ISSN 2464-4617 (print)
ISSN 2464-4625 (CD-ROM)
WSCG 2016 - 24th Conference on Computer Graphics, Visualization and Computer Vision 2016

Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility

Kevin Keul
University of Koblenz-Landau, Koblenz, Germany
keul@uni-koblenz.de

Stefan Müller
University of Koblenz-Landau, Koblenz, Germany
stefanm@uni-koblenz.de

Paul Lemke
University of Koblenz-Landau, Koblenz, Germany
lemke@uni-koblenz.de

ABSTRACT

We propose an efficient approach to precompute and reuse visibility information based on existing spatial data structures by using a precomputed data structure: the line space. This data structure provides an additional skip condition by checking whether the subnodes in a hierarchical spatial data structures need to check for intersection with the ray. We evaluate this method on different test scenes and show that it is able to achieve a remarkable speed-up by using this skip condition. Furthermore we describe algorithms for fast set-up and traversal in detail and discuss important strategies for this approach.

Keywords
Visualization, Computer Graphics, Ray Tracing, Data Structures, Visibility Algorithms

1 INTRODUCTION

The basic principle of ray tracing is that every visual effect is computed with rays that search for the nearest primitive in a given direction from a known starting point. When this intersection is found, more rays starting from there on can be processed. With this it is easy to calculate effects like shadows, reflexions and refractions with only one additional ray for each effect. With even more additional rays one can compute complex visual effects such as ambient occlusion or indirect lighting.

However, the quality of rendering comes with long rendering times, where even the slightest improvement can make a significant difference. The main limiting factor is the time it needs to compute the nearest intersection with the scene geometry. Therefore it is important to use an acceleration data structure which supports the task of finding the nearest intersection in an efficient way. Many of the data structures used today aim to subdivide the scene or the world space in such a way that the scene is equally distributed over every subunit in the data structure.

While this is already a studied field of research our approach goes beyond that. We try to precompute visibility tests on possible directional shafts additionally to the main data structure. Those precomputations should support the data structure by providing it with an additional condition to decide if it is possible to skip the main intersection computations. With this we achieve a performance speed-up compared to the already created acceleration data structure. Though it is a directional precomputation, we compute every possible direction and so enable visibility tests on the whole space with every possible starting and end point. Through this it is not only a speed-up for the initial coherent rays, but for every possible ray.

Moreover we try to combine the ideas of existing spatial data structures and extend the used traversal algorithms to optimize the achieved speed-up. For this we build a tree with a higher branching factor compared to the typically used data structures like the octree. In this paper we call it the *N*-tree because of the arbitrarily branching factor which can be dynamically chosen. With a higher factor it is possible to skip more spatial groups of elements thanks to a single test with our directional data structure.

2 RELATED WORK

In the past decades numerous data structures for accelerating ray tracing have been created and improved. Most of them aim to reduce intersection computations with the scene geometry by using spatial subdivision of the scene itself. An obvious way for this is to divide the total space with a simple cartesian grid, called the uniform grid, where every cell, called voxel, has the same size. For the traversal of this data structure it is possible to use known algorithms which are mostly based on Bresenham's 2D line drawing algorithm [Bre65], for example the 3D-DDA (3D Digital Differential Analyzer) algorithm introduced by Amanatides and Woo [AW87]. Today the use of grids benefits from quite efficient voxelization algorithms [ED06][ED08][SS10]. The biggest disadvantage of uniform grids is the variance between cells, so that in most cases there not only exist cells containing many scene candidates but also cells which are completely empty. Nevertheless it was shown that in some cases the use of uniform grids results in a significant performance gain in ray tracing [HKH11].

Hierarchical data structures are one kind of improvement. The goal is to have a high level of hierarchy and therefore a high resolution in those areas where there are many scene objects. Recursive grids were shown to work well with objects of varying density by recursively subdividing those voxels of the grid containing many scene candidates. Jevans and Wyvill [JW88] used an adaptive subdivision method where the branching factor of a voxel was higher the more scene candidates it contains. Octrees have a constant branching factor of 8 subvoxels per voxel. All subvoxels within a voxel have the same size so the subdivision of a voxel is exactly in the center point. The traversal can be done bottom-up, as by Samet [Sam89], or top-down, as introduced by Revelles, Ure*ñ*a and Lastra [RUnL00]. By using KD-trees [Hav00] one tries to achieve better distributions of scene objects to subvoxels as octrees. For this the split of the voxel is not necessarily in the center point but along the axis aligned plane, which seperates the containing scene objects in half. Extensions try to improve the scalability via SIMD commands [WPS*03][RSH05] and GPU advantages in stack based implementations [EVG04][FS05] as well as stackless implementations [PGSS07]. Binary space partitioning (BSP) trees, subdividing the space along arbitrary axes, have been used [SS92][KM07] and as shown in [TI08] they are more efficient as KD-Trees but need longer build times due to more complex construction algorithms.

Another approach to reduce computational overhead is to use bounding volumes around scene objects instead of spatial ordering. Bounding volume hierarchies (BVH) [KK86] apply k-DOPs, spheres or other kinds of proxy geometry and the traversal applies typical tree search and sorting techniques to reduce the complexity. Bounding interval hierarchy (BIH), a variation of axis-aligned bounding box trees, was used to great extent [NS04][WK06]. Advancements of these try to use SIMD parallelism [RSH05]. One way to do this is to use multi bounding volume hierarchies (MBVH) which in contrast to regular BVH store an arbitrarily number of subnodes according to the level of SIMD instructions [EG08]. Recently there have been implementations for the GPU using stackless MBVH [ÁSK14] and GPU accelerated construction [KA13].

Other acceleration methods try to take the visibility into account. Arvo and Kirk presented 5D volume structures, starting at a 3D object with a 2D angle [AK87]. They achieved a notable performance gain but due to its camera dependence it needs to be rebuild regularly whereas our data structure is independent from the camera position. Visibility preprocessing for urban scenes was used in the way of identifying blocker primitives by Bittner et al. [BWW01] and Leyvand et al. [LSCO03]. They also use the notation "line space" but it has a different meaning compared to our usage. Visibility precomputations have been a big topic in radiosity calculations [CW12]. In this context Drettakis and Sillion [DS97] used line space computations to precompute visibilities in a very similar way as we do. In their paper a line is considered as a link between two arbitrary surface elements surrounded by a shaft, covering all potential rays between both surface elements. Shaft culling was further used to optimize radiosity calculations by Haines and Wallace [HW94].

3 OVERVIEW

Our goal is to extend typical hierarchical acceleration data structures by precomputated visibility tests based on lines and shafts. With this the extended data structure performs just one additional visibility operation per node traversal for a given ray, which is done right before the intersection tests of the ray with the subjects within the current node. If this operation fails, the following intersection tests of the ray and the node subjects can be skipped completely. Note that the subjects of the node can be the objects of the scene contained by this node as well as all its own subnodes. Like most acceleration data structures we do not aim to work with dynamic scenes, so the set-up of the data structure does not need to be able to compute in real time. Our goal is to speed up ray tracing of static scenes and therefore only compute the data structure once initially.

3.1 N-tree as initial data structure

As the base data structure we use the N-tree, a variation of the recursive grid [JW88] with fixed branching factor, which benefits the most from our visibility data structure, due to reasons which are explained later on. Every edge of one N-tree node is divided in N equally long parts. We need to have our subnodes equal in size for our visibility test, which is also explained later. Therefore, we are not able to use arbitrary splitting points like in KD-Trees, where different subnodes of one node may differ in size. Although it is possible to store scene objects (the candidates) in every hierarchical level of the N-tree, our performance results suggest that only leaf nodes should contain candidates. Every node of the N-tree is either a leaf node and contains scene objects as candidates or consists of $N \times N \times N$ subnodes.

One can easily observe that the two main variables, N and the maximum depth of the tree (for further examples d), can be arbitrarily chosen and different selections of the values can give similar results. For example, do either $N = 2, d = 6$ (which resembles the typical octree) as well as $N = 8, d = 2$ result in a resolution of $64 \times 64 \times 64$ entries on the deepest hierarchy level. One observable difference lies in memory consumption in sparsely filled trees, where a higher N results in more memory usage due to a higher number of empty subnodes.

Algorithm 1 The traversal algorithm

```
1:  procedure TraverseNode(Ray r, Node n)
2:      $p \leftarrow 0$
3:      if n has primitives then
4:          $p \leftarrow$ nearest primitive intersecting $r$ within $n$
5:      else if n has subnodes then
6:          while $p = 0$ and subnodes left do
7:              $s \leftarrow$ next subnode in direction of r
8:              if s is non-empty then
9:                  $p \leftarrow$ TraverseNode$(r, s)$
10:             end if
11:         end while
12:     end if
13:     return p
14: end procedure
```

The pseudo code of the traversal algorithm for the N-tree is shown in Algorithm 1. In principle it is divided into two parts. At first, the exact start node has to be found. Starting at the root node the next inner subnode is chosen until the leaf node is reached. With this the main traversal starts. Every processed node has either candidates, which are tested for intersection with the current ray (lines 3 and 4), or has subnodes, which are recursively processed. All candidates within a leaf node have to be tested, but if at least one intersection is found, the traversal algorithm can stop. The step from one node to the subnodes follows a top-down strategy as proposed by Revelles et al. [RUnL00]. Like explained above, in our case it is not possible that a node has both, candidates and subnodes. As proposed by Amanatides et al. [AW87], the subnodes are traversed in a grid like manner (line 7). If a subnode neither contains candidates nor subnodes, it does not need to be processed at all and can be skipped in the traversal (lines 8-10). The algorithm continues with the next subnode. In the following, those subnodes are called "empty". The loop can stop if a primitive is found within a subnode (lines 6).

Figure 4a shows an exemplary traversal of the N-tree. The ray starts at the origin O within the node starting from S. At this point, every intersected subnode needs to be checked although neither the geometry nor any subnode containing the geometry is intersected by the ray.

3.2 Visibility Information with the line space

The line space builds upon the presented N-tree and extends it with an additional visibility test which decides whether a node can be skipped in the traversal. Note that this additional skip condition still works if the node has both, candidates and subnodes. Like explained above, a node contains of $N \times N \times N$ subnodes. Furthermore, each side of the nodes' bounding volume divides in $N \times N$ smaller sides with equal size, which makes a total of $6 \times N \times N$ smaller sides in the volume. These smaller sides are countable and each of these gets its own identifiable index. It is now possible to create shafts from every possible index to every other possible index. For each of those shafts it is decidable whether there exists at least one subnode partially or in total within the shaft that contains either candidates or subnodes itself. If a shaft has only empty subnodes, in other words the shaft does not intersect any subnode that is non-empty, the shaft itself is called empty.

The line space for a given node contains the information whether a shaft is empty or non-empty for every possible shaft within this node. It can be represented as 2D array or texture where the first axis stands for every possible start index and the second axis stands for every possible end index of sides. So, the pixel with the coordinates x and y denotes the shaft starting at the side with the index x and ending at the side with the index y. The value of the pixel represents whether the corresponding shaft is empty or intersects with at least one non-empty subnode.

In the step of deciding whether a shaft is empty, we use subnodes instead of the discrete scene geometry for two reasons. On the one hand, the scene geometry is already arranged in the subnodes of the N-tree and possibly quite many primitives of the scene result in just a few subnodes. On the other hand, the correspondence between the shafts and all intersected subnodes can be precomputed resulting in masking, which is further explained in the next paragraph. For these reasons it is possible to accelerate the construction of the line space effectively. One drawback to this is that there might be some subnodes within a shaft that only contain primitives of the scene that do not intersect with the shaft. Therefore, it would not be necessary to mark the corresponding entry in the line space. Anyway, it is marked because of the intersection between the subnode and the shaft. This results in possibly longer calculation times during traversal but especially with a big N it becomes negligible.

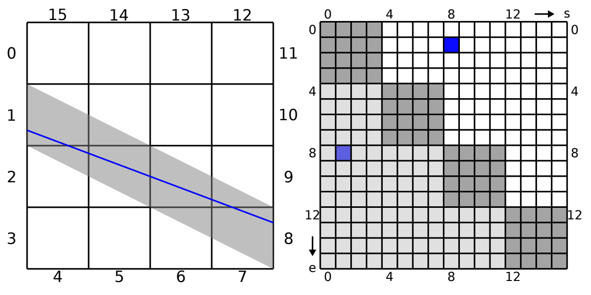


Figure 1: (a) An empty 2D scene with one exemplary ray and the belonging shaft between the start index 1 and the end index 8 and (b) the corresponding line space where the shaft is marked in blue.

Figure 1 demonstrates the relation between a node containing subnodes and the corresponding line space in 2D. There, the bounding box of a scene is subdivided into a 2D N-tree with $N = 4$ consisting of $4 \times N = 16$ elements. The border edges are numbered from 0 to 15. Each shaft is identified by the tuple of the start index and the end index of the sides. As a result the line space is of size 16×16, where each index tuple represents a shaft in the scene. The blue line in the left image is represented by index (1,8) or (8,1) respectively.

A few trivial properties help to reduce the memory capacity of the line space:

1. *LS(s; e) = LS(e; s)*: The line space is symmetric and the upper right triangle grants sufficient information.
2. *LS(s; s) = 0*: The elements of the diagonal characterize degenerated shafts with zero volume and can therefore be omitted.
3. *Coplanarity (Collinearity in 2D case)*: Shafts between coplanar sides are also degenerated, leading to blocks around the diagonal.

In 2D each of the 4 bounding sides contains N subsides so the total number of entries in the line space is $4N \times 4N = 16N^2$. Using the collinearity this can be reduced by $4N^2$ and afterwards divided in half due to symmetry, resulting in a total number of entries of size $6N^2$. In 3D each of the 6 bounding sides of the bounding box of the node contains $N \times N$ subsides and therefore the line space has $6N^2 \times 6N^2 = 36N^4$ entries in total. In the same way as for the 2D case this can be reduced to a total of $15N^4$ entries due to coplanaraty and symmetry. Note that this is only the size of the line space for a single node and therefore the memory consumption is quite high with a big N. In our test cases we found that $N = 10$ is sufficient for most cases and the memory consumption is appropriate. All entries for one line space are stored in a list and accessed with an identifier. This identifier is independent from symmetry and results in the same entry for both of the symmetric cases.

3.3 Set-up of the line space

Figure 2 shows the relevance of one non-empty subnode (marked in red) to the line space. On the left side for each possible start index it is shown which shafts count as non-empty because of the marked subnode. The right side shows the corresponding line space where exactly those pixels are marked that belong to non-empty shafts. Note that if only the marked subnode is non-empty, the line space would always result in this outcome. It is not relevant how many scene primitives are contained in this subnode or how they are located. So, the resulting line space which is presented can serve as a mask for this subnode.

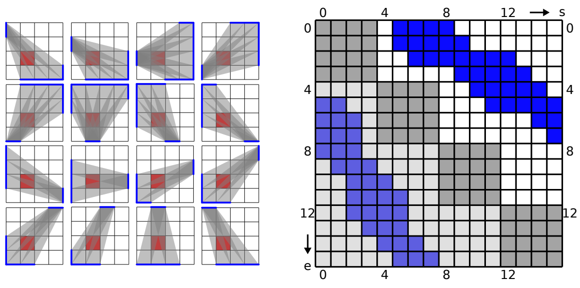


Figure 2: (a) All shafts covering one subnode (red) in 2D and (b) the resulting line space (bit mask). Every shaft with start index x and end index y fills the corresponding pixel in the line space. Symmetry and collinearity of the line space are quite obvious.

With this it is possible to precompute the masks for every possible subnode within a node and combine them to a mask atlas. This results in N^3 masks (one for each possible subnode) for the mask atlas, which then contains $N^3 \times 15N^4$ entries in total.

The pseudo code for the set-up algorithm for all line spaces of every node in the N-tree is shown in Algorithm 2. Our approach works in a top-down way starting with the root node. A line space for a node is only necessary, if the node itself contains subnodes. Every line space is computed with the help of the mask atlas. For every non-empty subnode of a node all entries of the corresponding masks are combined and result in the line space of the current node (lines 4-6). In the binary case, where it only matters whether a shaft is empty or not, this combination can be done with a simple "or" operation for every entry of the mask with the corresponding entry in the line space. The line spaces of every subnode consisting of subnodes itself are then computed recursively (lines 7-9).

Algorithm 2 Calculation of Line Space starting in the root node

```
1:  procedure CALCLINESPACE(Node n)
2:      LS ← create LineSpace for n
3:      for all subnodes $s \in n$ do
4:          if s is non-empty then
5:              mask ← mask denoted by s in n
6:              ADDMASKTOLINESPACE(LS, mask)
7:              if s has subnodes then
8:                  CALCLINESPACE(child)
9:              end if
10:         end if
11:     end for
12: end procedure
```

Figure 3 presents an example for a 3D line space. As with the previous examples, N is set to 4. It is obvious that the line space is much more complex compared to a 2D line space. Where in the 2D case every side is subdivided in 4 smaller parts, making it a total of 16 subsides, in the 3D case every of the 6 bounding sides is subdivided in 16 smaller sides and therefore making a total of 96 possible start and end sides. Figure 3b shows the mask for one subnode (marked in red). For every start index from $s \in 0..95$, a one bit entry provides the information, whether the shaft to end index $e \in 0..95$ intersects this subnode. In the shown example, we have 9 resulting shafts for the starting patch $s = 37$ which can be seen in the red column of the line space.

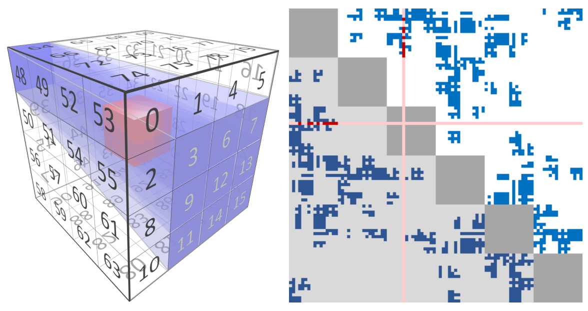
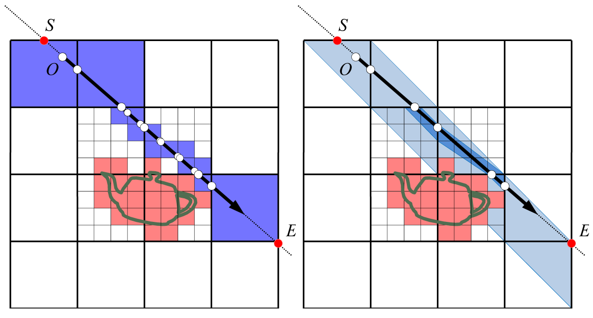

(a) (b)

Figure 3: (a) All shafts in 3D intersecting the red subnode from the start index with index 37. (b) Line space bit mask (4^3 subnodes with 962 LS-entries) for the red subnode. Note that the subnode itself can be subdivided as well and can therefore include its own line space.

3.4 Traversal of the line space

The traversal of the line space is mostly equivalent to the traversal presented in algorithm 1. Indeed the presented algorithm is just extended by another skip condition, which can be added before the subnodes are processed (after line 5 in algorithm 1). The skip condition checks, whether the line space entry corresponding to the current node is marked. If this is not the case, it means that all subnodes within the current shaft are empty and therefore no subnode needs to be processed with the current ray. The shaft itself is determined by the precise start and end index within the node which are intersected by the ray. These have to be computed first in order to identify the shaft the current ray belongs to.



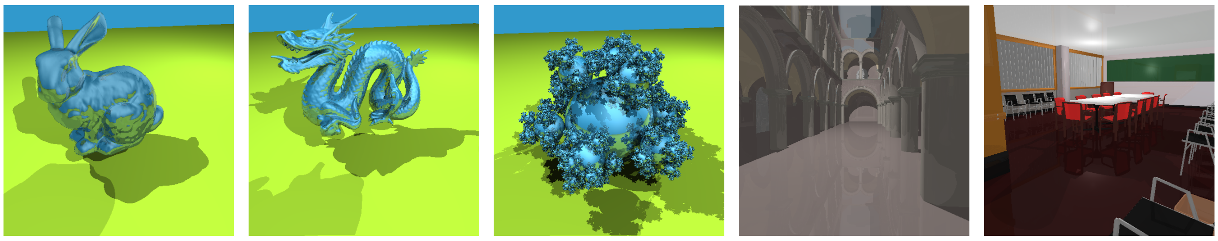
(a) (b)

Figure 4: (a) Traversal of the N-tree. Although no subnode containing geometry (red) is intersected by the ray, the algorithm traverses every possible subnode (dark blue) intersected by the ray. (b) traversal of the line space. Instead of testing every subnode intersected by the ray, it is first checked if the corresponding shaft intersects any non-empty subnodes. If this is not the case (like shown with the darker blue shafts), all subnodes within the shaft are skipped.

Figure 4 presents an exemplary traversal using the line space. For a given ray, we compute the intersection with the root node to determine the initial start index S and end index E. The x-, y-, z-coordinates of S and E are mapped to side indices of the root node surface, yielding the indices for the top level line space. In the example the top level shaft contains non-empty subnodes. Therefore, we select the subnode covering the ray origin O and from there on we start the traversal of the subnodes similar to the traversal of the N-tree. If one of these inner nodes is not subdivided, we check the candidate list of this node (if any) for intersection and continue with the next inner node, if no intersection is found. If the node is subdivided, we check the next level line space first with new start and end indices. If the shaft is not empty, we proceed with the traversal with smaller increments. In the example all inner shafts (in dark blue) corresponding to subnodes indicate that there are no non-empty inner subnodes and therefore these inner subnodes can be skipped at all.

4 RESULTS AND DISCUSSION

Our method was implemented in C++, exploiting SIMD operations (SSE) and multi-threading on a CPU. The results were evaluated on a PC with AMD Phenom II X6 1090T (6 cores, 3.5GHz) and 16 GB DDR3 RAM.



(a) BUNNY (69k triangles) (b) DRAGON (871k triangles) (c) SPHEREFLAKE (597k spheres) (d) DUBROVNIK SPONZA (66k triangles) (e) CONFERENCE ROOM (331k triangles)

Figure 5: Test scenes used for the performance measurements. Those include individual objects with a varying number of triangles (Bunny and Dragon), a fractal scene using spheres instead of triangles and architectural scenes with different number of triangles (Dubrovnik sponza and Conference room). The images were rendered using 3 light sources and multiple levels of reflection.

The used ray tracer computes intersection points for primary rays and up to 10 levels of reflections, where every primitive of the scene geometry is reflecting the ray. For every intersection with scene geometry 3 light sources are used for lighting and for each of those one shadow ray is evaluated. By using reflections and shadow rays we mostly work on more or less incoherent rays, which are traced by our method with no difference in comparison to coherent rays. All scenes were rendered with a resolution of 512×512 using different camera angles. The result is the average run time.

Multiple well-known test scenes with different characteristics and of different size of primitives have been used for evaluation (shown in figure 5). We divide those scenes in scenes showing individual objects only (Bunny and Dragon), architectural scenes (Dubrovnik sponza and Conference room) and a fractal scene (sphere flake using spheres instead of triangles). The individual objects represent the quality of the data structure for a single object only, where many primitives are concentrated in small space. For this purpose the Bunny is a model with a rather low number of primitives, whereas the Dragon consists of a lot of primitives. The architectural scenes represent conventional scenes, which may for example be used in games or films. We use the sponza as a scene with few primitives and the Conference room as scene with quite many primitives. The sphere flake is a fractal scene, which consists of a lot of primitives (spheres in our example). Those primitives are not concentrated in the center of the object, but are equally distributed.

For the N-tree and the line space we evaluate the size of the data structure and the runtime performance within our ray tracer. We compare those with the standard implementations of the uniform grid and the octree to show that the use of visibility information is an improvement of typical well-known spatial data structures. Furthermore we vary in the values of the two paremeters of the N-tree and the line space, which are the branching factor N and the maximal depth d, and investigate the differences in size and performance.

The results of the tests are shown in table 1. We evaluated several parameter sets for all data structures and only the best for each scene were considered. Note that the value of d belongs to the maximal depth of the data structure, which is not always needed. In scenes with a small number of primitives it is therefore possible that a big value of d does not provide any benefit.

The uniform grid grants good performance, especially in rather small scenes. The memory size used is in all test cases among the smallest. The optimal resolution for the uniform grid in most test scenes is 128^3 voxels in total. A higher resolution results in a higher traversal cost and a much higher memory consumption of the grid structure and might therefore only be beneficial in big scenes (like the dragon). In comparison to the uniform grid, the octree has a better performance in those big scenes (dragon and conference room), but worse in small scenes. The memory consumption depends on the value of d, where a small value results in a smaller memory consumption. In big scenes a big value of d is beneficial for performance but unfavorable for the required memory size.

The N-tree has a better performance than the octree, due to the higher branching factor, where every node is traversed in a grid-like manner. In most cases the N-tree performs similar to or better than the uniform grid, especially in the architectural scenes or in scenes with a high number of primitives. While a high value of N grants better performance, the higher branching factor results also in a bigger memory consumption, especially in sparsely filled N-trees. If a node is subdivided, it results in quite a lot of subnodes (N^3), even if only a few of them are actually needed. The optimal parameters of the N-tree in respect to the performance have been achieved with a value of N between 6 and 10. The optimal value of d is mostly either 3 or 4.

The line space, as an extension to the N-tree, is beneficial in all cases. Mostly it achieves a performance gain of up to 30% in comparison to the N-tree. In all test scenes the optimal parameters were the same as for the conventional N-tree. Obviously the additional usage

Scene		Uniform Grid	Octree	N-tree	Line Space
	parameters	128^3	d $\rightarrow$ 7	$N \rightarrow 9, d \rightarrow 3$	$N \rightarrow 9, d \rightarrow 3$
BUNNY	time per frame (s)	0,111	0,137	0,123	0,101
(69k triangles)	memory (MB)	78,4	55,2	82,5	106,7
	parameters	256^3	d $\rightarrow$ 9	$N \rightarrow 7, d \rightarrow 4$	$N \rightarrow 7, d \rightarrow 4$
DRAGON	time per frame (s)	0,327	0,332	0,297	0,253
(871k triangles)	memory (MB)	441,0	438,1	823,2	929,6
	parameters	128^3	d $\rightarrow$ 7	$N \rightarrow 8, d \rightarrow 3$	$N \rightarrow 8, d \rightarrow 3$
SPHEREFLAKE	time per frame (s)	0,151	0,208	0,179	0,129
(597k spheres)	memory (MB)	200,8	187,9	511,6	644,0
	parameters	128^3	d $\rightarrow$ 10	$N \rightarrow 10, d \rightarrow 3$	$N \rightarrow 10, d \rightarrow 3$
SPONZA	time per frame (s)	1,224	1,771	1,414	1,192
(66k triangles)	memory (MB)	80,4	55,1	220,0	294,7
	parameters	128^3	d $\rightarrow$ 10	$N \rightarrow 10, d \rightarrow 3$	$N \rightarrow 10, d \rightarrow 3$
CONFERENCE	time per frame (s)	1,395	1,593	1,300	1,089
(331k triangles)	memory (MB)	213,3	190,8	236,8	249,9

Table 1: Performance evaluations for the test scenes shown in figure 5. All scenes were rendered using 3 light sources and up to 10 reflections. We have compared typical data structures (uniform grid and octree) with the N-tree without and with the usage of the line space. Only the best parameter set in terms of traversal time for each data structure and each scene is shown.

of the line space results in a bigger memory consumption, where a high value of N is especially bad, because of the high number of possible shafts ($15N^4$) for every subdivided node. Due to the fact that only non-leaf nodes need a line space, this increment in memory size is quite acceptable in comparison to the total required memory size. While high values of N and d are a disadvantage in terms of memory consumption, they can be beneficial for traversal performance. A big value of N leads to long but slim shafts referring to many but small subnodes. If the shaft is empty, it therefore allows for a quick skip of many subnodes in just one computation. Moreover, long and slim shafts contain small subnodes. Even if these subnodes are intersecting the shaft only for a small part, the amount of subnode space outside of the shaft is just small in comparison to the length of the shaft.

An important observation is that the traversal performance and the memory consumption significantly depend on the values of the branching factor N and the maximal depth d. While the table shows only the best values for every data structure, it is observable that the results are different for the cases where the values for all data structures lead to the same resolution. One example for those values are a resolution of 512^3 for the uniform grid, a maximal depth of 9 for the octree and the values $N = 8$ and $d = 3$ for the N-tree and the line space. In those cases the size of the data strucutre and the performance of the traversal are way better for the N-tree in comparison to the uniform grid and the octree.

The main benefit of the N-tree comes with the usage of the line space. For this we evaluated the performance gain of the line space in comparison to the N-tree for different values of N and d. The results are shown in table 2. The evaluated test scene is the Bunny, but the results for the other scenes are similar and indicate the same results. In all test cases it is observable that the usage of the line space for a small value of N ($N < 5$) brings little to no benefit. The same applies to big values of N ($N > 10$). As explained above the reason for the former is that the shafts are wider if N is small and the amount of subnode space outside of the shaft is bigger in comparison to its length. While this is unproblematic for long shafts with a big value of N, the problem there is that the shaft loses the potential of prediction because of its length. One non-empty subnode is sufficient to mark the corresponding shaft, so that the traversal needs to check all subnodes . It is observable that big values of d may not make any difference in performance. The reason for this is that the geometry is sufficiently stored in higher nodes and therefore the maximum depth of d is not needed. Moreover, if the values of N and d are too big ($N > 10, d > 5$) the data structure is too memory consuming and therefore not usable. The benefit of the line space as well as the optimal choice of the parameters are scene dependant, but in all choices of parameters the usage of the line space results in better performance than the corresponding N-tree without line space.

5 CONCLUSION AND FUTURE WORK

We have presented a novel and effective extension to existing spatial data structures. First, the N-tree, a variation of the Octree, has been discussed. Based on this we introduced the line space as an advancement for the N-tree by taking directional visibility information into account. Algorithms for the set-up and the improved

Parameters		N-tree	LS	Δ
$N \to 5$,	time (s)	0,342	0,333	-2,7%
$d \to 3$	size (MB)	40,3	40,8	+1,1%
$N \to 5$,	time (s)	0,137	0,123	-9,9%
$d \to 4$	size (MB)	57,1	60,3	+5,7%
$N \to 5$,	time (s)	0,136	0,126	-6,9%
$d \to 5$	size (MB)	57,6	61,9	+7,5%
$N \to 6$,	time (s)	0,198	0,180	-8,8%
$d \to 3$	size (MB)	42,7	44,2	+3,6%
$N \to 6$,	time (s)	0,144	0,126	-12,9%
$d \to 4$	size (MB)	96,9	112,2	+15,7%
$N \to 6$,	time (s)	0,145	0,126	-12,9%
$d \to 5$	size (MB)	96,8	112,4	+16,0%
$N \to 7$,	time (s)	0,148	0,131	-11,6%
$d \to 3$	size (MB)	47,5	52,4	+10,4%
$N \to 7$,	time (s)	0,144	0,127	-11,9%
$d \to 4$	size (MB)	109,5	132,8	+21,3%
$N \to 8$,	time (s)	0,151	0,118	-21,9%
$d \to 3$	size (MB)	56,8	70,5	+24,0%
$N \to 9$,	time (s)	0,123	0,101	-18,2%
$d \to 3$	size (MB)	82,5	106,7	+29,4%
$N \to 10$,	time (s)	0,166	0,112	-32,9%
$d \to 3$	size (MB)	123,3	172,8	+40,1%

Table 2: Performance comparison between the N-tree without and with the usage of the line space (LS) for different parameter sets of N and d. It is shown that higher values for these parameters result in a bigger memory consumption but leads mostly to a smaller traversal time with the usage of the line space. The used scene is the Bunny as individual object, other scenes produce similar results.

traversal were shown. By using binary information for the possible emptiness of all shafts within one node we conclude whether it is necessary to test the subnodes of the current node or if we are able to skip them. This additional skip condition results in a notable speed-up for all shown test cases. From there on there exist multiple paths for further study.

The binary entries in the line space are enough for estimating whether a ray from one point to another might be intersected by scene geometry. With this information it is possible to compute approximated shadows without testing the scene geometry for intersection at all. This might be sufficient for shadow computations of non-primary rays. Even for primary rays the resulting error may become negligible with a high value of d. This technique might even be used in rasterization where the computation of soft shadows is a rather tough topic.

By using a counter instead of the binary entries within the line space the data structure can be updated during runtime and therefore it can possibly be fast enough to handle dynamic scenes in realtime. The counter is incremented for each object intersecting a shaft. Thus, the line space can efficiently be rebuilt by decrementing the counter if geometry is removed and by incrementing the counter if geometry is added.

An obvious option for faster set-up or better runtime performance is to port the data structure and the traversal to latest generation GPU architectures since many necessary tasks could benefit from parallel computation.

In this paper we presented that the line space as directional visibility data structure is able to improve existing spatial data structures. Moreover, in future work we try to extend the impact of directional visibility conditions to current state-of-the-art data structures like Bounding Volume Hierarchies. We think that some kind of line space structure could even improve these data structures resulting in a win in performance for latest generation ray tracing data structures.

Another attempt would be to not only save binary or integer information in the line space, but to save the list of candidates directly in the shafts instead of saving them in the nodes of the N-tree. This would result in several advantages. First, the candidates within the shaft can be sorted beforehand which would improve performance during runtime. Moreover, the traversal itself would work without the typical node structure based on voxels but rather based on shafts which is more accurate and efficient.

6 REFERENCES

[AK87] ARVO J., KIRK D.: Fast ray tracing by ray classification. In *ACM Siggraph Computer Graphics* (1987), vol. 21, ACM, pp. 55–64.

[ÁSK14] ÁFRA A. T., SZIRMAY-KALOS L.: Stackless multi-bvh traversal for cpu, mic and gpu ray tracing. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 129–140.

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. *Eurographics '87* (1987), 3–10.

[Bre65] BRESENHAM J. E.: Algorithm for computer control of a digital plotter. *IBM Syst. J. 4*, 1 (Mar. 1965), 25–30.

[BWW01] BITTNER J., WONKA P., WIMMER M.: Visibility preprocessing for urban scenes using line space subdivision. In *Computer Graphics and Applications, 2001. Proceedings. Ninth Pacific Conference on* (2001), IEEE, pp. 276–284.

[CW12] COHEN M. F., WALLACE J. R.: *Radiosity and realistic image synthesis*. Elsevier, 2012.

[DS97] DRETTAKIS G., SILLION F.: Interactive update of global illumination using a linespace hierarchy. In *Proceedings of ACM SIGGRAPH* (Aug. 1997), Annual Conference Series, pp. 57 – 64.

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (Redwood City, United States, 2006), ACM SIGGRAPH, pp. 71–78.

[ED08] EISEMANN E., DÉCORET X.: Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008* (Toronto, Ont., Canada, Canada, 2008), GI '08, Canadian Information Processing Society, pp. 73–80.

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), IEEE, pp. 35–40.

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization 2004* (2004), pp. 255–262.

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM, pp. 15–22.

[Hav00] HAVRAN V.: *Heuristic ray shooting algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[HKH11] HAPALA M., KARLIK O., HAVRAN V.: When it makes sense to use uniform grids for ray tracing. *Proceedings of WSCG'2011, Communication Papers* (Feb. 2011), 193–200.

[HW94] HAINES E. A., WALLACE J. R.: Shaft culling for efficient ray-cast radiosity. In *Photorealistic rendering in computer graphics*. Springer, 1994, pp. 122–138.

[JW88] JEVANS D., WYVILL B.: Adaptive voxel subdivision for ray tracing.

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), ACM, pp. 89–99.

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. *SIGGRAPH Comput. Graph. 20*, 4 (Aug. 1986), 269–278.

[KM07] KAMMAJE R. P., MORA B.: A study of restricted bsp trees for ray tracing. *Symposium on Interactive Ray Tracing 0* (2007), 55–62.

[LSCO03] LEYVAND T., SORKINE O., COHEN-OR D.: Ray space factorization for from-region visibility. In *ACM Transactions on Graphics (TOG)* (2003), vol. 22, pp. 595–604.

[NS04] NAM B., SUSSMAN A.: A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on* (June 2004), pp. 171–180.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum* (2007), vol. 26, Wiley Online Library, pp. 415–424.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. In *ACM Transactions on Graphics (TOG)* (2005), vol. 24, ACM, pp. 1176–1185.

[RUnL00] REVELLES J., UREÑA C., LASTRA M.: An efficient parametric algorithm for octree traversal. *Journal of Winter School of Computer Graphics 8* (2000), 212–219.

[Sam89] SAMET H.: Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics 13* (1989), 445–460.

[SS92] SUNG K., SHIRLEY P.: Ray tracing with the bsp tree. In *Graphics Gems III*, Kirk D., (Ed.). Academic Press, 1992, pp. 271–274.

[SS10] SCHWARZ M., SEIDEL H.-P.: Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 179:1–179:10.

[TI08] THIAGO IZE INGO WALD S. G. P.: Ray tracing with the bsp tree. *IEEE Symposium on Interactive Ray Tracing* (2008), 159–166.

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques 2006* (2006), 139–149.

[WPS*03] WALD I., PURCELL T. J., SCHMITTLER J., BENTHIN C., SLUSALLEK P.: Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports 1*, 3 (2003).