memory requests. This multi-threading and packet-based approach performs very well because of the high coherence between adjacent rays. The memory bandwidth is reduced further by using dedicated first level caches to store B-KD tree nodes (see [SWW*04, WSS05]).

The Traversal Unit is fully pipelined and completes one packet traversal step per clock cycle (throughput 1). The packet traversal algorithm always operates on one B-KD tree node per packet and requires an active vector for the packet, that indicates which rays overlap this current node. To derive a traversal decision for the packet of rays, first a traversal decision for each ray of the packet is generated in parallel. This yields 3 bits for each ray that indicate if the ray wants to traverse the first and/or the second child and in which order it wants to traverse them.

The joint packet traversal decision for the packet is computed as follows: The packet goes to the first child and/or second child if there exists an active ray in the packet that want to go there, and the packet goes from left to right if there is an active ray that wants to traverse in this order. A ray is set active in a child if it is active in the parent node and wants to traverse that child.

Rays may disagree on the traversal order but this is irrelevant for the correctness of the algorithm and only influences its performance. Thus we have no issues with inconsistent rays that cause problems in the packet traversal of KD trees [SWS02].

Our traversal stack has a depth of 32 entries and stores the address of the node to traverse later, which rays are active in that node, and the traversal interval for the node. The Traversal Unit also includes a different ray stack of depth two for storing transformed rays for instanced geometry, thus maximally one level of transformation nodes can currently be handled in our implementation.

Compared to a Traversal Unit for KD trees we need more resources in terms of memory and arithmetic units. Stack memory requirements are twice as high as an interval instead of a single scalar needs to be stored [SWW*04]. The logic complexity is four times higher because four multiplications (with the reciprocal of the direction) and four additions are required to compute the distances to the four planes versus only one plane. This higher cost pays off as the B-KD tree is much more flexible and enables the handling of most kinds of dynamic scenes.

4.5. Geometry Unit

The Geometry Unit is responsible for sequentially intersecting the rays of a packet with triangle geometry using the Möller-Trumbore algorithm [MT97], or to sequentially transform rays to the local coordinate space defined by a transformation node of the B-KD tree. This transformation mode requires no additional arithmetic units as them can be shared with the ones used for triangle intersection. The Geometry Unit is fully pipelined, thus can perform one ray triangle intersection or one ray transformation per clock cycle. Thus a number of 4 cycles are required to transform or intersect the four rays of a packet. Unfortunately, we could not fully pipeline the Geometry Unit in our implementation because of space limitations of the FPGA.

5. Prototype Implementation

Using FPGA technology, we implemented a high performance prototype of our hardware architecture with fixed-function shading capabilities running at 66 MHz. Our prototype platform uses a Xilinx Virtex-II 6000-4 FPGA [Xil03], that is hosted on the Alpha Data ADM-XRC-II PCI-board [Alp03]. The FPGA has access to a 64-bit wide DDR memory interface that can deliver a peak bandwidth of 1.0 GB/s at 66 MHz. Especially the bandwidth consuming skinning and updating benefits from this high bandwidth, while the rendering requires typically only 100 to 200 MB/s for our test scenes as caching is very effective.

The hardware description of our design is about 5300 lines

Scene	tris	DynRT	SaarCOR	RPU	OpenRT
Scene6	0.8 k	45.0	44.6	23.4	12.9
Office	34.3 k	27.9	35.9	17.2	10.4
Gael	52.5 k	14.4	18.6	9.2	8.0

Figure 5: *Performance comparision of our DynRT prototype with three different ray tracing platforms: the fixed-function SaarCOR prototype, the RPU prototype, and the OpenRT software ray tracer running on an Intel Pentium-4 with 2.66 GHz. The performance numbers of the three hardware architectures are scaled to 66 MHz and a single FPGA to compare the architecture and not the quality of the implementation. All numbers are given in frames per second and the scenes map to the images from left to right.*

of ML [MTH90] code using an ML library for hardware description [WBS]. The specification is fully parameterizable, which means that each of the parameters of the prototype, like floating point accuracy, packet size, number of threads, latencies, and caches, can be changed by simply adjusting a configuration file. We fixed the configuration to achieve the best performance with our FPGA platform.

The rendering part of the prototype architecture is widely multi-threaded, as there are 32 packets with four rays each in the system. This number of threads is sufficient to achieve comparatively high usage rates of the arithmetic units. Each arithmetic units in the chip computes with 24 bit floating point accuracy. Floating point numbers are stored in memory in IEEE32 format, thus need to be converted on reads and writes. The traversal and vertex cache are both direct mapped caches that hold 1024 B-KD tree nodes or vertices, respectively.

Because of limited space on the FPGA the Geometry Unit could not be fully pipelined. Thus we need two clock cycles to transform a ray or to intersect it with a triangle. This reduces performance as it causes the Geometry Unit to become the bottleneck during rendering. For the same reason we implemented only a very simple fixed-function shader instead of a programmable shading unit such as described in [WSS05]. Our fixed function shader uses a constant color per triangle or can shade the triangle with the barycentric coordinates of the intersection point. To evaluate the quality of the B-KD trees, we can also shade a ray according to the number of traversal steps performed per packet or the number of ray triangle intersections.

Our prototype architecture occupies about 99% of the logic cells, 65% of the block memories, and 33 of the 144 18-bit multipliers of the FPGA chip. The Traversal Unit uses about 35% of the FPGA resources while the Geometry Unit requires 41%. The Update Processor uses only 8% of the resources while the Skinning Unit is mostly shared with the Geometry Unit. The remaining resources are spend for the infrastructure, such as memory access, shading, PCI access etc. The latencies of the longest pipelines of the design are 13 stages for the Traversal Unit and 36 stages for the geometry pipeline.

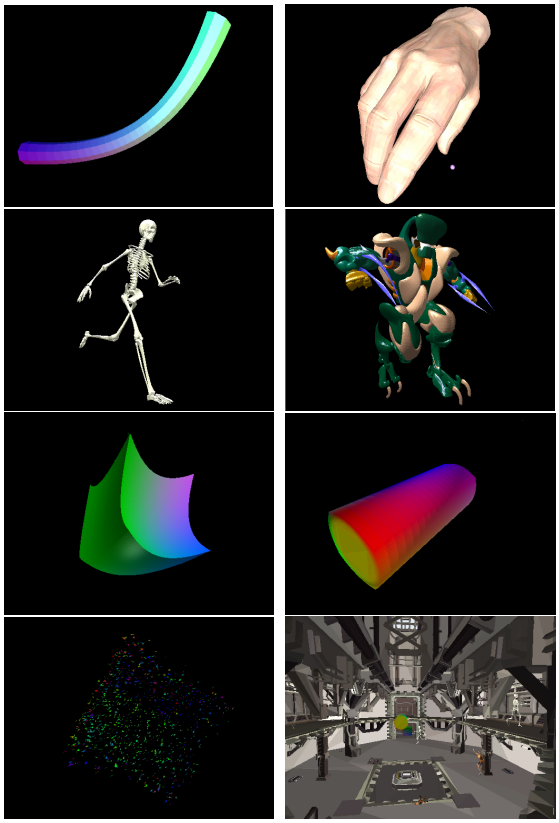

Figure 6: *Several dynamic scenes rendered in realtime with up to 35 frames per second at a resolution of 512x384 on the prototype FPGA clocked at 66 MHz: Pipe (0.5k triangles), Hand (17k triangles), Skeleton (16k triangles), Helix (78k triangles), Rotating Cube (18k triangles), Morph (4.3k triangles), Random (4.3k triangles), and the DynGael scene containing several dynamic objects (52k static + 45k dynamic triangles).*

The prototype can update a peak number of 66 million B-KD tree nodes per second and perform up to 66 million vertex matrix multiplications per second in skinning mode. The upload of scene data and the download of frame buffer data are performed by DMA data transfers. We achieve a peak data transfer rate of 94 MB/s using our configuration of a 64 bit PCI bus and the PCI bridge of the FPGA board.

Scene	triangles	vertices	skinning matrices	cycles skinning	cycles update	cycles render	relative cycles skinning	relative cycles update	relative cycles render	frame-rate
Pipe	0.5k	0.26k	2	2.3k	2.9k	1332k	0.17%	0.21%	99.6%	32.6
Hand	17k	9.3k	-	-	104k	1404k	-	6.8%	93.2%	20.9
Skeleton	16k	8.3k	-	-	107k	1414k	-	7.0%	93.0%	25.5
Helix	78k	50.1k	-	-	553k	3466k	-	13.7%	86.3%	12.3
Rotating Cube	18k	17.7k	2	133k	154k	2637k	4.5%	5.2%	90.2%	18.5
Morph	4.3k	2.1k	3	22k	26k	1359k	1.6%	1.8%	96.6%	32.3
DynGael	97k	184k	-	76k	176k	4547k	1.6%	3.6%	94.8%	13.0
Random	4.3k	12.9k	3	133k	58k	23503k	0.5%	0.2%	99.3%	2.8

Table 1: *This table shows the clock cycles required for to perform skinning and updating of the B-KD tree and for rendering in 512x384 resolution. The relative amount of cycles and the framerate (including a small driver overhead) are also shown.*

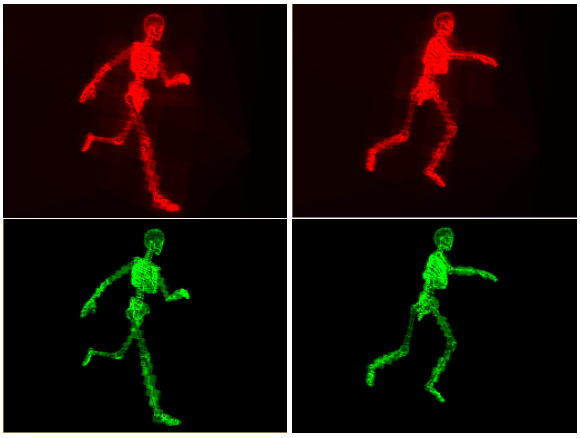

Figure 7: *The quality of the spatial index structure stays close to optimal during the entire animation, which is illustrated by these images showing the skeleton model in two different poses (from left to right). The number of traversal steps and ray triangle intersections are visualized from top down. Fully red is equivalent to 128 traversal steps while fully green stands for 128 ray triangle intersections (also see video).*

6. Results

For static scenes we compare the rendering performance of our hardware architecture against three different ray tracing engines: the fixed-function SaarCOR ray tracing prototype [SWW*04], the programmable RPU architecture [WSS05], and the OpenRT ray tracing system [WBS03]. We use three static scenes of different complexity for the comparison: Scene6 (0.8k triangles), Office (34.2k triangles), and the Gael level (52.5k triangles) from a current computer game (see Figure 5). We tried to make that comparison as fair as possible by using flat triangle shading only, the same KD tree, and the same number of FPGAs clocked at 66 MHz. For the DynRT architecture we converted the KD tree to a B-KD tree by supporting empty cells and encoding lists of triangles into small B-KD subtrees. The Figure 5 shows that the rendering performance of our architecture is slightly lower than the fixed-function SaarCOR prototype, which is mainly because of our lower intersection performance of 2 cycles per ray triangle intersection compared to 1.25 cycles per intersection for SaarCOR. For the same reason the RPU architecture performs worse than the DynRT. Compared to OpenRT the DynRT ray tracer is 2 to 3 times faster than the used Pentium-4 with 2.66 GHz.

We use a different set of benchmark scenes to evaluate the performance for dynamic scenes (see Figure 6). We use three Poser [Pos06] animations with varying complexity to evaluate the performance for game-like characters: Hand (17k triangles), Skeleton (16k triangles), and Helix (78k triangles). As we have no skinning model for these scenes, the vertex positions have been pre-computed by Poser and are uploaded to the FPGA via DMA, but updated by the hardware.

Four deforming objects show the computations of the complete animation using the skinning capabilities of the prototype: a deforming pipe (Pipe), a rotating deforming cube (Rotating Cube), a morphing sequence between a cube, a sphere, and a cylinder (Morph), and a morphing sequence between a cube, a random triangle distribution, and a cylinder (Random).

To show the combination of dynamic objects and a static environment, the DynGael scene contains the static gael environment (52k triangles), two skeleton instances (2x16k triangles), a rotating cube (4k triangles), a morph object (4k triangles), and a bouncing cube (4k triangles). The B-KD tree nodes on top of these 6 object instances are recomputed by the driver for each frame.

Figure 7 shows the number of traversal steps (red) and ray/triangle intersections (green) per pixel of the Skeleton object in two different poses. Even as the second pose is quite different from the inital one, the work per pixel does not change very much, because the B-KD tree adapts well to the geometry.

Table 1 analyzes the number of cycles required for the skinning, updating, and rendering in 512x386 resolution for each

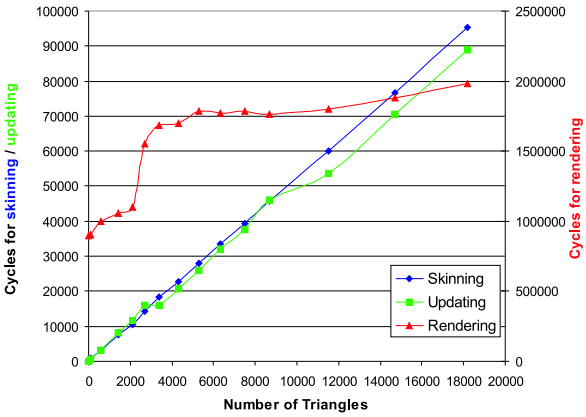

Figure 8: *This figure shows the number of clock cycles spend in skinning, updating and rendering for the morphing sequence between 3 objects computed with different geometry complexity.*

of the demo scenes. It shows that rendering is the most expensive operation requiring more than 90% of the clock cycles. This is due to the large amount of pixels that need to be processed at 512x386 resolution. One can also see that skinning is linear in the number of vertices times the number of matrices connected to them. Updating is dependent on the number of B-KD tree nodes to update (which is about 2 times the number of triangles) and the number of vertices to read.

As the Morph and the Random scene have a different number of vertices, they also have a quite different update time, despite the number of triangles are the same. As expected, the rendering time of the Random scene is the worst of all sample scenes despite it contains only 4.3k triangles. The reason is that this Random morph sequence morphs between two stages with similar structure (cube and cylinder) and a random triangle distribution whose structure does not match the initial B-KD tree structure any more. Rendering this state performs very bad, as the bounds of the B-KD nodes overlap much, and mainly the complete tree needs to be traversed per ray hitting the random distribution. Obviously, in the cube and the cylinder stage the rendering performance is high, but the table row shows the random state only.

We also analyzed the architecture for scalability with scene complexity. Figure 8 shows the number of skinning, update, and rendering cycles for the Morph object rendered with different geometry complexity. It shows that skinning and updating is linear in the number of triangles (as long as the vertex/triangle ratio is constant). One can also see that rendering depends sublinear on the number of triangles. The jump in the number of rendering cycles at 2000 triangles is due to caching effects.

The paper [WSS05] showed that a nearly linear scaling of the rendering part of a ray tracing hardware architecture for static scenes is easily possible if each rendering FPGA stores a copy of the complete scene data and renders different parts of the image. A similar speedup for the pure rendering time is also possible with our approach if only static scenes are considered.

Scaling the performance in case of many dynamic scene changes is more challenging, as these dynamic changes need to be computed by or send to the different parallel FPGAs for rendering. An approach where each FPGA computes all scene changes would only allow for scaling the rendering, not the setup time to compute the dynamics. A different approach would be to distribute the skinning and update computations to the parallel FPGAs followed by a final distribution of all updated B-KD trees and vertices to all FPGAs. This would result in a communication bottleneck as skinning and updating together produce a very high output bandwidth.

To solve this parallelization issue skinning and updating could be performed on demand during rendering. The parallel FPGAs then only compute dynamic changes that are required for the pixels rendered by them. This approach could be used if the application provides a conservative bound of the objects that are updated on demand.

7. Conclusions and Future Work

This paper presented B-KD trees, a new spatial index structure that combines the advantages of KD trees and Bounding Volume Hierarchies into a single, simple to handle, and homogeneous data structure.

B-KD trees can be used to handle dynamic scenes by maintaining the structure of the tree while updating the two bounding intervals of each node when geometry changes. We can efficiently handle all important types of dynamic motion like characters, morphing sequences, or water surfaces. Random movements of many objects are problematic as they result in large overlaps among the B-KD nodes representing these objects. The overlap is as large as the range of the movement and results in a sequential intersection with each of the involved subtrees.

B-KD trees fit well to an hardware implementation as they are more homogeneous than KD trees, making the special treatment of lists or mail-boxing unnecessary. B-KD trees can also be used as spatial index structure for static scenes, similar to KD trees.

We proposed a hardware architecture to render dynamic scenes that accelerates all expensive and time critical computations in hardware. We support skinning of meshes using a fixed-function general skinning approach, updating of the B-KD tree, and the rendering using ray tracing.

The novel spatial index structure together with the hardware architecture closes the largest remaining gap between

ray tracing and rasterization hardware: the handling of dynamic scenes. The techniques presented in this paper brings ray tracing very close to rasterization also in respect to the support for dynamic scenes while maintaining all the advantages of ray tracing including the direct and simple handling of global effects. We are confident that fast, hardware accelerated ray tracing will become widely available, as it provides a robust, easy to use, and powerful basis for advanced 3D graphics.

References

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

[App68] APPEL A.: Some Techniques for Shading Machine Renderings of Solids. *SJCC* (1968), 27–45.

[Arv88] ARVO J.: Linear-time voxel walking for octrees. *Ray Tracing News (available at htpp://www.acm.org/tog/resources/RTNews/html/rtnews2d.html 1*, 5 (Mar. 1988).

[ASR98] ANDREA SANNA P. M., ROSSI M.: *A Flexible Algorithm for Multiprocessor Ray Tracing*. Tech. rep., 1998.

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*. Eurographics Association, 1987, pp. 3–10.

[CWBV83] CLEARY J., WYVILL B., BIRTWISTLE G., VATTI R.: A Parallel Ray Tracing Computer. In *Proceedings of the Association of Simula Users Conference* (1983), pp. 77–80.

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques* (1980), ACM Press, pp. 124–133.

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *HWWS '05 Proceedings* (2005), ACM Press, pp. 15–22.

[FvDFH97] FOLEY, VAN DAM, FEINER, HUGHES: *Computer Graphics – Principles and Practice, second edition in C*. Addison Wesley, 1997.

[Gla84] GLASSNER A. S.: Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics 30*, Annual Conference Series (1996), 171–180.

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

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

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

[Jan86] JANSEN F. W.: Data structures for ray tracing. In *Proceedings of the workshop on Data structures for Raster Graphics* (1986), pp. 57–73.

[Kaj86] KAJIYA J. T.: The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (1986), vol. 20, pp. 143–150.

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (/1998), 21–36.

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

[KK86] KAY T. L., KAJIYA J. T.: Ray Tracing Complex Scenes. *Computer Graphics (Proceedings of ACM SIGGRAPH) 20*, 4 (1986), 269–278.

[KR00] KALTE P., RÜCKERT: *Using a Dynamically Reconfigurable System to Accelerate Octree Based 3D Graphics*. Tech. rep., System and Circuit Technology, University of Paderborn, 2000.

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

[MF99] MÜLLER G., FELLNER D. W.: *Hybrid Scene Structuring with Application to Ray Tracing*. Tech. rep., Institute of Computer Graphics, TU Braunschweig, Germany, Mai 1999.

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

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

[MTH90] MILNER R., TOFTE M., HARPER R.: The Definition of Standard ML, 1990.

[MTLT88] MAGNENAT-THALMANN N., LAPERRIERE R., THALMANN D.: Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics interface '88* (Toronto, Ont., Canada, Canada, 1988), Canadian Information Processing Society, pp. 26–33.

[MTT91] MAGNENAT-THALMANN N., THALMANN D.: Human body deformations using joint-dependent local operators and finite-element theory. 243–262.

[Neb97] NEBEL J.-C.: A Mixed Dataflow Algorithm for RayTracing on the CRAY T3E. In *Third European CRAY-SGI MPP Workshop* (September 1997).

[PG95] PALMER I. J., GRIMSDALE R.: Collision detection for animation using sphere trees. In *Computer Graphics Forum, 14(2)* (1995), pp. 105 – 116.

[Pos06] POSER: Poser Web Page. *http://www.e-frontier.com* (2006).

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

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

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

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH).

[RW80] RUBIN S. M., WHITTED T.: A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[SF90] SUBRAMANIAN K. R., FUSSEL D. S.: *A Search Structure based on K-d Trees for Efficient Ray Tracing*. Tech. Rep. PhD Dissertation, Tx 78712-1188, The University of Texas at Austin, Dec. 1990.

[Smi98] SMITS B.: Efficiency Issues for Ray Tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14.

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

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware* (2004).

[TL03] THOMAS LARSSON T. A.-M.: *Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models*. Tech. rep., February 2003.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using AABB trees. In *Journal of Graphics Tools, 2(4)* (1997), pp. 1–14.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS] WOOP S., BRUNVAND E., SLUSALLEK P.: HWML: RTL/Structural Hardware Description using ML. to appear.

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

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

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *CACM 23*, 6 (1980), 343–349.

[WPS*03] WALD I., PURCELL T. J., SCHMITTLER J., BENTHIN C., SLUSALLEK P.: Realtime Ray Tracing and its Use for Interactive Global Illumination. In *Eurographics State of the Art Reports* (2003), pp. 85–122.

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *SIGGRAPH 2005 Conference Proceedings* (2005), pp. 434 – 444.

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