Distributed Interactive Ray Tracing of Dynamic Scenes

Ingo Wald Carsten Benthin Philipp Slusallek
Computer Graphics Group, Saarland University
{wald,benthin,slusallek}@cs.uni-sb.de

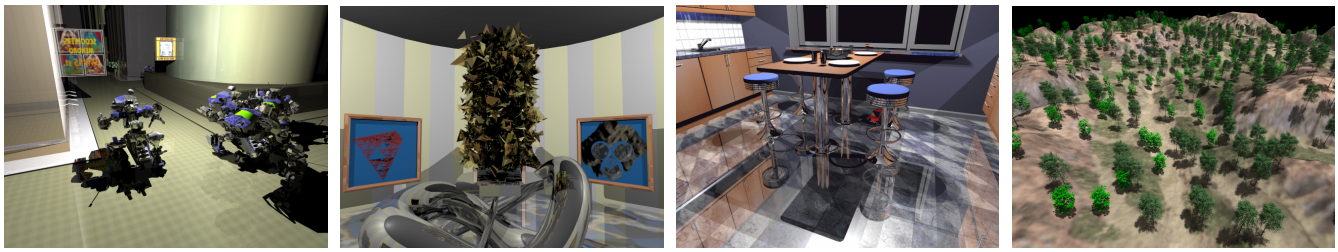

Figure 1: Examples of dynamic scenes interactively ray traced with our method: a.) Hierarchical animation in the "BART Robots" scene, b.) Unstructured motion in the "BART Museum", c.) "BART Kitchen", d.) one thousand instances of two kinds of complex trees (rendered with shadows), more than ten million triangles total. All pictures are live screenshots from our system running interactively on a cluster of PCs.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing, Animation, Color, shading, shadowing, and texture; I.3.2 [Graphics Systems]: Graphics Systems—Distributed/network graphics; I.3.3 [Graphics Systems]: Picture/Image Generation—Display algorithms;

Keywords: Interactive Ray Tracing, Distributed Rendering, Dynamic Scenes, Scalability and Performance

Abstract

Recently developed interactive ray tracing systems combine the high performance of todays CPUs with new algorithms and implementations to achieve a flexible and high-performance rendering system offering high-quality, but nonetheless interactive 3D graphics. However, due to its history in off-line rendering, interactive ray tracing is usually limited to static scenes and simple walkthroughs. In order to become truly interactive ray tracing *must* efficiently support dynamic scenes.

In this paper, we present a simple and practical method that allows to interactively ray trace dynamic scenes in a distributed PC cluster environment. Our method separates the scene into independent objects with common properties concerning dynamic updates — similar to OpenGL display lists and scene graph libraries. Three classes of objects are distinguished: Static objects are treated as before, objects undergoing affine transformations are handled by transforming rays, and objects with unstructured motion are rebuilt whenever necessary.

We present performance and scalablity results of our system using a variety of test scenes stressing a wide range of dynamic behaviour.

1 Introduction

Methods for creating computer generated images can be broadly classified into two different approaches, both with different strengths and weaknesses. On one side, triangle rasterization is easy to build in hardware, is cheaply available on todays graphics cards, and clearly dominates todays interactive graphics market. On the other side, ray tracing is well-known for achieving superior image quality, but is also infamous for its high computational cost, and has therefore traditionally been used only for off-line rendering.

Recently, the speed of ray tracing has been improved to interactive rates, e.g. [Wald et al. 2001a; Parker et al. 1999; Wald et al. 2003]. For a number of applications, interactive ray tracing even starts to challenge the dominating role of triangle rasterization: Due to its logarithmic behavior in scene complexity, ray tracing becomes increasingly efficient for complex environments [Wald et al. 2001b; Wald et al. 2002a]. It also offers a much more flexible image generation algorithm than rasterization, supporting features that are often hard to achieve with rasterization hardware, including arbitrary programmable shading and exact shadows, reflections, and refractions. Recently, even full global illumination has been computed at interactive rates [Wald et al. 2002b; Benthin et al. 2003]. New hardware architectures for real-time ray tracing are currently beeing investigated [Purcell et al. 2002; Carr et al. 2002; Schmittler et al. 2002].

Today, approaches to interactive ray tracing range from supercomputers [Parker et al. 1999] over PC clusters [Wald et al. 2001a; Wald et al. 2003; DeMarle et al. 2003], to special-purpose hardware [Schmittler et al. 2002] and implementations on GPUs [Purcell et al. 2002]. Even though it is not yet clear which of these different approaches eventually prove to be the most successful, they indicate that in the not too far ahead future, interactive ray tracing will probably play a bigger role in interactive 3D graphics.

1.1 Ray Tracing in Dynamic Environments

Even though ray tracing is a relatively old and well-understood technique, its use for interactive applications is still in its infancy. Ray tracing research so far almost exclusively concentrated on accelerating the process of creating a *single* image, which could take from minutes to hours. Most of these approaches relied on doing extensive preprocessing by building up complex data structures to accelerate the process of tracing a ray. This preprocessing was then amortized over the remainder of a frame.

This approach was very successful for off-line computations,

where the cost for building the data structures was negligible compared to the cost for the actual rendering phase. At interactive frame rates however, this approach is not feasible any more.

As such, building the acceleration data structure becomes the bottleneck due to its super-linear behavior with respect to scene complexity. In dynamic scenes, where the acceleration structure needs to be rebuilt for every frame, this preprocessing alone would often exceed the total time available per frame in an interactive setting.

Even worse, this preprocessing phase cannot easily be parallelized. Though the actual ray tracing phase can be parallilized trivially, such preprocessing usually has to be performed on each client, and thus does not parallelize. This poses a major problem for interactively ray tracing dynamic scenes, as virtually all of todays interactive ray tracing systems have to rely on massive parallelization to achieve interactive frame rates [Wald et al. 2001b; Parker et al. 1999; DeMarle et al. 2003].

Therefore, it is not surprising that all of those systems mainly concentrate on the actual ray tracing phase and do not target dynamic scenes. Without methods for interactively modifying the scene, however, interactive ray tracing will be limited to simple walkthroughs of static environments, and can therefore hardly be termed truly interactive, as real *interaction* between the user and the environment is possible.

In this paper, we propose a method to efficiently support dynamic scenes in a distributed interactive ray tracing system. The main goal of our method is to offer good performance and high scalability – in both frame rate and in the number of CPUs – in a distributed PC cluster environment. While our method has been specially designed for a distributed architecture, it should be directly applicable to other architectures, e.g. [Schmittler et al. 2002; Purcell et al. 2002]. In fact, many of the remaining limitations of our system would diminish in a non-distributed environment.

2 Previous Work

Ray tracing has first been used by Appel [1968], and has been adopted and extended by many other researchers, e.g. [Whitted 1980; Cook et al. 1984; Glassner 1989]. Since then, speeding up ray tracing has attracted much attention, and has lead to dozens of algorithms. Most of these algorithms rely on reducing the required ray/object intersection tests by building an *acceleration structure* over the scene's geometry

[Glassner 1989; Havran 2001].

Quite recently, ray-tracing has been accelerated to the point where interactive frame rates can be achieved at least for moderate screen resolutions. By exploiting the inherent parallelism of ray-tracing Muuss [1995] and Parker et al. [1999; 1998] achieved interactive ray tracing performance on shared memory supercomputer systems by massive parallelization and low-level optimizations. Wald et al. [2001a] have then shown that interactive ray-tracing performance can also be obtained on inexpensive, off-the-shelf PCs. As long as the scene remained static, this system has been shown to scale well in a distributed memory environment using commodity PCs and networks [Wald et al. 2001b].

Unfortunately, all these systems are closely tied to acceleration structures that have been designed for static environments, thus limiting interactive ray tracing systems to simple walkthroughs of static scenes. Some methods have been proposed for the case where predefined animation paths are known in advance, e.g. [Glassner 1988; Gröller and Purgathofer 1991]. These however are not applicable to our target setting of totally dynamic, unpredictable changes to the scene. Little research is available for truly interactive systems.

Excellent research on ray tracing in dynamic environments has recently been performed by Lext et al. with the BART project [Lext et al. 2000]. They provide an excellent analysis of the problems arising in dynamic scenes. Based on this analysis, they proposed a representative set of test scenes designed to stress the different aspects of ray tracing dynamic scenes. This *BART benchmark suite* provides an excellent tool for evaluating and analyzing a dynamic ray tracing engine, and will be used extensively in our experiments. In their research, the behavior of dynamic scenes was classified into two inherently different classes: One form is *hierarchical motion*, where a whole group of primitives is subject to the same transformation. The other class is *unstructured motion*, where each triangle moves without relation to all others. For a closer explanation of the different kinds of motion, see [Lext et al. 2000].

In a first step, Parker et al. [1999] excluded moving primitives from their acceleration structure and checked them individually for every ray. This of course is only feasible for a small number of moving primitives.

Another approach would be to efficiently update the acceleration structure whenever objects move. Because objects can occupy a large number of cells in an acceleration structure this may require costly updates to large parts of the acceleration structure for each moving primitive. To overcome this problem, Reinhard et al. [2000] proposed a dynamic acceleration structure based on a hierarchical grid. In order to quickly insert and delete objects independently of their size, larger objects are being kept in coarser levels of the hierarchy. As a result, objects always cover approximately a constant number of cells, thus allowing to update the acceleration structure in constant time. However, their method resulted in a rather high overhead, and also required their data structure to be rebuilt once in a while to avoid degeneration. Furthermore, their method mainly concentrated on unstructured motion, and is not optimal for hierarchical animation.

Recently, Lext and Akenine-Moeller [2001] proposed a way for quickly reconstructing an acceleration structure in a hierarchically animated scene. Though their method was developed independently, it is closely related to the way that our method handles hierarchical animation. To our knowledge, their method has never been used in an interactive context.

3 Our Approach

Our approach is motivated by the same observations as Lext et al. [2001] of how dynamic scenes typically behave: Large parts of a scene often remain static over long periods of time. Other parts of a scene undergo well-structured transformations such as rigid motion or affine transformations. Yet other parts are changed in a totally unstructured way. This common structure within scenes can be exploited by maintaining geometry in separate *objects* according to their dynamic properties, and handling the different kinds of motion with different, specialized algorithms that are then combined into a common architecture.

Each object can consist of an arbitrary number of triangles. It has its own acceleration structure and can be updated independently of the rest of the scene. Of course, an additional top-level acceleration structure must then be maintained that accelerates ray traversal between the objects in a scene. Each ray then first starts traversing this toplevel structure. As soon as a leaf is found, the ray is intersected with the objects in the leaf by simply traversing the respective objects local acceleration structures.

For *static objects*, ray traversal works as before by just traversing the ray with our usual, fast traversal algorithm.

For *hierarchical animation*, the ray in world-space has to be intersected with an object that has been transformed with an affine transformation. For affine transformations, however, it is a well-known fact that exactly the same result can be reached by not transforming the object, but instead intersecting the inversely transformed ray with the untransformed object [Lext and Akenine-

Moeller 2001]. This slightly increases the per-ray cost (e.g. for the transformation), but totally removes the reconstruction cost for hierarchically animated objects.

To enable this scheme, all triangles that are subject to the same set of transformations (e.g. all the triangles forming the head of the animated robot in Figure

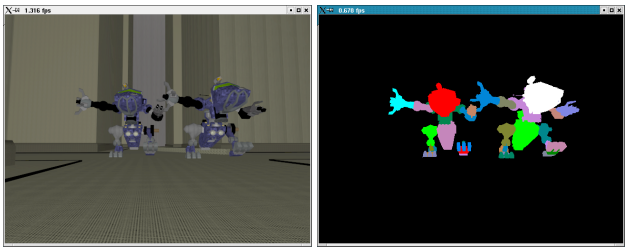

Figure 2: Robots (left), with color-coded objects (right). Triangles of the same color belong to the same object.

While this simple trick of transforming rays instead of triangles elegantly avoids any reconstruction cost for hierarchical motion, it does not work for *unstructured motion*, as there the acceleration structure has to be rebuilt for every frame. Even so, if triangles under unstructured motion are kept in a separate object, the BSP reconstruction cost can be localized to only those triangles that have actually been transformed in an unstructured way. The local acceleration structures of such objects are discarded and rebuilt from the transformed triangles whenever neccessary. Even though this process is costly, it is only required for objects with unstructured motion and does not affect any of the other objects. Our method allows to specify several different objects with unstructured motion in the same scene. Thus, only those objects have to be rebuilt that have actually been changed.

Additionally, object BSP reconstruction could be done *on demand*, i.e. only as soon as a ray actually requests intersection with such an object. This would further reduce the cost of this reconstruction step.

Even though our method totally avoids any BSP reconstruction cost for hierarchically animated objects, hierarchical animation still invalidates the toplevel BSP, and as such makes it very likely that this toplevel structure has to be rebuilt every frame. However, the cost for rebuilding this toplevel BSP only depends on the number of *objects* in a scene, and is totally independent of the actual number of triangles contained in these objects. As such, even multi-million-triangle objects can be transformed with negligible cost. Furthermore, the toplevel BSP usually contains only rather few objects (in the order of a few hundred) that allow to quickly rebuild that structure without major impact on the total systems performance. Additionally, we use highly optimized algorithms for building and traversing the toplevel BSP.

4 Implementation

Before discussing the actual algorithms for quickly building and traversing acceleration structures, we first give a brief overview of how objects may be specified by an application. This description is based on the proposed OpenRT API for interactive ray tracing [Dietrich et al. 2003]. Similar to OpenGL, OpenRT operates on a very low level that allows it to be used from almost any application or scene graph library. For this paper we concentrate on only the small aspect of OpenRT relevant for dynamic scenes.

4.1 OpenRT API

Using our method is similar to the way optimized applications make use of OpenGL [Neider et al. 1993]: Primitives are grouped into display lists depending on their (dynamic update) properties. All triangles inside display lists can then be transformed efficiently by adjusting the transformation stack before calling the display list. Triangles with unstructured motion would simply not be kept in a display list, but rather be rendered directly in immediate mode.

With OpenRT, an application operates in a very similar way, by using *objects* instead of *display lists*. The main difference between them is that OpenRT objects do not allow for side effects. In both cases, it is the application that needs to organize its geometry accordingly. Instead of using immediate mode for primitives with unstructured motion, special objects are used that may be redefined whenever necessary. As with OpenGL display lists, the exact way that the primitives are grouped into different objects can affect performance. For scenegraph-like applications, however, this grouping is often trivial.

The OpenRT API for defining geometry is very similar to OpenGL: Objects are defined by calls to *rtGenObjects*, *rtNewObject*, and *rtEndObject*, which closely correspond to the OpenGL functions for specifying display lists. This similarity simplifies porting of OpenGL based applications. Within an object, primitives are then specified using functions like *rtBegin/End(RT_TRIANGLE)* and *rtVertex3f()*, with all the functionality for transformations (e.g. *rtRotatef()*) and matrix stack handling (e.g. *rtPushMatrix()*) being supported.

After an object has been defined, it can be instantiated any time with a call to *rtInstantiateObject()* (corresponding to *glCallList()* using the transformation currently on the transformation stack. Each such *instance* of an object consists of a reference to the original object and of the transformation applied to it. These instances are then organized in a toplevel hierarchy as depicted in Figure

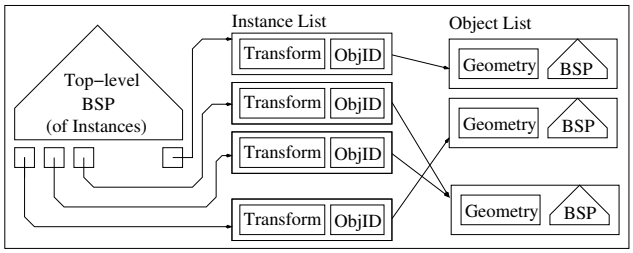

Figure 3: Two-level hierarchy as used in our approach: A top-level BSP contains references to instances, which contain a transformation and a reference to an object. Objects in turn consist of geometry and a local BSP tree. Multiple instances can refer to the same object with different transformations.

4.2 Object Construction and Traversal

During traversal of the data structure, rays have to be intersected with objects. As described above, each object consists of a set of geometric primitives as well as its own acceleration structure for fast traversal within that object.

Within each object, traversal is identical to traditional ray tracing in a static environment. Consequently, we use exactly the same algorithms for building and traversing that acceleration structure. In our case, we use exactly the BSP tree, data structures, and algorithms described in [Wald et al. 2001a].

For both static objects and for those with hierarchical motion, the local BSP tree must only be built once directly after object definition. Thus, the time for building these objects is not an issue, allowing us to use sophisticated and slow algorithms for building the acceleration structures[1] already used in [Wald et al. 2001a].

[1]Though the object (and thus its BSP) remains static in its local coordinate system, its instance in the world coordinate system can still be trans-

An extensive study of such algorithms can for example be found in [Havran 2001].

4.3 Fast Handling of Unstructured Motion

The mentioned algorithms for creating highly optimized BSP trees (e.g. with sophisticated cost-functions) may require several seconds even for moderately complex objects. Thus, they are not applicable to unstructured motion, where the object BSP has to be rebuilt every frame (and thus in fractions of a second). For these cases we trade traversal speed for construction speed by using less expensive, simple heuristics for BSP plane placement, and by using different quality parameters for BSP construction.

A particularly important cost factor for BSP tree construction is the subdivision criterion of the BSP. This criterion typically consist of a maximum tree depth and a target number of triangles per leaf cell. Subdivision continues on cells with more than the target number of triangles up to the maximum depth. Typical criteria specify 2 or 3 triangles per cell and usually result in fast traversal times – but also in deeper BSPs, which are more costly to create. Particularly costly are degenerate cases, in which subdivision can not reduce the number of triangles per cell, for example if too many primitives occupy the same point in space, e.g. at vertices with a valence higher than the maximum numbers of triangles.

In order to avoid such excessive subdivisions in degenerate regions, we modified the subdivision criterion (for unstructured object BSPs): The deeper the subdivision, the more triangles will be tolerated per cell. We currently increase the tolerance threshold by a constant factor for each level of subdivision. Thus, we generally obtain significantly lower BSP trees and larger leaf cells than for static objects. Even though this slows down the traversal of rays hitting such objects, this slowdown is more than made up by the significantly shorter construction time. With these compromises on BSP construction, unstructured motion for moderate-sized objects can be supported by rebuilding the respective object BSP every frame.

4.4 Efficient Traversal

Having a separate acceleration structure for every object allows for efficiently intersecting each ray with its geometry. We also use a BSP tree for the top-level acceleration structure (Figure

As with the original implementation, a ray is first clipped to the scene bounding box and is then traversed iteratively through the top-level BSP tree. As soon as it encounters a leaf cell, it sequentially intersects the ray with all instances in this cell: For each instance, the ray is first transformed to the local coordinate system of the object. The transformed ray is then intersected with the object using the original algorithms. The cost of intersecting a ray with the same instance multiple times can efficiently be avoided with mailboxing [Havran 2001].

As our traversal and intersection algorithms do not require normalized ray directions, transforming a ray is relatively cheap, as no costly re-normalization of the transformed rays is necessary. The ray-matrix multiplications themselves can very efficiently be done using SSE [Int 2002].

Of course, our method also works with our fast SSE packet traversal code. The only caveat is that this packet traversal code requires that all rays directions in a packet have the same signs [Wald et al. 2001a]. As a rotation can change these signs, an additional check has to be done after each transformation, and such a packet might sometimes have to be split up. However, this is trivial to check and work around, and happens rarely. As such, its cost is negligible.

formed with affine transformations.

4.5 Fast Top-Level BSP Construction

While traversing the top-level BSP requires only minor changes to the original implementation, this is not the case for the construction algorithm. A scene can easily contain hundreds or thousands of instances (see Figure

Fortunately, the task of building the top-level BSP is simpler than for object BSPs: Object BSPs require costly triangle-in-cell computations, careful placement of the splitting plane, and handling of degenerate cases. The top-level BSP however only contains instances represented by an axis-aligned bounding box (AABB) of its transformed object. Considering only the AABBs, optimized placement of the splitting plane becomes much easier, and any degenerate cases can be avoided.

For splitting a cell, we follow several observations [Havran 2001]:

1. It is usually beneficial to subdivide a cell in the dimension of its maximum extent, as this usually yields the most well-formed cells.

2. Placement of the BSP plane only makes sense at the boundary of objects contained within the current cell. This is due to the fact that the cost-function can be maximal only at such boundaries.

3. It can been shown that the optimal position for the splitting plane lies between the cells geometric center and the object median.

Following these observations, the BSP tree can be built such that it is both suited for fast traversal by optimized plane placement, and can still be built quickly and efficiently: For each subdivision step, we try to find a splitting plane in the dimension of maximum extent (observation 1). As potential splitting planes, only the AABB borders will be considered (observation 2). To find a good splitting plane, we first split the cell in the middle, and decide which side contains more objects, i.e. which one contains the object median. From this side, we choose the object boundary closest to the center of the cell. Thus, the splitting plane lies inbetween cell center and object median, which is generally a good choice (observation 3).

As each subdivision step removes at least one potential splitting plane, termination of the subdivision can be guaranteed without further termination criteria. Degenerate cases for overlapping objects cannot happen, as only AABB boundaries are considered, and not the overlapping space itself. Choosing the splitting plane in the described way also yields relatively small and well-balanced BSP trees.

For our simplified problem, this simple heuristic is usually almost as good as a surface area heuristic (SAH, see [Havran 2001]), but much faster to evaluate. Thus, we get an optimized toplevel BSP that can be traversed quickly, while still offering a fast and efficient construction algorithm.

The whole algorithm for toplevel BSP reconstruction can be expressed in only a few lines of pseudo-code:

```
BuildTree(instances,voxel)
for d = x,y,z in order of maximum extent
    P = {i.min_d, i.max_d | i ∈ instances}
    if (||P|| = 0) continue;
    c = center of voxel
    if (more instances on left side of c than on right)
        p = max({p ∈ P|p < c})
    else
        p = min({p ∈ P|p >= c})
    Split Cell (instances,cell) in d at p into
          (leftvox,leftinst),(rightvox,rightinst)
    l = BuildTree(leftinst,leftvox);
    r = BuildTree(rightinst,rightvox);
    return InnerNode(d,p,l,r);
end for
# no valid splitting plane found
return Leaf(instances)
```

5 Experiments and Results

As the main emphasis of our system lies on efficiently coping with the distribution issues of a loosely-coupled PC cluster environment, we will put special emphasis on evaluating the scalability of our method. To allow for representative results, we have chosen to use a wide range of experiments and test scenes. Therefore, we have chosen to use the BART benchmark scenes [Lext et al. 2000], which represent a wide variety of stress factors for ray tracing of dynamic scenes. Additionally, we use several of the scenes that we encountered in practical applications [Wald et al. 2002b], and a few custom-made scenes for stress testing. Snapshots of these test scenes can be found in Figure

All of the following experiments have been performed on a cluster of dual AMD AthlonMP 1800+ machines with a FastEthernet network connection. The network is fully switched with a single gigabit uplink to a dual AthlonMP 1700+ server. The application is running on the server and is totally unaware of the distributed rendering happening inside the rendering engine. It manages the geometry in a scene graph, and transparently controls rendering via calls to the OpenRT API. All examples are rendered at video resolution of 640×480 pixels. Ray tracing is performed with costly programmable shaders featuring shadows, reflections, and texturing.

5.1 BART Kitchen

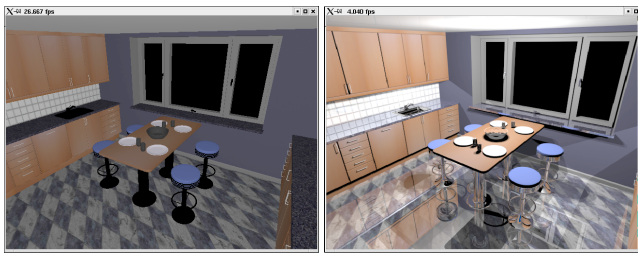

Figure 4: Two snapshots from the BART kitchen. a.) OpenGL-like ray casting at $>$ 26 fps on 32 CPUs. b.) full-featured ray tracing with shadows and 3 levels of reflections, at $>$ 7 fps on 32 CPUs.

The Kitchen scene contains hierarchical animation of 110.000 triangles organized to 5 objects. It requires negligible network bandwidth and BSP construction overhead. Overlap of bounding boxes may results in a certain overhead, which is hard to measure exactly but is definitely not a major cost factor.

The main cost of this scene is due to the need for tracing many rays to evaluate shadows from 6 point lights. There is also a high degree of reflectivity on many objects. Due to fast camera motion and highly curved objects (see Figure

We achieve interactive frame rates even for the large amount of rays to be shot. A reflection depth of 3 results in a total of 3.867.661 rays/frame. At a measured rate of 912.000 rays traced per second and CPU in this scene, this translates to a frame rate of 7.55 fps on 32 CPUs. Scalability is almost linear (see Table

CPUs	2	4	8	16	32
OpenGL-like	3.2	6.4	12.8	25.6	$>$ 26
Ray Tracing	0.47	0.94	1.88	3.77	7.55

Table 1: Scalability in the Kitchen scene in frames/sec.

5.2 BART Robots

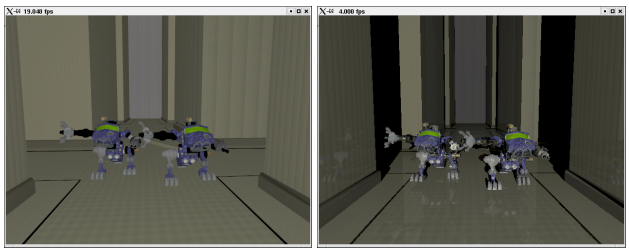

Figure 5: BART Robots: 16 robots consisting of 161 objects rendered interactively. a.) Simple ray casting at $>$ 26 fps on 32 CPUs. b.) Full ray tracing at $>$ 8 fps at 32 CPUs.

The Robots scene mainly stresses hierarchical animation: 16 Robots move through a complex city with hierarchical animation of their body parts that are organized into 161 different objects. All dynamic motion is hierarchical with no unstructured motion at all. Therefore, the BSP trees for all objects have to be built only once, and only the top-level BSP has to be rebuilt for every frame.

Using the algorithms described above, rebuilding the top-level BSP is very efficient taking less than one millisecond. Furthermore, updating the transformation matrices requires only a small network bandwidth of roughly 20 kb/frame for each client.

CPUs	2	4	8	16	32
OpenGL-like	2.8	5.55	10.8	21	$>$ 26
Ray Tracing	0.54	1.07	2.15	4.3	8.6

Table 2: Scalability in the Robots scene in frames/sec.

With such a small transmission and reconstruction overhead, we again achieve almost linear scalability (see Table

5.3 BART Museum

The museum has been designed mainly for testing unstructured motion and is the only BART scene featuring non-hierarchical motion. In the center of the museum, several triangles are animated on predefined animation paths to form differently shaped objects. The number of triangles undergoing unstructured motion can be configured to 64, 256, 1k, 4k, 16k, or 64k. Even though the complete animation paths are specified in the BART scene graph, we do not make use of this information. User controlled movement of the triangles – i.e. without knowledge of future positions – would create the same results.

num tris	64	256	1k	4k	16k	64k
reconst.time	$< 1ms$	2ms	8ms	34ms	0.1s	> 1s
bandwidth/client	6.4k	25.6k	102k	409k	1.6M	6.5M

Table 3: Unstructured motion in different configurations of the museum scene. Entries specify reconstruction time for the object BSP, and data sent to each client for updating the triangle positions.

As expected, unstructured motion gets costly for many triangles. Building the BSP tree for the complex version of 64k triangles already requires more than one second (see Table

Furthermore, the reconstruction time is strongly affected by the distribution of triangles in space: In the beginning of the animation, all triangles are uniformly and almost-randomly distributed in space. This is the worst case for BSPs, which are best at handling uneven distributions, and construction is consequently costly. During the animation, the triangles organize themselves to form a single surface. This results in much faster reconstruction time. The numbers given in Table

Apart from raw reconstruction cost, significant network bandwidth is required for sending all triangles to every client. Since we use reliable unicast (TCP/IP) for network transport, using 4096 triangles and 16 clients (32 CPUs), requires to transfer roughly 6.5 Mb (16 clients $\times$ 408kb, see Table

This scene also requires the computation of shadows from two point lights as well as large amounts of reflection rays. All of the moving triangles are reflective and incoherently sample the whole environment (see Figure

OpenGL-like	1	2	4	8	16
Robots	2.8	5.55	10.8	21	26($\star$)
Kitchen	3.2	6.4	12.8	25.6	26($\star$)
Terrain	1.3	2.5	4.8	8	15
Museum/1k	2.7	5.4	10.2	19.5	26($\star$)
Museum/4k	2.5	4.5	7.5	4.5	2.5
Museum/16k	1.6	2.4	1.7	1	0.5

Ray Tracing	1	2	4	8	16
Robots	0.54	1.07	2.15	4.3	8.6
Kitchen	0.47	0.94	1.88	3.77	7.55
Terrain	0.9	1.77	3.39	6.5	12
Museum/1k	0.6	1.2	2.4	4.8	9.3
Museum/4k	0.55	1.1	2.2	4.2	2.5
Museum/16k	0.45	0.9	1.65	0.98	0.53

Table 4: Scalability of our method in the different test scenes. "$\star$" means that the server's gigabit network connection is completely saturated due to the transfer of RGB color data, and thus no higher performance can be achieved. The numbers in the upper table correspond to OpenGL-like shading, the lower ones are for full ray tracing including shadows and reflections.

Even with all these effects – unstructured motion, shadows, and highly incoherent reflections in the animated triangles – the museum can be rendered interactively: Using 8 clients, we achieve 4.8 fps for 1024 triangles and still 4.2 fps for 4096 triangles at video resolution (see Table

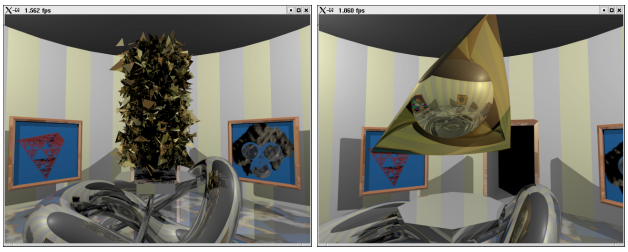

Figure 6: Unstructured motion in the BART museum: Up to 64,000 triangles are moving incoherently through the museum. Note how the triangles reflect the entire environment.

5.4 Outdoor Terrain

The Terrain scene has been specially designed to stress scalability with a large number of instances and triangles. It contains 661 instances of 2 different trees, which correspond to more than 10 million triangles after instantiation. A point light source creates highly detailed shadows from the leaves (see Figure

The large number of instances results in construction times for the top-level BSP of up to 4 ms per frame. This cost — together with the transmission cost for updating all 661 instance matrices on all clients — slightly limits the scalability for a large number of instances and clients (see Table

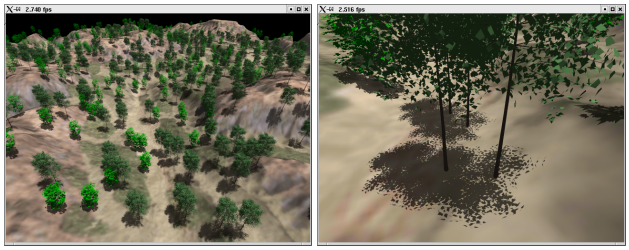

Figure 7: Terrain scene with up to 1000 instances of 2 kinds of complex trees (661 instances in the shown configuration, as some trees have been interactively moved off the terrain). Without instantiation, the scene would consist of roughly 10 million triangles. a.) Overview of the whole scene, b.) Detailed shadows from the sun, cast by the leaves onto both floor and other leaves.

6 Discussion

The above scenes have been chosen to stress different aspects of our dynamic ray tracing approach. Together with the terrain experiment, our test scenes contain a strong variation of parameters – from 5 to 661 instances, from a few thousand to several million triangles, from simple shading to lots of shadows and reflections, and from hierarchical animation to unstructured motion of thousands of triangles (for an overview, see Figure

Transformation Cost For mainly hierarchical animation, the core idea of our method was to trade the cost to reconstruct the data structure for the cost to transform the rays to the local coordinate system of each object. This implies that every ray intersecting an object has to be transformed via matrix-vector multiplications for both ray origin and direction (for every object encountered), potentially resulting in several matrix operations per ray. As our system shoots up to a million rays per second on an AthlonMP 1800+ CPU, this can amount to hundreds of thousands of matrix-vector multiplications. For example, the *terrain* and *robots* scenes at 640×480 pixels require 1.6 and 1 million matrix ops, respectively. Furthermore, additional transformations are often required during shading, e.g. by transforming the shading normal or for calculating procedural noise in the local coordinate system.

However, the cost for these transformations in practice is rather small. Even for a straight-forward C-code implementation, a matrix-vector operation costs only 23 cycles on an AMD AthlonMP CPU, which is rather small compared to the cost for tracing a ray (which is in the order of several thousand cycles). Using fast SSE code [Int 2002] reduces this cost even further, and makes the transformation overhead almost negligible.

Unstructured Motion As could be expected, the Museum scene has shown that unstructured motion remains costly for ray tracing. A moderate number of a few thousand independently moving triangles can easily be supported, but larger numbers still lead to high reconstruction times for the respective objects (see Table

To support such scenes, algorithms for faster reconstruction of dynamic objects have to be developed. Note that our method could also be combined with Reinhards approach [Reinhard et al. 2000] by using his method only for the unstructured objects. Even then, lots of unstructured motion would still create a performance problem due to the need to send all triangle updates to the clients. This is not a limitation of our specific method, but would be similar for any kind of algorithm in a distributed environment.

Bounding Volume Overlap One of the stress cases identified in [Lext et al. 2000] was *Bounding Volume Overlap*. In fact, this does creates some overhead, as in the overlap area of two objects, *both* objects have to be intersected sequentially by a ray – even a successful intersection in the first object may be invalidated by a closer one in the other object.

Even though it is easy to construct scenarios where this would lead to excessive overhead, it is rarely significant in practice. In fact, bounding volume overlap *does* happen in *all* our test cases, but has never shown to pose a major performance problem. In fact, overlapping objects are exactly what happens all the time with bounding volume hierarchies (BVHs) [Haines 1991], which have also proven to work rather well in practice.

Teapot-in-a-Stadium Problem The teapot-in-a-stadium problem is handled very well by our method: BSPs automatically adapt to varying object density in a scene [Havran 2001]. This is true for both object and toplevel BSPs. In fact, our method can even increase performance for such cases: If the 'teapot' is contained in a separate object, the shape of the 'stadium' BSP is usually much better, as there is no need any more for several BSP subdivisions to tightly enclose the teapot.

Over-Estimation of Object Bounds Building the top-level BSP requires an estimate of the bounding box of each instance in world coordinates. As transforming each individual vertex would be too costly, we conservatively estimate these bounds based on the transformed bounding box of the original object.

This somewhat over-estimates the correct bounds and thus results in some overhead: During top-level BSP traversal, a ray may be intersected with an object that it would not have intersected otherwise. However, this overhead is restricted to only transforming and clipping the ray: After transformation to the local coordinate system, such a ray is first clipped against the correct bounding box, and may immediately be discarded without further traversal.

Scalability with the number of Instances Apart from unstructured motion, the main cost of our method results from the need to recompute the top-level BSP tree. As such, a large number of instances is expensive, as can be seen in the Terrain scene. Thus, the number of instances should be minimized in order to achieve optimal performance. Usually, it is better to use a few, large objects instead of many small ones. All static triangles in a scene should best be stored in a single object, instead of using multiple objects. Still, even the thousand complex instances can be rendered interactively, and toplevel reconstruction has not yet proven a real limitation in any practical application. For moderate numbers of objects, toplevel reconstruction is virtually negligible.

On the other hand, supporting instantiation (i.e. using exactly the same object multiple times in the same frame) is a valuable feature of our method, as this allows to render complex environments very efficiently: With instantiation, memory is required only for storing the two original trees of the terrain scene and the top-level BSP, allowing to render even the several million triangles with a small memory footprint. For OpenGL rendering, all triangles would still need to be handled individually by the graphics hardware even when using display lists.

Scalability in a Distributed Environment As can be seen by the experiments in Section

In the terrain scene, using 16 clients would require to send 676 Kb[2] per frame simply for updating the 661 transformation matrices on the clients. Though this data can be sent in a compressed form, load balancing and client/server communication further adds to the network bandwidth. Without broadcast/multicast functionality on the network, the server bandwidth increases linearly with the number of clients. For many clients and lots of updated data, this creates a bandwidth bottleneck on the server, and severely limits the scalability (see Table

Total Overhead In order to estimate the total overhead of our method, we have compared several scenes in both a static and dynamic configuration. As there are no static equivalents for the BART benchmarks, we have taken several of our static test scenes, and have modified them in a way that they can be rendered in both a static configuration with all triangles in a single, static BSP tree, as well as in a dynamic configuration, where triangles are grouped into different objects that could then be moved dynamically.

For the scenes that are available in both static and dynamic configurations, we find that our method typically creates an overhead of only about 10 to 20 percent. We believe this overhead to be a reasonable price for the added flexibility gained through supporting dynamic, and fully interactive scenes.

7 Conclusions

In this paper, we have presented a simple and practical method for integrating support for dynamic scenes into an interactive distributed ray tracing engine. Our method can handle a large variety of dynamic scenes, including all the BART benchmark scenes (see Figure

For unstructured motion, our method still incurs a high update cost per frame, that makes it infeasible for a large number of incoherently moving triangles. For a moderate amount of unstructured motion however (in the order of a few thousand incoherently moving triangles), it is well applicable and results in frame rates of several frames per second at video resolution.

For mostly hierarchical animation — as used in almost all of todays scene graph libraries — our method is highly efficient and achieves interactive performance even for highly complex models with hundreds of instances, and with millions of triangles per object [Wald et al. 2002a].

With our proposed method, we have been successfully able to interactively ray trace all the dynamic scenes we have encountered so far. To our knowledge, this is the first time that it was possible to interactively ray trace the BART benchmark suite at all. With the unique advantages of ray tracing — now combined with the flexibility to handle dynamic environments — we believe that realtime ray tracing is a significant step closer to be a viable alternative to triangle rasterization for future interactive 3D graphics.

8 Future Work

Currently, the main remaining scalability bottleneck lies in communicating all scene updates to all clients, making the total bandwidth

[2] 661 instances $\times$ 16 clients $\times (4 \times 4)$ floats

linear in the number of clients. Thus, future work will investigate to use network broadcast/multicast to communicate the scene updates. As almost all of the updated data is the same for every client, this should effectively remove the network bottleneck.

On the clients, the main bottleneck is the cost for reconstructing objects under unstructured motion. This could be improved by designing specialized algorithms for cases where motion is spatially limited in some form, such as for skinning, predefined animations, or for keyframe-interpolation.

A VRML97 implementation based on our system is already up and running [Wald et al. 2003], and future work will also include interfacing to other scenegraph libraries such as OpenInventor. Finally, it is an obvious next step to integrate our techniques into a hardware ray tracing architecture such as the SaarCOR architecture [Schmittler et al. 2002].

Acknowledgements

This project has been supported by Intel Corp.

References

APPEL, A. 1968. Some Techniques for Shading Machine Renderings of Solids. *SJCC*, 27–45.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2003. A Scalable Approach to Interactive Global Illumination. to be published at Eurographics 2003.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *Proceedings of Graphics Hardware 2002*, Eurographics Association, 37–46.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed Ray Tracing. In *ACM SIGGRAPH Computer Graphics*, vol. 18, 137–144.

DEMARLE, D. E., PARKER, S., HARTNER, M., GRIBBLE, C., AND HANSEN, C. 2003. Distributed Interactive Ray Tracing for Large Volume Visualization. (submitted for publication).

DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*. Available at http://www.openrt.de.

GLASSNER, A. 1988. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications 8*, 2, 60–70.

GLASSNER, A. 1989. *An Introduction to Raytracing*. Academic Press.

GRÖLLER, E., AND PURGATHOFER, W. 1991. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics '91*, Elsevier Science Publishers, 103–113.

HAINES, E. 1991. Efficiency Improvements for Hierarchy Traversal in Ray Tracing. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 267–272.

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University.

INTEL CORP. 2002. *Intel Pentium III Streaming SIMD Extensions*. http://developer.intel.com/vtune/cbts/simd.htm.

LEXT, J., AND AKENINE-MOELLER, T. 2001. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pp. 311–318.

LEXT, J., ASSARSSON, U., AND MOELLER, T. 2000. BART: A Benchmark for Animated Ray Tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Goeteborg, Sweden, May. Available at http://www.ce.chalmers.se/BART/.

MUUSS, M. J. 1995. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*.

NEIDER, J., DAVIS, T., AND WOO, M. 1993. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusetts, U.S.A.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. P. 1998. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, 233–238.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. P. 1999. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics (I3D)*, 119–126.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics 21*, 3, 703–712. (Proceedings of SIGGRAPH 2002).

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, 299–306.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, 27–36.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–164. (Proceedings of Eurographics 2001).

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive Distributed Ray Tracing of Highly Complex Models. *Rendering Techniques 2001*, 274–285. (Proceedings of the 12th Eurographics Workshop on Rendering).

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2002. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Tech. rep., Saarland University. http://graphics.cs.uni-sb.de/Publications.

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, 15–24. (Proceedings of the 13th Eurographics Workshop on Rendering).

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*.

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *CACM 23*, 6 (June), 343–349.

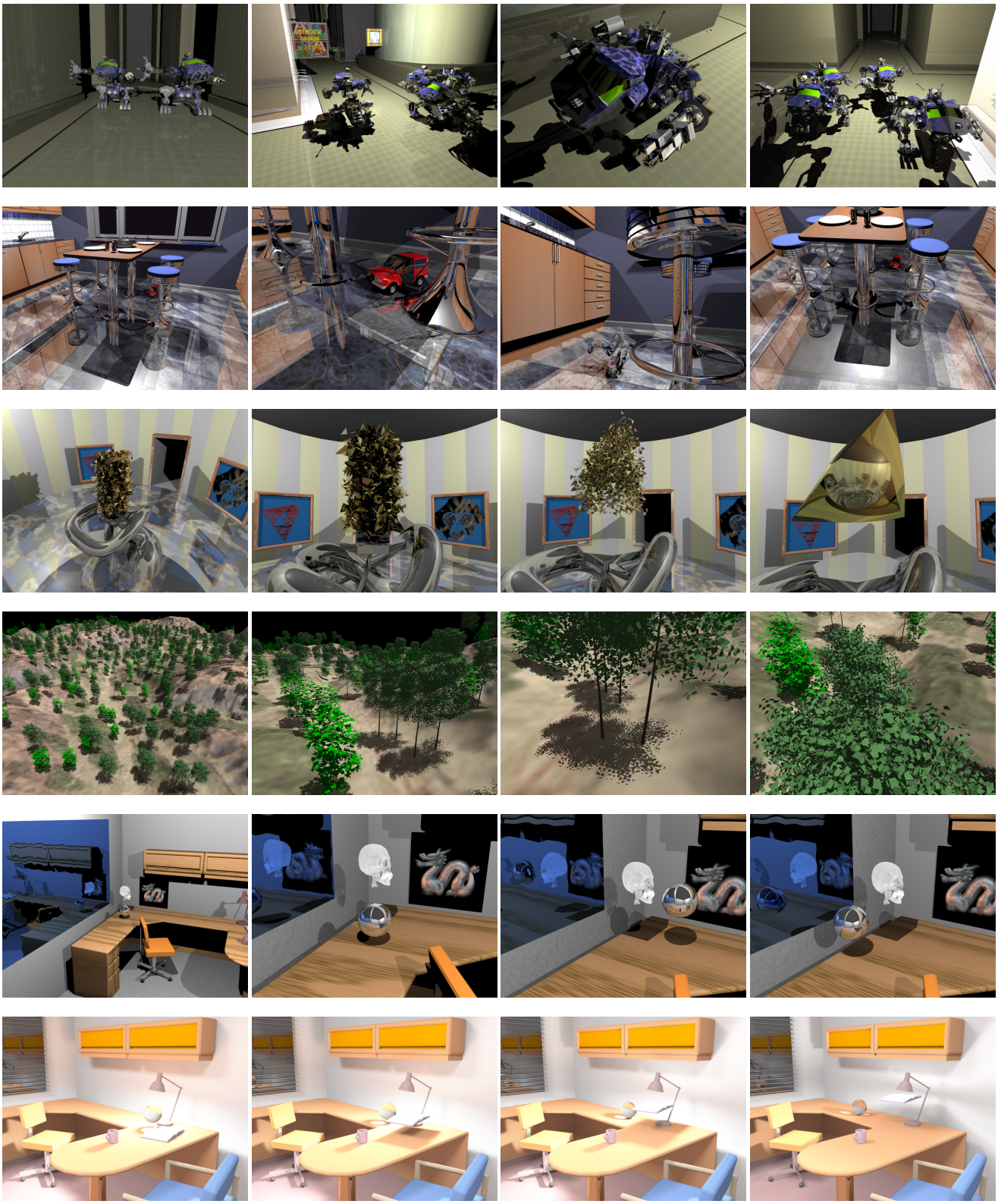

Figure 8: Several example frames from some of our dynamic scenes. From top to bottom: a.) "BART robots" contains roughly 100,000 triangles in 161 moving objects, b.) "BART kitchen", c.) "BART museum" with unstructured motion of several thousand triangles. Note how the entire museum reflects in these triangles. d.) The "terrain" scene uses up to 661 instances of 2 trees, would contain several million triangles without instantiation, and also calculates detailed shadows. e.) The "office" scene in a typical ray tracing configuration, demonstrating that the method works fully automatically and completely transparently to the shader. f.) Office with interactive global illumination.