

# Ray Tracing with Sparse Boxes

Vlastimil Havran\*  
Czech Technical University

Jiří Bittner†  
Czech Technical University  
Vienna University of Technology

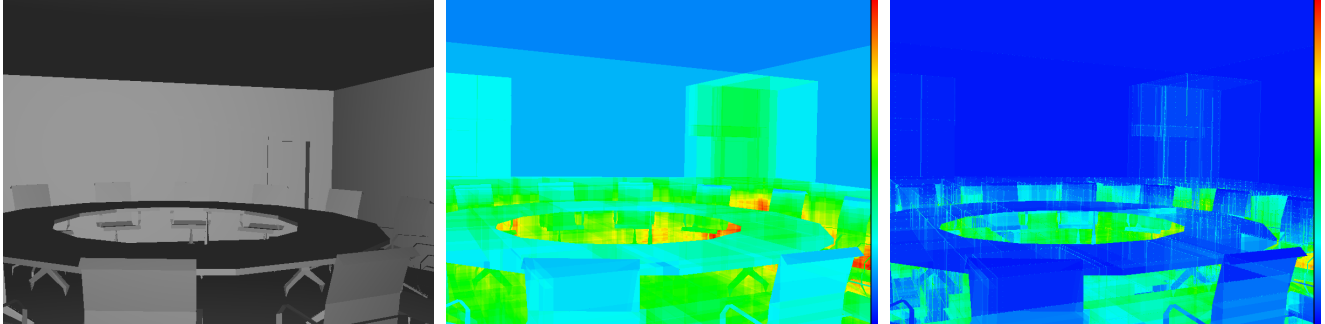


Figure 1: (left) A ray casted view of interior of a larger scene. (center) Visualization of the number of nodes traversed per ray for the traditional traversal algorithm. (right) The same visualization for our new algorithm. The dark blue color corresponds to zero traversal steps, dark red corresponds to 255 or more traversal steps per ray. Note that the new algorithm significantly reduces the number of traversals per ray.

## Abstract

We present a new method for ray tracing acceleration by modification of current spatial data structures. We augment a spatial hierarchy such as kd-tree with sparsely distributed bounding boxes that are linked from bottom to top. The augmented data structures can be viewed as a hybrid between kd-trees and uniform grids. The augmentation by boxes is used in new traversal algorithms, which are not restricted to start the hierarchy traversal at the root node. This is in particular efficient for short rays where we significantly reduce the number of traversal steps through the hierarchy. Further we make use of ray coherence as well as explicit knowledge of ray distribution and initiate the traversal near the leaf nodes where the intersection is expected.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

**Keywords:** ray tracing, data structures, visibility

## 1 Introduction

Ray tracing forms a basis of many current global illumination algorithms. Due to its logarithmic dependence on scene complexity it also becomes popular for rendering huge data sets in interactive applications.

\*e-mail: havran@fel.cvut.cz

†e-mail: bittner@fel.cvut.cz

The fundamental task of ray tracing is to determine visibility along a given ray, i.e. to find the closest intersection of the ray with the scene. In order to find the intersection as fast as possible the search must be limited only to the proximity of the ray. This can be achieved by organizing the scene in a data structure such as regular grid, kd-tree, octree, or bounding volume hierarchy. In particular kd-trees have become very popular for handling static scenes [Wald 2004] and bounding interval hierarchies for handling dynamic scenes [Woop et al. 2006]. The major advantage of hierarchies is their ability to adapt to irregular distribution of objects in the scene [MacDonald and Booth 1990]. However there is a cost for this ability: during ray tracing, every ray has to traverse the interior nodes of the hierarchy in order to reach the leaf nodes where the actual objects are stored. The number of traversed leaves is typically very small and thus the traversal of the interior nodes of the hierarchy is often the bottleneck. This problem has been addressed by using neighbor-links [MacDonald and Booth 1990; Havran et al. 1998]. However the neighbor-links have several issues which reduce or even eliminate their benefit: (1) They have considerable memory footprint, which reduces the savings of the traversal due to limited cache sizes, (2) every traversal step for neighbor-links is slightly more costly than the traversal of interior nodes, (3) they are costly to construct on the fly, which limits their usage to static scenes.

Newer methods of ray traversal optimization exploiting traversal coherence by casting whole packets of coherent rays. These methods rely on packets sharing a common origin [Reshetov et al. 2005] or packets with well defined boundaries [Havran and Bittner 2000].

In this paper we propose a simple method which significantly reduces the number of traversal steps in spatial hierarchies. Compared to neighbor-links the new method has significantly smaller memory footprint and performs faster traversal steps. Compared to ray bundle methods the proposed technique does not rely on well defined bundles, however it can still make use of ray coherence.

## 2 Related Work

Ray tracing has been dealt extensively in the field of computer graphics and computational geometry. Techniques aimed at the *worst case complexity* have been shown to have pre-processing time and storage  $O(n^{4+\epsilon})$  to achieve  $O(\log N)$  query time [Szirmay-Kalos and Márton 1998], where  $N$  denotes the number of objects in the scene. These algorithms count for all the possible worst case scenarios and provide theoretical guarantees [Agarwal 2004]. However, their space requirements severely restricts their use in practice.

The algorithms used in practical rendering packages try to reduce *average case* time complexity without any guarantees on the worst case scenarios. The algorithms are based on spatial decompositions organized typically as a hierarchy. The ray then traverses a hierarchy from a root along the ray path and identifies all the leaf nodes pierced by a ray. The objects in leaf nodes are tested against the ray until the first intersection along the ray is found.

The depth of the hierarchy is  $\text{const} \cdot \log_2 N$ , where  $N$  is the number of objects. According to the experimental data the number of visited leaves along a ray is 4 to 8 on average, and the number visited interior nodes is 20 to 200 on average (for scenes with 1,000 to 1,000,000 objects). While in the visited leaves actual intersections of the ray with scene objects are computed, the visited interior nodes constitute an overhead following from the usage of spatial hierarchy. MacDonald and Booth [MacDonald and Booth 1990] proposed a method for reducing the number of traversal steps of interior nodes for kd-trees. This method extends the original kd-tree with the links from the faces of the cells corresponding to the leaves to the neighboring leaves. The method is either referred to *neighbor-links* or ropes. A traversal algorithm then does not require a traversal stack and it can follow the sequence of leaves by repetitive point location of moving point along the ray path. The properties of the traversal algorithm based on neighbor-links and neighbor-link-trees have been further investigated by Havran et al. [Havran et al. 1998].

## 3 Algorithm Overview

Our new method builds on two main ideas: (1) augmenting the spatial hierarchy by sparse geometrical and topological information, (2) using ray coherence together with the augmented information to eliminate traversal of interior nodes of the hierarchy.

The augmentation of the spatial hierarchy consists of adding bounding boxes for selected hierarchy nodes as well as links to parent nodes which have also been augmented. This allows to initiate the hierarchy traversal from bottom of the tree and thus to avoid much of the traversal which would be performed when starting from the root node.

The ray coherence is exploited in different ways. For rays sharing a common origin we store a deepest augmented node containing the origin of the ray. This node is then used to initiate the traversal of the hierarchy. For secondary rays the traversal is initiated from the node containing the termination point of the generating ray (as this is the node containing the origin of the secondary ray). For shadow rays we store the node of the last shadow caster and use it to initiate the search for the shadow caster for subsequent shadow ray. For general rays which are assumed to be coherent we aim to reuse the sequence of deepest augmented node which have been traversed.

The paper is further organized as follows: Section 4 describes the augmentation of the spatial hierarchy with sparse boxes. Section 5

presents the new traversal algorithms based on sparse boxes. Section 6 contains the results from experimental measurements and their discussion. Finally, Section 7 concludes the paper.

## 4 Sparse Boxes

Our new traversal algorithms make use of explicit information about the geometry associated with nodes of a spatial hierarchy. This idea is not new as it has been used in algorithms based on neighbor-links [MacDonald and Booth 1990] (ropes) or rope-trees [Havran et al. 1998]. However, the published methods require significant memory footprint for storing both the geometrical and the connectivity information. In particular storing a bounding box requires 24 bytes and storing the neighbor-links another 24 bytes. In total the memory overhead is 48 bytes per node. For interior nodes this makes about 5 times more memory than required in the basic representation (splitting axis + splitting plane + 2 pointers). In turn the increased memory consumption negatively affects memory cache utility when traversing through the hierarchy which degrades the performance of the traversal algorithm.

Additional problem of neighbor-links is the setup-cost for the construction of neighbor-links which becomes important in the case of dynamic scenes.

In order to reduce the memory overhead, but keep the advantages of explicit geometrical information we distribute the bounding boxes over the spatial hierarchy only *sparsely*. We call the nodes augmented by a bounding box *augmented nodes*. We do not use neighbor-links between the augmented nodes, but instead we only store a single pointer to the parent augmented node (see Figure 2).

The augmented nodes are placed in the tree during the construction. The augmented node is constructed if the distance to the last augmented node is greater than  $d_{min}$ . If this criterion is not met, we construct a simple interior node. In our experiments we used  $d_{min} \in \langle 1, 3 \rangle$ , see the results in Section 6.

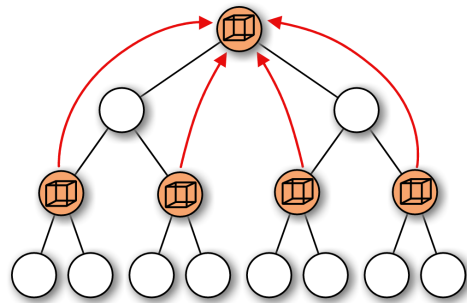


Figure 2: A spatial hierarchy augmented with bounding boxes. The bounding boxes are only stored at some levels of the hierarchy in order to reduce their memory footprint.

## 5 Traversal Algorithms

In this section we first briefly recall the hierarchical traversal algorithm, then we describe the extension that uses sparse boxes, and finally we describe the algorithm that reads and saves the sequence of boxes for coherent rays.

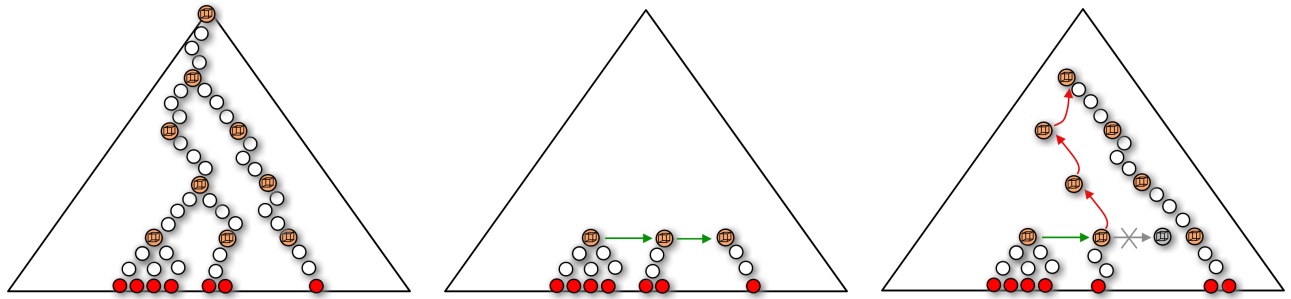


Figure 3: Comparison of hierarchical traversal algorithm and the new method. (left) Traditional traversal algorithm starts at the root node and traverses the tree downwards to the leaf node. In this example it traverses 42 interior nodes. (center) The new traversal algorithm starts the traversal at the nodes recorded for the previous ray. If a similar ray pierces the same sequence of nodes, most hierarchy traversal is avoided (12 interior nodes traversed). (right) If another ray pierces different nodes, we need to traverse up in the hierarchy to find first valid starting point for the down-traversal. In this example in total 22 interior nodes were traversed.

### 5.1 Traditional traversal algorithm

The traditional hierarchical traversal algorithm always starts from the root node of a kd-tree. In each interior node (including root node) the four cases are distinguished by two conditional operations:

1. traverse only to the left child node,
2. traverse only to the right child node,
3. first traverse to the left child node and then traverse to the right child node,
4. first traverse to the right child node and then traverse to the left child node.

If the two child nodes are traversed, the farther child is stored on the stack together with the exit signed distance. When a child node is visited, the ray is tested against the objects references in the leaves. After the leaf is accessed we take the first node from the stack and we continue the traversal until the ray-object intersection is found or the stack is empty (= no intersection). When we access the augmented interior nodes with the boxes, the information about boxes can be simply ignored during the traversal. Further details and the issue of robustness of traversal algorithms are elaborated in [Havran 2001].

### 5.2 Bottom-up traversal algorithm

The traditional hierarchical traversal algorithm described above always starts from the root node. If the hierarchy is deep, many traversal steps are needed to reach leaf nodes which contain the actual geometry.

The traversal algorithm with sparse boxes can start the traversal at any augmented node in the hierarchy. This node can be determined either by high-level knowledge (all primary or shadow rays start at a common point, secondary rays start at the termination of the generator ray) or by coherence (subsequent rays will likely start the traversal from the same node). If this node is deep inside the hierarchy and the number of traversed leaves is small, we can save most traversal of the interior of the hierarchy.

The traversal algorithm proceeds as follows: push the node on the traversal stack and perform the stack-based traversal. When the traversal stack becomes empty and no intersection has been found we need to determine the next node to continue the traversal from.

In order to do that we first compute an exit point with respect to the last traversed augmented node. This exit point is treated as a new origin of the ray. We follow the link to the parent augmented node until we find a node whose box contains the new ray origin. This node is used as a new starting node and the algorithm continues until an intersection is found or the ray leaves the scene. The latter case can be identified in a situation where the new origin of the ray lies outside of the bounding box associated with the root node.

### 5.3 Coherence based traversal algorithm

The first ray starts the traversal from the root node using the hierarchical stack-based traversal algorithm described in Section 5.1. During this traversal we store a sequence of deepest augmented nodes which have been traversed. These nodes are used to initiate the traversal for the next ray.

The traversal of subsequent rays proceeds as follows: if the origin of the ray lies inside the first node of the traversal sequence, we push this node on the traversal stack and start the stack-based traversal. If the origin does not lie inside the box or the sequence is empty, we follow the link to the parent augmented node until we find a node which contains the origin. The found node is pushed on the stack and the stack-based traversal starts from this node.

When the traversal stack becomes empty and no intersection has been found we need to determine the next node to continue the traversal from. In order to do that we first compute an exit point with respect to the last traversed augmented node. This exit point is treated as a new origin of the ray. If the traversal sequence is not empty, we remove the first node from the sequence. Then we apply the traversal algorithm described above. An example of the traversal using the hierarchical traversal algorithm as well as the new traversal algorithm is shown in Figure 3.

Note that during the traversal we store a traversal sequence of augmented nodes which will be used for the next ray. This is easily achieved by maintaining double buffer for the traversal sequences. Note that storing the traversal sequence can be relatively expensive as it requires writing to the main memory. In order to amortize this cost we can update the sequence with lower frequency, i.e. only for every  $m$ -th ray. The optimal selection of  $m$  depends on the particular hardware architecture as well as the scene properties. In our experiments using  $m = 2$  provided stable results. The pseudocode of the coherence based traversal algorithm is outlined in Algorithm 1.

**Algorithm 1:** Coherence based ray traversal

```

CastRay()
begin
  current = 0 ;           /* index in the input box sequence */
  last = 0 ;             /* index in the output box sequence */
  tentry = 0 ;          /* entry signed distance */
  textit = 0 ;         /* exit signed distance */
  ComputeMinMaxT(ray, seq[0], tentry, textit);
  ReadFromSequence();
  while not(stack.empty()) do
    while not(stack.empty()) do
      stack.pop(node, lastAugmentedNode, tentry, textit);
      if node.IsLeaf() then
        if ComputeIntersections(node) then
          return "object intersected";
        end
        tentry = textit + eps ;      /* shift the ray origin */
        WriteToSequence();
      else
        TraverseNodeDown(node);
      end
    end
    ReadFromSequence();
    if stack.empty() then
      TraverseNodeUp();
    end
  end
  return "no object intersected"
end

WriteToSequence()
begin
  if writeSeq[last]  $\neq$  lastAugmentedNode then
    writeSeq[last++] = lastAugmentedNode;
  end
end

ReadFromSequence()
begin
  repeat
    current++;
    ComputeMinMaxT(ray, seq[current], tmin, tmax);
  until tentry > tmax ;
  ; /* skip nodes which are before the ray origin */
  if tentry  $\geq$  tmin then
    stack.clear() ; /* reset the stack (if any) */
    stack.push(seq[current].node, seq[current].node, tmin+eps, tmax);
  else
    do nothing ; /* this entry is not usable (yet) */
  end
end

TraverseNodeDown(node)
begin
  t = Intersection(node.plane, ray);
  if IsAugmented(node) then
    lastAugmentedNode = node;
  end
  if Second Intersected Child then
    stack.push(Second Intersected, lastAugmentedNode, t, textit);
  end
  stack.push(First Intersected, lastAugmentedNode, tentry, Min(textit, t));
end

TraverseNodeUp()
begin
  /* Find the first parent augmented node containing the shifted ray origin */
  node = lastAugmentedNode;
  while node.parent do
    ComputeMinMaxT(node.parent, tmin, tmax);
    if tmin  $\leq$  tentry < tmax then
      stack.push(node.parent, node.parent, tentry+eps, tmax);
      return true ; /* found an augmented node */
    end
  end
  return false ; /* did not find an augmented node */
end

```

## 6 Results and Discussion

We have evaluated the new algorithms inside an optimized ray tracer. In the evaluation we have used two different scenes: a model of a sports stadium consisting of 1.5M triangles (arena) and a model of a city containing 0.8M triangles (city). For all tests we have used kD-tree constructed according to the surface area heuristics [MacDonald and Booth 1990]. Note that the time required for the spatial data structure construction is the same for the traditional algorithm and as well as the new algorithm.

The first set of tests evaluated the reduction of traversal steps, the speedup of the traversal and the speedup of the overall rendering time. We have measured 10 representative viewpoints inside the tested scenes. Table 1 summarizes the results for primary rays.

method	$r_{ITM}$ [-]	$N_L$ [-]	$N_I$ [-]	$T_R$ [s]	ratio [-]	speedup [-]
arena, 1.4M triangles, 7.1M kD-nodes						
<i>HTR</i>	25.4	6.8	70.77	1.69	1.0	1.0
<i>BTR</i>	25.4	6.8	50.76	1.47	0.72	1.15
<i>STR</i>	25.4	6.8	16.99	1.25	0.24	1.35
city, 0.8M triangles, 4.0M kD-nodes						
<i>HTR</i>	16.3	6.0	48.39	1.10	1.0	1.0
<i>BTR</i>	16.3	6.0	33.87	1.12	0.70	0.98
<i>STR</i>	16.3	6.0	29.62	1.16	0.61	0.95

Table 1: Summary of results for different traversal algorithms. *HTR* is the traditional hierarchical traversal algorithm described in Section 5.1, *BTR* is the bottom-up traversal algorithm described in Section 5.2, *STR* is coherence based traversal algorithm described in Section 5.3.  $r_{ITM}$  is the number of ray-object intersections per ray,  $N_L$  is the number of traversed leaves per ray,  $N_I$  is the number of traversed interior nodes per ray, and  $T_R$  is the computation time in seconds. By 'ratio' we denote the number of traversed interior nodes for the proposed algorithm and for the original algorithm, by 'speedup' we denote the speedup obtained from  $T_R$ .

The second set of tests analyzed the behavior of the method with respect to its parameters. In particular we have changed the number of bounding boxes distributed inside the spatial hierarchy and the update rate of the traversal sequences. The results for primary rays are summarized in Table 2.

The results show significant reduction of traversal steps for interior nodes of the hierarchy. However the savings in total rendering time do not directly correspond to the saved traversal steps. This follows from the induced overhead of the new traversal algorithm since traversing the augmented nodes is more costly. In particular restarting the traversal in the augmented nodes requires computation of the exit point and the verification of the position of the exit point with respect to the next candidate node (parent node or the next node from the traversal sequence).

In the arena scene the new algorithm achieves a total speedup of rendering up to 1.35. This scene has a very high depth complexity and many views with relatively short rays, which create a good potential for the new method. The profiling indicates that in this scene about 40% of time for rendering is spend in the actual traversal routine (the rest of the time corresponds to computing ray triangle intersections and shading). The speedup of rendering of 1.35 thus means that the actual speedup of the traversal is 1.6.

The city scene has a smaller depth complexity and longer rays and for this scene the new method actually results in a slowdown of 0.98. This indicates that the current version of the algorithm brings speedup only for scenes with very high depth complexity and short rays.



Figure 4: The view of the 'arena' scene and the 'city' scene.

$d_{min}$	$N_B$	$M_{KD}$	$N_L$	$N_I$	$T_R$
arena – HTR					
–	0	89MB	6.8	70.77	1.69
arena – BTR					
1	3.6M	231MB	6.8	50.25	1.75
2	2.0M	171MB	6.8	50.76	1.47
3	0.84M	122MB	6.8	53.48	1.50
arena – STR					
1	3.6M	231MB	6.8	15.54	1.28
2	2.0M	171MB	6.8	16.99	1.25
3	0.84M	122MB	6.8	23.93	1.32
city – HTR					
–	0	42 MB	6.0	48.39	1.10
city – BTR					
1	2.0M	122MB	6.0	32.98	1.16
2	0.86M	76MB	6.0	33.87	1.12
3	0.66M	68MB	6.0	35.63	1.13
city – STR					
1	2.0M	122MB	6.0	28.21	1.16
2	0.86M	76MB	6.0	29.62	1.16
3	0.66M	68MB	6.0	32.38	1.16

Table 2: The results for the traversal algorithm for different distribution of boxes in the kd-tree. The  $d_{min}$  shows the depth difference between the augmented nodes used in the kd-tree construction algorithm,  $N_B$  is the number of boxes stored in kd-tree,  $M_{KD}$  is total memory required for the kd-tree in MBytes,  $N_L$  is the number of traversed leaves per ray,  $N_I$  is the number of traversed interior nodes per ray, and  $T_R$  is the computation time in seconds.

## 7 Conclusion

We presented a method for ray tracing acceleration, which speeds up the traversal of the spatial hierarchy. In particular the proposed method reduces the overhead of traversing interior parts of the hierarchy. The method is based on augmenting the hierarchy by sparsely distributed bounding boxes of hierarchy nodes and linking these augmented nodes together. The sparse distribution of the boxes reduces the memory cost for the augmented data, while still allowing significant reduction of the traversal steps.

We proposed new traversal algorithms which exploit the sparse boxes and their topology. These algorithms are not restricted to traversal starting at the root node of the hierarchy and thus they can make use of various types of ray coherence. The major advantage to the already published algorithms is that the additional data storage is very low and that there is no need of pre-processing.

We discussed algorithms for coherent rays sharing a common origin or more general coherent rays. The implementation of the method is straightforward. We have shown that the method brings a small improvement of ray tracing performance especially for short rays.

In the future we want to focus on analysis of the behavior of the

method in dependence on the distribution of the bounding boxes inside the hierarchy. Additionally, we want to modify the algorithm for tracing packets of rays in a single traversal pass.

## Acknowledgments

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program MSM 6840770014 and LC-06008 (Center for Computer Graphics) and by the EU under the project no. IST-2-004363 (GameTools). The Arena scene is a courtesy of Digital Media Production a.s.

## References

- AGARWAL, P. 2004. Range Searching. In *CRC Handbook of Discrete and Computational Geometry* (J. Goodman and J. O'Rourke, eds.), CRC Press, New York.
- HAVRAN, V., AND BITTNER, J. 2000. LCTS: Ray shooting using longest common traversal sequences. *Computer Graphics Forum (Proc. Eurographics '2000)* 19, 3 (Aug), C59–C70.
- HAVRAN, V., BITTNER, J., AND ŽÁRA, J. 1998. Ray tracing with rope trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)*, 130–139.
- HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague.
- MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer* 6, 6, 153–65. criteria for building octree (actually BSP) efficiency structures.
- RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics* 24, 3, 1176–1185. (Proceedings of ACM SIGGRAPH).
- SZIRMAY-KALOS, L., AND MÁRTON, G. 1998. Worst-case versus average case complexity of ray-shooting. *Computing* 61, 2, 103–131.
- WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.
- WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware (2006)*, 67–77.



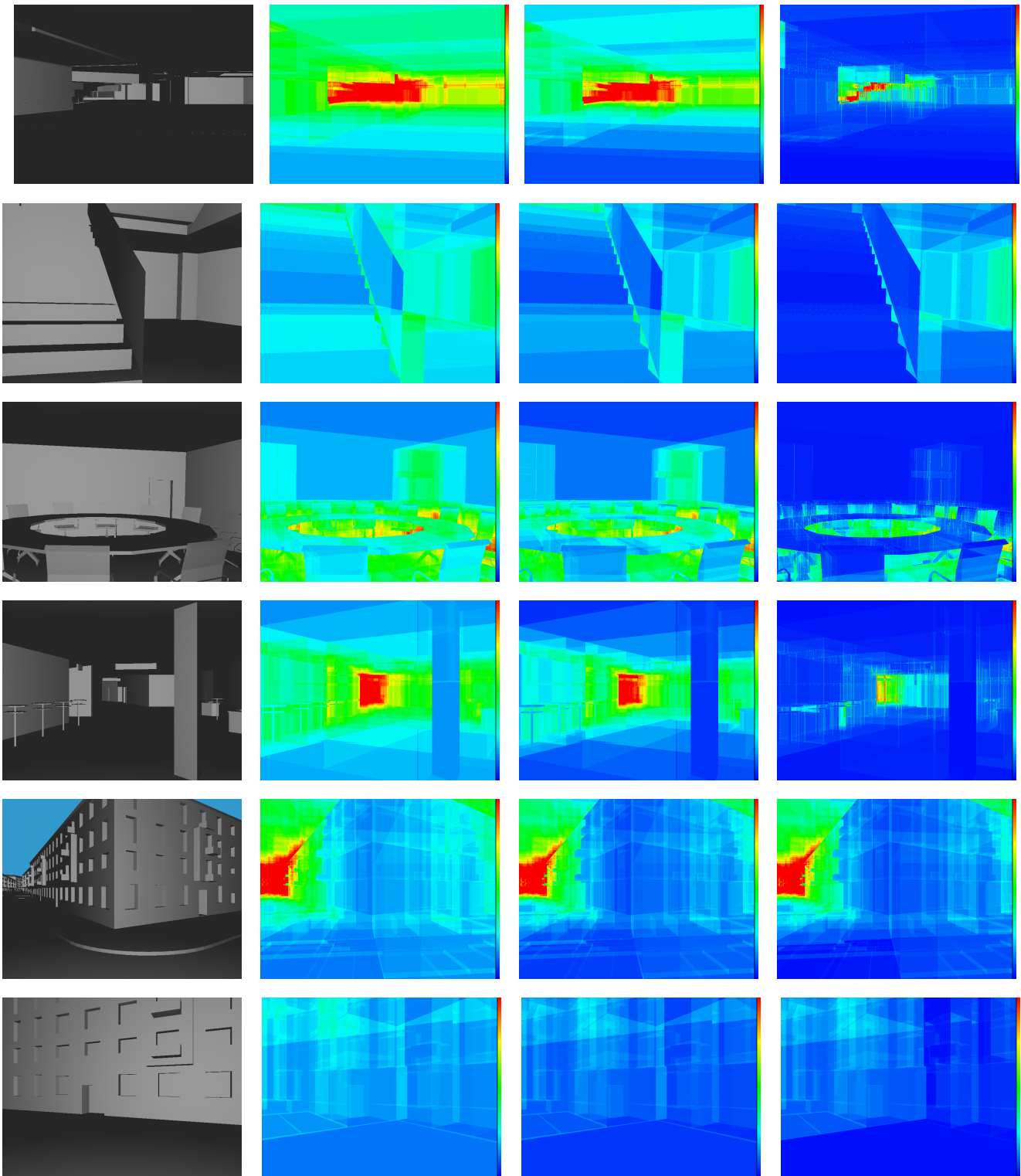


Figure 5: (left) Several views used in our tests. (center left) Visualization of the number of nodes traversed per ray for the traditional hierarchical ray traversal algorithm. (center right) Visualization of number of nodes traversed per ray for the bottom-up traversal algorithm described in Section 5.2. (right) The visualization of number of nodes traversed per ray for coherence based traversal algorithm described in Section 5.3. The false coloring is as follows: dark blue corresponds to zero traversal steps, dark red corresponds to 255 or more traversal steps per ray, for other values see the color bar on the right of the image. First four rows show the 'arena' scene and last two rows the 'city' scene.