GPU-based Ray Tracing of Dynamic Scenes

Martin Reichl   Robert Dünger   Alexander Schiewe
Thomas Klemmer   Markus Hartleb   Christopher Lux   Bernd Fröhlich

Prof. Dr. rer. nat. Bernd Fröhlich
Bauhausstr. 11
Fakultät Medien
Bauhaus-Universität Weimar
99423 Weimar
Tel.: +49 (0) 3643 - 58 3732
FAX: +49 (0) 3643 - 58 3709
E-Mail: bernd.froehlich@medien.uni-weimar.de

Abstract: This paper presents the design and implementation of a GPU-based ray tracing system for dynamic scenes consisting of a set of individual, non-deformable objects. The triangles of each object are organized in a separate Kd-tree. A bounding volume hierarchy (BVH) is built on top of these Kd-trees. The BVH is updated and uploaded into GPU memory on a frame-by-frame basis, whereas the Kd-trees are uploaded only once. We use a single ray tracing kernel for handling all ray generations. Code execution path divergence is limited by the use of a ray stack, which treats all ray types in the same way. We make effective use of the very limited high bandwidth memory of the GPU's multiprocessors by using a smart stack for the acceleration structure traversals. The results show that our GPU implementation of a two-level acceleration approach performs between 40 and 105 percent as fast as a single Kd-tree containing the entire scene.

Keywords: ray tracing, multi-level hierarchy, animation, dynamic scenes

1 Introduction

Interactive ray tracing of non-trivial scenes is just becoming feasible on single graphics processing units (GPU). Recent work in this area focuses on building effective acceleration structures, which work well under the constraints of current GPUs. Most approaches are targeted at static scenes and only allow navigation in the virtual scene. So far support for dynamic scenes has not been considered for GPU implementations.

We have developed a GPU-based ray tracing system for dynamic scenes consisting of a set of individual objects. Each object may independently move around, but its geometry and topology are static. We use a two-level acceleration structure for this constrained scenario similar to the approach taken by Wald et al. [WBS03]. Instead of using Kd-trees for both levels, we organize the individual objects in a bounding box hierarchy (BVH), which is built on top of Kd-trees for the individual objects. On the CPU the BVH is rebuilt every time the position of an object changes and updated in GPU memory. Our implementation uses a single kernel on the GPU to handle

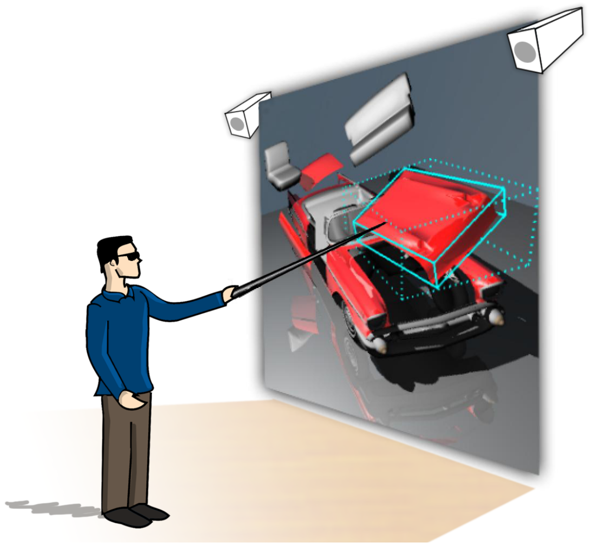


Figure 1: Our work is motivated by assembly planning and product evaluation applications. In these scenarios the interactive manipulation of individual objects is an important requirement.

all ray generations, except the primary rays which are handled by a separate rasterization pass. The implementation on the GPU requires particular attention to limit the divergence of the code execution paths across multiple rays. We use a ray stack to avoid branching into particular cases for the treatment of reflection, refraction and shadow rays. The GPU's multiprocessors only provide a very limited amount of extremely fast on-chip memory, which is important for the stack-based traversal of the BVH and the Kd-trees. Our smart stack uses on-chip memory as long as possible and overflows into the slower global memory only when necessary.

Ray tracing is an important technology for virtual environments since it greatly improves the visual quality and enhances depth perception. In virtual environments, we mostly deal with a set of individual objects, which can be translated, rotated and otherwise manipulated. This paper reports on the experiences of our GPU-based ray tracing system for such dynamic scenes consisting of individual objects or parts. We analyze the behavior of our implementation in detail and discuss the advantages, disadvantages and limitations of a two-level acceleration structure. We have also compared the performance of our BVH/Kd-tree combination to a single Kd-tree implementation, which cannot be rebuilt on a frame-by-frame basis and thus does not support dynamic objects. The results show that the two-level hierarchy performs between 40 percent and 105 percent as fast as the pure Kd-tree implementation. While this is an encouraging result, it still leaves room for improvement of the data structures and traversal approaches for dynamic scenes.

2 Related Work

First approaches to utilize GPUs for accelerating ray tracing were limited by the capabilities of the early generations of programmable graphics processors. With the Ray Engine Carr et al. [CHH02] were the first to implement a real-time ray tracing technique augmented by the GPU. In their approach the GPU was only used for ray-triangle intersections. The performance of this approach was limited by the required memory transports for geometry download and intersection results upload. At the same time Purcell et al. [PBMH02] simulated the use of the GPU as a stream processor to enable the implementation of a complete ray tracing algorithm on a GPU. This approach decomposed the ray tracing algorithm into smaller subtasks (i. e. ray generation, traversal, intersection tests, shading) which were processed in multiple rendering passes. A regular grid was chosen as the acceleration structure due to simple traversal computations.

For static scenes the Kd-tree is one of the most efficient acceleration structures [Hav00], in particular if its construction is based on a surface area heuristic (SAH) [**?**] to minimize traversal costs. Traversal of such advanced hierarchical data structures requires the use of a stack which is still difficult to efficiently implement on todays GPUs. Foley and Sugerman [FS05] presented two techniques for stackless Kd-tree traversal (kd-restart, kd-backtrack) which require a number of redundant traversal steps. To reduce this traversal overhead, Horn et al. [HSHH07] added a small fixed-size stack. Recently, Popov et al. [PGSS07] introduced a stackless, GPU-based Kd-tree traversal algorithm, which requires significantly less traversal steps than stack-based methods or kd-restart. With the additional storage of links to adjacent nodes called *ropes* a high amount of down-traversal steps are avoided since traversal may start at a leaf node.
Besides Kd-trees, bounding volume hierarchies (BVH) can be used as acceleration structures on the GPU. Thrane and Simonsen [TS05] presented a stackless BVH traversal algorithm, which can be efficiently implemented on the GPU. Their approach outperformed stackless kd-restart and kd-backtrack traversals on moderately sized scenes. Recently, Günther et al. [GPSS07] presented a BVH-based packet traversal algorithm using a shared stack. They were able to achieve near real-time results for large static scenes.

Recent work on ray tracing of animated and interactive scenes mostly focused on acceleration structures, which can be quickly build or rebuild [WMG$^+$07]. Wald and Havran [WH06] showed how to rebuild a BVH for the entire scene on a per-frame basis. Yoon et al. [YCM07] presented a technique for locally restructuring parts of a BVH instead of rebuilding the entire structure, which works well if only small portions of the scene are manipulated. Other approaches used nested or multi-level hierarchies where a top-level hierarchy maintains only movable scene objects while an efficient acceleration structure is used as a low-level hierarchy holding the object geometries. This allows for manipulation of scene objects while avoiding unnecessary reconstruction of acceleration structures for the non-deformable scene objects. Lext and Akenine-Möller [LAM01] showed how to use a grid as a top-level hierarchy which allows very fast rebuilds. A similar approach was taken by Wald et al. [WBS03] using Kd-trees for both the top-level and low-level hierarchies. While all these techniques for dynamic or animated scenes focus on CPU implementations, we have explored the use of multi-level hierarchies for dynamic scenes in the context of GPU-based real-time ray tracing.

3 Two-Level Hierarchy

Common virtual reality applications allow the interactive manipulation of objects or parts of objects in the scene. Most manipulations do not change the shape of objects and in most cases only

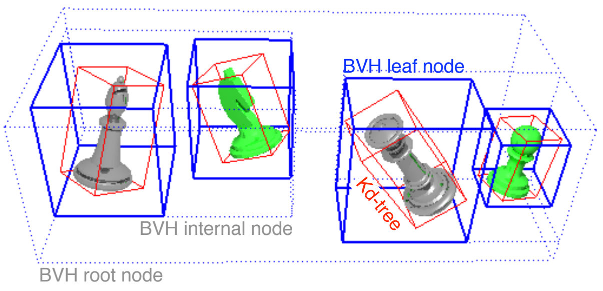


Figure 2: A two-level hierarchy consisting of a BVH maintaining the top-level scene structure while static scene geometries in the leaf nodes are organized in Kd-trees.

rigid body transformations are applied to individual objects. Our goal is to support such scenarios. Assembly planning in the automotive industry is a typical example, which requires the manipulation of individual car and engine parts. In a typical scene graph-based scenario objects are organized in hierarchical structures to achieve hierarchical transformations. In the following we will only consider scenes consisting of a set of individual objects, where each object may be associated with an affine transformation. Each object is static with respect to its mesh connectivity and geometry. Most scene graph hierarchies can be flattened to create the required structure. This allows the use of efficient acceleration structures on the level of static scene objects while maintaining the dynamic scene structure in a separate top-level acceleration structure. This approach of using multiple nested acceleration structures of potentially different types is typically called multi-level hierarchy (TLH) [LAM01, WBS03]. In our implementation, we use only a two-level hierarchy (TLH) of acceleration structures, which also allows single-level instancing, but does not support multi-level instancing schemes.

Using a two-level hierarchy allows to combine acceleration structures offering different characteristics with respect to their creation time. Choosing an acceleration structure, which allows for the most efficient ray traversals at the cost of extended build time, is favorable for the static object geometries. The acceleration structures for these objects remains unchanged after the initial build process and therefore only needs to be constructed once. This enables us to employ algorithms generating traversal-cost optimized structures in a preprocess for each interactive scene object. In contrast, due to user interaction with the scene, the top-level structure needs to be potentially rebuild on a frame-to-frame basis. Therefore it requires an acceleration structure with very fast reconstruction or restructuring times at the expense of being less optimized for ray traversals.

For the GPU-implementation of the two-level hierarchy we chose to combine a bounding volume hierarchy (BVH) of axis-aligned bounding boxes (AABBs) used as the dynamic top-level hierarchy with Kd-trees used for organizing the static object geometries (cf. figure 2). Both structures are constructed utilizing a surface area heuristic (SAH) cost estimation to minimize the costs for traversal and intersection tests. The top-level BVH is reconstructed from scratch every time the scene structure is changed through user input or animation. On these events, we employ a method for fast BVH rebuilds based on the technique presented by Wald et al. citeWald:07:RTDSBVH.

Note, that the reconstruction is done entirely on the CPU.

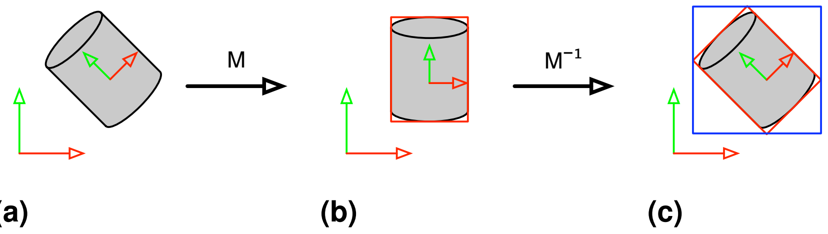


Figure 3: Initial transformations are computed for all static scene objects which results in more tightly fitting bounding boxes in the object's coordinate system. Figure **(a)** shows the object orientation in global space which is computed in a preprocess. Figure **(b)** shows the built Kd-tree and AABB in object space. The resulting Kd-tree OBB and the BVH AABB in global space are shown in figure **(c)**.

In a pre-process we compute oriented bounding boxes (OBB) for the individual scene-objects using Gottschalk's fitting technique [GLM96] and assign an initial transformation matrix for each object to place it appropriately in the global coordinate system. The Kd-trees for the individual scene objects are built in the local coordinate system of each object. The BVH is built in top-down order based on axis-aligned bounding boxes around the OBBs of the individual objects (cf. figure 3) on a frame-by-frame basis. Each leaf node of the BVH holds a transformation matrix and a reference to the contained Kd-tree. Thus instancing is directly supported which avoids unnecessary geometry replications.

The top-level BVH structure needs frequent updates in GPU-side memory, whereas the static Kd-tree structures containing the actual scene geometries are uploaded only once. We chose a full binary BVH layout where each inner node has exactly two descendants. For a fixed scene size the BVH data structure has a constant memory footprint. Thus the memory has to be allocated only once on the GPU side avoiding memory fragmentation and minimizing memory management during run-time. The following section will describe details of our implementation.

4 Two-Level Hierarchy on the GPU

For the GPU implementation we chose to use NVIDIA's CUDA API [NVI07] for directly accessing the compute features of the NVIDIA G80 family of GPUs. This enables us to take advantage of particular hardware features which are not accessible through common graphics APIs. This section introduces the characteristics of the Compute Unified Device Architecture (CUDA) before describing our TLH GPU implementation in detail. Data layout and algorithmic details (e. g. traversal, ray generation, stacks) are presented as well as memory access optimizations introduced by our approach.

4.1 G80 Architecture & CUDA

Today's GPUs may be viewed as highly parallel streaming architectures. More precisely, they are multi-processor/multi-alu machines. Each multiprocessor can be classified as a Concurrent Read, Concurrent Write Parallel Random Access Machine (CRCW PRAM) [PGSS07].

CUDA enables direct access to the compute capabilities of G80+ GPUs without the requirement to express algorithms in terms of graphic primitives such as triangles or textures. Using CUDA general memory scattering operations have finally become possible, while they are still very limited with current high-level shading languages. Scattering allows writing to arbitrary memory locations, and therefore enables more flexible algorithm implementations and in particular the use of stacks and other dynamic data structures. CUDA programs are expressed as so called kernels which are executed in chunks of threads that are running in parallel. These chunks are called warps[1] which again are grouped into blocks running on individual multiprocessors. These blocks share all resources of a multiprocessor. However, the atomic scheduling unit remains a warp which executes threads in SIMD fashion.

The G80 architecture exposes different types of memory with highly different bandwidth and latency characteristics. All multiprocessors share direct uncached access to the global device memory. While this is suited for intra- and inter-processor communication memory fetches suffer from a relatively high latency. An alternative way to access global memory is through the CUDA texture interfaces, which speeds up recurring and spatially coherent fetches from global memory regions by using an on-chip cache. In addition, there are constant and shared memory regions attached to each multiprocessor. These memories exhibit high bandwidth and very small access latencies. Finally, threads have access to a set of dedicated registers. Each multiprocessor possesses only a limited set of overall resources for constant, shared and register memory. As a result the requirements of multiprocessor resources per kernel limits the amount of runnable (active) threads per block.

4.2 Ray Tracing Kernel

We employ a single kernel for implementing the complete ray tracing algorithm including acceleration structure traversal, ray-triangle intersection, shading and secondary ray generation. For each pixel a thread is created which executes the ray tracing kernel. We use a rasterization pass to generate the intersections for the primary rays to accelerate the ray tracing system. The results of this rendering pass are used by the ray tracing kernel to compute secondary effects and shading.

4.2.1 Data Layout

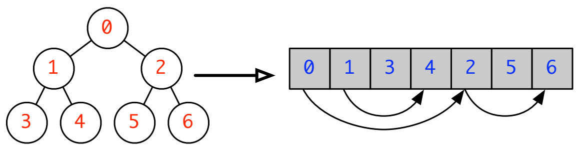


Figure 4: Depth-first left ordered binary tree serialization.

.

[1] On current GPUs the warp size is 32 threads.

For GPU-ray tracing the geometry as well as the acceleration structures are serialized and uploaded to the GPU global memory. We employ the CUDA texture interface to benefit from cached memory access. The BVH and the Kd-tree structures are serialized in depth first order as shown in figure 4. Using this memory layout for binary trees the first child of each inner node is located right next to the parent requiring only a single link per node pointing to the second child. As a result, for any node fetched from the global memory the first child is potentially placed in the cache. Special handling of null pointers is not required since we use a binary tree layout for the BVH as well as for the Kd-tree (i. e. each inner node has exactly two child nodes).

The Kd-tree leaf nodes store only references to individual triangles to prevent triangle replication. The Kd-tree node layout strictly follows the 8 byte scheme proposed by Wald [Wal04]. In contrast, a BVH node needs to additionally store references to a transformation matrix and an axis-aligned bounding box requiring 32 byte per node.

The CUDA-OpenGL interoperation layer provides only limited support for shared data usage. Unfortunately, the scene geometries need to be duplicated to be available for the primary OpenGL rasterization pass and for the actual CUDA-based ray tracing. Furthermore, since the offscreen buffers which holding the primary ray intersection results are updated each frame, they need to be transferred from dedicated OpenGL memory to CUDA-mapped memory each frame which is an unexpected bottleneck with current drivers[2].

4.2.2 Primary Ray Generation

To take advantage of the rasterization power of current GPUs we employ a rasterization pass to generate the triangle index and intersection point for each primary ray. The scene is rendered using OpenGL using a fragment program which writes out triangle indices, interpolated barycentric coordinates and a Kd-tree reference. We also use the BVH structure to render the scene objects in front-to-back order to take advantage of hardware-supported early-z culling. The CUDA ray tracing kernel reads back the per-pixel information from the rasterization pass to reconstruct the exact intersection point p. Since p is defined in the Kd-tree coordinate system it is then transformed into the world coordinate space.

4.2.3 Shading

After the reconstruction of the primary intersection point the shading equation is applied to this point. This implies fetching associated materials, normals and texture coordinates from global memory. As CUDA is currently not supporting the use of indexed texture objects we use a texture atlas and extend the material structures with a texture index. The texture coordinates are on the fly transformed into the actual atlas texture coordinates by using a texture description table residing in on-chip constant memory.

[2] We measured an effective bandwidth of 7GiB/s which is at around 10% of the maximum bandwidth capabilities of an NVIDIA GeForce 8800GTX GPU.

4.2.4 Secondary Ray Traversal

Subsequently to primary intersection point shading secondary rays are generated and traced through the TLH. The SIMD GPU execution model leads to sets of implicitly generated ray packets (warps) of secondary rays, which are handled in parallel. Each thread in a warp executes the same instructions at the same time, since all the threads in a warp are executed on a single multiprocessor in SIMD fashion. In case of conditional branching the whole warp is executing all required executions paths by masking individual threads as inactive. Thus it is particularly important to avoid too many possible execution paths. We use a ray stack in global memory to maximize the number of active threads. Each generated ray and the corresponding ray parameters are pushed onto the stack which is then processed according to listing 1. Thus the code is not branching for the different ray types and cases for secondary rays. Instead all secondary rays basically follow the same execution path which limits the number of inactive threads per warp. Nevertheless, if no reflective or transparent material is hit these threads cannot do meaningful work.

```
while (!stack.empty()) { // secondary ray stack
 Ray ray = stack.pop();
 Result hit = traverseTLH(ray);
 if (!hit) continue;

 if (isTransparent(hit))
  stack.push(refract(ray));
 if (isReflective(hit))
  stack.push(reflect(ray));

 Result shadow = traverseTLH(toLight(hit));
 color += shade(ray.intensity, shadow, hit);
}
```

Listing 1: Secondary rays are processed using a ray stack instead of branching into different cases for shadow, reflection and refraction rays. The ray stack limits code execution divergence.

The secondary rays traverse through the TLH structure in the same way as it would be implemented on the CPU. For each intersected BVH node intersections with the two child nodes are calculated and depth sorted. Then the traversal descends into the BVH following the first hit child node (cf. 5a). In case of BVH leaf nodes the ray is transformed to the coordinate system associated with the referenced Kd-tree. Before traversing the actual Kd-tree the axis-aligned bounding box of the Kd-tree is intersected to avoid unnecessary traversals. After returning from the Kd-tree traversal, the remaining sub-trees with potential intersections closer to the ray origin than a potentially found intersection point are processed.

4.3 Memory Access Optimization

We employ two techniques to reduce the cases where we have to wait for data to be fetched from global memory: At first, we store the geometry data in separate lines of a 2D texture to increase texture cache utilization. But as others, we observed that cache hits become more and more unlikely with subsequent ray generations. The performance advantage of using the small 16Kb texture cache is split among all uses such as storing geometry data, acceleration structures as well as materials thus it may be negligible in the end.

Scene	BVH			Kd		
	avg	max	var	avg	max	var
BART robots	1.0	6	1.33	1.40	7	1.09
BART kitchen	0.01	1	0.01	3.63	10	1.14
The chevy	0.72	5	1.1	0.22	5	0.37
Chess	0.27	2	0.39	1.53	6	1.02

Table 1: Actual maximum stack usage for the two acceleration structures used in the approach. Data is taken for all primary rays. The maximum allowed tree depth for the Kd-tree is 20, the depth of the BVH is $log_2(n)$ where n is the number of objects.

The second method for memory access optimization is the use of a *smart stack*, which uses on-chip shared memory as long as possible and overflows into global memory if necessary. While

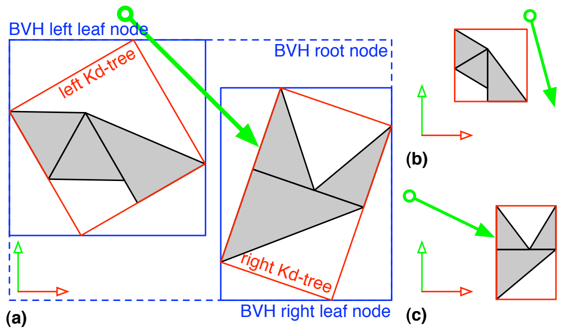
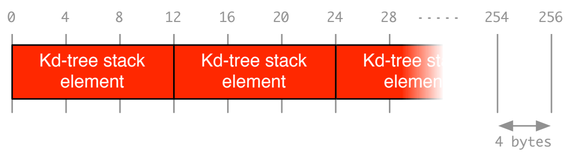
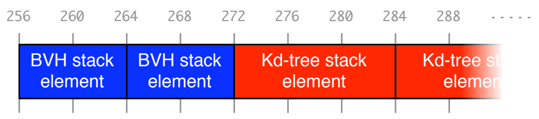


Figure 5: Detailed schematic view of the TLH traversal. Figure **(a)** Shows the top-level BVH within the global coordinate system. During TLH traversal the ray intersects both BVH child nodes therefore it has to be transformed to both local Kd-tree coordinate systems. Figures **(b)** and **(c)** show the traversal of the Kd-trees in their local coordinate space.



(a) upper part of the stack is stored in fast shared device memory



(b) oldest stack elements are stored in slow global device memory

Figure 6: Per thread mixed BVH/Kd-tree stack layout. Each BVH stack element takes 8 bytes of stack memory for its corresponding node index and the ray parameter for the nearest intersection point with the AABB. Kd-tree stack elements need additional 4 bytes to store the ray parameter for the exit intersection point. The stack in the global memory will only be used as a swap region for the oldest stack elements if the maximum size of the shared memory is reached.

a maximum stack size of the height of the tree can be required, the average stack utilization is significantly lower as can be seen in Table 1. Therefore, Horn et al. [HSHH07] introduced a small, fixed-size stack called short-stack. In their implementation, if the short-stack underflows they fall back to a strategy called kd-restart, which restarts tree traversal with a shortened ray. If the stack overflows, they simply overwrite the oldest entry. We adapted the concept of the short-stack to create a fast-access version of the BVH and Kd-tree stacks which reside in the on-chip shared memory of the multiprocessor. As our kernel implementation allows for an active thread count of 64 threads per block there are 256 bytes of shared memory per thread available. This would allow to implement for either a short BVH stack of 32 elements or a short Kd-tree stack of 21 elements. Since both stacks are not used at the same time, we have adapted the stacks to work in the same memory region. This is possible since the Kd-tree stack is always empty as a thread runs BVH traversal. During Kd-tree traversal updates of the BVH stack do not occur. Figure 6 shows the memory layout and addressing scheme used. If the short stack would grow beyond the memory region dedicated to a thread, this thread uses a second memory region in the global memory as a swap area for the oldest stack elements. When reaching the size limit of the short stack, the oldest element respectively the two oldest elements are transferred to the swap area and this space in shared memory is used for the new stack entry. Once the shared memory stack runs empty, the youngest stack element from the swap area is popped back to the shared memory region. Note that if any thread of a warp has to fall back to global memory, all threads in the warp are punished with high memory latencies due to the swapping operations to global memory. However, these cases are extremely rare. For the examples in Table 1 they did not occur.

We also considered to put the ray stack for secondary ray handling into shared memory. As this memory region is clearly limited in size, this approach would constrain the memory dedicated to the smart stack even more. Since secondary ray stack operations occur considerably less often than traversal stack operations the speedup of the smart ray stack was negligible.

5 Results and Discussion

In this section we will present and discuss the results of the conducted performance analysis to evaluate the impact of the TLH approach compared to using a single Kd-tree for the entire scene. The chosen test scenes (cf. figure 7) exhibit different characteristics regarding rigid object count, polygon count and the spatial distribution of objects. Especially the BART scenes provide different stress scenarios for ray tracing of dynamic scenes like the ”teapot in a stadium” problem, hierarchical animations and varying frame-to-frame coherence. Additionally, the chevy scene contains multiple objects with overlapping bounding volumes which requires a large percentage of the rays to traverse more than one object Kd-tree. The chess scene has been chosen because of a uniform distribution of objects which show no bounding volume overlaps as well as nearly the same triangle count as the chevy scene.

The tests were conducted using a Intel Core2 Duo 2.4GHz workstation with 2GiB RAM and a NVIDIA GeForce 8800GTX graphics board running CUDA 1.1 (driver version 174.55) under

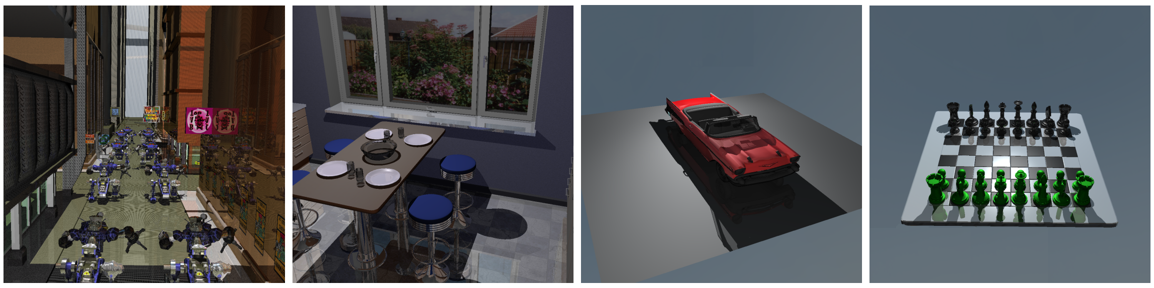
Windows XP. All tests were conducted using an image resolution of 512x512 pixels.

5.1 Performance Results

Table 2 shows our performance measurements for the test scenes under different ray recursion depth configurations. We also show the primary ray shading performance to give an estimate of the baseline performance without ray traversals. These numbers include the OpenGL rasterization pass as well as the transfer of the results to CUDA mapped memory and the intersection point reconstruction and shading performance of the CUDA kernel. For our scenes the rasterization approach is up to six times faster than tracing primary rays, which is due to the relatively small number of triangles (only up to 110k). As noted in section 4.2.1 this approach is still limited by CUDA-OpenGL inter operation constraints and surprisingly low device memory transfer performance, which is expected to be improved with newer CUDA and driver revisions.

Scene		Primary Shading	Shadow Only	Reflection Only	5 Bounces + Shadow
Robots	Kd	123.21	11.45	6.02	1.18
	TLH		8.83	4.32	0.77
Kitchen	Kd	125.02	11.55	13.05	2.53
	TLH		12.78	13.61	2.8
Chevy	Kd	208.53	55.26	52.6	26.53
	TLH		23.03	26.45	11.19
Chess	Kd	230.51	45.9	27.08	7.89
	TLH		38.43	22.26	6.55

Table 2: Performance comparison for the different test scenes using either the TLH or a single Kd-tree for the entire scene. The results are measured in frames per second for different depths of the ray tree. Primary ray shading only involves the rasterization pass and a single kernel pass for reconstructing and shading the primary intersection point.



(a) BART robots **(b)** BART kitchen **(c)** Chevy **(d)** Chess

Figure 7: Test scene overview. **(a)** 162 objects, 110K triangles **(b)** 6 objects, 71K triangles **(c)** Chevy, 79 objects, 43K triangles **(d)** Chess, 34 objects, 46K triangles

The rebuild time for the BVH is significantly influenced by the number of dynamic objects in the scene. Our SAH-based implementation, which is similar to Wald et al. [WBS07], exhibits $O(nlog^2n)$ runtime behavior and allows us to build a BVH for 1000 objects in 9.8 *ms*. As the test scenes consist of considerably less dynamic objects the BVH rebuild time is negligible to performance considerations. As we are employing a already published construction algorithm more comprehensive performance data can be found in the original publication.

The performance of the TLH approach turns out to be an almost constant fraction of the performance of the Kd-tree implementation for a specific scene. While Wald et al. observed an overhead of 10 to 20% comparing an TLH to a static Kd-tree in a CPU environment [WBS03] we observe performance penalties of 10 to 60% for the TLH approach depending on scene characteristics. Interestingly, the chevy model runs only at 42% and the BART robots scene achieves 65% of the Kd-tree performance while consisting of about twice as much dynamic objects. The reason for the limited peformance of the chevy model is the significant number of overlapping bounding volumes of the various car parts.

In case of overlapping bounding volumes the probability increases that more than one Kd-tree needs to be traversed for finding the first intersection point of a ray. Generally, two extreme cases can be observed: On the one hand, without overlap the BVH is able to separate two children and no traversal overhead is observed (as evident in the BART kitchen scenario). On the other hand, in case of complete overlap the BVH degenerates to a linear list whose children all need to be processed. The average case between these two extrema profits from the BVH early-out techniques as outlined in section 4.2.4. The issue of overlapping bounding volumes can be reduced by using oriented bounding boxes. However, a complete avoidance is often not possible.

The chevy model consists of densely arranged objects which results in many overlapping bounding volumes and thus the lowest performance ratio compared to other scenes consisting of even more dynamic objects. Figure 8 illustrates the amount of Kd-tree traversals measured for the primary shadow rays comparing the chevy model and the BART robots scene. It is evident that the amount of rays traversing four or more Kd-trees before finding the correct intersection is much higher for the chevy scene than for the robots scene. The average number of Kd-tree traversals for the chevy scene is 3.44 in contrast to 1.3 in the robots scene, which clearly explains the lowest performance ratio for the TLH approach compared to a single Kd-tree for the entire scene.

5.2 GPU Utilization

We found three primary factors influencing the performance of our GPU ray tracing algorithm: the number of memory accesses, the active GPU occupancy and the code execution divergence.

We found the biggest influence on performance is the large number of memory accesses needed for the traversal of the TLH structure (e. g. bounding volumes, transformation matrices) as well as the access to the actual geometry data for intersection tests and shading (e. g. vertices, normals, texture coordinates). Due to the chosen serialized memory layout for binary trees (cf. section 4.2.1) access to the nodes of a BVH or a Kd-tree is more coherent than access to geometry data. This is the case since we chose a binary tree structure for the BVH and the Kd-tree and store the first

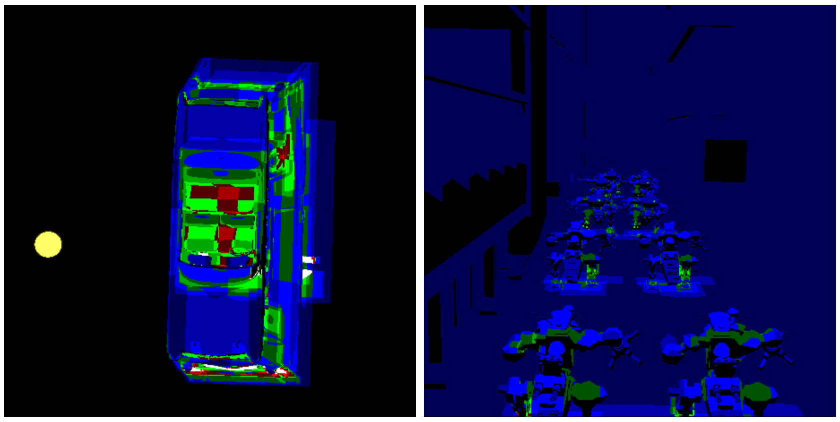


Figure 8: These images illustrate the amount of Kd-tree traversals for the first shadow ray traversal for the chevy scene and the BART robots scene. Blue colors indicate up to three, green colors up to six, red colors up to nine and white color more than nine Kd-tree traversals per ray.

child with the parent node. Thus the acceleration structures make more efficient use of the texture cache than the geometry data. An experiment selectively disabling the texture cache interface confirmed this assumption. Disabling the texture cache for the acceleration structures resulted in a 20% decrease in performance while disabling the texture cache for the geometry data showed no noticeable effect. Thus there is clearly potential for improving the memory layout for the geometry data by optimizing it with respect to the layout of the Kd-trees.

We analyzed the ratio of clock cycles spent for actual computations in comparison to waiting cycles. We found that on average 70 to 75% of the overall clock cycles account to memory access waiting cycles clearly dominating the actual computation time per thread. Typically, such waiting cycles can be hidden by the GPU scheduler by swapping blocked with runnable warps. The amount of runnable warps depends on the amount of active threads on a multiprocessor. As stated before, the number of active threads on a multiprocessor is limited by the kernel's requirements on multiprocessor resources (i. e. registers and shared memory). The kernel of our TLH traversal approach requires 96 registers limiting the number of active threads per block and multiprocessor to 64. This results in only two active warps per multiprocessor. Due to this fact, the memory access latencies can not be effectively hidden and the multiprocessors have to be stalled. Experiments using an early beta version of the CUDA 2.0 API resulted in an reduction to 59 registers for the same kernel. This allows to double the number of active warps per multiprocessor enabling the scheduler a more effective hiding of memory access latencies. First tests showed an increase in performance of about 30% for all our test scenes.

The SIMD execution of all threads belonging to a warp further reduces the GPU utilization since individual threads may be following different code execution paths. While bad branching behavior is not a new phenomenon on programmable GPUs [HSHH07], the problem will get even worse as growing warp sizes are an expected way to raise computation power of future GPUs. We analyzed the actual code execution divergence[3] of our TLH implementation using the NVIDIA Visual Profiler [NVI07]. We found that the divergence is around 10% for the BART robots scene

[3] Defined as the relation of the number of divergent branches divided by the number of branches.

and around 5% for the chevy model using a ray recursion depth of three. This indicates that TLH traversal using the global ray stack does not introduce large divergences. Sadly, the profiler tool does not provide any information about how large the run-time penalties are for the existing divergence numbers.

5.3 Scalability

Our approach rebuilds the top-level BVH on a frame-to-frame basis, which clearly limits the number of dynamic objects in the scene. When only manipulating small portions of the scene it might not be necessary to rebuild the entire BVH structure. Yoon et al. [YCM07] presented a technique to locally restructure parts of a BVH. Ize et al. [IWP07] showed how to asynchronously rebuild a BVH over multiple frames while relying on refitting for the intermediate BVHs. The use of a parallel build algorithm may increase the number of dynamic objects even further.

Since the BVH and the Kd-trees are binary trees, doubling the number of scene objects and triangles results in only one additional traversal step. While this behavior is quite acceptable scalability is additionally limited by the size of the fast on-chip memory for the stack usage and potentially also less coherent access to tree nodes and geometry data.

Another problem are large scene objects whose bounding volumes are very frequently intersected by smaller objects. The BART kitchen and robots scenes are an example for such scenarios. They contain large background objects which contain the smaller dynamic objects. This typically leads to unnecessary multiple decents into the Kd-tree hierarchies. This problem can be solved by subdividing the large scene objects into smaller parts which are sorted into the top-level hierarchy. Even if this does not completely remove the intersection of bounding volumes it should result in significantly less unnecessary decents at the cost of more top-level objects.

During BVH construction we split the set of objects until every leaf contains exactly one object. One benefit of using a SAH-based tree construction is that object subdivision is canceled if a subdivision would not improve the estimated intersection cost. Ignoring this decision criterion may possibly reduce the BVH quality. On the other hand, allowing for more than one Kd-tree being assigned to a leaf node would require each ray to iterate through a list and result in a more complicated data layout. Along with this, one more point of possible code execution divergence would be created due to the necessary loop over list elements. Nevertheless it is hard to predict the impact of this design decision.

6 Conclusion and Future Work

We have reported on the design and implementation of a two-level acceleration structure for ray tracing dynamic scenes on the GPU. We used a bounding volume hierarchy for the top level to organize the bounding boxes of the dynamic objects and encapsulated each object in a Kd-tree. The results indicate that our approach is feasible for a GPU implementation and that the performance may reach up to the performance of a single Kd-tree implementation for the entire scene. The BVH can be very quickly build, but performance may be reduced if there is much overlap between the bounding boxes of the individual objects in the scene. Our approach is applicable for specific virtual reality applications such as assembly planning or rigid body physics, which do not require deformable objects and vertex animations.

On desktop systems users alternate between view point manipulation and object manipulation since both cannot be controlled at the same time. During view point manipulation, a single Kd-tree for the entire scene seems most appropriate, but also view point coherence should be exploited. During object manipulation, often only small parts of the scene change and thus efficient update techniques for a global acceleration structure would be required. At the same time, object manipulation may change only a small subset of the light exchange paths in the scene. Kurz et al. [KLSF08] have recently shown an approach which makes use of this fact by storing ray paths and only updating those which change. We should combine such an approach with our two-level hierarchy to make use of frame-to-frame coherence.

Today there are more and more CPUs and CPU cores in combination with two, three or even more graphics cards build into a single machine. The challenge is to split the ray tracing algorithm into parts such that these abundant CPU and GPU resources are utilized in a balanced and parallel way. The increased bandwidth between CPUs and GPUs as well as approaches to place both processor types on a single die make such approaches feasible and efficient. Nevertheless, appropriate data structures for handling large dynamic scenes on such systems still need to be developed.

References

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 37–46. ACM, 2002.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*, pages 15–22. ACM, 2005.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. volume 30, pages 171–180. ACM, 1996.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of Eurographics 2007*, pages 113–118. IEEE, 2007.

[Hav00] Vlastimil Havran. *Heuristic ray shooting algorithms*. Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174. ACM, 2007.

[IWP07] Thiago Ize, Ingo Wald, and Steven G. Parker. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*. Eurographics, 2007.

[KLSF08] Daniel Kurz, Christopher Lux, Jan P. Springer, and Bernd Fröhlich. Improving interaction performance for ray tracing. In *Eurographics'08, Annex to the Conference Proceedings, Short Papers*, pages 283–286. Eurographics, 2008.

[LAM01] Jonas Lext and Thomas Akenine-Möller. Towards rapid reconstruction for animated ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2007*, pages 311–318. Eurographics, 2001.

[NVI07] NVIDIA. The cuda homepage. 2007.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. volume 21, pages 703–712. ACM, July 2002.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless Kd-tree traversal for high performance GPU ray tracing. volume 26. Eurographics, 2007.

[TS05] Niels Thrane and Lars Ole Simonsen. *A comparison of acceleration structures for GPU assisted ray tracing*. Master's Thesis, University of Aarhus, 2005.

[Wal04] Ingo Wald. *Realtime ray tracing and interactive global illumination*. Ph.D. Thesis, Computer Graphics Group, Saarland University, 2004.

[WBS03] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 11. IEEE, 2003.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. volume 26, page 6. ACM Transactions on Graphics, January 2007.

[WH06] Ingo Wald and Vlastimil Havran. On building fast Kd-trees for ray tracing, and on doing that in O(N) log(N). In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69. IEEE, 2006.

[WMG$^+$07] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*, pages 89–116. Eurographics, 2007.

[YCM07] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 55. ACM Press, 2007.