Dynamic Acceleration Structures for Interactive Ray Tracing

Erik Reinhard, Brian Smits and Charles Hansen
University of Utah
reinhard|bes|hansen@cs.utah.edu

Abstract. Acceleration structures used for ray tracing have been designed and optimized for efficient traversal of static scenes. As it becomes feasible to do interactive ray tracing of moving objects, new requirements are posed upon the acceleration structures. Dynamic environments require rapid updates to the acceleration structures. In this paper we propose spatial subdivisions which allow insertion and deletion of objects in constant time at an arbitrary position, allowing scenes to be interactively animated and modified.

1 Introduction

Recently, interactive ray tracing has become a reality [8, 9], allowing exploration of scenes rendered with higher quality shading than with traditional interactive rendering algorithms. A high frame-rate is obtained through parallelism, using a multiprocessor shared memory machine. This approach has advantages over hardware accelerated interactive systems in that a software-based ray tracer is more easily modified. One of the problems with interactive ray tracing is that previous implementations only dealt with static scenes or scenes with a small number of specially handled moving objects. The reason for this limitation is that the acceleration structures used to make ray tracing efficient rely on a significant amount of preprocessing to build. This effectively limits the usefulness of interactive ray tracing to applications which allow changes in camera position. The work presented in this paper is aimed at extending the functionality of interactive ray tracing to include applications where objects need to be animated or interactively manipulated.

When objects can freely move through the scene, either through user interaction, or due to system-determined motion, it becomes necessary to adapt the acceleration methods to cope with changing geometry. Current spatial subdivisions tend to be highly optimized for efficient traversal, but are difficult to update quickly for changing geometry. For static scenes this suffices, as the spatial subdivision is generally constructed during a pre-processing step. However, in animated scenes pre-processed spatial subdivisions may have to be recalculated for each change of the moving objects. One approach to circumvent this issue is to use 4D radiance interpolants to speed-up ray traversal [2]. However, within this method the frame update rates depend on the type of scene edits performed as well as the extent of camera movement. We will therefore focus on adapting current spatial subdivision techniques to avoid these problems.

To animate objects while using a spatial subdivision, insertion and deletion costs are not negligible, as these operations may have to be performed many times during rendering. In this paper, spatial subdivisions are proposed which allow efficient ray traversal as well as rapid insertion and deletion for scenes where the extent of the scene grows over time.

The following section presents a brief overview of current spatial subdivision techniques (Section 2), followed by an explanation of our (hierarchical) grid modifications (Sections 3 and 4). A performance evaluation is given in Section 5, while conclusions are drawn in the final section.

2 Acceleration Structures for Ray Tracing

There has been a great deal of work done on acceleration structures for ray tracing [5]. However, little work has focused on ray tracing moving objects. Glassner presented an approach for building acceleration structures for animation [7]. However, this approach does not work for environments without *a priori* knowledge of the animation path for each object. In a survey of acceleration techniques, Gaede and Günther provide an overview of many spatial subdivisions, along with the requirements for various applications [4]. The most important requirements for ray tracing are fast ray traversal and adaptation to unevenly distributed data. Currently popular spatial subdivisions can be broadly categorized into bounding volume hierarchies and voxel based structures.

Bounding volume hierarchies create a tree, with each object stored in a single node. In theory, the tree structure allows $O(\log n)$ insertion and deletion, which may be fast enough. However, to make the traversal efficient, the tree is augmented with extra data, and occasionally flattened into an array representation [10], which enables fast traversal but insertion or deletion incur a non-trivial cost. Another problem is that as objects are inserted and deleted, the tree structure could become arbitrarily inefficient unless some sort of rebalancing step is performed as well.

Voxel based structures are either grids [1, 3] or can be hierarchical in nature, such as bintrees and octrees [6, 11]. The cost of building a spatial subdivision tends to be $O(n)$ in the number of objects. This is true for both grids and octrees. In addition, the cost of inserting a single object may depend on its relative size. A large object generally intersects many voxels, and therefore incurs a higher insertion cost than smaller objects. This can be alleviated through the use of modified hierarchical grids, as explained in Section 4. The larger problem with spatial subdivision approaches is that the grid structure is built within volume bounds that are fixed before construction. Although insertion and deletion may be relatively fast for most objects, if an object is moved outside the extent of the spatial subdivision, current structures would require a complete rebuild. This problem is addressed in the next section.

3 Grids

Grid spatial subdivisions for static scenes, without any modifications, are already useful for animated scenes, as traversal costs are low and insertion and deletion of objects is reasonably straightforward. Insertion is usually accomplished by mapping the axis-aligned bounding box of an object to the voxels of the grid. The object is inserted into all voxels that overlap with this bounding box. Deletion can be achieved in a similar way.

However, when an object moves outside the extent of the spatial subdivision, the acceleration structure would normally have to be rebuilt. As this is too expensive to perform repeatedly, we propose to logically replicate the grid over space. If an object exceeds the bounds of the grid, the object wraps around before re-insertion. Ray traversal then also wraps around the grid when a boundary is reached. In order to provide a stopping criterion for ray traversal, a logical bounding box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation whenever an object moves far away, the cost of maintaining the spatial subdivision will be substantially lower. On the other hand, because rays now may have to wrap around, more voxels may have to be traversed per ray, which will slightly increase ray traversal time.

During a pre-processing step, the grid is built as usual. We will call the bounding box of the entire scene at start-up the ’physical bounding box’. If during the animation an object moves outside the physical bounding box, either because it is placed by the user in a new location, or its programmed path takes it outside, the logical bounding box is extended to enclose all objects. Initially, the logical bounding box is equal to the physical bounding box. Insertion of an object which lies outside the physical bounding box is accomplished by wrapping the object around within the physical grid, as depicted in Figure 1 (left).

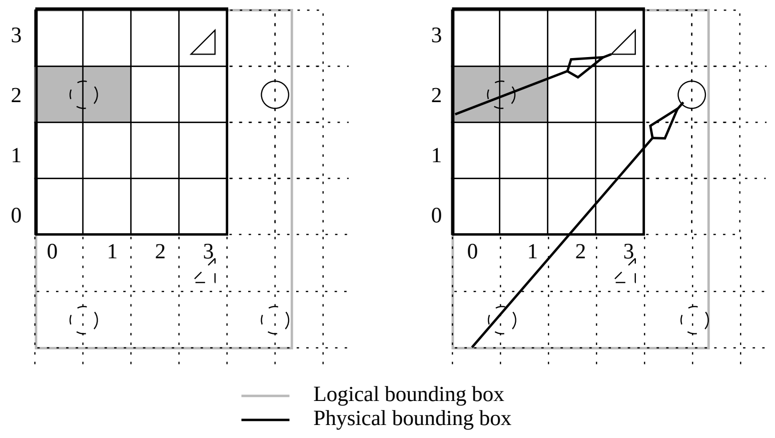

Fig. 1. Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.

As the logical bounding box may be larger than the physical bounding box, ray traversal now starts at the extended bounding box and ends if an intersection is found or if the ray leaves the logical bounding box. In the example in Figure 1 (right), the ray pointing to the sphere starts within a logical voxel, voxel (-2, 0), which is mapped to physical voxel (0, 2). The logical coordinates of the sphere are checked and found to be outside of the currently traversed voxel and thus no intersection test is necessary. The ray then progresses to physical voxel (1, 2). For the same reason, no intersection with the sphere is computed again. Traversal then continues until the sphere is intersected in logical voxel (4, 2), which maps to physical voxel (0, 2).

Objects that are outside the physical grid are tagged, so that in the above example, when the ray aimed at the triangle enters voxels (0, 2) and (1, 2), the sphere does not have to be intersected. Similarly, when the ray is outside the physical grid, objects that are within the physical grid need not be intersected. As most objects will initially lie within the physical bounds, and only a few objects typically move away from their original positions, this scheme speeds up traversal considerably for parts of the ray that are outside the physical bounding box.

When the logical bounding box becomes much larger than the physical bounding box, there is a tradeoff between traversal speed (which deteriorates for large logical bounding boxes) and the cost of rebuilding the grid. In our implementation, the grid is rebuilt when the length of the diagonals of the physical and logical bounding boxes differ by a factor of two.

Hence, there is a hierarchy of operations that can be performed on grids. For small to moderate expansions of the scene, wrapping both rays and objects is relatively quick without incurring too high a traversal cost. For larger expansions, rebuilding the grid will become a more viable option.

This grid implementation shares the advantages of simplicity and cheap traversal with commonly used grid implementations. However, it adds the possibility of increasing the size of the scene without having to completely rebuild the grid every time there is a small change in scene extent. The cost of deleting and inserting a single object is not constant and depends largely on the size of the object relative to the size of the scene. This issue is addressed in the following section.

4 Hierarchical grids

As was noted in the previous section, the size of an object relative to each voxel in a grid influences how many voxels will contain that object. This in turn negatively affects insertion and deletion times. Hence, it would make sense to find a spatial subdivision whereby the voxels can have different sizes. If this is accomplished, then insertion and deletion of objects can be made independent of their sizes and can therefore be executed in constant time. Such spatial subdivisions are not new and are known as hierarchical spatial subdivisions. Octrees, bintrees and hierarchical grids are all examples of hierarchical spatial subdivisions. However, normally such spatial subdivisions store all their objects in leaf nodes and would therefore still incur non-constant insertion and deletion costs. We extend the use of hierarchical grids in such a way that objects can also reside in intermediary nodes or even in the root node for objects that are nearly as big as the entire scene.

Because such a structure should also be able to deal with expanding scenes, our efforts were directed towards constructing a hierarchy of grids (similar to Sung [12]), thereby extending the functionality of the grid structure presented in the previous section. Effectively, the proposed method constitutes a balanced octree.

Object insertion now proceeds similarly to grid insertion, except that the grid level needs to be determined before insertion. This is accomplished by comparing the size of the object in relation to the size of the scene. A simple heuristic is to determine the grid level from the diagonals of the two bounding boxes. Specifically, the length of the grid's diagonal is divided by the length of the object's diagonal, the result determining the grid level. Insertion and deletion progresses as explained in the previous section.

The gain of constant time insertion is offset by a slightly more complicated traversal algorithm. Hierarchical grid traversal is effectively the same as grid traversal with the following modifications. Traversal always starts at a leaf node which may first be mapped to a physical leaf node as described in the previous section. The ray is intersected with this voxel and all its parents until the root node is reached. This is necessary because objects at all levels in the hierarchy may occupy the same space as the currently traversed leaf node. If an intersection is found within the space of the leaf node, then traversal is finished. If not, the next leaf node is selected and the process is repeated.

This traversal scheme is wasteful because the same parent nodes may be repeatedly traversed for the same ray. To combat this problem, note that common ancestors of the current leaf node and the previously intersected leaf node, need not be traversed again. If the ray direction is positive, the current voxel's number can be used to derive the number of levels to go up in the tree to find the common ancestor between the current and the previously visited voxel. For negative ray directions, the previously visited voxel's number is used instead. Finding the common ancestor is achieved using simple bit manipulation, as detailed in Figure 2.

```
bitmask = (raydir_x > 0) ? x : x + 1
forall levels in hierarchical grid
{
  cell = hgrid[level][x>>level][y>>level][z>>level]
  forall objects in cell
    intersect(ray, object)
  if (bitmask & 1)
    return
  bitmask >>= 1
}
```

Fig. 2. Hierarchical grid traversal algorithm in C-like pseudo-code. The bitmask is set assuming that the last step was along the x-axis.

As the highest levels of the grid may not contain any objects, ascending all the way to the highest level in the grid is not always necessary. Ascending the tree for a particular leaf node can stop when the largest voxel containing objects is visited.

This hierarchical grid structure has the following features. The traversal is only marginally more complex than standard grid traversal. In addition, wrapping of objects in the face of expanding scenes is still possible. If all objects are the same size, this algorithm effectively defaults to grid traversal. Insertion and deletion can be achieved in constant time, as the number of voxels that each object overlaps is roughly constant[1].

5 Evaluation

The grid and hierarchical grid spatial subdivisions were implemented using an interactive ray tracer [9], which runs on an SGI Origin 2000 with 32 processors. For evaluation purposes, two test scenes were used. In each scene, a number of objects were animated using pre-programmed motion paths. The scenes as they are at start-up are depicted in Figure 5 (top). An example frame taken during the animation is given for each scene in Figure 5 (bottom). All images were rendered on 30 processors at a resolution of 512^2 pixels.

To assess basic traversal speed, the new grid and hierarchical grid implementations are compared with a bounding volume hierarchy. We also compared our algorithms with a grid traversal algorithm which does not allow interactive updates. Its internal data structure consists of a single array of object pointers, which improves cache efficiency on the Origin 2000.

From here on we will refer to the new grid implementation as 'interactive grid' to distinguish between the two grid traversal algorithms. As all these spatial subdivision methods have a user defined parameter to set the resolution (voxels along one axis and maximum number of grid levels, respectively), various settings are evaluated. The overall performance is given in Figure 3 and is measured in frames per second.

[1]Note that this also obviates the need for mailbox systems to avoid redundant intersection tests.

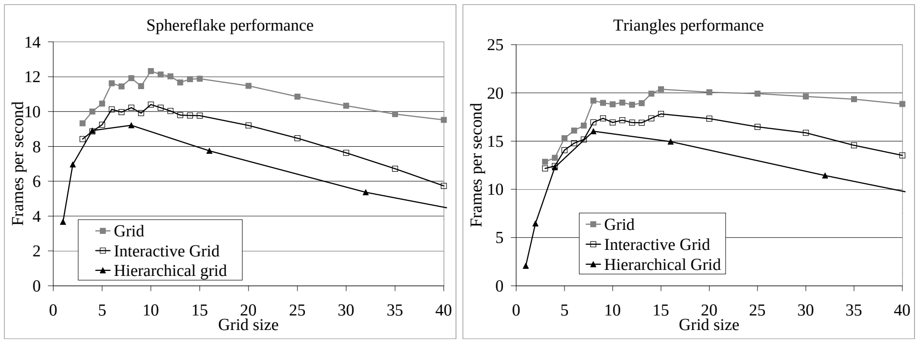

Fig. 3. Performance (in frames per second) for the grid, the interactive grid and the hierarchical grid for two static scenes. The bounding volume hierarchy achieves a frame rate of 8.5 fps for the static sphereflake model and 16.4 fps for the static triangles model.

The extra flexibility gained by both the interactive grid and hierarchical grid implementations results in a somewhat slower frame rate. This is according to expectation, as the traversal algorithm is a little more complex and the Origin's cache structure cannot be exploited as well with either of the new grid structures. The graphs in Figure 3 show that with respect to the grid implementation the efficiency reduction is between 12% and 16% for the interactive grid and 21% and 25% for the hierarchical grid. These performance losses are deemed acceptable since they result in far better overall execution than dynamically reconstructing the original grid. For the sphereflake, all implementations are faster, for a range of grid sizes, than a bounding volume hierarchy, which runs at 8.5 fps. For the triangles scene, the hierarchical grid performs at 16.0 fps similarly to the bounding volume (16.4 fps), while grid and interactive grid are faster.

The non-zero cost of updating the scene effectively limits the number of objects that can be animated within the time-span of a single frame. However, for both scenes, this limit was not reached. In the case where the frame rate was highest for the triangles scene, updating all 200 triangles took less than 1/680th of a frame for the hierarchical grid and 1/323th of a frame for the interactive grid. The sphereflake scene costs even less to update, as fewer objects are animated. For each of these tests, the hierarchical grid is more efficiently updated than the interactive grid, which confirms its usefulness.

The size difference between different objects should cause the update efficiency to be variable for the interactive grid, while remaining relatively constant for the hierarchical grid. In order to demonstrate this effect, both the ground plane and one of the triangles in the triangle scene was interactively repositioned during rendering. The update rates for different size parameters for both the interactive grid and the hierarchical grid, are presented in Figure 4 (left). As expected, the performance of the hierarchical grid is relatively constant, although the size difference between ground plane and triangle is considerable. The interactive grid does not cope with large objects very well if these objects overlap with many voxels. Dependent on the number of voxels in the grid, there is one to two orders of magnitude difference between inserting a large and a small object. For larger grid sizes, the update time for the ground plane is roughly half a frame. This leads to visible artifacts when using an interactive grid, as during the update the processors that are rendering the next frame temporarily cannot intersect this object (it is simply taken out of the spatial subdivision). In practice, the hierarchical grid implementation does not show this disadvantage.

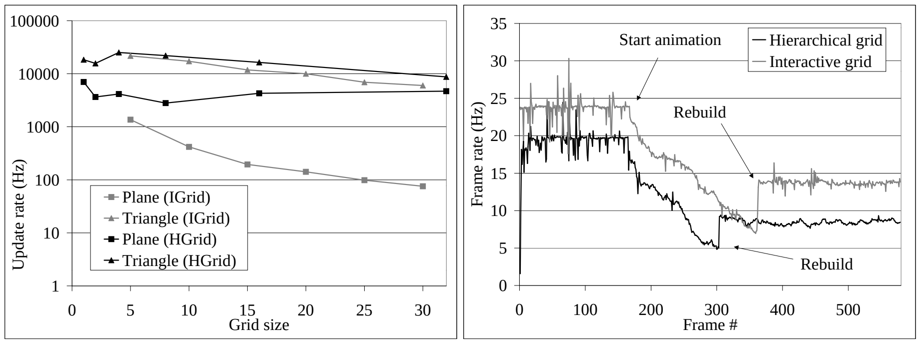

Fig. 4. Left: Update rate as function of (hierarchical) grid size. The plane is the ground plane in the triangles scene and the triangle is one of the triangles in the same scene. Right: Frame rate as function of time for the expanding triangle scene.

The time to rebuild a spatial subdivision from scratch is expected to be considerably higher than the cost of re-inserting a small number of objects. For the triangles scene, where 200 out of 201 objects were animated, the update rate was still a factor of two faster than the cost of completely rebuilding the spatial subdivision. This was true for both the interactive grid and the hierarchical grid. A factor of two was also found for the animation of 81 spheres in the sphereflake scene. When animating only 9 objects in this scene, the difference was a factor of 10 in favor of updating. We believe that the performance difference between rebuilding the acceleration structure and updating all objects is largely due to the cost of memory allocation, which occurs when rebuilding.

In addition to experiments involving grids and hierarchical grids with a branching factor of two, tests were performed using a hierarchical grid with a higher branching factor. Instead of subdividing a voxel into eight children, here nodes are split into 64 children (4 along each axis). The observed frame rates are very similar to the hierarchical grid. The object update rates were slightly better for the sphereflake and triangle scenes, because the size differences between the objects matches this acceleration structure better than both the interactive grid and the hierarchical grid.

In the case of expanding scenes, the logical bounding box will become larger than the physical bounding box. The number of voxels that are traversed per ray will therefore on average increase. This is the case in the triangles scene[2]. The variation over time of the frame rate is given in Figure 4 (right). In this example, the objects are first stationary. At some point the animation starts and the frame rate drops because the scene immediately starts expanding. At some point the expansion is such that a rebuild is warranted. The re-computed spatial subdivision now has a logical bounding box which is identical to the (new) physical bounding box and therefore the number of traversed voxels is reduced when compared with the situation just before the rebuild. The total frame rate does not reach the frame rate at the start of the computation, because the objects are more spread out over space, resulting in larger voxels and more intersection tests which do not yield an intersection point.

Finally, Figure 6 shows that interactively updating scenes using drag and drop interaction is feasible.

[2]For this experiment, the ground plane of the triangles scene was reduced in size, allowing the rebuild to occur after a smaller number of frames.

6 Conclusions

When objects are interactively manipulated and animated within a ray tracing application, much of the work that is traditionally performed during a pre-processing step becomes a limiting factor. Especially spatial subdivisions which are normally built once before the computation starts, do not exhibit the flexibility that is required for animation. The insertion and deletion costs can be both unpredictable and variable. We have argued that for a small cost in traversal performance flexibility can be obtained and insertion and deletion of objects can be performed in constant time.

By logically extending the (hierarchical) grids into space, these spatial subdivisions deal with expanding scenes rather naturally. For modest expansions, this does not significantly alter the frame rate. When the scenes expand a great deal, rebuilding the entire spatial subdivision may become necessary. For large scenes this may involve a temporary drop in frame rate. For applications where this is unacceptable, it would be advisable to perform the rebuilding within a separate thread (rather than the display thread) and use double buffering to minimize the impact on the rendering threads.

Acknowledgements

Thanks to Pete Shirley and Steve Parker for their help and comments and to the anonymous reviewers for their helpful comments. This work was supported by NSF grants CISE-CCR 97-20192, NSF-9977218 and NSF-9978099 and by the DOE Advanced Visualization Technology Center.

References

1. J. Amanatides and A. Woo, *A fast voxel traversal algorithm for ray tracing*, in Eurographics '87, Elsevier Science Publishers, Amsterdam, North-Holland, Aug. 1987, pp. 3–10.
2. K. Bala, J. Dorsey, and S. Teller, *Interactive Ray Traced Scene Editing using Ray Segment Tree*, in Rendering Techniques '99, pp. 31-44. Springer-Verlag, 1999.
3. A. Fujimoto, T. Tanaka, and K. Iwata, *ARTS: Accelerated ray tracing system*, IEEE Computer Graphics and Applications, 6 (1986), pp. 16–26.
4. V. Gaede and O. Günther, *Multidimensional access methods*, ACM Computing Surveys, 30 (1998), pp. 170–231.
5. A. S. Glassner, ed., *An Introduction to Ray Tracing*, Academic Press, 1989.
6. A. S. Glassner, *Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications, 4 (1984), pp. 15–22.
7. A. S. Glassner, *Spacetime ray tracing for animation*, IEEE Computer Graphics and Applications, 8 (1988), pp. 60–70.
8. M. J. Muuss, *Towards real-time ray-tracing of combinatorial solid geometric models*, in Proceedings of BRL-CAD Symposium, June 1995.
9. S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen, *Interactive ray tracing*, in Symposium on Interactive 3D Computer Graphics, April 1999.
10. B. Smits, *Efficiency issues for ray tracing*, Journal of Graphics Tools, 3 (1998), pp. 1–14.
11. J. Spackman and P. Willis, *The SMART navigation of a ray through an oct-tree*, Computers and Graphics, 15 (1991), pp. 185–194.
12. K. Sung, *A DDA octree traversal algorithm for ray tracing*, in Eurographics '91, W. Purgathofer, ed., North-Holland, sept 1991, pp. 73–85.

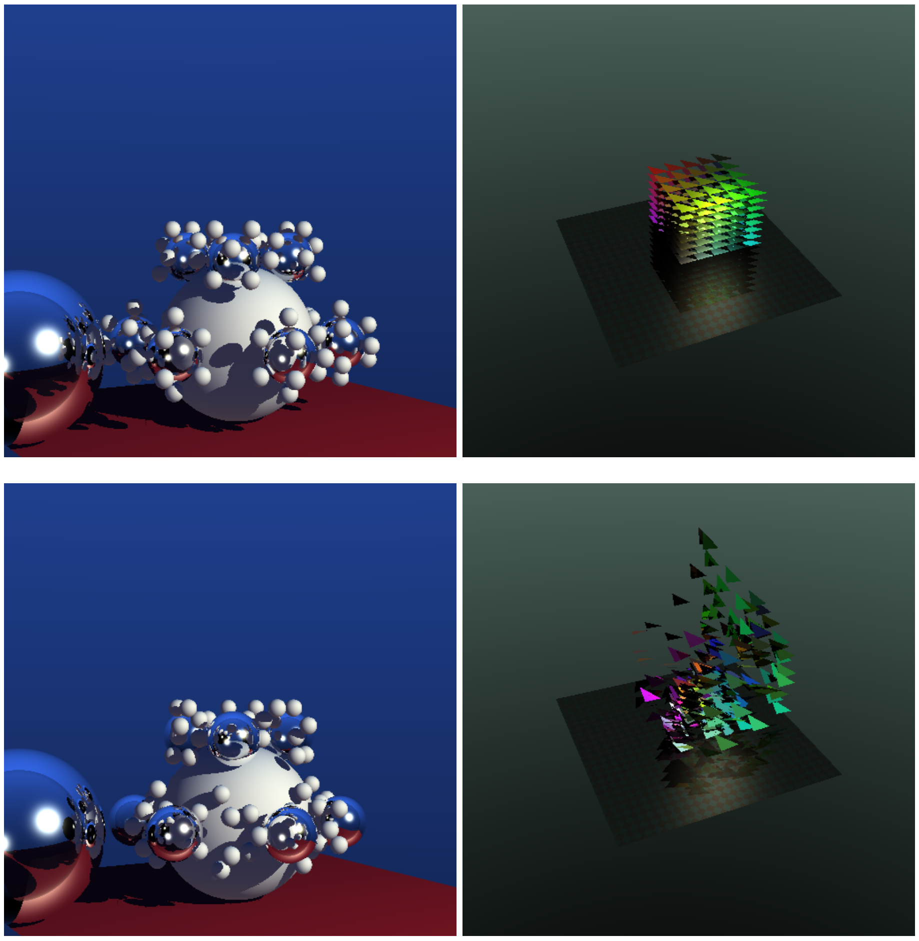

Fig. 5. Test scenes before any objects moved (top) and during animation (bottom).

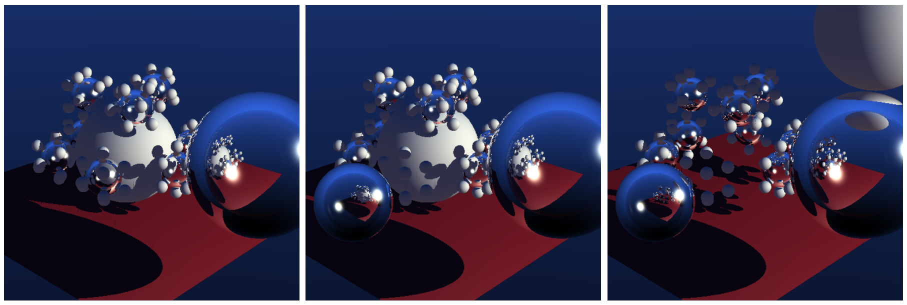

Fig. 6. Frames created during interactive manipulation.