BART: A Benchmark for Animated Ray Tracing

Jonas Lext, Ulf Assarsson, and Tomas Möller

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{lext, uffe, tomasm}@ce.chalmers.se

Technical Report 00-14

May 2000

Abstract

Due to the advent of ray tracing at interactive speeds and because there is an absence of a way to measure and compare performance and quality of ray traced scenes that are animated, we present an organized way to do this objectively and accurately in this proposal for *BART: A Benchmark for Animated Ray Tracing*. This is a suite of test scenes, placed in the public domain, designed to stress ray tracing algorithms, where both the camera and objects are animated parametrically. Equally important, BART is also a set of rules on how to measure performance of the rendering. We also propose how to measure and compare the error in the rendered images when approximating algorithms are used.

1 Introduction

In order to measure performance, benchmarks are needed, as they allow people to compare performance in a more accurate and objective way. A benchmark is often a suite of test cases or executables together with a specified procedure of how to report performance with that benchmark. There is a need for benchmarks in computer graphics in a variety of different areas such as radiosity, global illumination, collision detection, animation, image-based rendering, polygon rendering, and all other areas where you need to be able to measure and compare performance.

Currently, there are only benchmarks in a few of these areas, and our effort is an attempt to bridge that gap. We saw the need for *BART: A Benchmark for Animated Ray Tracing*, because there currently is no benchmark for this, and because at least two groups [1, 2] have been ray tracing fairly complex and realistic scenes at interactive speeds (by "interactive speeds", we mean any rate above one frame per second). Another reason is because acceleration data structures for animated ray tracing has not been studied much, but probably will be in the future. The main contributions of BART are: a set of parametrically animated test scenes that are designed to stress ray tracing algorithms and that are easy for anyone to use, and a set of reliable performance measurements that allows the user of BART to compare performance of different ray tracing algorithms. For approximating algorithms (that is, algorithms that might not produce entirely correct values for each pixel but rather approximate values), we also define how to measure the quality of the approximated images.

The organization of this paper is as follows. In the next section we will briefly review the different, currently available benchmarks for graphics. Section 3 identifies events and scenarios that potentially stress different existing ray tracing algorithms. These are then implemented in the test scenes, described in section

4. In section 5, we discuss and motivate the measurements that we propose should be reported when the benchmark is used. Short implementation notes follows in section 6. Finally, we conclude and present some future work.

2 Related Work

For graphics hardware vendors and implementors there are many different benchmarks. The *Standard Performance Evaluation Corporation* (SPEC) has a subgroup called *Graphics Performance Characterization Group* (GPC) [3] which has a set of graphics benchmarks. Their *SPECglperf* is a way to measure the performance of rendering low-level primitives, such as points, lines, triangles etc. with OpenGL. In contrast, GPC's *SPECviewperf* for OpenGL is a way to measure the rendering performance of a set of real models. Both *SPECglperf* and *SPECviewperf* are targeted towards vendors and implementors of graphics hardware as it is not allowed to alter the program (e.g., implement an occlusion algorithm) to make the execution more efficient. GPC also have benchmarks for a few commercial programs, but these require fully licensed versions of the programs and are thus not available for everyone. For PC's there are also several other benchmarks for measuring the performance of the graphics subsystem: *3D WinBench*, *3DMark*, and *QuakeIII*, to mention a few.

Would it be possible to use the scenes from these benchmarks, which are targeted towards graphics hardware systems, for evaluating the performance of ray tracing algorithms? We believe that this would be unsuitable, or even impossible in some cases. The scenes in these benchmarks may be surrounded by license agreements, which limit their use for other purposes then the original ones. For example, a user of the SPEC benchmarks must report measured results strictly in accordance with the rules published by SPEC. As different performance parameters might be interesting for ray tracing algorithms and graphics hardware, this would rule out the use of SPEC models for comparing the performance of ray tracing algorithms. Naturally, these benchmarks are also constructed for the single purpose of testing graphics hardware specific features, or specific applications (e.g., CAD applications or game engines) well suited for graphics hardware. For example, CAD applications often deal with single static objects hanging in free space. However, to really stress ray tracing algorithms one would want scenes representing complete environments. Another example is the Quake scenes, which are optimized to make the rendering as fast as possible using the Quake game engine; the scene description is in a binary format and contains an acceleration data structure hard coded into it. On the contrary, the scenes presented in our benchmark are specifically designed with the intent of stressing ray tracing algorithms, and are described using a simple and readable format for greater flexibility.

In [4], a framework for a performance evaluation system for real-time rendering algorithms in Virtual Reality is presented. This benchmark is also targeted against polygon rendering hardware, and there seems not to be any scenes available on the Internet.

Worth a mention is also Pete Shirley's eleven scenes for testing radiosity algorithms: `http://radsite.lbl.gov/mgf/scenes.html`. These were for benchmarking quality instead of speed and is not animated.

For ray tracing, there is, to our knowledge, only one recognized benchmark and it is called *Standard Procedural Database* (SPD) by Haines [5] from 1987. The SPD is targeted for ray tracing algorithms for single static images. Other drawbacks are that the images are not necessarily realistic, and that almost the entire geometry of each scene is located in the view frustum of the camera. This is not normally the case. While perfectly valid and widely used for over a decade, progress in computer architecture and algorithms has advanced beyond that of what SPD first was intended for. For example, there are at least two projects that have ray traced fairly complex and realistic images at interactive speeds [1, 2].

Formella and Gill [6] present an alternative benchmark measuring the performance of tracing rays in a ray tracer. That benchmark is also for single static images, and their scenes consists of a cube and a set of distributed spheres or parallelograms. To our knowledge, this benchmark has not been widely used.

A vast number of ray tracing algorithms and variants have been developed over the years. In this paper, we will only reference a few of those that we think apply.

3 Potential Stresses

In order to construct a benchmark with relatively long lifetime, we first set out to identify what stresses existing ray tracing algorithms and thus decreases performance. The goal was then to implement each of these potential stresses into the benchmark. The following scenarios or events tend to stress different efficiency schemes for ray tracing:

1. Hierarchical animation using translation, rotation, and scaling.
2. Unorganized animation of objects (i.e., not just combinations of translation, rotation, and scaling).
3. "Teapot in the stadium" problem.
4. Low frame-to-frame coherency.
5. Large working-set sizes.
6. Overlap of bounding volumes or overlap of their projections.
7. Changing object distribution.

Below we describe each of the items above, why we believe that they should in some way be incorporated in a benchmark for ray tracing, and which ray tracing acceleration scheme(s) in particular they stress.

Stress 1 : Hierarchical animation

During modeling, the easiest and most natural way is to model each object in its own frame of reference. Building a scene from such objects, a hierarchical representation of it offers a simple and flexible way to express how objects are positioned, oriented and how they move relative to each other.

However, in an animated or interactive ray tracer, the hierarchy of local coordinate systems and hierarchical animation may also stress the scene rendering. To investigate intersection between a ray and an object in the scene, both must be expressed in the same frame of reference. A straight forward solution would be to continuously transform rays to the coordinate systems of the individual objects. However, transformation involves matrix multiplication, which is a very floating point intensive operation as compared to e.g., checking a bounding volume (BV) for intersection, thus stressing the rendering time. A solution to this problem would be to transform the camera and all the objects in the scene to the global coordinate system once, before the spawning of rays. However, this may require a lot of extra memory; an advantage with keeping the hierarchy is that if a complex object appears multiple times in a scene, memory can be saved by keeping only one copy of the object model in memory, and referencing it using pointers in the hierarchy.

When adding animation to the scene, whole or parts of the acceleration data structures will most likely have to be reconstructed between frames. Depending on the amount of changes in the scene, this could seriously stress the reconstruction phase when using octrees [7], uniform grids [8], recursive grids [9], hierarchical grids [10, 11, 12], BSP trees [13, 14] and bounding volume hierarchies (BVHs) [15, 16]. In our benchmark, we have excluded the animation of light sources, because it simplified our animations, and the same stress can be achieved by animated objects. Therefore, this is also a serious stress for light buffers [17].

Stress 2 : Unorganized animation

In order to cope with transforms, ray tracers often transform the ray with the inverse transform instead of transforming the object itself and its efficiency data structures. Thus, for some acceleration schemes (e.g., a static grid or a BVH around an object), we do not have to rebuild the efficiency data structures

each frame. This is easily done for translations, rotations, and scalings, but often other kinds of "less organized" animation is used, and then this approach cannot be used. Since, to our knowledge, all currently available types of acceleration schemes for ray tracing (e.g., [7, 8, 9, 12, 14, 16]) must rebuild their efficiency structures for such animations, this will be a serious stress on all ray tracing algorithms.

Stress 3 : Teapot in the stadium

The "teapot in the stadium" problem [18] refers to when a small detailed object (teapot) is located in a relatively large surrounding object (stadium). This tends to stress uniform grid-based [8] algorithms and octree-based schemes [7], because the uniform grid has finite sized voxels and because the octree has finite depth, because the teapot will be located in one or only a few voxels or octree nodes. For example, if the viewer is looking at the teapot such that it covers most of the screen, then only one or a few voxels/octree nodes will be traversed and each will contain many primitives, which thus degrades performance enourmously.

Stress 4 : Low frame-to-frame coherency

Situations where the frame-to-frame coherency is low tend to stress reprojection algorithms [19, 20, 21, 22], since those use information from previous frames. If the difference between two frames is too big then the performance of such algorithms is worse or the quality of the rendered images gets worse. Similarly, frameless rendering techniques [23, 24] will produce images of poorer quality when the frame-to-frame coherency is low. In this paper, we use the name *approximating algorithms* for all algorithms that may generate images that are not entirely correct for every frame they generate.

Stress 5 : Large working-set sizes

An important problem in computer architecture is the increasing gap between the computational speed of the processor and the speed with which the memory system can feed the processor with data. The conventional solution to this problem is using a cache hierarchy between the processor and memory, and relying on spatial and temporal coherence in the data access patterns [25]. Current computer architectures are usually equipped with two levels of caches (L1 and L2) between the processor and main memory.

Typical sizes for the L1 and L2 caches range from 16–64 kB and 128 kB–2 MB, respectively. This means that the sizes of the scenes of BART should be significantly larger than the cache sizes found in contemporary microprocessors. However, a scene size much larger than a cache size does not necessarily imply a problem. Only the size of the scene data actually used when rendering an individual frame in an animation, a so called working set [26], is of importance. For example, if the working set of one frame in an animation is larger than the L2 cache, this will most likely expunge data that could have been reused in the following frame, raising the L2 cache miss ratio. This would be the case if the majority of the primitives of a large scene is potentially visible in one single frame. Note that this also could happen indirectly; due to reflecting objects, much more of the scene primitives might be used than the ones that actually occurs in the view frustum. Thus, the reflection of rays most likely increase the working set encountered in a frame.

The same situation applies for the L1-cache. Here the working set might be a set of primitives visible during a set of consecutive scan lines. If the number of primitives visible is much larger than the L1 cache, the L1 cache miss ratio will be high. Therefore, it might be desirable to develop algorithms that tries to exploit data coherency as much as possible, to increase the achieved performance of the memory system.

Stress 6 : Bounding volume overlap

This problem might occur when a number of BVs or local grids overlap (perhaps due to animation). If a ray penetrates all of the overlapping volumes, it is not necessarily the first one reached that contains the closest

object intersection. Therefore, a number of BVs or grids may have to be traversed before encountering the true intersection point.

The BVs or grids do not necessarily have to overlap to get this effect. If a ray intersects several BVs or grids, and these contain a lot of empty space around the object they cover, again, a number of rays may have to traverse a large number of BVs or grids before an intersection is found. If the number of rays that encounter these situations is large enough, this could be a potential stress.

Stress 7 : Changing object distribution

Due to animation, the distribution of objects in the scene might change over time. This might stress ray tracing algorithms regarding which efficiency data structure should be used. For example, one static grid covering the whole scene might work well if there is an even distribution of objects in the scene. However, hierarchical grids or recursive grids are probably a better choice for an unbalanced distribution of objects. Therefore, if the distribution of objects in the scene changes over time, the most suitable data structure may also change.

A solution might be to recreate or update the data structure each frame. Cazals et al. presents an algorithm that automatically creates an efficient data structure for ray tracing given a particular distribution of geometry as input [11]. It outputs a hierarchy of uniform grids with cell sizes and depth optimized for the given scene.

4 Animated Test Scenes

Here we present the three test scenes, called kitchen, robots, and museum, of BART. All test scenes are parametrically animated, by which we mean that the user of BART easily can vary the number of frames in an animation. One use of this feature is to try how well an algorithm can handle different levels of frame-to-frame coherency by simply decreasing the number of frames to test lower frame-to-frame coherency.

4.1 Kitchen

The main subject in this scene is a toy car moving around in a kitchen. The camera, initially overlooking the scene from one of the upper corners of the room, descends to meet the car and then follows it on its path through the room. Figure 3 shows images taken from this scene.

The toy car is hierarchically animated using translations, rotations, and scalings (stress 1). The hierarchy is at most three levels deep, and the scaling occurs at the end of the animation when the car crashes into a cupboard, and is thus scaled along the current driving direction.

The kitchen scene should be subject to the teapot-in-the-stadium problem (stress 3) : the walls, roof and floor is modeled using only a few large triangles. However, the scene also contains quite a lot of complex objects, which vary in size from small (e.g., door knob) to medium (e.g., chairs).

Low frame-to-frame coherency (stress 4) should be likely to appear at two different instances during the kitchen animation. At one instant the camera is almost still. The toy car passes quickly in front of the camera, abruptly increasing the visible number of primitives during a small number of consecutive frames. A similar effect is achieved when the camera moves rapidly, very close to the table edge; during one frame, the table edge obscures the whole camera view and only a few triangles are visible. In the next frame, the items on the table come into view. Therefore, the number of visible primitives differ drastically between the two frames, and thus the frame-to-frame coherence is very low.

The kitchen scene contains six point light sources and is modeled using 110,561 polygons. These require a total of over 15 MB[1] of memory to store. Furthermore, eight texture maps are used, ranging in size from

[1]The memory usage is 3 vertices and 3 normals per triangle patch. Each normal or vertex occupies 3 doubles (8 bytes per double). This gives $(3+3)*3*8*110,559 \approx 15$ MB.

96 kB to 3.1 MB and requiring 7.25 MB of memory in total. In all, the kitchen scene might require more than 22 MB of memory to store. Therefore, the complex and highly reflective kitchen furniture should be able to stress the memory hierarchy system (stress 5) on most contemporary processors.

Finally, when the toy car moves under the table and around the chairs in the animation, there should be the possibility of bounding volume overlap (stress 6). However, this depends on the acceleration data structure used, and how it is applied to the object in the scene.

4.2 Robots

The robot scene consists of ten animated warrior robots and a static downtown environment with skyscrapes with a total of 71,708 polygons. Each robot consists of 6,249 polygons with 18 moving parts. The city is built of 9,218 polygons. One light source is used as the sun. Background lighting is implemented by using an ambient contribution. Snapshots from the robots test scene is shown in figure 4.

The robots are spread out in the city at start of the animation and walk down the streets to finally gather in the middle of the scene. We have implemented stress 7 in this scene by letting the distribution of robots change drastically from fairly balanced at start of the animation to highly unbalanced at the end. This also gives the ”Teapot in the stadium”-stress (stress 3). The hierarchical animation of the robot ensures stress 1.

For a few seconds at the end of the animation the camera is in a static position, looking down at all robots with only a few of them moving. This gives an opportunity for algorithms to exploit frame-to-frame coherency.

Stress number 6 is implemented by the moving parts of the robots since spatial data structures will overlap in the joints. Furthermore, data structure overlap might occur between the robots and the city. Unless the spatial data structures for the parts of the robots are very tight fitting, overlap will also occur between different robots when they are clustered at the end of the animation and when several robots, marching down the street, are viewed head on.

Each robot of 6,249 polygons will fit in most current L2 caches, but all ten robots together probably will not. Therefore, in the frames were many or all robots are visible at the same time, stress 5 should occur. On the other hand, since all ten robots are identical except for the positions and rotations of the parts, this could be utilized to save memory. Only the information about the transforms must be handled separately. In this way large scenes may still fit in the caches, which can be essential for speed.

4.3 Museum

While the kitchen and robots test scenes include hierarchical animation of objects and animation of the camera, this test scene’s goal is to stress the building of the efficiency data structures, which typically is done as a preprocess before ray tracing an image. In order to create such a stress (stress 2), we have included a very simple manner to animate objects such that every type of efficiency data structure that we know of must be rebuilt each frame in order to obtain good performance. For this kind of animation, a triangle patch (which is a triangle with normals at each vertex) is interpolated into another triangle patch.

Therefore, the museum scene consists of a small room (in a museum). The main subject in this room is an animated piece of abstract art, which consists of a number of triangle patches that are interpolated from one constellation into four others. For example, in one of these constellations, the triangle patches are uniformly distributed, and randomly rotated inside a cylinder, and at a later time, they form a sphere. This test scene uses two light sources, and snapshots from it is shown in figure 5.

Another potential stress in this test scene is that the number of primitives inside the view frustum is approximately the same for the first 90% of the frames in this animation. For the remaining 10% of the frames, the number of primitives inside the view frustum changes drastically to only a few large polygons. This will be a challenge to “constant frame rate” algorithms since the workload changes drastically.

If the animated piece of art in the middle of this scene is excluded, then this scene has 10,143 polygons and 8 cones. There are five 512×512 RGB textures, which together occupy 3.75 MB of memory. In a

sense, this is a very small scene in terms of numbers of primitives. To be able to test scenes with different number of primitives, we provide different complexity levels of the animated piece of art in this scene. More specifically, there are 6 different versions, consisting of 2^{2k} animated triangle patches, where $k = 3, \ldots, 8$. This means that the lowest complexity level consists of 64 triangle patches and the highest $65,536$. All these triangle exist at five different times, i.e., they are interpolated into five different constellations. The memory usage for the highest complexity level is 45 MB. So, all in all, the highest complexity level of this scene occupies at least 50 MB. Due to the fact that the complex and highly reflective abstract art and pedestal are fully visible in the majority of the frames in this animation stress 5 might occur.

5 Performance Measurements

The main idea of using a benchmark is that people should be able to compare the performance of different algorithms in an objective way. To do that, the users of a benchmark must report the same performance measurements and in the same manner. In this section, we first summarize the parameters and the measurements that we propose the user of the benchmark should report, and then motivate and describe the non-obvious measurements and parameters.

5.1 Proposal of Measurement Report

In the table below, the parameters and some of the measurements that we propose should be reported are presented, together with actual numbers and parameters from a simple measurement. Please note that this should not be seen as a serious attempt to achieve good rendering time. The animated museum test scene was rendered using the publicly available Rayshade ray tracer [27] (adapted to read our file format), and we simply used a uniform grid with fixed size, which was rebuilt each frame.

Model:	museum	**Frames:**	300
Primitives:	11,175	**Complexity level:**	5
Resolution:	800×600	**Mode:**	interactive
Average frame time:	176.6 s	**Worst frame time:**	294 s
Deviation:	0.37	**Continuity (opt):**	0.18
Total time:	52,966.0 s	**Preprocessing time:**	0.0 s
Machine:	Sun UltraSparc 10, 333 MHz	**Memory:**	128 MB
Scene memory:	8.5 MB	**Efficiency memory:**	0.5 MB

Model is the name of the model, *Frames* is the number of frames in the animation, *Primitives* is the number of primitives in the scene, *Complexity level* is the complexity level number used in the scene, and *Resolution* is the resolution of the rendered images. See section 5.2.1 for a description of the *Mode* parameter. *Total time* is the time it takes to render all the frames, *Average frame time* is the time it takes to render a frame on average, and *Worst frame time* is the time it takes to render the frame in the animation that takes the most time. In an interactive, or near interactive, ray tracer, this parameter would be useful to get an absolute measure of the worst possible frame rate one can expect. *Preprocessing time* is the time preprocessing takes, which is done once before the rendering of the entire animation starts. Finally, see section 5.2.2 or a description of the *Deviation* and the *Continuity* measurements.

It is also reasonable and interesting to report what kind of machine has been used to render the images, how much memory it has, how much memory the scene occupies, and how much memory the efficiency data structures occupy.

In addition to the information in the table, a graph can be supplied that shows the total number of rays as a function of the frame number. Optionally, this graph could be divided into the following classes of rays: eye, reflection, refraction, and shadow rays.

Also, a rendering time diagram as a function of the frame number should be given. As an option, the rendering time can be divided into *shading time*, which is the amount of the rendering time that is spent on shading and lighting a frame, *visibility time*, which is the amount of the rendering time that is spent finding the intersection with the closest object, and *rebuild time* which is the time it takes to rebuild the efficiency data structures each frame. See figure 1 for example diagrams. Note that the rendering time is the sum

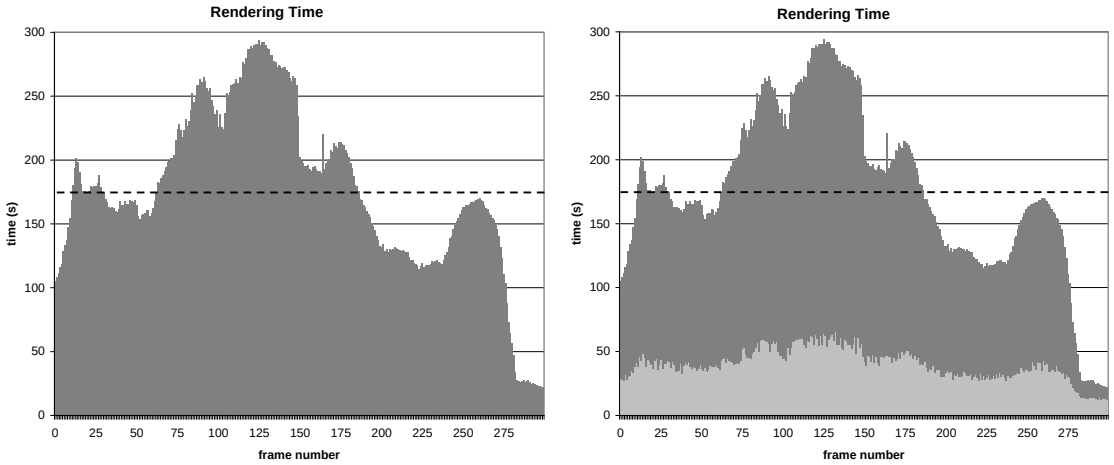

Figure 1: Rendering times as a function of frame number for the museum test scene. The 300 images of this animation was rendered using the publicly available Rayshade ray tracer [27], and the time was measured using the Unix time command. The left diagram shows only the total rendering time per frame, while the right shows what it might look like when it has been divided into shading time (light gray) and visibility time (dark gray). The dashed horizontal lines mark the average rendering time. The right diagram could also be divided into one more part, namely rebuild time, which is the time it takes to rebuild the efficiency data structures per frame. However, this was negligible in our test, and so does not show in the diagram.

of the visibility, the shading time, and the rebuild time. The main reason to include those is mostly for the algorithm developer's own sake; he/she can gain insight about where the bottleneck in his/hers algorithm lies, and where best to optimize.

If the user of BART is comparing his (new) algorithm against an old algorithm, it is recommended that a speed-up diagram of $t_{new}(k)/t_{old}(k)$ is given. Here, $t_{new}(k)$ is the time it took for the new algorithm to render frame k, and $t_{old}(k)$ is the time it took for the old algorithm to render frame k. For approximating algorithms, which may introduce errors in the rendered images, a PNSR diagram as a function of the frame number, and the APNSR should also be reported (see section 5.2.3).

5.2 Motivation

In this section, we present motivation for certain measurements that are non-obvious, and that we propose are reported when BART is used. Some of the measurements are optional.

5.2.1 Predetermined vs. Interactive Mode

The benchmark can be used in two different modes: *interactive* and *predetermined*. In *predetermined* mode the user of the benchmark is allowed to look into the future, that is, information about frames that have not been rendered is allowed to be retrieved. For example, assume that frame k is about to be rendered. In *predetermined* mode, the position and the orientation of the camera for, say, frame $k+1$, $k+2$, and $k+3$ may be investigated (in order to see if that information can be exploited for faster rendering).

Due to the difficulties in specifying a truly interactive animation (where things usually change due the the input from the user of the system) which gives the same amount of work each time the benchmark is used, we have included a "fake" interactive mode. In this mode, which is simply called *interactive*, the user is not allowed to look into the future, i.e., you are not allowed to retrieve information about frames that appear after the frame that is currently being rendered.

5.2.2 Deviation and Continuity

It is quite common to use the standard deviation as a measure on how much a set of samples deviates from the average of the samples. The formula for the standard deviation is

$$s = \sqrt{\frac{1}{n-1}\sum_{i=0}^{n-1}(t_i - t_{avg})^2} \tag{1}$$

where n is the number of frames is the scene, t_i is the time for frame i, t_{avg} is the average frame time. Instead of using s, we propose that the following is used instead

$$d = \frac{s}{t_{avg}} \tag{2}$$

which we call the *Deviation*. The reason to use d instead of s is that d is dimensionless, and because it is, at least theoretically, invariant of the average frame time. This means that d is the same if you run a renderer on machines with different performance.

The continuity measurement is computed as

$$\max_k(\text{abs}(t_k - t_{k+1}))/t_{avg}, \tag{3}$$

which thus is the maximum of the absolute value of the difference in rendering time between two subsequent frames divided by the average frame time.

Both these measurements are invariant of the average frame time. (This was experimentally verified by computing both the *Deviation* and the *Continuity* for a 167 MHz and a 333 MHz machine, and the observed values were very close. The *Deviation* measure differed by 2% and the *Continuity* by 5%.) This implies that a researcher can compare deviations and continuity just by reading another researcher's paper. In a perfect world, all measurement would be such, which would lessen the burden of the researcher. Instead the researcher has to implement algorithms in order to compare performance. Unfortunately, all timings, such as average frame time, total time, etc cannot be made such without losing their meaning.

We recommend that both *Deviation* and *Continuity* are reported. *Deviation* is reasonable to report because it reports a deviation globally, while *Continuity* is good because it catches frame-to-frame anomalies, which for frame rates higher than one per second is distracting to the human visual system. Therefore, if interactive rates are not achieved, then there is probably no use in reporting *Continuity*.

5.2.3 Approximating Algorithms

In order to render images rapidly or to maintain a constant frame rate, approximating techniques can be used. Examples include reprojection methods [19, 20, 21, 22] and frameless rendering techniques [23, 24].

More algorithms along this line are likely to be developed to meet the need for speed. However, errors may be introduced in the rendered images, and we therefore recommend that certain error values are reported when BART is used with such algorithms. To do this we assume that the user of BART renders a reference set of images of the animation without approximating techniques and with the highest possible quality (high anti-aliasing, high ray depth, etc), or at least states how the reference set was rendered. Due to differences in shading, anti-aliasing, etc, in different renders, we believe strongly that it is best that the user generates his/her own set of reference images. In image analysis and compression, it is common to measure differences between two images using the *peak signal to noise ratio* (PNSR), which in this case is a good way to measure the rendering error caused by the approximation. To compute that, we first define the *mean square error* (MSE) for RGB images as:

$$\mathrm{MSE} = \frac{1}{3wh}\sum_{x=0}^{w}\sum_{y=0}^{h}((\mathbf{a}(x,y)_r - \mathbf{c}(x,y)_r)^2 + (\mathbf{a}(x,y)_g - \mathbf{c}(x,y)_g)^2 + (\mathbf{a}(x,y)_b - \mathbf{c}(x,y)_b)^2) \quad (4)$$

where w is the width and h is the height of the rendered image measured in pixels, $\mathbf{a}(x,y)$ is the pixel at (x,y) of the approximated image, and $\mathbf{c}(x,y)$ is the pixel at (x,y) of the correct (reference) image of the same frame. The RGB components of a pixels are assumed to be in the interval $[0,1]$, and thus that $\mathrm{MSE} \in [0,1]$. The individual color components are accessed as $\mathbf{c}(x,y)_c$, where c could be any of r,g,b. Note that the squares in Equation 4 penalize large differences in the individual pixels which usually is more distracting. Given the MSE, the PNSR is:

$$\mathrm{PNSR} = 10\log_{10}(1.0^2/\mathrm{MSE}) = -10\log_{10}(\mathrm{MSE}) \quad (5)$$

Note that the lower PNSR the worse approximation. There may be some other measure that rewards rendered images that are perceptually of better quality. For example, if an image is shifted one pixel to the left, then PNSR will be low, even though the image is "perceptually pleasing". Unfortunately, we know of no such measure that takes every possible aspect into account. The average of the PNSR, denoted APNSR, is computed as the PNSR of the average of the MSE of all images in an animation:

$$\mathrm{APNSR} = -10\log_{10}\left(\frac{1}{n}\sum_i \mathrm{MSE}_i\right) \quad (6)$$

where n is the number of images in the animation, and MSE_i is the mean square error for image i. Note that a PNSR of 0.0 means the maximum possible error, which may occur when all pixels in an image are black when they in fact should be white. A totally correct image implies a PNSR value of ∞, which is unreasonable to draw in a diagram. Assume therefore that we render an image of 1280×1024 pixels with 8 bits per color component and that exactly one pixel has an error in the least significant bit in one color component. The PNSR is then 114, and represents a negligible error. Therefore we say that $\mathrm{PNSR} \in [0,120]$, where we assume that every value above 120 represents an error-free image. The value 120 may be adjusted for other resolutions.

When approximating algorithms are used, we recommend that APNSR and a diagram of PNSR_i as a function of the frame number i, where $i \in [startframe, stopframe]$ are reported. The reason for this is to be able to see how good the approximation really is, which is something that has previously been neglected at large in the computer graphics community. Thus, when comparing algorithms in a fair way, these are important measurements. See Figure 2 for an example of a PNSR diagram.

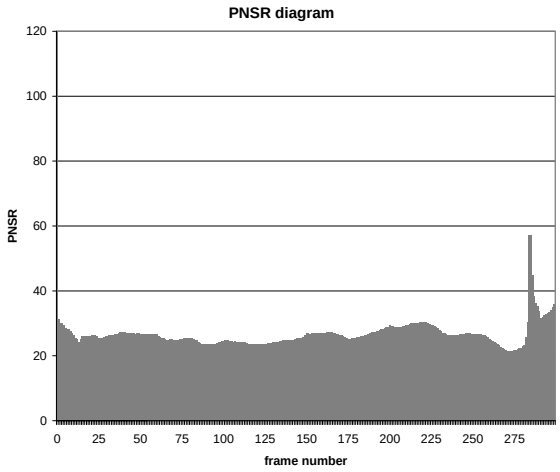

Figure 2: PNSR diagram for rendering of the museum scene. The 300 images of this animation was rendered using the publicly available Rayshade ray tracer [27], and a PNSR-value was calculated for each frame, reusing every fourth pixel from the previous frame. This made for faster rendering, but worse image quality as can be seen in this diagram. Note that only the first image has $\mathrm{PNSR} = 120$, which means that this is the only image that was 100% correct.

6 Implementation Notes

We have placed the three models of BART into the public domain, so that they are free for everyone to use. We wanted to keep the file format as simple as possible and yet reasonably flexible. Therefore the NFF format [28] was enhanced in several ways to handle animated scenes. The new format, called *Animated File Format*, is described on our web site (`http://www.ce.chalmers.se/BART/`). Kochanek-Bartels splines [29] were used for animating both the viewer and the objects. This included rotations, translations, and scalings organized hierarchically. Due to this, there is a notation of objects in AFF, since an object usually is grouped under a transform (static or animated). We used Eberly's implementation [30]. This approach was chosen as it avoids file reads between frames, which should be avoided for interactive programs and because it was simple.

In addition to the files, we provide the following, in order for the user of BART to quickly use it:

- A simple parser of the file format (written in C), which is easy to add to a renderer (it is our experience that it takes less than half a day).
- Routines for spline interpolation.
- C-code for reading texture files.
- MPEGs of the animations (for comparison purposes).

7 Conclusion and Future Work

We have provided a proposal for a suite of test scenes for use when comparing and measuring performance of ray tracing algorithms that can handle animated scenes. These scenes have been designed to stress the performance of acceleration schemes for ray tracing. We also provide a list of measures that we recommend

should be reported when comparing, testing, and measuring performance and possibly image quality of different algorithms. All of this is important in order to find good algorithms for rendering animated scenes.

It is difficult for a benchmark to cover every aspect, and this is just a first step, that we hope will be extended in the future. Possible extensions that would be interesting to include would be some kind of parametric patches (e.g., NURBS, B-spline surfaces or Bézier triangles) and subdivision surfaces. It would also be nice to extend BART to include a large architectural building and a complex outdoor scene. It would also be good to have extremely large scenes in BART, say a few magnitudes larger than they currently are. Furthermore, the interactive ray tracers discussed in [1, 2] require powerful parallel computer systems to achieve interactive rendering speeds. As parallel computer systems most likely will be used in the future, special scenes could be added that stresses multi-processor algorithms for ray tracing. Over time we expect that what should be reported with BART will evolve; some measurements may not be needed, and some new ones may appear.

The goal for users of BART is real-time ray tracing (i.e., > 15 fps) with a constant frame rate, and it will be very exciting to see when this will happen and what kind of algorithms will be used. Finally, we encourage everyone to participate in the usage and the development of BART, which we hope and expect to grow and evolve over time.

References

[1] Muuss, Michael John, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models", In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[2] 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.

[3] *The Graphics Performance Characterization Group*, `http://www.specbench.org/gpc/`.

[4] Ping Yuan, Mark Green and Rynson Lau, "A Framework of Performance Evaluation of Real-time Rendering Algorithms in Virtual Reality", ACM Symposium on Virtual Reality Software and Technology'97, Sept. 1997, Switzerland.

[5] Haines, Eric, "A Proposal for Standard Graphics Environments", *IEEE Computer Graphics and Applications*, vol. 7, no. 11, pp. 3–5, November 1987.

[6] Formella, Arno, and Christian Gill, "Ray Tracing: A Quantitative Analysis and a New Practical Algorithm", *The Visual Computer*, vol. 11, no. 9, pp. 465–476, May 1995.

[7] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15–22, October 1984.

[8] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata, "Arts: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications*, pp. 16–26, April 1986.

[9] Jevans, David, and Brian Wyvill, "Adaptive Voxel Subdivision for Ray Tracing", *Graphics Interface '89*, pp. 164–172, June 1989.

[10] John M. Snyder and Alan H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4), pp. 119-128, July 1987

[11] 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.

[12] 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.

[13] Jansen, F.W, “Data Structures for Ray Tracing”, *Data Structures for Raster Graphics*, Springer-Verlag, pp. 57–73, 1986.

[14] Arvo, James, “Linear-time Voxel Walking for Octrees”, in Eric Haines, ed., *Ray Tracing News*, vol. 1, no. 5, `http://www.acm.org/tog/resources/RTNews/html/`, 1988.

[15] Timothy L. Kay and James T. Kajiya, “Ray Tracing Complex Scenes”, *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4), pp. 269-278, August 1986.

[16] Goldsmith, Jeffrey, and John Salmon, “Automatic Creation of Object Hierarchies for Ray Tracing”, *IEEE Computer Graphics & Applications*, vol. 7, no. 5, pp. 14–20 , May 1987.

[17] Haines, Eric A., and D. P. Greenberg, “The Light Buffer: a Shadow Testing Accelerator”, *IEEE Computer Graphics & Applications*, vol. 6, no. 9, pp. 6–16 , 1986.

[18] Haines, Eric, “Spline Surface Rendering, and What’s Wrong with Octrees”, in *Ray Tracing News*, vol. 1, no. 2, January 1988.

[19] Badt, Sig Jr, “Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing”, *The Visual Computer*, vol. 4 no. 3, pp. 123–132, September 1988.

[20] Adelson, Stephen J., and Larry F. Hughes, “Generating Exact Ray-Traced Animation Frames by Reprojection”, *IEEE Computer Graphics & Applications*, vol. 15 no 3, pp. 43–53, May 1995.

[21] Myszkowski, Karol, Przemyslaw Rokita, and Takehiro Tawara, “Perceptually-Informed Accelerated Rendering of High Quality Walkthrough Sequences”, in Gred Ward Larson and Dani Lischinsky (eds), *Proceedings of of 10th Eurographics Workshop on Rendering*, pp. 13–26, June 1999.

[22] Walter, Bruce, George Drettakis, and Steven Parker, “Interactive Rendering using the Render Cache”, in Gred Ward Larson and Dani Lischinsky (eds), *Proceedings of of 10th Eurographics Workshop on Rendering*, pp. 27–38, June 1999.

[23] 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.

[24] Scher Zagier, Ellen J., “Frameless antialiasing”, UNC Technical Report TR95-026, May 1995.

[25] Hennessy, John L., and David A. Patterson, *Computer Architecture: A Quantitative Approach, second edition*.

[26] Singh, Jaswinder P., A. Gupta, and M. Levoy, “Parallel Visualization Algorithms: Performance and Architectural Implications”, *Computer*, vol. 27, no. 7, pp. 45–55 , July 1994.

[27] “Rayshade”, `http://www-graphics.stanford.edu/ cek/rayshade/`.

[28] Haines, Eric, “Standard Procedural Databases (and NFF)”, `http://www.acm.org/tog/resources/SPD/overview.html`.

[29] Kochanek, Doris H.U., and Richard H. Bartels, “Interpolating Splines with Local Tension, Continuity and Bias Control”, *Computer Graphics (SIGGRAPH ’84 Proceedings)*, vol. 18, pp. 33–41, July 1984.

[30] Eberly, David, code from `http://www.magic-software.com/`.

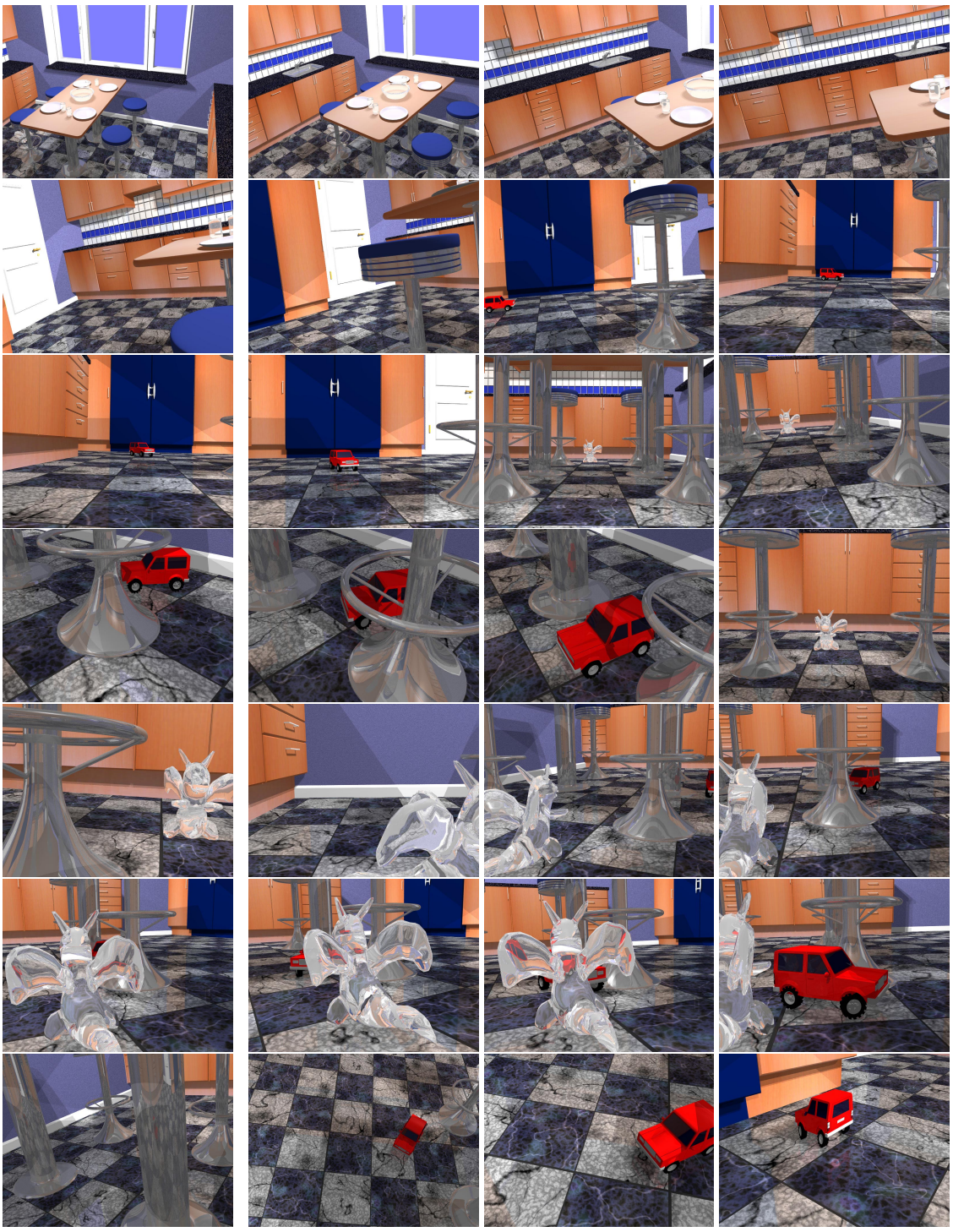

Figure 3: The kitchen test scene. The animation starts at the upper left, and should be read from left to right and top to bottom.

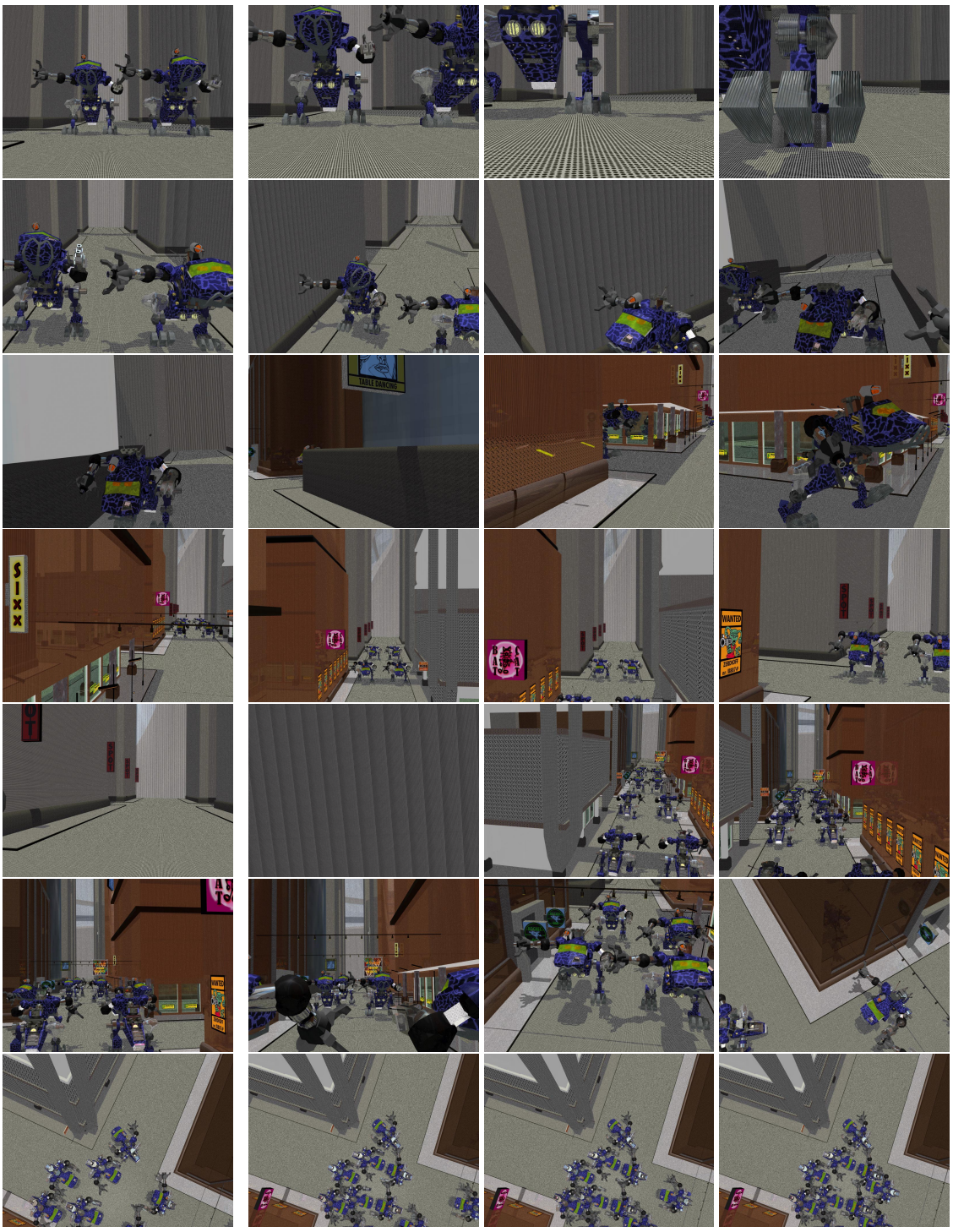

Figure 4: The robots test scene. The animation starts at the upper left, and should be read from left to right and top to bottom.

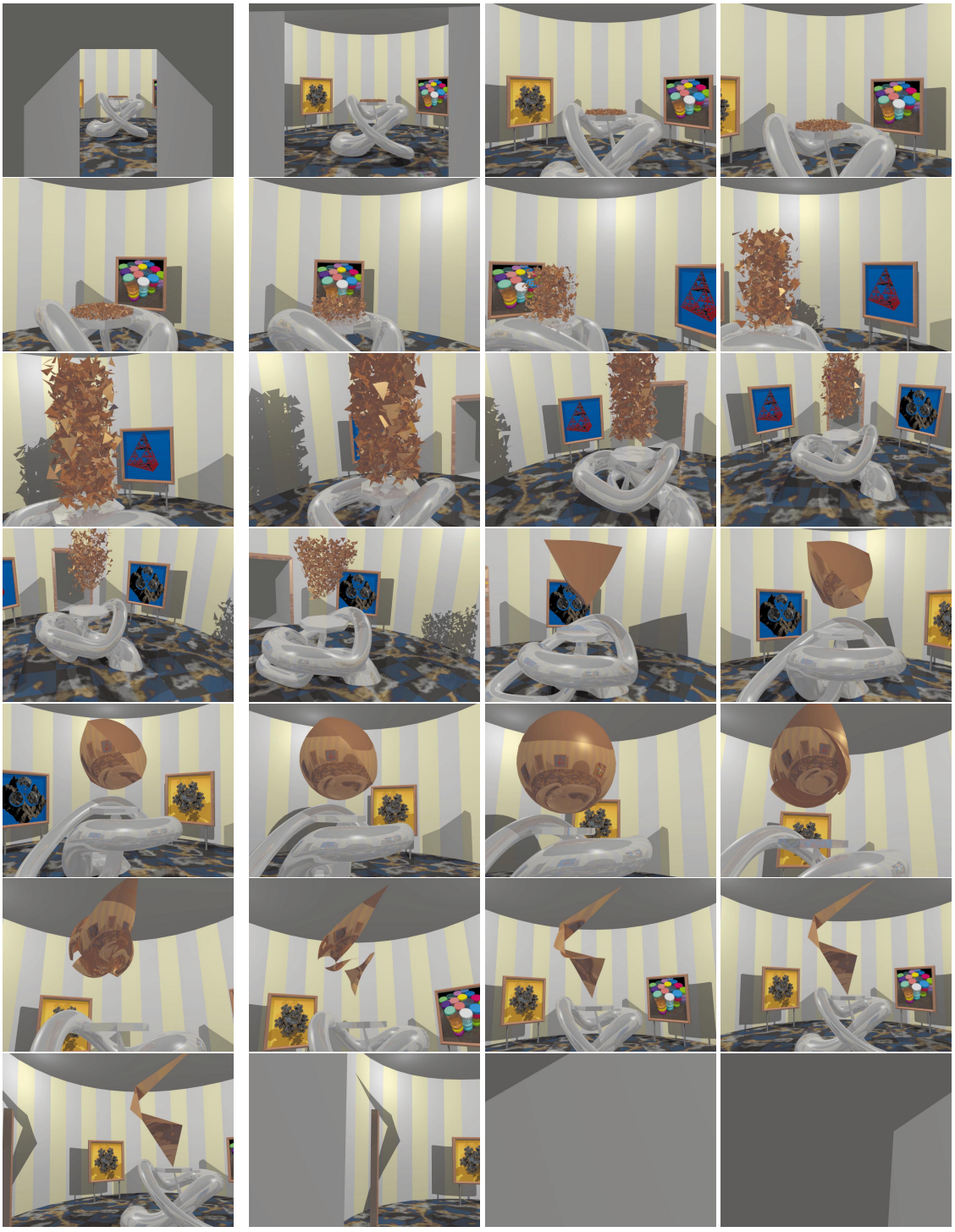

Figure 5: The museum test scene. The animation starts at the upper left, and should be read from left to right and top to bottom.