Stackless Ray Traversal for kD-Trees with Sparse Boxes

Vlastimil Havran
Czech Technical University
e-mail: *havranat fel.cvut.cz*

Jiri Bittner
Czech Technical University
e-mail: *bittnerat fel.cvut.cz*

November 17, 2007

Contents

Abstract

We present a new stackless ray traversal algorithm which makes use of kD-tree augmented with sparsely distributed bounding boxes. The proposed traversal algorithm is not restricted to start the hierarchy traversal at the root node and so it can exploit the knowledge of the ray origin. The algorithm achieves traversal times comparable with the traditional hierarchical traversal as well as recently introduced bottom-up traversal with sparse boxes. The stackless nature of the algorithm lends itself towards implementation of the method on the architectures with limited number of registers.

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.

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 [9] and bounding interval hierarchies for handling dynamic scenes [10]. The major advantage of hierarchies is their ability to adapt to irregular distribution of objects in the scene [6]. 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 [6, 4]. 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.

In this paper we extend a recent method using sparse boxes for ray traversal [3] which focused on reducing the number of traversal steps while using small memory footprint of the spatial data structure. In particular we present a stackless traversal algorithm using sparse boxes. The results indicate that the usage of sparse boxes makes the performance of this algorithm comparable with the stack based methods. However the proposed stackless version is more suitable for potential future implementation on the computer architectures, where the efficient implementation of stack is not feasible.

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 $O(n^{4+\varepsilon})$ to achieve $O(\log N)$ query time [8], where N denotes the number of objects in the scene. These algorithms count for all the possible worst case scenarios and provide theoretical guarantees [1]. 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 typically organized as a hierarchy. The ray then traverses the hierarchy from the root node 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 $const \cdot \log_2 N$, 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 [6] 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. [4]. The stackless algorithm for kd-trees based on the neighbor-links was published recently by Popov et al.[7]. The disadvantage of this algorithm is substantially higher memory requirements and also the increased construction time of the hierarchy.

In this paper we extend the method based on sparse boxes proposed by Havran and Bittner[3]. In particular we propose a new stackless bottom-up ray traversal algorithm using sparse boxes. We thoroughly compare the stackless algorithm with the previously proposed bottom-up traversal and the classical hierarchical traversal.

3 Overview

The methods described in this paper build on the following 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 some interior nodes of the hierarchy, (3) using the new data structure for efficient stackless ray traversal.

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 many traversal steps 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 and shadow 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). Optionally, for shadow rays we can store the node of the last shadow caster and use it to initiate the search for the shadow caster for subsequent (coherent) shadow ray.

We propose an efficient stackless ray traversal algorithm which makes use of the proposed data structure. The performance of the CPU implementation of this algorithm is only slightly worse than the performance of the stack based algorithm. However the stackless algorithm has a very good potential for an efficient implementation on computer architectures, where it is difficult or impossible to implement a stack based traversal algorithm.

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 [6] (ropes) or rope-trees [4]. 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 orientation + splitting plane position + 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. The 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 1).

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 $d_{min} \in \langle 1, 12 \rangle$, see the results in Section 6.

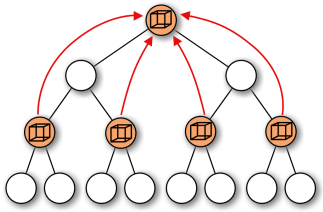

Figure 1: A spatial hierarchy augmented with bounding boxes. The bounding boxes are only stored at some levels of the hierarchy. This significantly reduces their memory footprint and has a positive influence on the performance of the newly proposed traversal algorithms.

5 Traversal Algorithms

In this section we first briefly recall the hierarchical traversal algorithm, then we describe the bottom-up traversal algorithm that uses sparse boxes, and finally we describe the stackless bottom-up traversal algorithm using the sparse boxes.

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

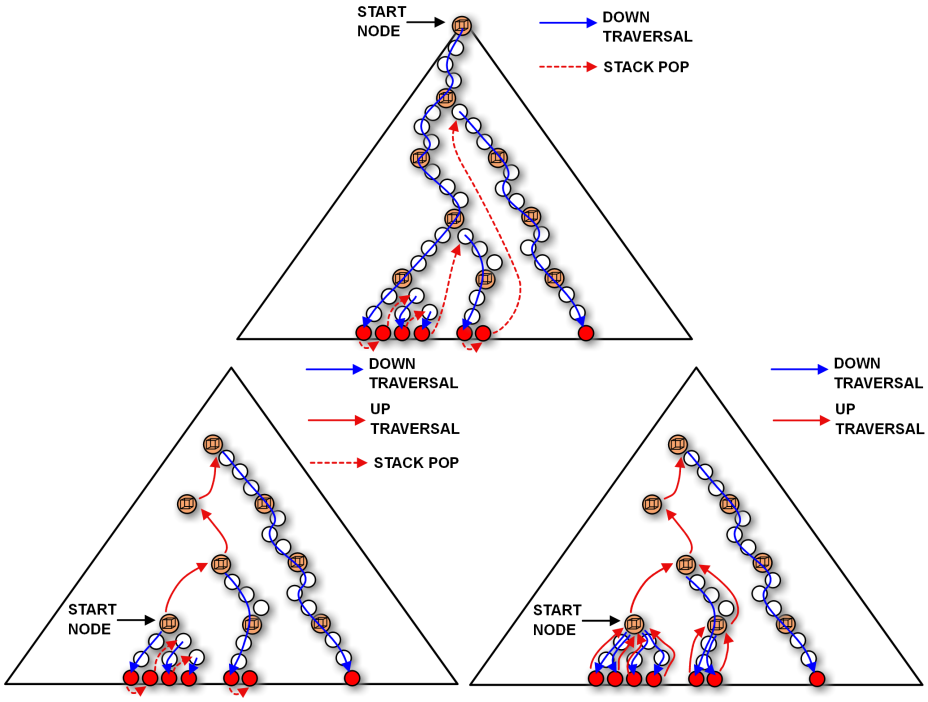

Figure 2: Illustration of different traversal algorithms. (top) Traditional stack based traversal algorithm starts at the root node and traverses the tree downwards to the leaf nodes using a stack. (bottom-left) The bottom up traversal algorithm starts at a given node in the hierarchy and traverses the hierarchy using the stack and the parent links between augmented nodes. (bottom-right) The stackless traversal algorithm starts at a given node in the hierarchy and traverses the hierarchy using only the down-links between nodes and parent links between augmented nodes. Note that this algorithm traverses some internal nodes more times.

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 [2]. The pseudocode of the hierarchical traversal algorithm is depicted in Algorithm 1.

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

Algorithm 1: Traditional hierarchical ray traversal algorithm.

```
CastRayHierarchical()
begin
    node = 0 ;                                          /* currently traversed node */
    tentry = 0; texit = 0 ;                             /* entry and exit signed distances */
    ComputeMinMaxT(ray, root, tentry, texit);
    stack.push(root, tentry, texit) ;                   /* initiate the stack */
    while not(stack.empty()) do
        stack.pop(node, tentry, texit);
        if node.IsLeaf() then
            if ComputeIntersections(node) then
                return "object intersected";
            end
        else
            TraverseDown();
        end
    end
    return "no object intersected"
end

TraverseDown()
begin
    t = Intersection(node.plane, ray);
    if Second Intersected Child then
        stack.push(Second Intersected, t, texit);
    end
    stack.push(First Intersected, tentry, Min(texit, t));
end
```

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 steps through the interior nodes 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. The pseudocode of the bottom-up traversal algorithm is given in Algorithm 2.

5.3 Stackless bottom-up traversal algorithm

The bottom-up traversal algorithm described in the previous section has the advantage that it can start the traversal at any augmented node in the hierarchy. However similarly to the traditional hierarchical traversal algorithm it needs a stack for performing the actual traversal. The need for stack causes problems for potential implementation of the algorithm on the current GPUs where the register space per thread is strictly limited.

Therefore we propose a modification of the bottom-up traversal algorithm which performs ray traversal without the need for stack. By using the idea of sparse boxes, i.e. sparsely distributed augmented nodes, we achieve the performance of the stackless traversal which is close to the performance of the stack based algorithms.

The stackless traversal algorithm proceeds as follows: the traversal is initiated at an augmented node containing the ray origin. We perform simple down traversal of the tree which finds the actual leaf containing the ray origin. If no intersection is found in this leaf, we need to restart the traversal in order to find the next leaf pierced by the ray. We first shift the ray origin to start just after the exit point of the ray with respect to the currently processed leaf. Then we find the closest parent augmented node in the tree which contains the new ray origin. The traversal continues from this augmented node using the simple down traversal. If there is no augmented node containing the new origin, the ray is leaving the scene without any intersection.

The actual search for the closest augmented node containing the new ray uses caching of the last visited augmented node. If the new ray origin is within the extent of this node, we start the down traversal from the cached node. If the new ray origin is outside of the cached node we follow the links to parent augmented nodes until we find a node in which the ray origin is contained.

Algorithm 2: Bottom-up ray traversal algorithm.

```
CastRayBottomUp(startNode)
begin
    node = 0 ;                                              /* currently traversed node */
    lastAugmentedNode = 0 ;                                 /* last augmented node traversed */
    tentry = 0; texit = 0 ;                                 /* entry and exit signed distances */
    ComputeMinMaxT(ray, startNode, tentry, texit);
    stack.push(startNode, startNode, tentry, texit) ;       /* initiate the stack */
    while not(stack.empty()) do
        while not(stack.empty()) do
            stack.pop(node, lastAugmentedNode, tentry, texit);
            if node.IsLeaf() then
                if ComputeIntersections(node) then
                    return "object intersected";
                end
            else
                TraverseDown();
            end
        end
        TraverseUp();
    end
    return "no object intersected"
end

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

TraverseUp()
begin
    tentry = texit + eps ;                                  /* shift the ray origin */
    /* Find the first parent augmented node containing the shifted ray origin */
    node = lastAugmentedNode;
    while node.parent do
        ComputeMaxT(node.parent, tmax);
        if tentry < tmax then
            stack.push(node.parent, node.parent, tentry, tmax) ;   /* augmented node
            found */
            return
        end
    end
    ;                                                       /* augmented node not found */
end
```

The pseudocode of the stackless bottom-up traversal algorithm is given in Algorithm 3. The illustration of all discussed traversal algorithms is given in Figure 2.

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 [6]. 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 tests were performed on a PC with AMD Athlon(tm) 64 X2 Dual processor with 1MB L2 cache running on 2GHz and 2GB or RAM. The program was running on Linux with a compiler GNU GCC 4.1.2. We used only a single core for our computations (512 KB L2 cache). The images were rendered at resolution 800×600 pixels.

The first set of tests evaluated the number of traversal steps and the total rendering time for different algorithms. We have measured 13 representative viewpoints inside the tested scenes. Table 1 summarizes the results for primary rays.

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 by modifying the d_{min} constant used in the construction of the hierarchy. The results for primary rays are shown in Figure 3.

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 parent node.

In the 'arena' scene the bottom-up traversal algorithm (BTR) achieves a total speedup of rendering to 1.10 on average compared to the hierarchical traversal algorithm (HTR). 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 spent in the actual traversal routine (the rest of the time corresponds to computing ray triangle intersections and shading). The speedup of rendering of 1.10 thus means that the actual speedup of the traversal 1.29. The use of stackless based traversal algorithm (SLTR) increases number of traversal steps by 46% which induces a total slowdown 1.38.

The 'city' scene has a smaller depth complexity and longer rays and for this scene the BTR method actually results only in a speedup of 1.04. Stackless version of the ray traversal algorithm (SLTR) yields in slowdown of 1.38. The SLTR algorithm compared to the sequential traversal algorithm of Kaplan[5] (SEQ) is faster by factor 1.35.

Algorithm 3: Stackless bottom-up ray traversal algorithm.

```
CastRayStackless(startNode)
begin
    node = startNode ;                                   /* currently traversed node */
    lastAugmentedNode = startNode ;                      /* last augmented node traversed */
    ComputeMinMaxT(ray, node, tentry, texit) ;           /* compute entry and exit signed
    distances */
    tup = texit ;                                        /* signed distance for up traversal */
    repeat
        /* find the leaf containing the origin                                          */
        while !node.IsLeaf() do  TraverseDown(node);
        if ComputeIntersections(node) then  return "object intersected";
        tentry = texit + eps ;                           /* shift the ray origin by epsilon */
    until RestartTraversal() ;
    return "no object intersected"
end

TraverseDown()
begin
    t = Intersection(node.plane, ray);
    texit = Min(texit, t) ;                              /* clip the ray by the plane */
    node = First Intersected Child ;                     /* traverse node containing the ray origin */
    if IsAugmented(node) then
        lastAugmentedNode = node ;                       /* update node for restarting the traversal */
        tup = texit ;                                    /* update the distance for up traversal */
    end
end

RestartTraversal()
begin
    if tentry < tup then
        texit = tup;
        node = lastAugmentedNode;
        return true ;                /* last augmented node contains the shifted ray origin */
    else
        return TraverseUp() ;                            /* ray leaves the node - traverse up */
    end
end

TraverseUp()
begin
    /* Find the first parent augmented node containing the shifted ray origin           */
    node = lastAugmentedNode;
    while node.parent do
        node = node.parent;
        ComputeMaxT(node, tmax);
        if tentry < tmax then
            texit = tmax; tup = tmax;
            lastAugmentedNode = node;
            return true ;                                /* found an augmented node */
        end
    end
    return false ;                                       /* did not find an augmented node */
end
```

method	N_B [-]	M_{KD} [-]	r_{ITM} [-]	N_L [-]	N_I [-]	T_R [s]
'arena', 1.4M triangles, 7.1M kD-nodes						
HTR	0	89M	35.93	9.45	86.85	1.71
BTR	0.725M	117M	31.96	9.46	73.67	1.55
$SLTR$	0.725M	117M	37.06	10.80	126.49	2.37
SEQ	0	89M	37.06	10.81	246.97	2.89
'city', 0.8M triangles, 4.0M kD-nodes						
HTR	0	42M	20.97	7.95	58.85	1.24
BTR	0.557M	64M	22.92	8.21	50.87	1.19
$SLTR$	0.557M	64M	21.01	9.15	85.67	1.66
SEQ	0	42M	21.01	9.15	195.83	2.24

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, $SLTR$ is stackless traversal algorithm described in Section 5.3, and SEQ is the sequential traversal algorithm of Kaplan [5]. N_B is the number of boxes stored in kd-tree, M_{KD} is total memory required for the kd-tree in MBytes, 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.

7 Conclusion

We presented a new stackless ray traversal algorithm for kD-trees. 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 leads to good performance of the stackless traversal while keeping the memory cost for the augmented data low.

We compared the proposed method with the traditional hierarchical and sequential traversal algorithm as well as the recently introduced bottom-up traversal algorithm with the sparse boxes. The results indicate that the new algorithm is slightly slower than the stack based methods, but faster than the sequential traversal when implemented on a CPU.

Currently we work on porting the algorithm to the GPU. We also plan 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). The 'arena' scene is a courtesy of Digital Media Production a.s.

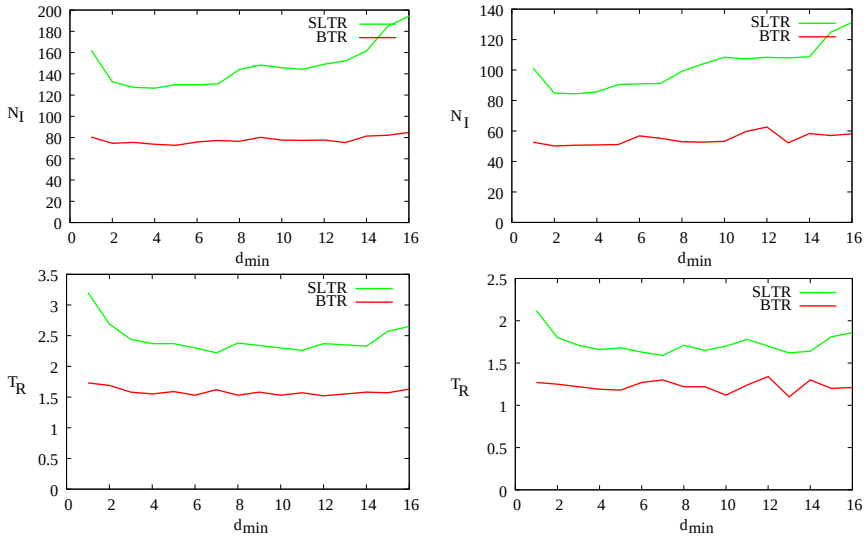

Figure 3: Dependence of the number of traversed interior nodes (top) and the rendering time (bottom) on the density of the augmented nodes (minimal distance between nodes d_{min}). (left) the 'arena' scene, (right) the 'city' scene.

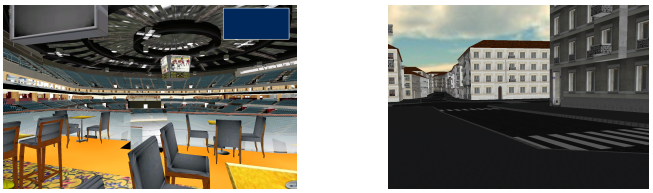

Figure 4: The view of the 'arena' scene and the 'city' scene.

References

[1] P. Agarwal. Range Searching. In *CRC Handbook of Discrete and Computational Geometry (J. Goodman and J. O'Rourke, eds.)*, New York, 2004. CRC Press.

[2] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[3] V. Havran and J. Bittner. Ray tracing with sparse boxes. In *Proceedings of SCCG'07 (Spring Conference on Computer Graphics)*, pages 49–54, Budmerice, Slovak Republic, April 2007.

[4] V. Havran, J. Bittner, and J. Žára. Ray tracing with rope trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)*, pages 130–139, Budmerice, Slovak Republic, April 1998.

[5] M. Kaplan. *Space-Tracing: A Constant Time Ray-Tracer*, pages 149–158. July 1985.

[6] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[7] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[8] L. Szirmay-Kalos and G. Márton. Worst-case versus average case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998.

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

[10] S. Woop, G. Marmitt, and P. Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware (2006)*, pages 67–77, 2006.

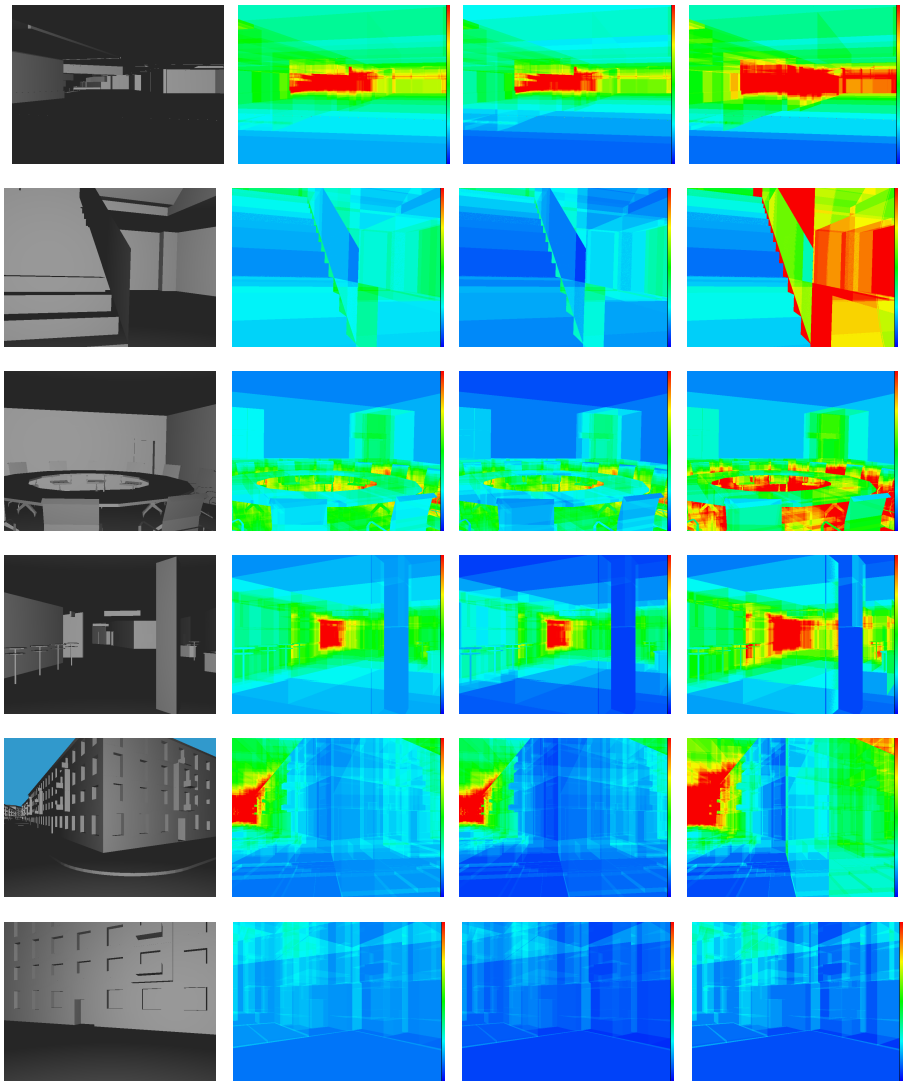

Figure 5: (left) Several views used in our tests. (center left) Visualization of the number of nodes traversed per ray for the traditional hierarchical ray traversal algorithm. (center right) Visualization of number of nodes traversed per ray for the bottom-up traversal algorithm described in Section 5.2. (right) The visualization of number of nodes traversed per ray for stackless ray 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.