Review: Kd-tree Traversal Algorithms for Ray Tracing

M. Hapala V. Havran

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Volume 30 (2011), Number 1, pp. 199–213 COMPUTER GRAPHICS *forum*

Review: Kd-tree Traversal Algorithms for Ray Tracing

M. Hapala V. Havran

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Abstract

In this paper we review the traversal algorithms for kd-trees for ray tracing. Ordinary traversal algorithms such as sequential, recursive, and those with neighbor-links have different limitations, which led to several new developments within the last decade. We describe algorithms exploiting ray coherence and algorithms designed with specific hardware architecture limitations such as memory latency and consumption in mind. We also discuss the robustness of traversal algorithms as one issue that has been neglected in previous research.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing

Keywords: kd-tree, ray tracing, traversal algorithm

1. Introduction

Many rendering algorithms aiming at high quality images are based on ray tracing [App68], i.e. finding the primitives along the oriented half-line or along a line segment which provides the means to propagate or gather energy in the scene. The naive algorithm for ray tracing with $O(N)$ complexity computes the intersections with all geometric primitives. This can be used efficiently only for a small number of primitives (N up to 100). For larger scenes we need to restrict the number of computed intersections along the ray path. This is achieved by various data structures which allow different structuring of spatial regions or objects of a scene. We have to pay the reduced number of computed intersections by the time spent on traversing the data structures.

A kd-tree [Ben75] is one of the hierarchical data structures often used for ray tracing algorithms. It is a special case of a BSP tree [FKN80], recursively partitioning space with planes that are perpendicular to the axes of a coordinate system. Every inner node of the tree has a defined splitting plane, which creates two separate half-spaces. These two half-spaces are included in the left and the right child of the original node and object primitives, such as triangles, are redistributed among both of its children. Those objects straddling the splitting plane need to be assigned to both children. The leaves contain the references to the geometric primitives. An example of a kd-tree can be found on Figure 1.

When working with kd-trees, or acceleration structures in general, one needs to take care of two separate algorithms: data structure build/construction and traversal. This paper is concerned with the latter, but the former is co-essential. A brief introduction into the build algorithm is given in Section 2.

The rest of this paper is structured as follows. In Section 3 we recall the basic traversal algorithms for kd-trees used in ray tracing. In Section 4 we discuss the traversal algorithms for coherent ray packets. In Section 5 we focus on the approaches that were motivated by specific issues of computer hardware. In Section 6 we discuss the modifications aimed at the minimization of computation time due to cache latency. In Section 7 we summarize the discussed algorithms. In Section 8 we conclude the paper.

2. Build Algorithms

A kd-tree is commonly constructed recursively in a top-down fashion, making local greedy decision about the positioning of the splitting planes. When visiting a node, it first decides whether to construct an inner node or to declare the node as a leaf. In the former case decisions must be made about the orientation (axis) and position of the splitting plane. The way how a kd-tree is constructed, in particular the positions of the splitting planes, is essential for the performance of the kd-tree. This is usually done by minimizing the cost function of a local greedy heuristics, called

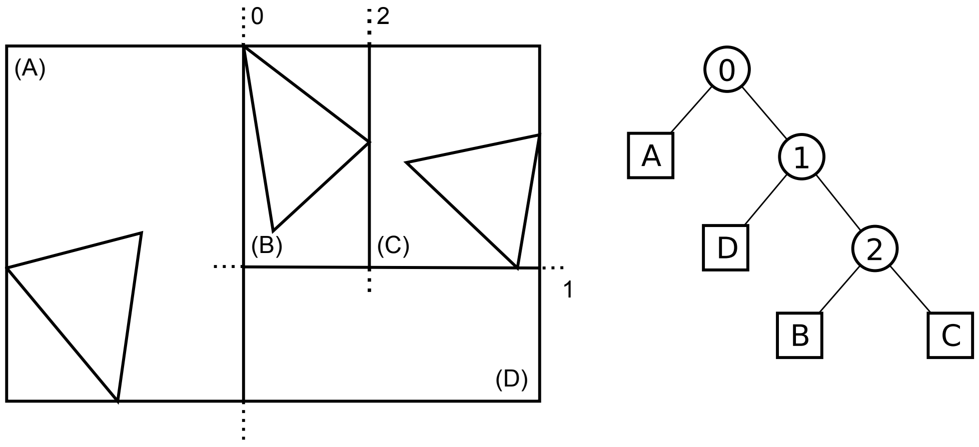

Figure 1: *Simple scene with three triangles with a corresponding kd-tree with three leaves (boxes) and three inner nodes (circles).*

surface area heuristics (SAH) [MB90]. The underlying idea behind the SAH is to put the splitting planes in such a way that they minimize locally estimated traversal and intersection costs. This is achieved by considering geometric probabilities of intersecting *axis aligned bounding boxes* (AABBs, further referred to as bounding boxes) associated with the tree nodes, expressed by the surface areas of these boxes, multiplied with the number of primitives incident to them. The estimated cost is computed for both lower and upper boundaries of every object on each of the coordinate axis and the splitting plane and a minimal value is picked. Wald and Havran [WH06] summarized the techniques for kd-tree construction and proposed an algorithm to minimize the SAH. The modifications to SAH for some special cases are discussed in detail in [Hav00]. SAH computation is quite costly as one has to evaluate the equation twice for each object on the split axis. We will now shortly describe two algorithms based on the approximation of the SAH.

A so-called binning [HMS06, PGSS06] algorithm was proposed, where the idea is to split an interval into a number of equally large bins along one axis. Then, in one pass over all objects in a certain node, it inserts each object into one of the bins according to its centroid, and increments the bin's object count. SAH is then computed only for the boundaries of the bins.

Furthermore, Shevtsov et al. proposed a min-max binning algorithm [SSK07] which gives a better approximation than the binning algorithm as given in [HMS06, PGSS06]. At the beginning, one creates two arrays of bins for minima and maxima of triangle bounding volumes. These are represented by bounding boxes. Then, in one sweep over the geometry object counts in bins are incremented for every triangle boundary that falls into the respective bin. The second pass is then over the bin boundaries, which are now splitting plane candidates. SAH is evaluated on each of them with a number of objects on the left computed from the min bins and a number of objects on the right computed from the max bins.

3. Basic Traversal Algorithms

In this section we describe several basic algorithms that were published prior to the year 2000: a sequential algorithm, a stack-based traversal algorithm, and the two based on neighbor-links placed into leaves. The sequential algorithm was described as the first and the most simple one. The recursive traversal algorithm based on a stack is probably the most known and used algorithm. A neighbor-link tree algorithm extends kd-tree nodes with information about the interior structure of the tree and has smaller traversal overhead when compared to the recursive one.

3.1. Sequential Traversal Algorithm

A basic traversal algorithm for kd-trees (see Algorithm 1) was devised by Kaplan [Kap85] in 1985. We call it a sequential traversal algorithm in this paper. First, a ray is intersected with an axis aligned box that covers the whole scene. This will return *entry* and *exit* distances. These will represent an interval on which an intersection of the ray with the scene is valid. When there is no intersection between the ray and the box the function returns no intersection. Otherwise we will continue by progressively finding all leaves that the ray pierces by point-location queries on the kd-tree.

We will start with a point that is on the entry of the ray into the scene box or if the ray origin is inside of the box this point will be the ray origin. This step is called *leaf location*. It is implemented by traversing the kd-tree from the root downwards: in each inner node we traverse to the child

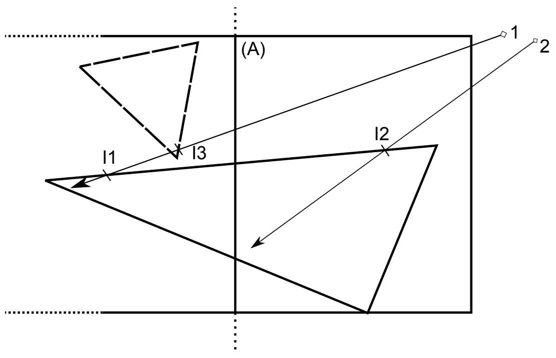

Figure 2: *Intersection with a triangle in a leaf. Ray 1 intersects the primitive (triangle) that lies partly in leaf A, the intersection (I1) lies outside of the leaf and thus is not valid. Instead a valid intersection (I3) is with the dashed triangle that will be found when the other part of the tree is traversed. Ray 2 has a valid intersection (I2) with the triangle as it lies inside of the leaf.*

which includes the half-space containing our search point. When a leaf is found the ray is intersected with all the objects referenced in the leaf (there may be none). If there is no valid intersection that lies inside the leaf box (see Figure 2), the next leaf location search occurs. This requires computing the exit point of the ray on the bounding box associated with the leaf. To avoid visiting the same leaf, the new point to search is moved along the ray path by a small epsilon outside the box. This is repeated until an intersection is found or the next point is outside the scene bounding box.

As the sequential traversal algorithm has to visit the exactly same sequence of nodes several times, it is not very efficient, though it does need constant local memory for its execution which is beneficial for parallel implementations. Moreover, the numerical stability of the algorithm is dependent on the choice of the epsilon. For scenes with very large models and/or ones that lead to kd-trees with cells having one extent equal to zero a choice of a wrong epsilon can lead to an endless loop due to the rounding of floating-point values.

Below we provide the improved algorithms with different design goals that are more involved but they provide algorithmically more efficient solutions or are designed with a particular computer hardware in mind.

3.2. Stack-Based Traversal Algorithm

A stack-based traversal algorithm (often called recursive traversal) remedies the major disadvantage of a sequential one: it traverses each node in a tree at most once and all necessary nodes exactly once (see Algorithm 2). This is achieved by the use of a stack data structure. When traversing a node, the algorithm chooses in which order the children need to be

```
1  Locate Leaf ( node, point)
2  begin
3      current node ← node;
4      if point lies outside node's AABB then
5          return no leaf exists;
6      end
7      while current node is not a leaf do
8          if point is to the left of the node's splitting plane then
9              current node ← current node's left child;
10         else
11             current node ← current node's right child;
12         end
13     end
14     return current node
15 end
16 Kd-tree Sequential Traversal:
17 begin
18     (entry distance, exit distance) ← intersect ray with root's
       AABB;
19     if ray does not intersect AABB then
20         return no object intersected;
21     end
22     if ray has origin in AABB then
23         point = ray origin;
24     else
25         point = ray origin + ray direction * ( entry distance +
           eps );
26     end
       /* this will locate first leaf                              */
27     current node = Locate Leaf ( tree root node, point);
28     while current node is leaf do
           /* current node is a leaf while point is inside tree root
           node's AABB, see Locate Leaf lines 4-6                   */
29         (entry distance, exit distance) ← intersect ray with
           current node's AABB;
30         if current node is not empty leaf then
31             intersect ray with each object;
32             if any intersection exists inside the leaf then
33                 return closest object to the ray origin;
34             end
35         end
           /* point just a bit outside the current node            */
36         point = ray origin + ray direction * ( exit distance +
           eps );
37         current node = Locate Leaf ( tree root node,
           point);
38     end
39     return no object intersected;
40 end
```

Algorithm 1: Sequential traversal.

Traversal decision	near only	far only	both
ray origin	(t $<$ 0) or (t $>$ exit)	(t $>$ 0) and (t $<$ entry)	(entry $<$ t) and (t $<$ exit)
cases:	(a)1, (b)1, (c)1	(a)2, (b)2, (c)2	(a)3, (b)3
ray dir	t $>$ exit	t $<$ entry	(entry $<$ t) and (t $<$ exit)
cases:	(a)1, (b)1, (c)2	(a)2, (b)2, (c)1	(a)3, (b)3

Table 1: *Conditions for the traversal of origin/direction node classifications. For the cases and entry/exit/t refer to Figure 3.*

traversed and if any of them can be skipped. The children are generally referred to as a left and a right child, based on their position with respect to the splitting plane. For the traversal algorithm the children have to be classified as *near* and *far* child nodes. For this classification of children there are only three possible cases of traversal: (i) visit only the near child, (ii) visit only the far child, (iii) visit the near child first and then the far child. The stack stores the far node in case both children need to be visited and this node is used when no intersection occurs in the near node. This can be translated to the original left/right classification into four cases: (1) visit only the left child, (2) visit only the right child, (3) visit the left child first and the right child second, (4) visit the right child first and the left child second.

There are three variants of the stack-based traversal algorithm. The first one uses a near/far classification based on ray-origin and was published by Jansen [Jan86] and also later by Arvo [Arv88]. The second one uses a near/far classification based on the ray direction and was given in the appendix of Keller's thesis [Kel97] and is also used in Wald's thesis [Wal04]. The last algorithm uses the left/right classification directly and was given by Havran et al. [HKBv97].

The near/far classification based on the ray origin uses the origin of the ray and the position of the splitting plane. As a result it treats left as near and right as far when the ray origin is to the left of the splitting plane and vice versa. The near/far classification based on the ray direction classifies the left node as near and the right node as far if the sign of the direction of the ray in the axis of the splitting plane is positive and vice versa if negative. The near/far classifications based on the ray origin and direction are depicted in Figure 3. In Table 1 we can find conditions for these classifications that are used in the recursive traversal algorithm.

There are subtle differences between the two ways in which child nodes are classified as near or far. As this was not discussed in the former publications, we provide short analysis below. The traversal algorithm based on the ray origin classification can suffer from robustness issues when the ray origin is embedded into the splitting plane, as was shown by Havran et al. [HKBv97]. It can thus happen that instead of the left child the right child is traversed and vice versa.

```
1  Kd-tree Recursive Traversal:
2  begin
3      (entry distance, exit distance) ← intersect ray with root's
       AABB;
4      if ray does not intersect AABB then
5          return no object intersected;
6      end
7      push ( tree root node, entry distance, exit distance) to
       stack ;
8      while stack is not empty do
9          (current node, entry distance, exit distance) ← pop
           stack;
10         while current node is not a leaf do
11             a ← current node's split axis;
12             t ← (current node's split position.a - ray origin.a)
               / ray dir.a;
13             (near, far) ← classify near/far with (split
               position.a > ray origin.a);
14             if t ≥ exit distance or t < 0 then
15                 current node ← near;
16             else if t ≤ entry distance then
17                 current node ← far;
18             else
19                 push ( far, t, exit distance) to stack;
20                 current node ← near;
21                 exit distance ← t;
22             end
23         end
24         if current node is not empty leaf then
25             intersect ray with each object;
26             if any intersection exists inside the leaf then
27                 return closest object to the ray origin;
28             end
29         end
30     end
31     return no object intersected;
32 end
```

Algorithm 2: Recursive traversal based on ray origin classification. For direction based classification change line 10 to use (ray dir.a $>$ 0) and remove condition (t $<$ 0) from line 11.

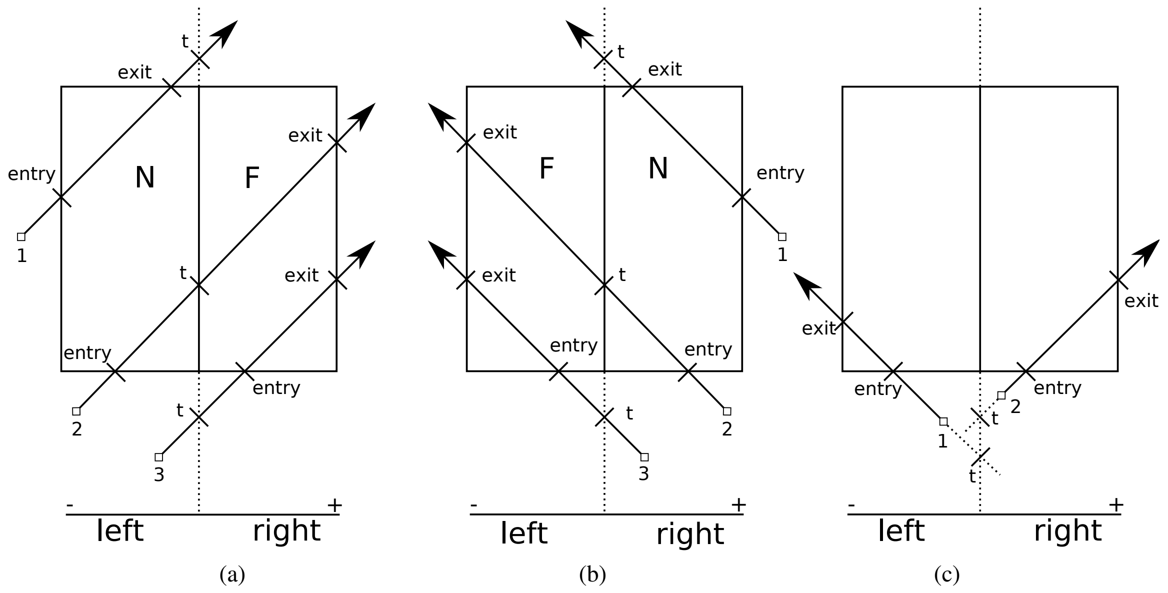

Figure 3: *Direction and origin based near/far classification. The classification for rays in (a) and (b) are the same. Rays in (c) have their origin closer to (or it may be even inside of) the node and thus on the other side of the splitting plane. Here for ray 1 the ray direction classifies left as far, but the ray origin classifies left as near and vice versa for ray 2.*

The same problem with robustness can also happen for the algorithm based on the ray direction, if the ray direction is zero for one of the coordinates. Fortunately, the robustness can easily be reinforced by a small trick before the ray traversal starts. The zero coordinates of the ray direction are found and are set to a small epsilon. The error due to the change of the ray direction is negligible and the correctness of algorithm can be shown for all of the cases. Such a trick can obviously not be used for the near/far classification based on the ray origin, as the comparison between the ray origin and the splitting plane is computed in each traversal step and this must be always checked for the case of an embedded ray origin in the splitting plane.

There is a robust algorithm provided by Havran [Hav00] that also minimizes the number of conditions performed for each traversal step. This is achieved by not storing only the distance along the ray, but also the coordinates of the point on the splitting plane. It assumes that the memory required to implement the stack is available and can be always quickly accessed. The method also approaches the theoretical limit on the processed conditions - there are only 4 traversal cases when visiting an interior node that can be distinguished by $log_2 4 = 2$ consecutive branches.

3.3. Neighbor-Links and Neighbor-Link Trees Traversal

As the recursive traversal improves the sequential one by not traversing nodes more than once, it still traverses many nodes inside the tree when finding consecutive leaves pierced by a ray. If we added information regarding the internal structure of the tree to the nodes, e.g. with which nodes a node shares its faces (sides), we would be able to traverse to neighboring nodes directly from and inner node without or with minimal traversal overhead.

Neighbor-links (or ropes) as proposed by MacDonald and Booth [MB90] and further investigated by Havran et al. [HBv98] use this idea. Six additional pointers are added to each node, providing links to neighboring spaces in the tree, where those spaces can be a number of inner and/or leaf nodes. The basic algorithm constructs links that will point to leaves wherever possible, i.e. where a leaf's face covers the whole of a node's face completely, thus the leaf is a sole neighbor (direct link). Otherwise a link to an inner node that covers all leaves that are connected to that node's face is created (indirect link, see Figure 4(b)).

We can also replace links to inner nodes by so-called neighbor-link trees. These are 2D kd-trees that contain all nodes from the inner node's sub-tree that can be accessed through the inner node's face. This will replace traversal from an inner node by a traversal through an optimized neighbor tree, where only relevant nodes are traversed as depicted on Figure 4(c).

3.3.1. Construction

The construction of neighbor-links is straightforward. First, one needs to build a kd-tree using the SAH or a different build algorithm. Then, for each face in each leaf's bounding

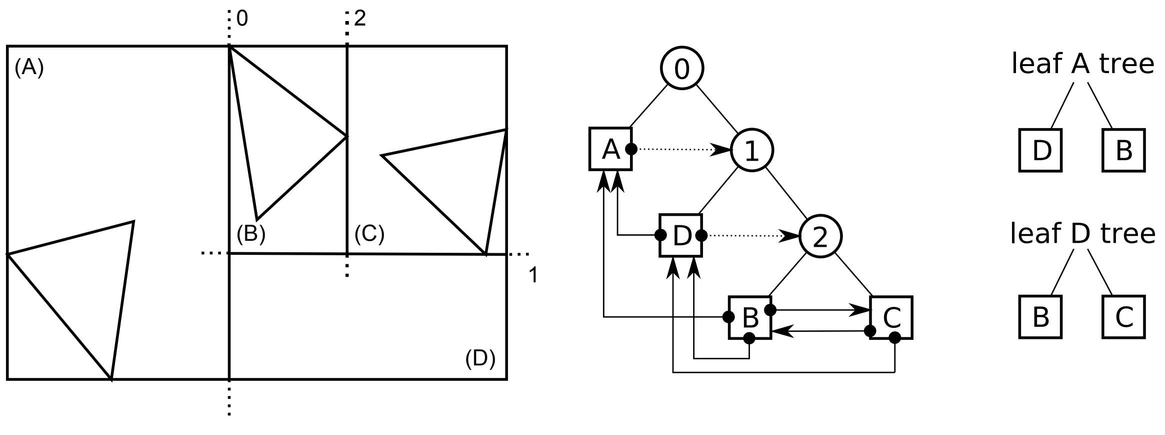

Figure 4: *Neighbor-links and neighbor-link trees. (a) source scene, (b) kd-tree with direct and indirect (dotted) neighbor-links, (c) neighbor-link trees that replace indirect links from the middle image.*

box in the kd-tree a single neighbor-link is set up. For a given face the kd-tree is searched, starting from the root node of the kd-tree. In each step the search continues in the sub-tree that contains the whole face. If the splitting plane splits the face, the search is terminated. The algorithm constructing the neighbor-link trees further replaces indirect neighbor-links by their corresponding 2D neighbor-link trees. This is done with a constrained depth-first-search (DFS) on the kd-tree. Only sub-trees corresponding to the cells that intersect the leaf's face are visited during the DFS and only the leaves where the faces intersect the leaf's face are added to the neighbor-link tree.

3.3.2. Traversal

The ray traversal algorithm for the kd-tree using neighbor-links replaces the redundant point-location search that always starts from the root of the kd-tree in the sequential ray traversal algorithm by the one that starts in the node pointed to by a neighbor-link. Compared to the stack-based traversal algorithm it also allows us to skip some inner nodes directly while it does not use any memory for the stack. However, it is necessary to store the neighbor-links/neighbor-link trees which can result in significantly increased memory consumption. In addition to the links it also requires to explicitly store the bounding boxes of leaves directly in the leaf node structure.

The ray traversal algorithm first finds the exit face from the leaf and then computes the exit point on this face. Four cases can occur, depending on which node the neighbor-link associated with the exit face is pointing to and what algorithm was used:

1. The node is a leaf (direct link) and it is directly used as a next leaf in the sequence that needs to be traversed.

2a. The node is an inner node (indirect link). This node is then used as a root for the point-location search until we find the next leaf.

2b. Or the node is a root of a neighbor-link tree (indirect link expanded to a tree). This 2D tree is searched for a leaf that contains the exit point.

3. The link is pointing to nowhere, the ray then leaves the scene box.

4. Ray Coherence and Packet Tracing

Basic ray tracing algorithms that trace single rays can be improved by taking advantage of *ray coherence*. Rays with a similar origin and a similar direction will traverse a common, sometimes very long part of the scene near to each other. This is depicted on Figure 5. The ray coherence can therefore be exploited by not tracing individual rays but by packaging several such rays together into a *ray packet*, and then tracing those together through the acceleration structures at once.

4.1. Longest Common Traversal Sequence

The use of ray coherence in the context of kd-trees was proposed by van der Zwaan et al. [ZRJ95] for pyramidal shafts formed by primary rays. It was further extended by progressively refining the shafts by Havran and Bittner [HB00] using the Longest Common Traversal Sequences (LCTS). It groups rays into convex shafts with four corner rays. These rays are then traversed through the data structure one by one and their traversal path is saved as a traversal sequence. If a common traversal sequence to all four rays is found, the traversal sequence is valid for all the rays inside the shaft. The empty leaves in the traversal sequence can be skipped. Moreover, if all corner rays intersect a single convex object, this (termination) object is also intersected by all inner rays.

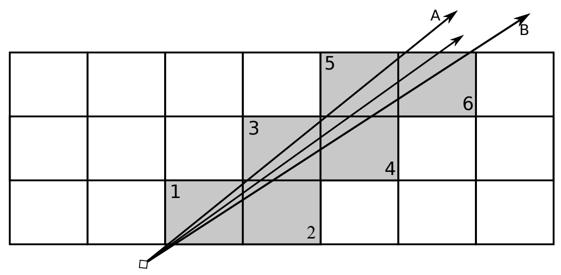

Figure 5: *Ray coherence concept.*

4.2. Packet Tracing

Another approach for a group of 2×2 rays (a packet) was proposed by Wald et al. [WBWS01]. They propose to use Single Instruction Multiple Data (SIMD) instructions implemented in contemporary CPUs and known as Streaming SIMD Extensions [TH99], which execute one operation over a vectors of operands at once. These instructions are then used to compute traversal of a whole packet of rays at once. One must be careful when the rays diverge, however, e.g. some of them want to traverse only the far child while the others will want to traverse both children. At this point a bitmask is usually used to save which rays are currently active in the traversal and the results of inactive rays need to be masked out not to trigger incorrect traversal steps or compute incorrect intersections. One also has to take into account the classification algorithm, where for different types of rays in a packet, e.g. rays traced from different surface hitpoints towards a common light source (shadow rays), the origin-based classification or the direction-based one will results in packets diverging sooner.

Currently the most common SIMD width on commodity CPUs is 4 which we can see rise to 8 with an expected support of Advanced Vector Extensions (AVX) in 2011. If we also take the current graphical processing units into account we can find SIMD widths of up to 32 on NVidia's Fermi architecture. Moreover, the best number of rays in a packet does not necessarily have to equal the SIMD width and the packets of 4×4 and 8×8 rays are mentioned already in [WBWS01].

4.3. Relevant Triangle Tracing

Relevant triangle tracing by Dmitriev et al. [DHS04] expanded the idea by Wald et al. by using SIMD instructions to traverse all four corner rays of a pyramidal shaft at once, allowing more rays to be included. Under certain conditions fulfilled by the corner rays of a shaft, it is possible to immediately and conservatively answer visibility queries for the inner rays with respect to a triangle in the leaf nodes of a kd-tree. This is in general possible for primary rays induced by a pinhole camera or shadow rays cast from a point light source. For other rays such as secondary rays it could be difficult or even impossible to use this technique.

4.4. Multi-Level Ray Tracing algorithm

The Multi-Level Ray Tracing algorithm (MLRTA) by Reshetov et al. [RSH05] introduced the entry point search, which is basically trying to find the LCTS of a ray shaft, though by a different search algorithm based on the geometry of the shaft. An entry point is the deepest node from which the actual intersection search can start. The shaft traverses the acceleration structure in a breadth-first search manner, keeping the topmost node with potential intersections in sub-trees of both children. Sub-trees without potential intersections are those with all leaves in them either empty or not intersected by the shaft.

The final entry point candidate is thus the node that is the common ancestor of all non-empty leaves that the shaft actually intersects. It is clear that the performance gain of the entry point search heavily relies on the number of empty leaves in the tree. Reshetov et al. thus proposed a kd-tree build method, which sets a bias for creating leaves with empty spaces and also works best for dominantly axis-aligned scenes with large occluders, such as in architectural scenes.

4.5. Image Plane Tiling

The coherence based traversal algorithms above, excluding packet tracing, require either regular or adaptive splitting of the image plane. The so-called adaptive tile splitting is an extension to the entry point search, where we start with pixels of an image split into a couple of uniformly sized squares. When these are connected to the eye origin they create the initial ray shafts. The number of nodes in a sub-tree of a kd-tree depends on the the complexity of the geometry enclosed in that part of a tree. Thus we can adaptively split ray shafts in places where geometry is complex.

4.6. Early exit

For coherent packets, we can allow an early exit strategy similar to that for single rays. The kd-tree is traversed in a front-to-back manner along a ray. This will assure that if an intersection is found, then it is the nearest one to the ray origin and the traversal of this ray can be terminated. This is not true for packets of incoherent rays, as the rays in a packet will have different classifications for near and far child nodes for a traversed node. Thus we can test a single node for an incoherent packet but we cannot assume anything about the other nodes on the stack, as some of them may be still closer to one of the rays in the packet. This can be solved by splitting incoherent packets into coherent subsets where for each the classification is computed separately.

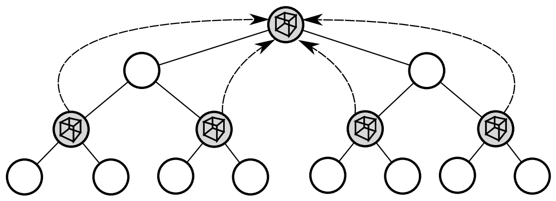

Figure 6: *Acceleration structure augmented with sparsely distributed augmented nodes. Links to parents are only between nodes with additional data.*

4.7. Omnidirectional Traversal

Reshetov [Res06] has shown an omnidirectional algorithm for tracing packets of incoherent rays which relaxes this early exit strategy by not terminating the traversal altogether when an occluded node is found, but only pops the occluded one from the stack and then continues until the stack is empty. Another approach for traversing packets with different directions at once was shown by Tsakok et al. [TBK08]. They use ray origin based classification and change negative intersection distances to positive infinity (see in Figure 3(c)). At the beginning of every traversal step all rays that have an entry distance larger than their exit distance are excluded from processing. This will correctly traverse packets without the need to check their coherence. Both Reshetov [Res06] and Tsakok et al. [TBK08] include source code of their respective algorithms.

5. Hardware Motivated Modifications

Recently, ray tracing algorithms, including those with kd-trees, were modified to suit specialized computer architectures, such as graphical processing units (GPUs) or the IBM Cell processor, which have proved to give better performance than general purpose ones. The modifications are based on the need to find a work-around for limitations imposed by these architectures. For example, some architectures possess no or very small low-latency memory (cache), some architectures have greater penalties for branched code, because they lack branch prediction capabilities, and almost all architectures now possess wide SIMD arithmetic units. Basically, the important issue on special hardware architectures is data and instruction parallelism. This has to be carefully considered when adopting a traversal algorithm to these architectures. Disregarding these limitations can lead to stalls during the traversal algorithm execution.

5.1. Stackless Traversal Algorithms

To stay efficient on specialized hardware one can avoid a stack altogether and work with a stackless algorithms. Foley and Sugerman [FS05] have shown two basic approaches: kd-restart and kd-backtrack. The kd-restart is basically a sequential traversal ported to the GPU, as already stackless (Section 3.1). Its main disadvantage is thus the same as it was already mentioned, i.e. to find each consecutive leaf it has to start from the root of the tree.

5.1.1. Kd-Backtrack

The kd-backtrack adds bounding boxes and pointers to a parent in each node. This allows it to avoid the restart of the traversal from the root node. The point is moved a bit forward along the ray path and the traversal backtracks upwards the tree until the point lies inside the bounding box of a node traversed (see Algorithm 3).

```
Backtrack To Leaf ( node, point)
begin
    current node ← node's parent;
    if point lies outside global AABB then
        return no leaf exists;
    end
    while point lies outside current node's AABB do
        current node ← current node's parent;
    end
    while current node is not a leaf do
        if point is to the left of the node's splitting plane then
            current node ← current node's left child;
        else
            current node ← current node's right child;
        end
    end
    return current node
end
```

Algorithm 3: Backtrack traversal.

5.1.2. Sparse Boxes

Havran and Bittner [HB07a] reduced the memory footprint of kd-backtrack by reducing the number of nodes with bounding boxes and parent links. The augmented nodes (with additional data) are sparsely distributed throughout the tree and only these are connected with parent links (see Figure 6). The last augmented node is always cached and used as first when the traversal restarts. If there is no intersection between its bounding box and the currently valid intersection interval, its parent link is used to find another augmented node further up the tree. The pseudo-code is given in the paper [HB07a] and generalized for coherent rays in [HB07b].

5.2. Kd-Restart and Kd-Push-Down Traversal

Horn et al. [HSHH07] expanded kd-restart to kd-push-down, which keeps information about the depth-wise lowest node that completely contains the valid intersection interval. Instead of the root, this node is then used when a traversal is restarted. This *pushed node* is updated to the currently traversed node until the ray no longer traverses only near nodes or far nodes. The pushdown algorithm is given in Algorithm 4.

```
Kd-tree Push-down Traversal:
begin
    (entry distance, exit distance) ← intersect ray with root's AABB;
    maximum distance ← exit distance;
    exit distance ← entry distance;
    pushed node ← tree root node;
    enable pushdown;
    while exit distance < maximum distance do
        current node = pushed node;
        entry distance ← exit distance;
        exit distance ← maximum distance;
        while current node is not a leaf do
            a ← current node's split axis;
            t ← (current node's split position.a - ray origin.a) / ray direction.a;
            if t ≥ exit distance or t < 0 then
                current node ← near;
            else if t ≤ entry distance then
                current node ← far;
            else
                current node ← near;
                exit distance ← t;
                disable pushdown;
            end
            if pushdown enabled then
                pushed node ← current node;
            end
        end
        if current node is not empty leaf then
            intersect ray with each object;
            if any intersection exists in the leaf node then
                return closest object to the ray origin;
            end
        end
    end
    return no object intersected;
end
```

Algorithm 4: Push-down traversal.

5.3. Neighbor-links Traversal

The concept of neighbor-links, which inherently requires no stack, has been used by Popov et al. [PGSS07] to implement a stackless traversal on Nvidia's G80 architecture [NVi06]. Neighbor-links are added to the tree during construction, where links from the faces of a node are copied to the faces of its relevant children, respectively their outer faces and the children are connected by through the common sides. To create neighbor-links suitable for single rays one needs to "push" these links down the tree as far as possible, linking neighboring leaves to each other. Popov et al. skipped this step for packet traversal, thus every leaf was left with a link to the root of the neighboring sub-tree.

This linking is implemented by a regrouping mechanism. When ray packets traverse to a child of a node, some of the rays may be deactivated as they do not intersect it. Thus it

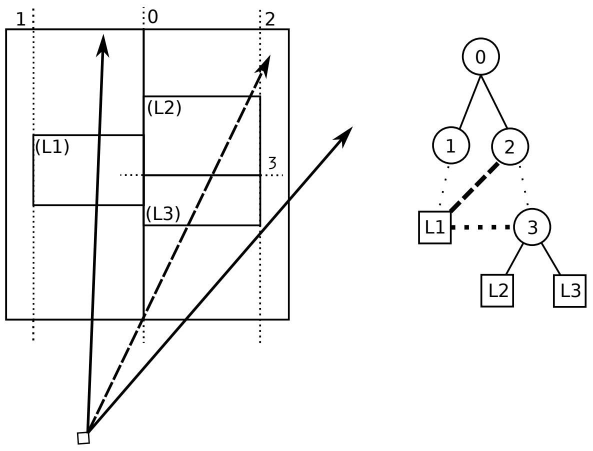

Figure 7: *Neighbor links for packet tracing. The wide dotted line is an ordinary neighbor link that leads to the lowest node that encapsulates all leaves that are neighboring one of the faces. The wide dashed line is a link that has not been pushed down the tree and hence points to the whole neighboring sub-tree.*

is needed, after the processing of the child is finished, to reactivate the whole packet and continue to the other child. In this case it is advantageous to have a rope linking the current node to the whole neighboring sub-tree. For simplicity, if we take a packet of rays with a common origin (see Figure 7), we can see that all the rays on the right side of the dashed one need to traverse the right child only, thus they will be deactivated on entry of the packet into the left one. When the packet ends processing in leaf L1 it is advantageous to have a neighbor-link to the root of the neighboring tree, so the packet can reactivate and continue processing from the nearest node to the origin.

5.4. Stack-Based Traversal Algorithms

One approach for architectures with a small on-chip memory is to use a stack that does not scale logarithmically with respect to the size of the acceleration structure built over the scene, but rather is of a fixed size. One pushes nodes onto a stack as usual and discards the bottom item when a stack overflow occurs. When an underflow occurs the algorithm has to fall back to a stackless version.

Horn et al. [HSHH07] provide a very fast short stack algorithm which falls back to a stackless algorithm with push-down, so when eventually a restart needs to be done, one can simply backtrack to the top of the smallest enclosing tree (see Section 4.1). They also state, that "the algorithm with short stack visits fewer than 3% more nodes than the one with unlimited stack". A different solution for alleviating the impact of a restart was shown by Novák [Nov09]. He used a technique called "stack spilling" (shown in Figure 8), where

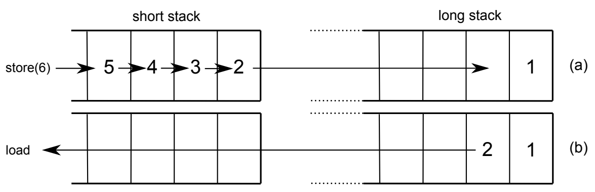

Figure 8: *Stack spilling. (a) stack overflow, (b) stack underflow.*

a regular large stack in the GPU memory is used as a place for nodes that were at the bottom of the short-stack at the moment of overflow. These nodes are, instead of being discarded, saved to the large stack. In the moment of underflow one uses all the nodes on the large stack before the need to do a restart. Novák claims that the technique with spilling is faster by 23% on average than the short-stack with restart on a GeForce 285 GTX. He has also shown that a conventional stack in memory is a viable option on current graphics hardware, where a large stack without spilling still performed better than a short-stack with restart.

Zlatuška and Havran [ZH10] studied the performance of three spatial data structures for ray tracing on a GPU. They also include the study of a kd-tree traversal algorithm with a stack on a GPU. In addition to the pointer to a node they also have to save the interval along the ray given by the entry and the exit distance. However, the entry distance can be taken as exit distance of the traversed node. Storing the two values instead of three leads to non-negligible performance increase.

5.5. Branch-Aware Traversal Algorithms

Mispredicted branches can be sometimes avoided by transforming branched code to a branchless one. This idea is based on performing computations in both the taken and not taken branch sequentially and then choosing the correct values according to the Boolean value that was previously used to branch the code. In SIMD this can be achieved by a "blending" instruction that will combine two vectors of values into one according to another vector.

A branchless kd-tree traversal algorithm, as it was implemented in [Hap09], is based on the idea that instead of using branches when deciding which node is going to be traversed or possibly pushed on the stack, one can push both nodes and by simply moving the stack pointer forward or backward skip those that are not needed (see Algorithm 5). The correctness of the code for all traversal cases shown in Figure 3 is documented in Table 2. A similar approach, in a more condensed form, has been shown by Benthin [Ben06]. A similar construction of a branchless algorithms can be created for other recursive variants described in Section 3.2.

```
c ← (t < exit distance) ;   /* c is either 0 or 1 */
d ← (t < entry distance) ;  /* d is either 0 or 1 */
stack [ stackPtr ].node ← far;
stack [ stackPtr ].entry distance ← max(t, entry distance);
stack [ stackPtr ].exit distance ← exit distance;
stackPtr ← stackPtr + c;
stack [ stackPtr ].node ← near;
stack [ stackPtr ].entry distance ← entry distance;
stack [ stackPtr ].exit distance ← min(t, exit distance);
stackPtr ← stackPtr - d;
/* next node to be processed is on the
   top of the stack                      */
```

Algorithm 5: Branchless traversal (direction based). One step for a single node.

6. Latency-aware Motivated Modifications

In this chapter we will discuss modifications that aim to reduce memory latency when traversing through the kd-tree. These methods try to have a maximum of data that will be needed at a similar time close together in memory and a minimal memory consumption. This will reduce cache misses that are caused by random memory access which keeps less relevant data in cache. In other words, the methods are based on either algorithmic improvements or spatial and temporal locality.

6.1. Implicit Pointers

Concerning the data layout, the size of the tree node can be minimized in order to have more of them fetched on each cache line. The cache line size (size of a block of memory transferred between main memory and cache at once) is between 32 to 256 bytes for current CPUs. To represent an inner kd-tree node we have to store the following variables: a pointer to the left child, a pointer to the right child, a splitting plane axis and a splitting plane position. Using 4 bytes per pointer and float or integer value and 2 bits for a splitting plane axis (3 possible values) we need a total of 13 bytes per node. A leaf needs only a pointer to the array of primitives and the number of primitives that it contains, so the leaf data can be saved into the same data structure as an inner node, when we use the fourth left-over value in the 2 bits for a splitting plane axis to signify a leaf node.

Wald et al. [WBWS01] proposed to reduce memory consumption so that addressing of all nodes in memory is aligned to a multiple of 8. This creates room to store additional data in the pointers to the left and the right child, because there will be bits in the address that will always be zero. The splitting plane axis can be easily stored in those bits. To reduce the node size by another 4 bytes to the final 8 bytes we avoid storing one of the pointers to a child. If the nodes are stored so that the left child always follows its parent node, then only a right child pointer is needed and the left child pointer can be computed as parent + 8 bytes.

	c	d	Save far	Inc with c	Save near	Dec with d	Decision
	values		line 3	line 6	line 7	line 10	
ray 1	0	0	$\boxed{\text{far}}$	$\boxed{\text{far}}$	$\boxed{\text{near}}$	$\boxed{\text{near}}$	near only
ray 2	1	0	$\boxed{\text{far}}$	far $\boxed{}$	far $\boxed{\text{near}}$	far $\boxed{\text{near}}$	both
ray 3	1	1	$\boxed{\text{far}}$	far $\boxed{}$	far $\boxed{\text{near}}$	$\boxed{\text{far}}$	far only

Table 2: *Stack content in different phases of the branchless traversal algorithm for different rays from Figure 3.* Save far, Inc with c, Save near *and* Dec with d *columns include current stack content after the operation has been completed, where a boxed value is the one pointed to by the stack pointer.*

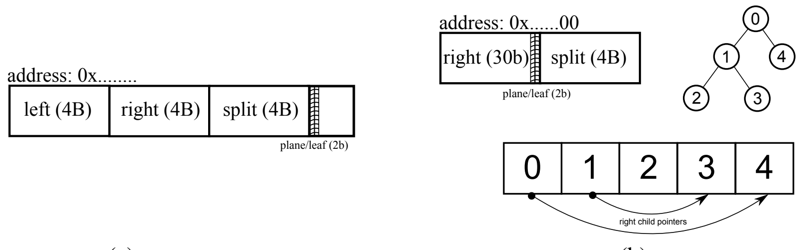

Figure 9: *Memory layout of a (a) kd-tree node for basic layout storing 13 Bytes and (b) memory aligned and condensed layout reduced to 8 Bytes to a node with an example of a simple tree.*

The node then is comprised of only a right child pointer (including 2 bits for splitting plane axis) and a splitting plane position. The node storage is depicted in Figure 9. To have this applicable in 64-bit address space one needs to convert all pointers to 4 byte memory offsets from a given static 8 byte memory address.

6.2. Regular Subtree Layouts

There are other methods to store nodes in memory, but usually this will leave the parent and the children, or the two children, far from each other. This reduces the effectiveness of the cache, because after a parent is processed it would be advantageous to have both of its children on the same cache line prepared to be processed. Havran [Hav99] proposed a solution (corresponding to van Emde Boas memory layout [vEB75]) that uses a custom memory allocation, which clumps together nodes from a sub-tree of a certain height, where the memory allocated for such is as large as the maximum possible number of nodes for that sub-tree. Only the nodes on the lowest level need pointers to their children, as inside the sub-tree the locations can be computed from their memory address. This will tradeoff memory size for latency, as some space in these sub-trees will not be occupied. The solution is also named treelet layout in the literature.

Szécsi [Szé03] expanded the idea of treelets and tried to decrease fragmentation of memory space that inevitably occurs in the leaf parts of the kd-tree. The idea is to occupy the memory that is left unused in the leaves of the treelets and move leaves in place of their pointers to them (see Figure 10).

6.3. Irregular Subtree Layouts

Yoon et al. [YM06] proposed a generalization of the subtree layout in a cache-oblivious model which adds spatial locality to the parent-child locality used by Havran [Hav99] and Szécsi [Szé03]. They show a probabilistic model to quantify these localities where for ray tracing applications the probability that a node would be accessed during a query is computed as a ratio of surface areas of its parent node and its grandparent node. They then cluster nodes to maximize the sum of probabilities of nodes in one cluster. These clusters are then reorganized to get spatially close clusters which are also close together in the final layout.

6.4. Nodes Prefetching

To avoid memory stalls when fetching the kd-tree data from the main memory to the cache a technique similar to one used in hardware called temporal multithreading can be employed. It was implemented on the IBM Cell processor and called *software hyperthreading* by Benthin et al. [BWSF06], but the idea can also be implemented on general-purpose

Algorithm (derived from)	Leaf location summary	Additional memory requirements
Sequential (-)	• To find leaves the algorithm uses separate point-location searches (always traversing from the root) starting with a point on the entry of the ray into the scene box. • After finding a leaf the next point for point-location is set just after the exit of the ray from that leaf.	Bounding box data in each leaf (faster, but not necessary).
Stack-based (Sequential)	• The algorithm chooses which nodes to traverse according to a comparison between the ray origin or ray direction and the splitting plane. • If both children of a node are to be traversed the second one is pushed on the stack. Each node is traversed at most once. • When a leaf is found the traversal continues with the node on the top of the stack.	One traversal stack of a maximal depth.
Neighbor links (Sequential)	• Each side of each leaf has associated either the only neighboring leaf (direct link) or the inner node that contains all neighboring leaves (indirect link). Traversal then follows these links to find consecutive leaves pierced by the ray. • On average less nodes are traversed than in Stack-based traversal.	Six pointers in each leaf, one for each side. Bounding box data in each leaf.
Neighbor-link trees (Neighbor links)	• Indirect links are replaced with 2D kd-trees that include only relevant nodes for the traversal to the leaf.	Six pointers and bounding box data in each leaf, shallow 2D kd-trees to replace indirect links.
Kd-backtrack (Sequential)	• After finding a leaf the next point-location is started from the closest node upwards in the tree that includes the next point. The node is found by following pointers to parent nodes and intersecting the point with the bounding boxes saved in them.	Bounding box data and pointer to parent in each node.
Sparse boxes (Kd-backtrack)	• Next point-location is started from the closest node with a bounding box upwards in the tree, that includes the next point.	Bounding box data and pointer to parent in some nodes.
Kd-pushdown (Sequential)	• Next point-location is started from a cached node, which is the last that completely encapsulates the currently valid intersection interval, i.e. from the point we were locating to the exit point of the ray from the scene box.	One pointer to a node.
Short stack (Sequential)	• The algorithm uses stack as in Stack-based, but does not scale its depth with the acceleration structure depth and is rather of a fixed size. • When an underflow occurs falls back to a form of a stackless traversal. • Nodes that overflow the stack are discarded. • Small on-chip memory friendly algorithm.	One traversal stack of a small constant size.
Stack spilling (Short stack)	• Overflown nodes are saved to a larger stack that is in the conventional memory. • In the moment of underflow it first uses all nodes from the larger stack before falling back to a stackless traversal.	One traversal stack of a small constant size and a large traversal stack of a maximal depth.

Table 3: *Summary of traversal algorithms.*

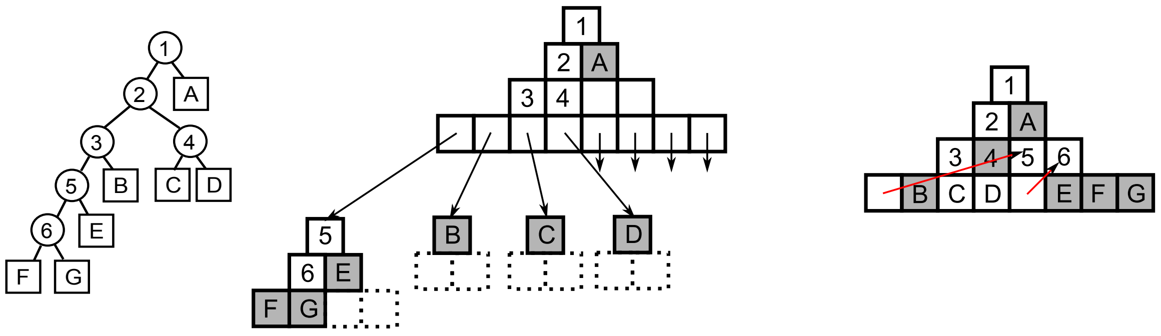

Figure 10: *Kd-tree layout with treelets. (a) Source kd-tree, (b) Havran [Hav99] based on van Emde Boas layout, (c) Szécsi expansion [Szé03] for incomplete trees.*

hardware. This method works on a pool of rays or ray packets at once. The algorithm always executes one packet at a time, but when the data is not available in cache, the execution of the packet is suspended and another packet is started instead. While the second packet is being processed, the data for the first one are copied to the cache by direct memory transfer (DMA). This reduces the memory latency as otherwise the program would have to wait for the data to be fetched from memory, and a memory stall would occur.

On general purpose hardware with support for data prefetch instructions, where are no means to find the availability of data in the cache, the traversal is carried out for a sequence of N rays from the pool. A single node is traversed for each ray, the data for the child node to be traversed are prefetched, and the computation moves to the next ray in the sequence. When the ray tracing for a single ray is finished, the result for the ray is stored and the data for a new ray are placed in the sequence until the whole pool is computed. This approach can be then used successfully for offline computation of ray tracing queries, i.e. with those queries known before the start of the algorithm. This has been tested with a performance gain between 5 to 12 percent by Bittner et al. [BMW*09], though the results depended strongly on the architecture and the compiler.

7. Summary

In this section we present Table 3, which summarizes the core traversal algorithms. For every algorithm we describe: its direct predecessor, how the algorithm finds consecutive leaves pierced by the ray and memory required by the algorithm. Algorithms described in Sections 4 and 6 are not included, as their purpose is to enhance performance, rather than to create a completely new approach.

8. Conclusion

We have summarized the traversal algorithms for kd-trees used in ray tracing. We have described the ordinary traversal algorithms and the changes that were developed for specialized cases. These include the ray coherence for a set of rays that can be formed by casting them from the camera or a point light source. Furthermore, we have described the modifications of the traversal algorithm motivated by specific hardware issues such as the lack of caches, the memory latency, and the penalty for performing branches. We have shown that the traversal algorithms can be interconnected to the kd-tree build algorithm as specific data may be needed to be precomputed and stored in the kd-tree representation. We have also discussed the memory layouts for this representation that can be optimized in particular for larger data sets or when limited by memory of a specific hardware. We also discussed the robustness issues for the traversal algorithms, which, when neglected, can lead to incorrect traversal. Finally, we have described several approaches for ray tracing of many rays at once. We believe that this survey will simplify the user design choices when an implementation of a kd-tree based ray tracer for a particular application and computer hardware is needed.

Acknowledgment

We would like to thank Lukas Novosad, Jiří Bittner, and Jan Novák for proofreading this paper and the anonymous reviewers for their kind remarks and comments. This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under research programs MSM 6840770014, LC-06008 (Center for Computer Graphics) and MEB-060906 (Kontakt OE/CZ), the Grant Agency of the Czech Republic under research program P202/11/1883, and the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13.

References

[App68] Appel A.: Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 37–45. 1

[Arv88] ARVO J.: Linear-time Voxel Walking for Octrees. *Ray Tracing News (available at http://www.acm.org/tog/resources/RTNews/html/rtnews2d.html 1*, 5 (1988). 4

[Ben75] BENTLEY J.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18* (1975), 509–517. 1

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006. 10

[BMW*09] BITTNER J., MATTAUSCH O., WONKA P., HAVRAN V., WIMMER M.: Adaptive global visibility sampling, Aug. 2009. Proceedings of ACM SIGGRAPH 2009. 13

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray Tracing on the Cell Processor. In *Proc. IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 15–23. 11

[DHS04] DMITRIEV K., HAVRAN V., SEIDEL H.-P.: *Faster Ray Tracing with SIMD Shaft Culling*. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, Dec. 2004. 7

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14* (July 1980), 124–133. 1

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *Proceedings of Graphics Hardware* (2005), pp. 15–22. 8

[Hap09] HAPALA M.: *Data Structures for Ray Tracing on Specialized Hardware*. Master's thesis, Czech Technical University, Feb. 2009. 10

[Hav99] HAVRAN V.: Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica 23*, 3 (May 1999), 203–210. 11, 13

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Nov. 2000. 2, 5

[HB00] HAVRAN V., BITTNER J.: LCTS: Ray Shooting using Longest Common Traversal Sequences. *Computer Graphics Forum 19*, 3 (Aug 2000), 59–70. (Proceedings of Eurographics 2000). 6

[HB07a] HAVRAN V., BITTNER J.: Ray Tracing with Sparse Boxes. In *23rd Spring Conference on Computer Graphics (SCCG 2007)* (Budmerice, Slovakia, May 2007), pp. 49–54. 8

[HB07b] HAVRAN V., BITTNER J.: Stackless Ray Traversal for kD-Trees with Sparse Boxes. *Computer Graphics & Geometry 9*, 3 (Dec. 2007), 16–30. 8

[HBv98] HAVRAN V., BITTNER J., ŽÁRA J.: Ray Tracing with Rope Trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)* (Budmerice, Slovak Republic, Apr. 1998), pp. 130–139. 5

[HKBv97] HAVRAN V., KOPAL T., BITTNER J., ŽÁRA J.: Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools 2*, 4 (Dec. 1997), 15–23. 4

[HMS06] HUNT W., MARK W., STOLL G.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. *Proc. IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), 81–88. 2

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *SI3D* (2007), pp. 167–174. 8, 9

[Jan86] JANSEN F. W.: Data Structures for Ray Tracing. In *Data Structures for Raster Graphics*, Kessener L. R. A., Peters F. J., van Lierop M. L. P., (Eds.). Springer-Verlag, New York, 1986, pp. 57–73. Eurographics Seminar. 4

[Kap85] KAPLAN M.: Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes* (July 1985), pp. 149–158. 2

[Kel97] KELLER A.: *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, University of Kaiserslautern, 1997. 4

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer 6* (1990), 153–65. 2, 5

[Nov09] NOVÁK J.: *Global Illumination Methods on GPU with CUDA*. Master's thesis, Czech Technical University, June 2009. 9

[NVi06] NVIDIA CORP.: Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, Nov 2006. 9

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94. 2

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007), 415–424. (Proceedings of Eurographics 2007). 9

[Res06] RESHETOV A.: Omnidirectional Ray Tracing Traversal Algorithm for kd-trees. In *Proc. IEEE Symposium on Interactive Ray Tracing 2006* (Sep 2006), pp. 57–60. 8

[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). 7

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404. (Proceedings of Eurographics 2007). 2

[Szé03] SZÉCSI L.: An Effective Implementation of the k-D tree. In *Graphics programming methods*. Charles River Media, Inc., Rockland, MA, USA, 2003, pp. 315–326. 11, 13

[TBK08] TSAKOK J., BISHOP W., KENNINGS A.: kd-Tree traversal techniques. In *Proc. IEEE Symposium on Interactive Ray Tracing 2008* (Aug. 2008), pp. 190–190. 8

[TH99] THAKKAR S. T., HUFF T.: Internet Streaming SIMD Extensions. *Computer 32*, 12 (1999), 26–34. 7

[vEB75] VAN EMDE BOAS P.: Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1975), IEEE Computer Society, pp. 75–84. 11

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 4

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics 2001). 7, 10

[WH06] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proc. IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 61–69. 2

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516. (Proceedings of Eurographics 2006). 11

[ZH10] Zlatuška M., Havran V.: Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In *Proceedings of WSCG'2010 (communication papers),* (Feb 2010), pp. 69–75. 10

[ZRJ95] Zwaan M., Reinhard E., Jansen F. W.: *Pyramid Clipping for Efficient Ray Traversal*. Tech. rep., University of Bristol, Bristol, UK, UK, 1995. 6