# Towards Rapid Reconstruction for Animated Ray Tracing

Jonas Lext† and Tomas Akenine-Möller

Department of Computer Engineering, Chalmers University of Technology, Sweden

**Abstract**

*This article discusses methods for avoiding that the reconstruction of the acceleration data structure becomes a bottleneck in animated or interactive ray tracing. Situations in which this could occur include trying to increase the frame rate by parallelization of the ray tracing phase or by techniques such as frameless rendering. Specifically, we explore a method for avoiding unnecessary reconstruction in rigid-body animated scenes. The method builds a hierarchy of oriented bounding boxes containing recursive grids by applying these to the rigid bodies found in different transforms in the scene graph. The oriented bounding boxes containing gridded objects are then kept intact during the complete animation. Before performing intersection tests, rays are transformed to the local coordinate system of an oriented bounding box. Using this technique, the reconstruction of the data structure can be performed an order of magnitude faster as compared to using a recursive grid that has to be rebuilt completely between each frame.*

## 1. Introduction

One of the long standing goals in computer graphics is to generate photo-realistic synthetic images. Global illumination algorithms, such as photon mapping[1], are the current state-of-the-art for this. At the other end of the spectrum is the goal of rendering images of three dimensional scenes in real-time, that is, faster than, say, 20 frames per second. The best way to do this currently is to use dedicated graphics hardware that rapidly rasterizes textured triangle primitives, and uses the Z-buffer to resolve occlusion.

In a perfect world, both goals should be met simultaneously. Meanwhile, there are two ways worth exploring:

- Use graphics hardware to "fake" or approximate global illumination effects
- Use simplified global illumination algorithms in order to reach higher frame rates

This paper focuses on the latter. As ray tracing is the foundation for the majority of global illumination algorithms, and due of the birth of ray tracing at interactive frame rates[2, 3, 4, 5], our main concern here is to deal with ray tracing at as rapid rates as is possible in an environment where multiple objects, including the camera, are animated. This would extend the use of ray tracing.

† Email: lext@ce.chalmers.se



As one starts to explore the realm of animated interactive ray tracing, lots of questions emerge that need to be answered. Many of these questions arise due to the fact that the ray tracing algorithm has to be *parallelized* in order to achieve good performance (see Stone's article for a treatment of some of these questions[5]). Another way to achieve higher frame rates, but worse image quality, is to use *frameless rendering*[6]. In such a framework, only a subset of the pixels, randomly distributed in the image, is updated each frame. Since both parallelism and frameless rendering reduces the amount of ray tracing work that has to be done per CPU per frame, the bottleneck may move from the ray tracing part to other parts of the algorithm that are not easily parallelizable. This can be seen in figure 1. Here, the x-axis shows the number of times the total number of pixels in an image is divided by. The result of this is the number of pixels ray traced by frameless rendering each frame. The horizontal curve is the time it takes to *reconstruct* the acceleration data structures (which are needed to get good performance in the first place for ray tracing) for the animated objects in the scene. The other curve shows the actual time per frame that is spent ray tracing.

Also important is that often the complexity of reconstructing the acceleration data structures is worse than the actual ray tracing. Many methods have linear complexity $O(n)$, while some methods has $O(n \log n)$ complexity[7, 8]. In con-

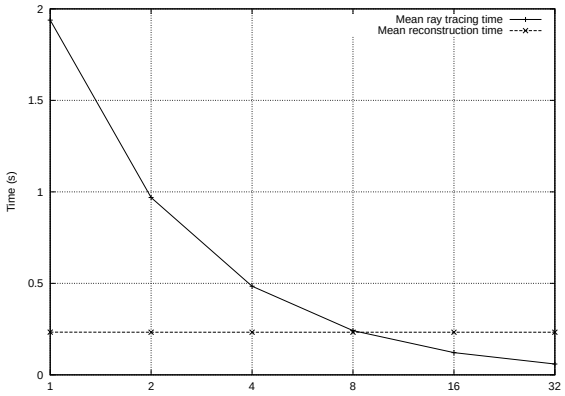


**Figure 1:** *The near horizontal curve is the reconstruction time, while the other curve is the ray tracing time as the number of pixels ray traced by frameless rendering is decreased, that is, if $x = 5$, that implies that only $100/5 = 20\%$ of the pixels are ray traced each frame.*

trast, the ray tracing phase often has $O(\log n)$ complexity per pixel. Therefore, it should be obvious that the acceleration data structure reconstruction phase could become a serious bottleneck when using a multiprocessor computer, frameless rendering, and when there is a reasonable amount of animated objects in the scene.

In this paper, we therefore focus on the *reconstruction problem*, i.e., on how to rebuild the acceleration data structures as rapidly as possible. More specifically, we use a hierarchy of oriented bounding boxes that each contain a local acceleration data structure, which in our case is a recursive grid. The benefit from this is that for rigid-body animated scenes, only the transform of the box need to be updated, thus avoiding updating the local acceleration data structure in the box. We show that this makes the reconstruction phase an order of magnitude faster than previous methods. To measure performance, we use BART[9], which is a benchmark for animated ray tracing, and a Linux PC with an AMD Athlon 1.1 GHz processor. We compare our approach with a recursive grid[10].

The rest of this paper is organized as follows. Next, we review some previous work, which is followed by section 3 where we discuss and present algorithms for updating the acceleration data structures. In section 4 implementation notes are given, followed by a presentation of some early results. Finally, we offer some conclusions and directions to future work.

## 2. Related Work

Here, we will briefly review the relevant previous work in the area of animated ray tracing, which at the point of writing is quite close to non-existing.

Muuss[2] and Parker et al.[3] have shown that it is possible to ray trace reasonably complex scenes at interactive rates using multiprocessors. However, they focused mostly on scenes where only the viewer was animated and not the objects. Parker et al.[3] could render few ($< 10$) dynamic objects by placing them outside the acceleration data structure, and testing these individually. They also use the concept of frameless rendering[6] in order to get higher frame rate at the cost of image quality.

Reinhard et al.[4] has presented an algorithm which we consider the first attempt at ray tracing dynamic scenes. Their data structure is essentially an octree with $O(1)$ insertion and deletion. The data structure is repeated in order to fill space. This means that as an object leaves the data structure on the right side, it is inserted on the left side. This implies that the entire data structure need not be rebuilt due to such movements.

Adelson and Hodges[11] use image-based reprojection techniques in order to exploit temporal coherency for faster ray tracing. Using this technique, up to 92% of the rendering time could be saved. However, all objects were required to be convex, and the objects could not be animated (only the viewer could), and this limits its use.

Glassner[12] uses a 4D bounding volume hierarchy to speed up ray tracing of animated scenes. The fourth dimension is time, and he thus employs a space-time hierarchy to exploit temporal coherency.

McNeill et al.[13] use lazy evaluation techniques to avoid building parts of the acceleration data structure that is not accessed. The upper levels of an octree is built as a preprocess, and the lower levels are built on demand as they are needed. They claim that this could also be used in a dynamic environment, but only use static environments for testing.

Oriented bounding boxes (OBBs) has been used in collision detection to speed up these queries, and in that work, OBBs have been suggested as an appropriate bounding volume for ray tracing as well[14]. Cazals et al.[15] have used OBBs in a ray tracing, but not in an animated framework, and no actual results were presented.

## 3. Update Strategies for Acceleration Data Structures

In this paper, we focus on two different approaches for achieving faster reconstruction of the acceleration data structure. These are based on separating dynamic and static scene data and lazy-evaluation techniques, respectively, and are discussed in the following two subsections. Both methods imply introducing a new phase between each frame, the *update phase*, in which animated objects are moved and the acceleration data structure is reconstructed accordingly. If possible, only minor modifications to the part of the structure where objects move should be performed. By leaving the rest of the structure intact, a lot of time can be saved. The animation process is illustrated in figure 2 below.

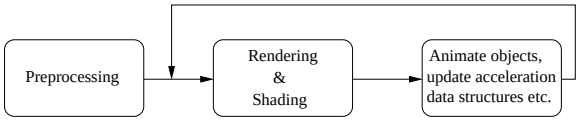


**Figure 2:** *Phases in the animation process.*

### 3.1. Separation of Dynamic and Static Scene Data

The traditional approach when rendering static images using ray tracing is to tear down the scene graph, by transforming the primitives it contains to the root system, and construct a new hierarchy from scratch that is better suited for ray tracing. In an animation, this will be required in each frame, and the update phase introduced previously will be identical to the preprocessing phase. This costly approach might be used because it is not known exactly which part of the scene that has changed and we must therefore reconstruct the acceleration data structure completely.

However, if the scene graph has been modeled using dynamic transforms, the moving part of the scene can easily be identified and thus separated from the static portion of the scene data. Furthermore, if the objects within a dynamic transform show spatial locality, they are directly suitable for enclosing in a single bounding volume or a more complex structure, e.g., a bounding volume hierarchy or a recursive grid. As it is known that the objects within a single dynamic transform do not move relative each other, we can build the structure once and for all in the preprocessing phase. The amount of work in the update phase is, at best, then reduced to simply updating the transform matrices associated with each node in the scene graph. However, as the scene objects remain at the different nodes in the scene graph, some extra work must be spent in the ray tracing phase transforming rays between the different nodes before performing intersection tests.

Different authors have argued that the scene graph which results from modeling should not be particularly good for constructing an acceleration data structure[8, 16]. However, we should be able to expect reasonable ray tracing performance using the proposed approach if the scene graph is constructed with some care or if the right conditions are fulfilled. For example, in truly interactive environments all objects that a user might pick up or move will be placed in a dynamic transform. Each dynamic transform will therefore by default be associated with a single solid object in which the triangles show good spatial locality. Constructing the scene graph with the intent of using it to guide in the construction of the acceleration data structure can be compared to game programming where it should be very common to strongly take advantage of the scene characteristics in, e.g., collision detection for achieving greater performance. However, we still view the proposed approach as fairly general in its applicability.

In section 4, we describe our specific implementation of these general ideas. We use OBBs to encapsulate the objects within a dynamic transform and also apply a recursive grid within each OBB to speed up the ray tracing phase.

### 3.2. Lazy Evaluation Techniques

If the scene graph cannot be used with good result as suggested in the previous section, we could resort to lazy evaluation techniques. The approach here is to minimize the amount of work done in the update phase by only performing changes in those part of the data structure that are visited by rays during the generation of a frame. This will typically mean a tighter integration of the ray tracing phase and the update phase as the function that updates, e.g., a voxel is called when a ray enters that particular voxel for the first time.

An example of this approach is given by McNeill et al.[13]. Here, a lazy-evaluation algorithm for octrees is proposed in which only the upper part of the octree is constructed wholly between frames. Only when rays enters subvoxels in the octree in which the number of primitives is larger than the threshold value, will the necessary lower part of the octree be built. This method should be quite straightforward to extend to a recursive grid, which is one of the classical data structures that we have implemented. We would then only construct the grid at the top level and only create subgrids for the voxels that rays actually intersects.

A possible extension that we also would like to investigate would be based on the observation that if the number of rays taking advantage of a complex acceleration data structure is very small, it might not be beneficial to spend an excessive amount of time constructing it. A better trade-off might be to construct a much simpler acceleration structure quickly or even skip the data structure altogether and resort to basic list-searching. We can envision a system keeping track of how many rays that traversed every bounding box or voxel and, by taking advantage of frame-to frame-coherency, make an intelligent choice on which acceleration data structure that should be used when updating a particular bounding box or voxel. The choice should be guided by both the number of rays that intersected the volume and the number of primitives that are associated with it.

## 4. Implementation

In this section, we describe our implementation of the general ideas presented in section 3.1. Specifically, we outline the implementation of the construction and update routines, respectively.

Our method finds a separate minimal area OBB for all the primitives in each static and/or dynamic transform in the scene graph and also applies a recursive grid within each OBB. If a dynamic transform has no dynamic transforms

among its parents, a special OBB-grid is also applied encapsulating all the grids created at lower levels in that part of the scene graph. These particular OBB-grids have to be reconstructed between each frame due to the movements of the subgrids they contain, and this is the main task performed in the update phase. An alternative would have been to use only a simple bounding volume at the highest level or nothing at all. However, in the Robot scene that we consider, the moving objects are robots composed of several parts, each of which is placed inside a grid. All of these grids will be placed in the outermost data structure and if this a simple bounding volume all will have to be tested for intersection for every ray that intersects the bounding volume. Therefore, we expect better performance using a grid.

Pseudocode for the construction routine is given below. The routine is called with a reference to the root node of the scene graph as input. The output of the routine is currently a list with references to a number of grids. One of these grids contains the static scene primitives and the remaining grids all encapsulate a cluster of moving objects.

Note that the minimum area OBB will be created inside the routine that creates a grid. We use a modified version of the algorithm by Eberly[17] to find the minimum OBBs. As a local coordinate system will be associated with each OBB, the contained objects must again be transformed to this local coordinate system before an ordinary grid or recursive grid can be created inside the OBB. In the ray tracing phase, rays are transformed to this local coordinate system before performing intersection tests. Also note that each grid must keep information about in which node in the scene graph it was created and also in which node it was itself put inside a supergrid as this information will be needed in the grid update routine. Finally, all grids are heterogeneous and the resolution is calculated as suggested by Klimaszewski et al.[16].

```
List Create( Node &node,
             List &objectsFather,
             List &gridsFather )
{
  List objectsLocal ;
  List gridsLocal ;

  for ( all objects found in node )
  {
    object.transformToRootFrom( node );
    objectsLocal.append( object );
  }

  for ( all child nodes of node )
  {
    Create( childnode, objectsLocal, gridLocal );
  }

  switch( node.type() )
  {
    case root :
      Grid grid = new Grid( node.objects, node );
      gridsLocal.append( grid );
      return gridsLocal ;

    case static :
      objectsFather.append( objectsLocal );
      gridsFather.append( gridsLocal );

    case dynamic :
      objectsFather.append( objectsLocal );

      if ( objectsFather.numberOf() > 0 )
      {
        objectsFather.transformFromRootTo( node );
        Grid grid = new Grid( objectsFather, node );
        grid.transformToRootFrom( node );
        gridsLocal.append( grid );
      }

      if ( node has no parents of type dynamic )
      {
        gridsLocal.transformFromRootTo( node );

        Grid supergrid = new Grid( gridsLocal, node );

        for ( all grids in local list of grids )
        {
          grid.setFatherGrid( supergrid );
          grid.setNodeFinal( node );
        }

        supergrid.transformToRootFrom( node );

        gridsFather.append( supergrid );
      }
      else
      {
        gridsFather.append( gridsLocal );
      }
  }
}
```

The update routine is quite simple, so we do not give any pseudo code for it. Instead, its implementation is outlined in the following.

First note that prior to calling the update routine for each grid in the list of grids returned by the construction routine, the scene graph has been traversed and in each node the transform matrices for transforming points from a node in the scene graph to the root system and vice versa has been recalculated corresponding to the new frame time. The matrices used during the previous frame is, however, also kept for easy access as these are needed in the update routine as well.

When a grid calls the update routine, the OBB of the grid (i.e. its axes and origin) is first transformed from the node in the scene graph that it currently resides in to the root system and from there to the node in which it was created. For both transforms, the transform matrices corresponding to the old frame time are used. We then check if this grid contains primitives or subgrids. If it contains subgrids, this grid must be reconstructed due to the movements of the subgrids. We therefore transform all the subgrids through the transform constructed from the OBB's current axes and origin. All subgrids then perform their incremental movements by calling the update routine themselves. Next, a new OBB is created encapsulating all the subgrids in their new positions and then the subgrids are transformed to the local coordinate system of the new OBB. Finally, we create a recursive grid inside the OBB.

In the end of the update routine, the OBB of the calling grid, whether it was created within this call of the routine or not, is transformed back to the node in the scene graph in which it resided when the routine was called. For these transformations, the transform matrices corresponding to the new frame time are used.

In the next section, the described method is compared to the simpler approach of using a single recursive grid that encapsulates the whole scene, including the moving objects, and thus will have to be reconstructed between each frame. Measurements are presented for both a simpler animation and for the complete Robot test scene from BART[9].

## 5. Results and Discussion

To more easily show the performance of the proposed OBB algorithm, we first present measurements on a simple animation featuring a single robot in empty space. In this animation, the robot walks along a circular path and the camera follows the robot so as the relative position is almost constant. Figure 5 to 7 show the first ray traced frame from this animation and also the data structure used. The images were rendered with a maximum allowed ray tree depth of two, i.e., one reflection was allowed, and the image size was $300 \times 300$ pixels. Only one level of subgridding was used in the recursive grids and ten primitives were allowed in a voxel before a voxel was subgridded. The rightmost picture indicates the time to ray trace each individual pixel. We use it to identify the most time consuming parts of a data structure.

As hinted in section 4, our implementation allows us to choose whether to apply recursive grids in dynamic transforms only or also in each static transform in the scene graph. In both cases, a reasonably good hierarchy resulted without having to make changes to the initial scene graph description as given in the BART files. Figure 5 illustrate the hierarchy when grids were applied in both static and dynamic transforms and this was the structure used in the measurements presented below. Figure 6 shows the resulting hierarchy when applying grids in dynamic transforms only.

For comparison purposes, the simple animation mentioned above was also rendered with a single recursive grid applied to the robot. Figure 7 shows the pictures resulting when using this data structure. Due to the movement of the robot, the recursive grid was fully reconstructed in the update phase. The two graphs in figure 3 show the resulting times spent in the ray tracing phase and update phase, respectively, for the two data structures.

The mean total time to generate a frame is very similar, however, the time spent in the two phases is quite different. The time required by the OBB algorithm in the update phase is much smaller because only a small part of the structure needs to be reconstructed.

The hill shaped form of the curve resulting when using the

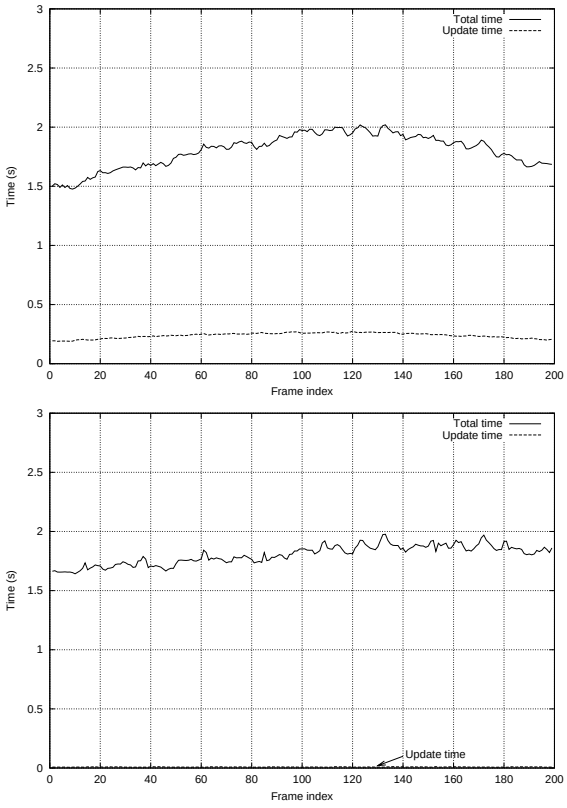


**Figure 3:** *Time spent in the ray tracing phase and update phase as a function of frame index for the single robot test scene. Top graph: single recursive grid, bottom graph: proposed algorithm using OBBs.*

recursive grid is probably due to two known disadvantages with axis aligned bounding boxes. As the robot turns, the volume of the grid grows slightly due to increased misfit, which translates into an increase in ray traversal time. Also, in the middle of the animation, most eye rays traverse the grid diagonally, increasing the traversal time as well.

Turning to the lower graph in figure 3, we see that, in general, the time spent in the ray tracing phase by the OBB-algorithm is larger than is needed by the recursive grid structure. One explanation might be the extra work introduced in this phase due to the transformations of rays. Empirical studies have made us come to the conclusion that these transformations can have a non-negligible impact on the performance, as the floating point unit temporarily becomes a bottleneck. Also, some voxels in the outermost OBB contain several subgrids and all of these must be tested for intersection when rays passes through these voxels.

Leaving the simple test scene, we now present measurements on the complete Robots test scene from the BART benchmark suite[9]. In these measurements, a maximum al-

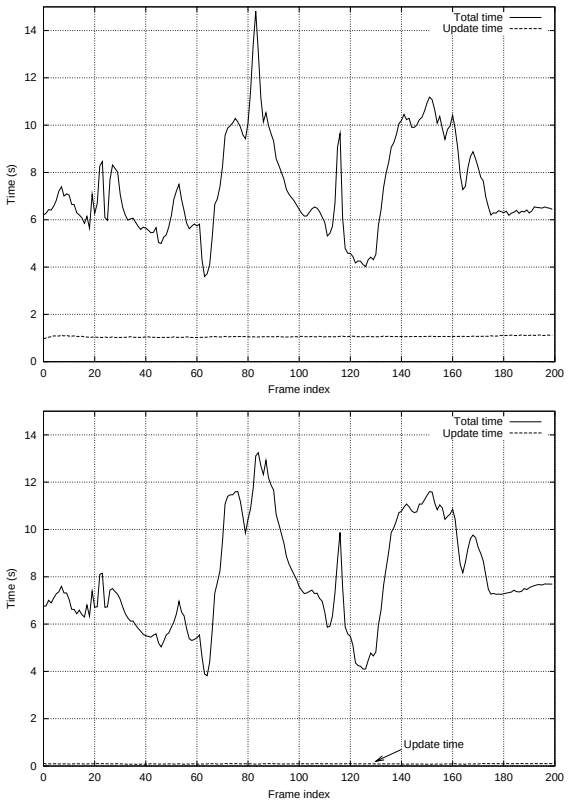


**Figure 4:** *Time spent in the ray tracing phase and update phase as a function of frame index for the Robots test scene from BART. Top graph: single recursive grid, bottom graph: proposed algorithm using OBBs.*

lowed ray tree depth of four were used. Figure 4 (top) shows the time spent in the ray tracing phase and the update phase when applying a recursive grid to the whole scene including the robots. In the bottom graph, the result when using the proposed OBB algorithm is presented. Similar to the case with a single robot, the OBB algorithm manages to reduce the time spent in the update phase with roughly a factor of ten as compared to reconstructing the recursive grid from scratch between each frame. Unfortunately, it again requires more time performing the ray tracing phase, particularly in the parts of the animation where many of the robots are visible.

An additional explanation for this lower performance, besides the two disadvantages with the OBB algorithm discussed earlier in this section, is the fact that we only allow an OBB-grid to contain either primitives or subgrids. The reason is that placing both primitives and moving subgrids in the same grid would force us to regrid all the primitives again as the subgrids moved, ruining the whole idea of our approach. This works fine when dealing with a single robot, however, when the complete Robot scene is used, the ten robot OBBs cannot be placed within the grid covering the city data, forcing the ten robot grids and the city grid into a single list. The result is that every ray currently must be tested against all robot OBBs in the scene, despite that only some of them is visible in most frames.

A simple remedy could be based on the fact that the robots always will be contained within the OBB covering the static city data, a characteristic that should hold for most realistic scenes with some static scenery and some objects moving around within the scene. We would then extend our current algorithm with some way to quickly remove and reinsert the robot OBBs into the grid covering the static city data. However, this will increase the amount of work in the update phase, and the performance gain is therefore not obvious.

Finally, returning to the graphs in figure 4, we see that in both cases the worst time to generate a complete frame is about 14 seconds. As the time to reconstruct the recursive grid fully is about 1.0 seconds, and hypothetically assuming a linear speedup when parallelizing the ray tracing phase, the ray tracing phase would be performed faster than the update phase already when using 16 processors. Doubling the number of processors to 32 would only result in a speed up of 33%, with diminishing returns for every extra doubling of the number of processors. If the OBB-algorithm is used instead, we could, at least in theory, use 128 processors before the two phases again would start to take a similar amount of time. On top on that, we could also apply frameless rendering to increase the frame rate, and make the arguments for our presented algorithm even stronger. Thus, having the ability to trade time between the two phases might allow us to better utilize large multiprocessor machines without having to parallelize the update phase.

## 6. Conclusion and Future Work

We show that using the scene graph to aid in the creation of an acceleration data structure for ray tracing of rigid body animations has the potential of allowing fast update times of the acceleration data structure. This can allow better performance when using frameless rendering or better utilization of large multiprocessor machines without the need to parallelize the construction of the acceleration data structure. Specifically, our implementation, using a hierarchy of OBBs, gave a tenfold speedup in the reconstruction phase as compared to rebuilding a recursive grid from scratch between each frame. This was the case when using both a simpler and a more advanced test scene.

Future work include improving the proposed OBB-algorithm so that the moving objects in a scene can be quickly removed from and reinserted into the grid structure enclosing the static scene primitives between frames. We currently believe that this could reduce the time spent in the ray tracing phase with only a small extra time added to the update phase. For comparison purposes, we would also

like to implement a lazy-evaluation technique as discussed in section 3.2.

It would be also interesting to adapt the reprojection method[11] to current computer architectures and to a multiprocessing environment in order to speed up the ray tracing process further. To be really useful, it would have to cope with moving non-convex objects as well.

### Acknowledgments

Our deepest thanks to Peter Rundberg for letting us use his personal computer for performing measurements.

This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

### References

1. Jensen, Henrik Wann, and Niels Jorgen Christensen, *A Practical Guide to Global Illumination using Photon Maps*, SIGGRAPH 2000 Course notes 8, July 2000. 1
2. Muuss, Michael John, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models", *Proceedings of BRL-CAD Symposium '95*, June 1995. 1, 2
3. Parker, Steven, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen, "Interactive Ray Tracing", *1999 Symposium on Interactive 3D Graphics*, pp. 119–126, April 1999. 1, 2, 2
4. Reinhard, Erik, Brian Smits, and Chuck Hansen, "Dynamic Acceleration Structures for Interactive Ray Tracing", *Proceedings of the 11th Eurographics Workshop on Rendering*, Brno, Czech Republic, pp 299-306, June 2000. 1, 2
5. Stone, John, "The Ups and Downs of Multithreaded Ray Tracing and Optimization", *Ray Tracing News*, vol. 12, no. 2, December 1999. `http://www.raytracingnews.com` 1, 1
6. Bishop, Gary, Henry Fuchs, Leonard McMillan, and Ellen j. Scher Zagier, "Frameless Rendering: Double Buffering Considered Harmful", *Computer Graphics (SIGGRAPH 94 Proceedings)*, pp. 175–176, July 1994. 1, 2
7. Cazals, Frédéric, George Drettakis, and Claude Puech, "Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes", *Computer Graphics Forum*, vol. 14, no. 3, pp. 371–382, August 1995. 1
8. Goldsmith, Jeffrey, and John Salmon, "Automatic Creation of Objects Hierarchies for Ray Tracing", *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, May 1987. 1, 3
9. Lext, Jonas, Ulf Assarsson, Tomas Möller, "A Benchmark for Animated Ray Tracing", *IEEE Computer Graphics and Applications*, pp. 22–31, March/April 2001 2, 5, 5
10. Jevans, David, and Brian Wyvill, "Adaptive Voxel Subdivision for Ray Tracing", *Graphics Interface '89*, pp. 164–172, June 1989. 2
11. Adelson, Stephen J., and Larry F. Hodges, "Generating Exact Ray-Traced Animation Frames by Reprojection", *IEEE Computer Graphics & Applications*, vol. 15 no 3, pp. 43–53, May 1995. 2, 7
12. Glassner, Andrew S., "Spacetime Ray Tracing for Animation", *IEEE Computer Graphics and Applications*, vol. 8 no. 2, pp. 60–70, March 1988. 2
13. McNeill, M.D.J., B.C. Shah, M-P. Hébert, P.F. Lister, and R.L. Grimsdale, "Performance of Space Subdivision Techniques in Ray Tracing", *Computer Graphics Forum*, vol. 11, no. 4, pp. 213–220, 1992. 2, 3
14. Gottschalk, Stefan, Ming Lin, and Dinesh Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection", *Computer Graphics (SIGGRAPH 96 Proceedings)*, pp. 171–180, August 1996. 2
15. Cazals, Frédéric, and Claude Puech, "Bucket-like space partitioning data structures with applications to ray-tracing", *ACM Symposium on Computational Geometry*, 1997 2
16. Klimaszewski, Krzysztof S., and Thomas W. Sederberg, "Faster Ray Tracing Using Adaptive Grids", *IEEE Computer Graphics and Applications*, vol. 17, no. 1, January/February 1997. 3, 4
17. Eberly, David H., *3D Game Engine Design – A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers, San Diego, 2001. 4

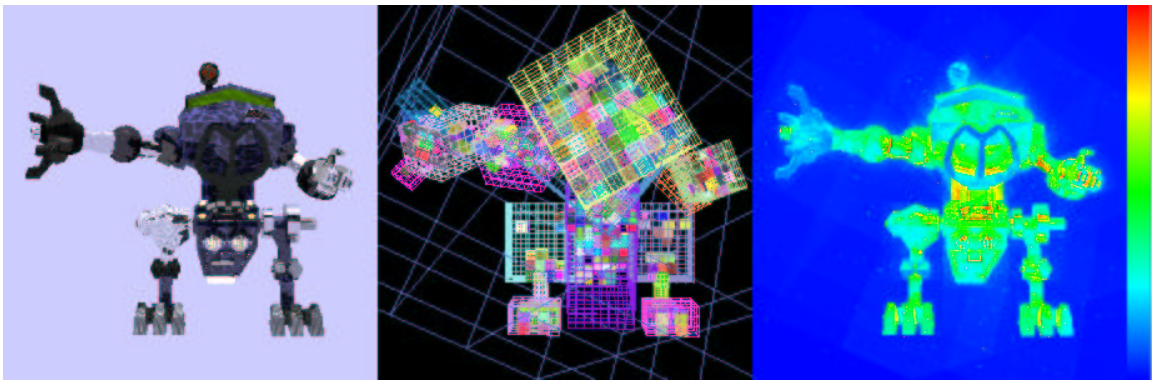

**Figure 5:** *Left: first frame in ray-traced animation. Middle: visualization of applied acceleration data structure; proposed algorithm with recursive grids applied in both dynamic and static transforms. Right: visualization of time spent in ray-tracing phase by each individual pixel.*

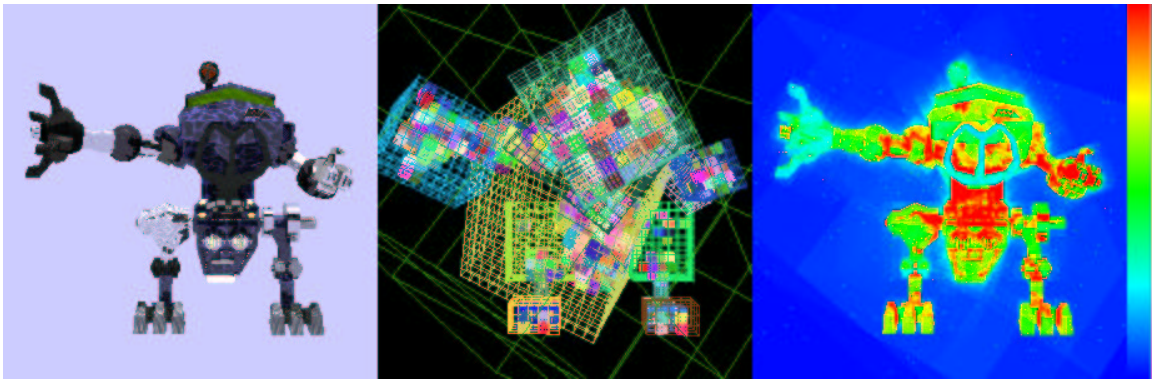

**Figure 6:** *Left: first frame in ray-traced animation. Middle: visualization of applied acceleration data structure; proposed algorithm with recursive grids applied in dynamic transforms only. Right: visualization of time spent in ray-tracing phase by each individual pixel.*

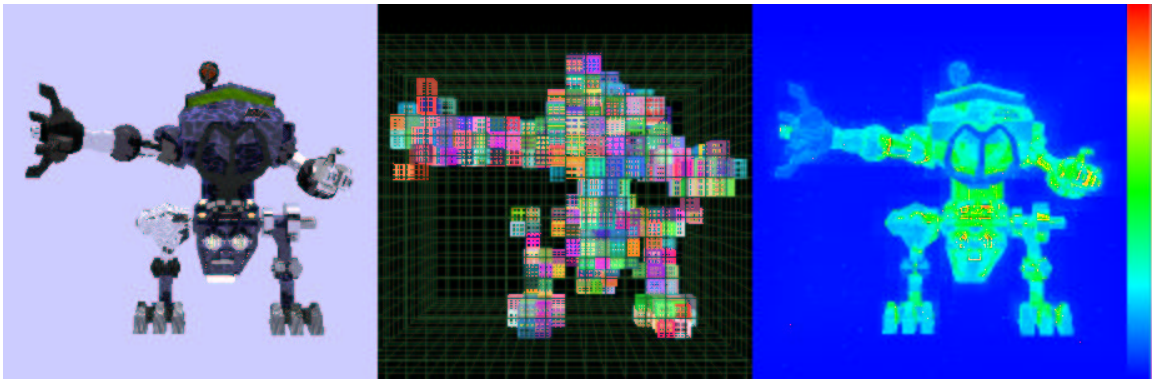

**Figure 7:** *Left: first frame in ray-traced animation. Middle: visualization of applied acceleration data structure; a single recursive grid. Right: visualization of time spent in ray-tracing phase by each individual pixel.*