Automatic Creation of Object Hierarchies for Ray Tracing of Dynamic Scenes

Martin Eisemann
Institute for Computer Graphics, TU Braunschweig, Germany
eisemann@cg.tu-bs.de

Thorsten Grosch
University of Koblenz-Landau, Germany
grosch@uni-koblenz.de

Marcus Magnor
Institute for Computer Graphics, TU Braunschweig, Germany
magnor@cg.tu-bs.de

Stefan Müller
University of Koblenz-Landau Braunschweig
stefan.mueller@uni-koblenz.de

ABSTRACT

Ray tracing acceleration techniques most often consider only static scenes, neglecting the processing time needed to build the acceleration data structure. With the development of interactive ray tracing systems, this reconstruction time becomes a serious bottleneck if concerned with dynamic scenes. In this paper, we describe two strategies for efficient updating of bounding volume hierarchies (BVH) for scenarios with arbitrarily moving objects. The first exploits spatial locality in the object distribution for faster reinsertion of the moved objects. The second allows insertion and deletion of objects at almost constant time by using a hybrid system, which combines benefits from both spatial subdivision and BVHs. Depending on the number of moving objects, our algorithms adjust a dynamic BVH six to one hundred times faster than it would take to rebuild the complete hierarchy, while rendering times of the resulting hierarchy remain almost untouched.

Keywords: Ray Tracing, Object Hierarchies, Bounding Volume Hierarchies, Animation, Dynamic Scenes

1 INTRODUCTION

Ray tracing is well known for its ability to create photorealistic images. Recently developed ray tracing systems are now already able to achieve interactive frame rates, but their efficiency relies heavily on precalculated acceleration data structures [15][22][4][17]. The complexity for reconstructing these acceleration data structures for a scene with n triangles is often $O(n \log n)$ or worse, with the final cost in the ray tracing phase being only $O(\log n)$ per pixel on average. This usually limits interactive ray tracing to static scenes or simple walkthroughs, so that the acceleration structure can be reused for all frames.

For a complete interactive ray tracing system, an efficient support of moving objects is necessary. Therefore, the acceleration data structures usually have to be rebuild for each frame. Techniques like frameless rendering [2] [3] and frustum traversal [17] reduce the amount of work that has to be done in the ray tracing phase and almost linear scalability for up to 128 processors has been shown [15]. But the reconstruction phase cannot be parallelized as easily, in fact very little research has been done on this topic [8]. Thus it becomes the bottleneck to interactive ray tracing of dynamic scenes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright UNION Agency – Science Press, Plzen, Czech Republic.

In this article we present two approaches to deal with the problem of ray tracing animated scenes, based on bounding volume hierarchies (BVH). The first, called *Dynamic Goldsmith and Salmon* (Dyn. G&S), exploits spatial coherence to rapidly update the BVH, while the second, called *Loose Bounding Volume Hierarchy* (LBVH), is a hybrid approach, which allows for reconstruction of the acceleration data structure in $O(n)$ by combining the benefits of a BVH with spatial subdivision.

The rest of the paper is organized as follows. In the next section we review some related work. Then we present our approaches of handling dynamic scenes in Sect. 3. Results and their discussion are given in Sect. 4, followed by a conclusion and directions to future work in the final section.

2 RELATED WORK

A large number of methods and algorithms to speed up ray tracing exist, but most of them are designed for static images or simple walkthroughs and not much attention has been spent on constructing these acceleration structures efficiently. Therefore, ray tracing of dynamic scenes is a rather new field of research, which gets more and more important as ray tracing gets more and more accelerated.

Quite early Glassner [5] developed a technique called *Spacetime Ray Tracing*. The idea was to intersect rays with static four-dimensional objects instead of dynamic

objects in three-space, whereas the fourth dimension is time. Unfortunately, this technique is only suitable for scenes with predefined movements.

Using multiprocessors Parker et al. [15] were able to ray trace reasonably complex scenes at interactive frame rates. Moving objects are tested separately for intersection, which therefore allowed only a small amount of them (≤ 10).

Reinhard et al. [16] used hierarchical grids for ray tracing of dynamic scenes. Their data structure is essentially a balanced octree, which keeps objects at different levels, depending on their size. This allows for insertion and deletion in almost $O(1)$ for an object. Depending on the motion, the entire data structure needs to be rebuild once in a while.

Lext and Akenine-Moeller [11] build hierarchies of oriented bounding boxes containing recursive grids. These grids include all primitives which underly the same affine transformation. It is therefore sufficient to build them once and transform the rays into the local coordinate system for performing intersection tests. We adapt this concept for both of our presented strategies in this paper.

Wald et al. [21] also exploit local coordinate systems to animate rigid bodies. But instead of using the scene graph for traversal between these entities, they rebuilt the whole top-level data structure every time a movement takes place. This results in quite long update phases between the frames. A complete rebuild for every frame is also proposed in [24], but the underlying data structure is a uniform grid which can be rebuild more efficiently, but might suffer from non-uniform object distribution.

Guenther et al. [7] use motion decomposition to ray trace deformable models. The connectivity does not change and the space of possible poses is known in advance. The model is decomposed into clusters which underly a similar transformation. Residual motion is captured in a single fuzzy kd-tree for the entire animation.

An example of a lazy evaluation strategy was given by McNeill et al. [14]. The upper levels of an octree are build in a preprocessing step, while the rest is build on demand. They propose to use this technique also for dynamic environments, but test results were only presented for static scenes.

Recently, Waechter et al. [20] showed that this concept can be applied to dynamic scenes and very complex scenes. They also proposed a simple, but fast, global heuristic for choosing the splitting plane of every node in their *Bounding Interval Hierarchy*, which is similar in spirit to our Loose Bounding Volume Hierarchy, even though we do not need to split larger objects into smaller pieces as they do.

Actually, Larsson and Akenine-Moeller [9] make strong use of lazy evaluation to ray trace deformable models, by utilizing the static connectivity between the triangles and refitting only the upper half of their preconstructed BVH. The rest gets refitted on demand. As the structure of the BVH is not allowed to change, the possible movement of the triangles is rather limited without degrading performance, even though it can be sufficient for ray tracing small to mid-sized deformable scenes [23].

To make this technique applicable for any kind of scenes, Lauterbach et al. [10] used the ratio between each parent node's surface area to the sum of the area of its two children to detect degradation of the BVH and rebuilt it on demand.

Another quite interesting approach was presented by Ulrich [18], called *Loose Octrees*. It has not been used in the context of ray tracing so far, only for collision detection and view frustum culling. These Loose Octrees are a variation of normal octrees which allow insertion in $O(1)$ by using overlapping voxels and choosing an insertion level depending on the size of the object. But this overlapping is also the reason why the scheme works better for collision detection than for view frustum culling.

3 OUR APPROACHES

We assume that the reader is somewhat familiar with the basic concepts of a BVH and ray tracing, if not see [1]. Rebuilding a BVH for every frame of an animation, using standard techniques, make it impossible to achieve interactive frame rates in rather complex scenes. A refitting of the bounding volumes (BV), a recomputing of the bounds of the BVs, can be done very quickly. But, depending on the movement of the objects, the quality of the BVH can arbitrarily decrease, resulting in inacceptable long rendering times for certain scenes.

In this paper we present two approaches to deal with ray tracing of dynamic scenes with non-predetermined movement. In Sect. 3.1 a method is presented, which exploits locality in the acceleration structure for a rapid update. The second approach in Sect. 3.2 presents a method for insertion and deletion of an object into a BVH in almost constant time. For both methods we introduce a new phase between each frame, the *update phase*, in which the animated objects are moved and the update of the BVH takes place.

For an easier understanding, we first clarify some terms. A *primitive* can be any kind of basic geometric shape, like a triangle or a parametric shape. An *object* is either a primitive or a collection of primitives within its own local coordinate system having its own acceleration structure, in our case a BVH. These objects are stored in a separate list. All leaf cells of our BVHs possess a pointer towards the object contained in them. These nodes are called *object nodes*. In addition, the objects have a *hierarchy pointer*, if necessary, which grants immediate access to the object node in the BVH

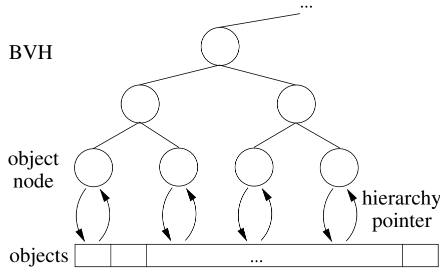

Figure 1: Overview of the notations. BVH: Bounding Volume Hierarchy

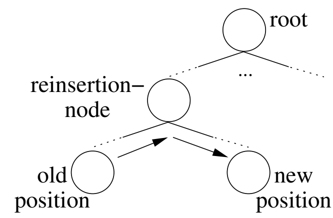

Figure 2: Beginning at its old position, the object node of the moving object is passed along its ancestors, until the reinsertion node is found, from where it is inserted again.

containing the object. This is one of the biggest advantages of BVHs compared to other acceleration structures, every object is contained in just a single node of the BVH, instead of several voxels, as it could be possible when using k-d trees, octrees or uniform grids. The relation of these terms is given in Fig. 1.

3.1 Dynamic Goldsmith and Salmon

In this section we describe an adaptive hierarchy, based on the BVH creation scheme introduced by Goldsmith and Salmon [6]. We used their technique to initially build the BVH and reinsert our objects later on, even though the described technique is not limited to this creation scheme. Others, like the surface area heuristic (SAH), could be applied as well [13]. Which scheme suits best is unfortunately always scene dependent. [6] is usually superior if the scene contains distinct objects, while the SAH is a more general approach, but has a very complex creation scheme.

Goldsmith and Salmon proposed a method to deal with dynamic changes in a scene, by deleting the object nodes from a BVH, adjusting the BVs and reinserting the object nodes beginning at the root, using a heuristic tree search to find the optimal insertion position (for more details see [6]). However, this insertion technique is a rather inefficient scheme. It is not necessary to completely remove a changed object node from a BVH. Since a certain spatial locality is given by the BVs in the same subtree of a BVH and since objects usually move only small distances compared to the scene extent, the object would either be inserted at its old position in the BVH or a position nearby for the most part. The term nearby here means a subtree which encloses both, the old and the new position of the object. Since this subtree is probably much smaller in depth compared to the whole BVH, starting the reinsertion of the object node at the root of this subtree will shorten the whole insertion process drastically. We will call the root of this subtree the *reinsertion node*. Choosing this node, we minimize the number of needed insertion steps, since no changes have to be made to any of its ancestors. This also implies that an insertion starting at the root would most likely lead to this node anyway, since the inheritance cost for this node, a term depicting the surface growth of the ancestor nodes, is still 0 (see [6]). In the worst case, the whole process is exactly the same as removing an object node completely from the BVH and reinserting it at the root node. In comparison to a complete rebuild, which is in general even faster than reinserting every object at the root, our reinsertion scheme can speed up the process of reconstructing the hierarchy by more than two orders of magnitude (see the test results in Sect. 4).

This process is visualized in Fig. 2. From its old position, the object node is incrementally passed along its parents until the reinsertion node is found. This is the first node on the path to enclose the object node. Then the object node is inserted again, leading to its new position.

Removing an object from a BVH and reinserting it may lead to an effect which we call *thinning*. This term describes a decrease of objects contained in a node, while its surface area remains almost unchanged. If the thinning continues, it is most likely that better BVHs could and should be created. An example is given in Fig. 3. Objects 2, 4 and 5 are moving as depicted by the arrows. The dashed circles are the target positions (see Fig. 3 on the left). At a certain point in time the insertion criteria would force object 2 to change into the right subtree (Fig. 3 in the middle). Even though object 3 did not move at all, it would result in a better BVH if it would change into the right subtree as well (Fig. 3 on the right).

To prevent such a degradation of a BVH, we introduce a quality criterion $Q(B)$ that can be efficiently calculated and effectively prevented thinning in our tests. We use the surface area of a node divided by the number of objects contained in its subtree, which is in some sense a measure for the packing density of this node. This is depicted in equation (1):

$$Q(B) = \frac{S(B)}{C_{obj}(B)} \quad , \tag{1}$$

where $Q(B)$ is the quality measurement of node B, $S(B)$ is the surface area of B and $C_{obj}(B)$ is the number of objects contained in the corresponding subtree of B. Note that this criterion is solely used for detecting the change in the BVH, it cannot and is not intended

 ISBN 978-80-86943-02-2

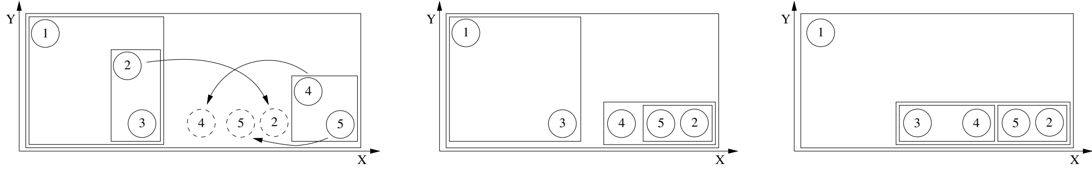

Figure 3: Example for the thinning of nodes. Left: The objects move as depicted by the arrows. Middle: resulting BVs after movement. Right: Rearranged BVH for a faster traversal.

to be used as a global quality criterion for ray tracing efficiency.

After the initial construction of the BVH, an initial value Q_{init} is calculated for every node in the hierarchy. This is also done during the update phase, if a new node is created. During the animation, if all reinsertions took place, the current value $Q_{current}$ of the changed nodes only is compared to their initial values. If it exceeds a predefined threshold, the corresponding nodes are deleted, the BVH gets adjusted and the children of the nodes are reinserted as described above, as the reinsertion scheme is not only limited to object nodes.

In our tests this quality criterion not only removes most of the threat coming from thinning of nodes, but can also decrease the ray tracing phase up to 34%, while only increasing the update time by about 16% compared to not dealing with thinned nodes. The benefit is proportional to the relative number of dynamic objects.

Combining both presented techniques leads to the following pseudocode:

Algorithm 1 Update Phase Dyn. G&S

1: **for all** objs **do**
2: animate objs
3: **end for**
4: **for all** animated objs **do**
5: remove obj node from hierarchy using the hierarchy pointer for instant access
6: incrementally search for the new insertion node and adjust BVs on the path
7: insert adjusted obj node using G&S's technique
8: **end for**
9: search for and remove thinned nodes from the BVH
10: reinsert all children of the thinned nodes

The underlying data structure is highly dynamic, which means keeping a good cache efficieny is a non-trivial task. For some high performance systems it might be a better solution to just mark all reinsertion nodes and rebuild the whole underlying part of the BVH. Since every node has a fixed memory footprint, this reconstruction can be done in place. This memory bound is not available for other spatial data structures, like kd-trees.

3.2 Loose Bounding Volume Hierarchy

Even though the method described in the last section results in a tremendous speedup to the update phase, its complexity is still no better than $O(m \log n)$ on average given m moving objects and n scene objects. In the following we present a hybrid approach, which allows insertion and deletion of objects in $O(1)$ by exploiting that every object lies exactly in one node of a BVH combined with a pseudo-spatial subdivision scheme. For a better understanding the simple version of the LBVH will be described first, followed by the extensions applied for a better performance.

Using a pre-built BVH with a fixed subdivision level of $3N$ and a branching factor of $k = 2$ with splitting axes chosen in a round robin fashion along the midpoints, we can think of the lowest level as a uniform grid which encloses the whole scene and a $2^N \times 2^N \times 2^N$ resolution. The insertion positions of the object nodes are based on their midpoints, the corresponding index i_x of the voxel in the x-direction can be calculated using the following equation:

$$i_x = \left\lfloor 2^N \left(\frac{O_{x_{mid}} - S_{x_{min}}}{S_{x_{max}} - S_{x_{min}}} \right) \right\rfloor \quad , \tag{2}$$

where $O_{x_{mid}}$ is the midpoint of object node O along the x-axis, and $S_{x_{min}}$ and $S_{x_{max}}$ define the minimum and maximum value of the scene extent along this axis. Similar computations are made for i_y- and i_z-axis. The actual index in the BVH is then be computed from these indices.

After object insertion the rest of the hierarchy can be refitted to assure the tree's correctness. The LBVH for a simple test scene is shown on the left in Fig. 4. Empty nodes in the graph are represented by dots. Dotted lines in the scene on the left represent extents of the voxels for insertion, bounding volumes are drawn with solid lines. Identical bounding volumes are drawn with different scales for clarity.

If we would insert all objects at the lowest level, we could not assure that the surface of a child's node is actually smaller than it's parent, which is a must have for a BVH with a reasonable performance. To solve this problem, we allow inner nodes to contain objects as well. For every object, the insertion level is calculated from its axis-aligned bounding box (AABB) using the following equation:

$$L = 3 \left\lfloor \log_2 \left(\min \left(\frac{S_x}{O_x}, \frac{S_y}{O_y}, \frac{S_z}{O_z} \right) \right) \right\rfloor \quad , \tag{3}$$

 ISBN 978-80-86943-02-2

where O_a is the extent of the AABB of object O along axis $a \in \{x,y,z\}$ and S_a is the extent of the AABB surrounding the scene along axis a. Using equation (3) we keep larger objects closer to the root and therefore assure, that the maximum possible extent along the splitting axis is reduced by 50% for every level of the hierarchy, which leads to a good spatial partitioning. If L is greater than the predefined subdivision level of the BVH, it is set to the maximum possible level.

Depending on L we can calculate the indices in the x-, y- and z-direction similar to equation (2), substituting N for $L/3$. For the x-direction this is shown in equation (4).

$$i_x = \left\lfloor 2^{L/3} \left(\frac{O_{x_{mid}} - S_{x_{min}}}{S_{x_{max}} - S_{x_{min}}} \right) \right\rfloor \tag{4}$$

Because of the limited number of possible indices, the easiest way to calculate the index in the BVH is to use a precalculated lookup table, based on i_x, i_y, i_z and L. Therefore any desired memory layout of the BVH, optimized for the chosen traversal method, can be used. The object node is then added to that node. The resulting BVH, using the same small test scene as before, is shown in Fig. 4 (middle). Object A will be assigned to the root node due to its great extent along the x-axis, while B stays at its old node.

If we assume constant scene extends, we can calculate the i_x, i_y and i_z even faster if we transform the whole scene into the $N \times N \times N$ volume, since the calculation of the index becomes a simple truncating of the midpoint coordinates in this volume.

The insertion process may lead to nodes with just one child. This can happen if small objects are surrounded by a large empty space. Because the object node and one or more of its ancestors are identical in this case, it would be a tedious task to test all of them for intersection. To avoid this problem *skip indices* can be used. If a ray intersects a node, usually all children have to be tested for intersection as well. Instead of testing the child directly, the node that its skip index points to is tested. This way all nodes with just one child and without objects can be skipped easily. An example for our simple test scene is shown on the right of Fig. 4. The calculation of the skip index can be efficiently done in the refitting process, which will be described in the following.

Until now objects are inserted into the hierarchy, but the BVH ist still inconsistent, since only the nodes containing at least one object could be adjusted so far. Assuring that the index of a node is less than the index of its children, as it is usually the case, all nodes in the BVH can be efficiently refitted by iterating over the array in reversed order, as suggested by van den Bergen [19]. During this refitting we mark empty nodes and if a node has just one child and contains no objects, its skip index is set to the skip index of this child, otherwise to itself. Please note, the resulting hierarchy, comparing the extends of the non-empty nodes, is similar to the one proposed by Waechter in [20], even though developed completely independent of each other. In general, their approach is better suited for very complex scenes, while ours is targeted at small to mid-size scenes with up to a few thousand moving objects.

Since adjusting one node and inserting an object takes almost constant time, the creation of the complete BVH can be done in time linear to the number of nodes and objects in the BVH.

The update phase between two consecutive frames is basically a reconstruction, but in $O(n)$, instead of the usual $O(n \log n)$. During the construction we saved every insertion index of all object nodes in the BVH. Using these indices, we can simply empty the hierarchy. Afterwards the objects are animated and the BVH gets rebuild as described before. Pseudocode for the update phase is given below.

Algorithm 2 Update Phase LBVH

```
1:  empty hierarchy
2:  for all objs o do
3:      animate o
4:      adjust obj node of o
5:  end for
6:  expand root node to enclose scene
7:  for all objs o do
8:      calc index in BVH for o using equation (4)
9:      insert o into BVH depending on the index
10: end for
11: refit hierarchy by reversed order iteration
```

4 RESULTS AND DISCUSSION

To evaluate our methods, we have used a variety of test scenes. Here we present the results for three of them, which we think reveal both, the benefits and weaknesses of our strategies. All tests were performed on a PC with a 2GHz Intel Pentium Mobile processor and 512 MB of memory. The maximum allowed ray tree depth was two, i.e. one reflection and refraction was allowed. The predefined depth for the LBVH is set to 18.

We compare the results of our approaches to a complete rebuild of the hierarchy every frame using the method of Goldsmith and Salmon [6]. The rebuild is done only once per frame, without further shuffling of the objects, since rebuilding the acceleration structure more than once is not feasible, if we want a fair comparison of the resulting update times. A simple refit of the BVH is not included in our statistics because of the drastic increase in ray tracing time for most of the scenes. The average (*avg*) timings per frame for the update phase (*up*), ray tracing phase (*rt*) as well as the average speedup achieved for the test scenes are presented in table 1.

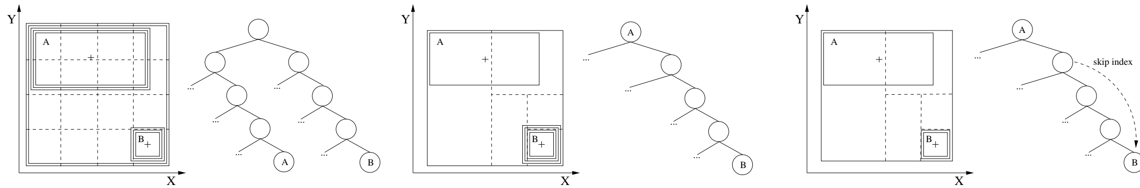

Figure 4: Left: Simple version of the Loose Bounding Volume Hierarchy, with objects inserted at the lowest level. Middle: Advanced version, keeping larger objects at higher levels. Right: Final version, using skip indices in addition to the advanced version to avoid intersecting unnecessary nodes.

Please note, that we assume no knowledge about possible movements of the objects. Therefore the kitchen scene e.g. should be judged as a scene with about $110k$ dynamic objects, and not as a scene consisting of almost only static objects and a few moving ones. This is important, since the information about what is static and what not might not always be available, e.g. in physics based simulations.

Kitchen	G&S	Dyn. G&S	LBVH
avg. up	1.759s	0.017s	0.157s
avg. rt	6.142s	6.082s	11.771s
speedup up	1.0	103.471	11.204
speedup rt	1.0	1.010	0.522
speedup tot	1.0	1.295	0.662
#tris 110k		resolution 300×225	

Museum	G&S	Dyn. G&S	LBVH
avg. up	1.933s	0.315s	0.125s
avg. rt	11.839s	7.950s	10.323s
speedup up	1.0	6.137	15.464
speedup rt	1.0	1.489	1.147
speedup tot	1.0	1.666	1.318
#tris 76k		resolution 800×640	

Falling tris	G&S	Dyn. G&S	LBVH
avg. up	7.478s	0.907s	0.404s
avg. rt	22.298s	11.420s	3.182s
speedup up	1.0	8.245	18.510
speedup rt	1.0	1.953	7.008
speedup tot	1.0	2.416	8.303
#tris 149k		resolution 512×512	

Table 1: Performance measurements from the three test scenes.

The first test scene is the kitchen scene from the BART benchmark suite [12]. Only a small toy car, consisting of 5 dynamic objects, is animated in an otherwise static surrounding. Test results are given in table 1 and Fig. 6, which shows a comparison of the ray tracing phase in the upper left graph and the update phase in the upper right graph.

The timings in the ray tracing phase between the complete rebuild and the Dyn. G&S method are almost equal. But when comparing the update timings, we achieved a dramatic decrease compared to a complete rebuild by more than two orders of magnitude. Showing the advantage of this method for scenes with small amounts of movement.

The update time for the LBVH also shows a decrease by more than an order of magnitude, even though the ray tracing time almost doubled. A closer statistical analysis showed that this is due to the so-called "teapot in the stadium" problem. The scene consists of many large and many small objects. Due to the predefined depth of the hierarchy, the average number of primitives per leaf node is 100, with a maximum value of 1339. Therefore a two level approach (e.g. [11] and [21]), or a lazy evaluation strategy, (e.g. [20]), should be used with the LBVH to avoid this problem. This is not currently implemented in our system.

The second test scene, the BART museum scene shows a museum room with a deforming piece of art in the middle, with mostly unstructured, random movement. The time spent in the update phase is reduced by roughly a factor of 6 to 15. The fact that the ray tracing time of the Dyn. G&S method does not exceed the ray tracing time for a BVH after a complete rebuild verifies our assumptions made in Sect. 3.1. In addition even a decrease in the time needed for the ray tracing phase is achieved, compared to a complete rebuild. In the case of the Dyn. G&S method, this is due to the possibility to rebuild the BVH several times in the beginning and shows the quality of our update routine. A local update of the acceleration data structure is sufficient to preserve the quality of a BVH.

The LBVH shows also good results in this test. It is not only able to ray trace the animation faster than the complete rebuild method, but it also takes only a fraction of the time needed in the update phase.

The last test scene consists of triangle patches, randomly assorted in a plane parallel to the xz-plane. During the animation, the triangles start falling from the ceiling at random times, speed and directions. The high number of animated primitives, as well as the highly changing object distribution, stresses our methods.

To avoid the advantage of the Dyn. G&S method of exploiting spatial locality too much, the amount of frames is reduced to eleven. Therefore, the triangles move rather fast through the scene compared to the complete scene extents.

 ISBN 978-80-86943-02-2

When using the Goldsmith and Salmon technique for the complete rebuild, the arrangements of the primitives and the fact that they are all of uniform size lead to a relatively unbalanced tree. In contrast we can rebuild our initial BVH for the Dyn. G&S method several times to have a better initial stand. This is not possible for the complete rebuild method as it would take too long during the update phase.

The time needed in the update phase of the LBVH is almost constant throughout the whole scene. The superior ray tracing time is due to the uniform size of the objects, which allows for a very good spatial partitioning.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented two methods for updating BVHs for ray tracing dynamic scenes. We have shown that the use of these two methods can greatly decrease the time needed in the update phase, compared to a complete rebuild. Speed-ups up to a factor of 103 in the update phase have been achieved. This allows for much better overall performance, especially when using multiprocessor machines or techniques like frameless rendering.

While the Dyn. G&S method showed very good overall performance in almost all of our test cases, its biggest advantages are achieved in scenes with much local movements. The LBVH is useful for time-critical ray tracing applications, as the update phase is almost constant in all test scenes and smaller scenes. Further optimizations are possible, e.g. calculating the insertion indices and parallelizing the creation process of the BVH is possible, in case of static scene extents.

As future work, we are planning to apply a lazy evaluation strategy to the lower levels of the LBVH, to circumvent the "teapot in the stadium" problem. And we would like to implement both of our methods in a high performance system.

REFERENCES

[1] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 206–208. Academic Press, 1989.

[2] G. Bishop, H. Fuchs, L. McMillan, and E. J. Scher Zagier. Frameless rendering: Double buffering considered harmful. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 175–176, NY, USA, 1994. ACM Press.

[3] A. Dayal, C. Woolley, B. Watson, and D. P. Luebke. Adaptive frameless rendering. In *Proceedings of the Eurographics Symposium on Rendering Techniques*, pages 265–275, 2005.

[4] M. Geimer. *Interaktives Ray Tracing*. PhD thesis, University of Koblenz-Landau, 2005.

[5] A. S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988.

[6] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[7] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek. Ray Tracing Animated Scenes using Motion Decomposition. *Computer Graphics Forum*, 2006. (Proceedings of Eurographics, to appear).

[8] V. Isler, C. Aykanat, and B. Özguç. An efficient parallel spatial subdivision algorithm for parallel ray tracing complex scenes. In *First Bilkent Computer Graphics Conference, ATARV-93*, Ankara, Turkey, 1993.

[9] T. Larsson and T. Akenine-Moeller. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Technical report, MRTC Maelardalen Real-Time Research Centre, Maelardalen University, February 2003.

[10] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Salt Lake City, Utah, 2006.

[11] J. Lext and T. Akenine-Moeller. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001 - Short Presentations*, pages 311–318, 2001.

[12] J. Lext, U. Assarsson, and T. Moeller. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications*, 21:22–31, March 2001.

[13] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166, 1990.

[14] M.D.J. McNeill, B.C. Shah, M.-P. Hébert, P.F. Lister, and R.L.Grimsdale. Performance of space subdivision techniques in ray tracing. *Computer Graphics Forum*, 11(4):213–220, 1992.

[15] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.

[16] E. Reinhard, B. Smits, and C. Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 299–306, June 2000.

[17] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics*, 24(3):1176–1185, 2005.

[18] T. Ulrich. Loose octrees. In *Game Programming Gems*, volume 1, pages 434–442. Mark DeLoura, 2000.

[19] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphic Tools*, 2(4):1–13, 1997.

[20] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the Eurographics Symposium on Rendering*, June 2006.

[21] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 77–86, Oktober 2003.

[22] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3), 2001.

[23] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version). *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023 (conditionally accepted at ACM Transactions on Graphics)*, 2006.

[24] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 2006. (Proceedings of ACM SIGGRAPH 2006, to appear).

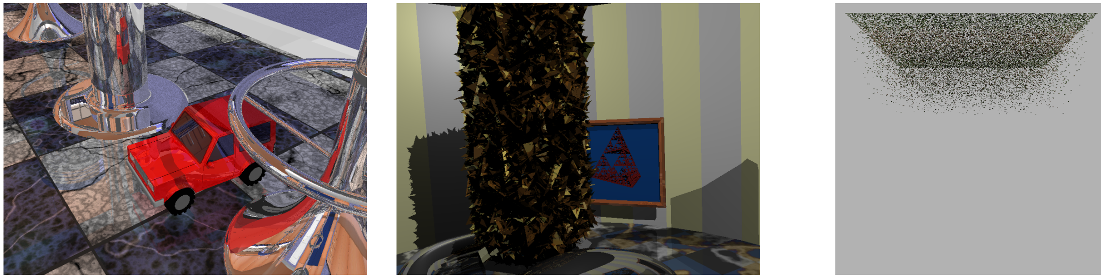

Figure 5:
Sample images from the three test scenes. Left: Kitchen, middle: Museum, right: Falling Triangles.

Testscene 1 - BART Kitchen Scene - Ray tracing time

G&S Rebuild
Dyn. G&S
Loose BVH

Testscene 1 - BART Kitchen Scene - Update time

G&S Rebuild
Dyn. G&S
Loose

Testscene 2 - BART Museum Scene - Ray tracing time

G&S Rebuild
Dyn. G&S
Loose

Testscene 2 - BART Museum Scene - Update time

G&S Rebuild
Dyn. G&S
Loose

Testscene 3 - Falling Triangles - Ray tracing time

G&S Rebuild
Dyn. G&S
Loose

Testscene 3 - Falling Triangles - Update time

G&S Rebuild
Dyn. G&S
Loose

Figure 6: Test results for the three test scenes. Top: Kitchen, middle: Museum, Bottom: Falling Triangles. Left column: Time spent in the ray tracing phase. Right column: Time spent in the update phase