THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

View Frustum Culling and Animated Ray Tracing: Improvements and Methodological Considerations

ULF ASSARSSON

Department of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2001

View Frustum Culling and Animated Ray Tracing: Improvements and Methodological Considerations

ULF ASSARSSON

Technical Report no. 396L

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:
Ulf Assarsson
ABB Robotics Products AB
Drakegatan 6
SE–412 50 Göteborg, Sweden

Phone: +46 (0)31–773 8513
Email: `uffe@ce.chalmers.se`
URI: `http://www.ce.chalmers.se/~uffe`

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2001

View Frustum Culling and Animated Ray Tracing: Improvements and Methodological Considerations

ULF ASSARSSON
Department of Computer Engineering, Chalmers University of Technology

Thesis for the degree of Licentiate of Engineering, a Swedish degree
between M.Sc. and Ph.D.

Abstract

Today's algorithms and computers are orders of magnitude too slow for photo-realistic rendering of complex scenes in real time. Even though the speed of graphics rendering hardware grows rapidly, there are strong reasons to believe that we will never get sufficient rendering power for naive algorithms. Algorithmic performance improving techniques are therefore essential.

This thesis presents new performance improving techniques, as well as tools and methodologies to be used in research aimed at performance improving algorithms. Four new algorithmic improvements for fast culling of objects that are outside the field-of-view (view frustum culling) are evaluated in combination with existing methods. The execution times are measured and compared between the implementations. The results show that the new techniques are successful in lowering the amount of work that needs to be done. Furthermore, different combinations of improvements are evaluated. The thesis also investigates how to utilize multiprocessors to speed up view frustum culling compared to using only one processor. A number of previously documented load distribution schemes are implemented and the amount of resulting parallelism is evaluated. The results show that the communication cost (communication/computation ratio) involved in the load distribution is too high for the schemes to provide significant speedup. However, with a number of straightforward tricks that are presented, this is circumvented, and a speedup of four is achieved on eight processors. With the same load distribution scheme applied on collision detection, the speedup is three times on seven processors. Finally, the thesis presents a benchmark and a methodology for comparing algorithms for animated ray tracing. The criteria for comparing ray tracing algorithms are identified. The potential stresses of existing ray tracing algorithms are categorized. The result is a benchmark, implementing the stresses into three scenes, and a methodology for comparing algorithms where image quality may be traded for speed.

Keywords: View frustum culling, parallel tree traversal, collision detection, animated raytracing, bench-mark, 3D computer graphics.

List of Appended Papers

The thesis is a summary of the following papers. References to the papers will be made using the Roman numbers associated with the papers.

I. Ulf Assarsson and Tomas Möller, “Optimized View Frustum Culling Algorithms for Bounding Boxes,” *Journal of Graphics Tools*, 5(1), Pages 9-22, 2000. Corresponding tech-report: “*Optimized View Frustum Culling Algorithms,*” Technical Report 99-3, Department of Computer Engineering, Chalmers University of Technology, http://www.ce.chalmers.se/staff/uffe/, March 1999.

II. Ulf Assarsson and Per Stenström, *”A Case Study of Load Distribution in Parallel View Frustum Culling and Collision Detection,”* Department of Computer Engineering, Chalmers University of Technology, Sweden, January 2001. Submitted for publication.

III. Ulf Assarsson, Jonas Lext and Tomas Möller, ”BART: A Benchmark for Animated Ray Tracing,” *IEEE Computer Graphics and Applications*, Pages 22-30, March/April, 2001.

1 Introduction

Computer graphics is the science of how to generate (render) images with computers. This includes algorithms for creating realistic images of three dimensional scenes by using geometrical models and shading models. One particular goal is to achieve the image quality of a photograph or even images that are indistinguishable from the reality. In real time computer graphics, the images should be updated fast enough to give the impression of smooth motions, such as that used in TV. The problem is that today's algorithms and computers are orders of magnitude too slow to be able to generate photo-realistic images in real time. Therefore, performance improvement techniques to make graphics algorithms run faster are important. This thesis considers two applications: view frustum culling and animated ray tracing.

A three dimensional scene is typically composed of several individual objects (like a table, a sofa, chairs etc) represented by geometry. Triangles are often used as a primitive to approximately catch the shape, since they can be rendered fast by graphics hardware. Upon traditional rendering, the objects are usually drawn to the image one by one, triangle by triangle, as seen from the desired view-point. Objects outside the field-of-view do not affect the result of the image (except if they cast shadows into the image or are reflected by visible objects - but that is usually handled separately), and thus need not be rendered. View frustum culling (VFC) is the technique for determining whether or not an object currently is located within the field-of-view and discarding objects outside.

Ray tracing is an alternative method to rendering objects one by one for generating images [12, 13]. Virtual rays of light are traced, typically backwards from the viewer's eye into the scene, in order to determine the color of each picture element (pixel). Ray-tracing is typically slower than hardware accelerated triangle rendering, but is able to create very realistic images with simple and general algorithms, and is therefore very popular. With the improvements of computer performance and algorithms, it is now possible to ray trace simple scenes in real time. However, there is an absence of a way to compare the performance of algorithms for animated ray tracing, i.e., ray tracing of scenes where objects are moving.

This thesis considers performance improvement techniques for view frustum culling and a methodology for comparing the performance of algorithms for animated ray tracing. The thesis is a summary of three papers, denoted with Roman numerals **I**, **II** and **III**. The contribution of **I** is new algorithm improvements and evaluation of combinations of improvements for view frustum culling. The contribution of **II** is a case study of parallel view frustum culling focused on how to utilize multiple processors to get significant speedup. In particular, we evaluate the usefulness of previously published load distribution strategies. The contribution of **III** is a methodology for evaluating ray tracing algorithms for animated scenes. It is also a set of rules on how to measure and compare rendering performance and image quality.

The thesis commences with explanations of the problems and contributions for each paper. The papers on view frustum culling (**I** and **II**) are treated in Section 2. Comparing performance of algorithms for animated ray tracing (**III**) is summarized in Section 3.

2 View Frustum Culling

The view frustum is the pyramid-shaped volume in front of the viewer within the field-of-view, and with the eye at the top. The four planes defining the sides of the pyramid virtually passes through the window borders. Usually a near plane and far plane is added [4], cutting the top off the pyramid and limiting the depth. In this case the view frustum is totally defined by 6 planes (see Figure 1).

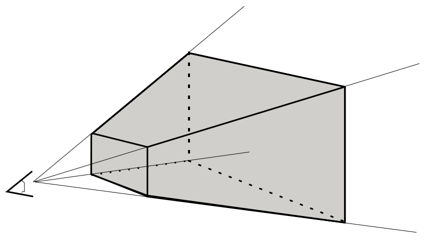


Figure 1: The view frustum is the area in grey in front of the eye or the camera. It is defined by the window extensions, the field-of-view, and the near- and far planes.

View frustum culling is a technique for determining which objects that are outside the view frustum. Objects outside the view frustum are not visible (possibly with the exception of reflections) and thus are unnecessary to render.

For each object a bounding volume is computed that is faster to test against the frustum than the object itself. The bounding volume (BV) should enclose the object completely, and at the same time be as tight-fitting as possible. Bounding spheres and bounding boxes are two popular entities. A test is devised such that the intersection between the BV and the six planes can be determined. The BV is tested geometrically against the six planes of the frustum. If the BV is inside all six planes, the object is totally inside the view frustum and should be renderered. If the BV is outside at least one of the planes, the object is totally outside the frustum and does not need to be rendered. Otherwise, the object is sent for rendering since it may be visible.

View frustum culling is often performed hierarchically, to avoid testing each individual object against the view frustum. Sometimes the scene's natural hierarchy is used, and sometimes a separate BV hierarchy is created. A BV hierarchy may be created in the following way. Objects close to each other are clustered into groups and for each resulting group, a bounding volume is computed, enclosing all its members BVs. Nearby groups are then clustered into larger groups, with their BVs computed equally, and so on until there is only one BV enclosing the whole scene. The hierarchy is represented as a tree structure. A natural structure could for instance be a root node representing a room with one of the child nodes representing a table in the room, and with its children representing the four legs and the board.

When rendering the scene, the BV-tree is traversed top-down, and for each node the bounding volume is tested for intersection with the view frustum. If the bounding volume is totally inside the frustum, the contents of the subgraph is marked for rendering. If it is totally outside, the subgraph is pruned. Otherwise, the traversal continues recursively.

A framerate of 70-85 frames per second (fps) is sufficient for smooth motions [8, 10]. This gives 12-13 milliseconds of computation time for each frame. If there are many objects that need to be tested, for instance in a large scene, the VFC may require a major part of this time. It is important to free CPU time for all other necessary tasks, such as animation, application logic, artificial intelligence etc. Thus it is important to improve the VFC algorithm for speed.

2.1 Optimized View Frustum Culling Algorithms

In some disciplines, *optimization* is often loosely meant as an improvement, in terms of execution time, of some algorithms. In this section we use optimization in that sense.

Paper **I** presents three new optimizations – *the plane-coherency test, the octant test,* and *the Translation and Rotation coherency test (TR test)* – and analyzes their efficiency. All presented techniques here involve adding computations in one step, assuming this can save computations in another step. The circumstances under which the optimizations are efficient sometimes overlap, and thus **I** also evaluates how to best combine the new and existing optimizations in order to maximize performance. In this paper only the most promising combinations are presented. In a corresponding technical report [2] more detailed results are reported.

The plane coherency test utilizes temporal coherence. An object that was outside a plane during the previous frame is likely still outside that plane during the next frame, and thus by testing against that plane first, testing of the other planes can often be avoided. The TR test uses previous results and considers the change in viewer position to determine which objects that must remain inside or outside the frustum.

The octant test exploits the fact that it is sufficient to test the BV against the three closest planes of a symmetric frustum.

The VFC algorithm used as a base for the added optimizations is derived from the idea that the problem can be transformed into testing one point against a swept volume [3]. For bounding boxes this results in an algorithm presented by Green [6]. This algorithm, in itself and together with different combinations of optimizations, is compared against another popular method where the BV is transformed into frustum space (perspective coordinate system) and surrounded by a new axis-aligned bounding box. In this system the frustum is also an axis-aligned bounding box and the resulting two bounding boxes are compared against each other.

The tests were performed on a personal computer using three industrial models as environments. Since the speedup of the algorithms are highly dependent of the 3D-environment, position of the viewer, and how the viewer is moved, four different paths for the viewer for each scene were used.

The results show that the plane coherency test should always be used, and preferably in combination with either the octant test or the TR test or both. The TR coherency optimization was especially fruitful when the navigation involves either pure rotation or pure translation and gave up to a ten-fold speedup. In this case the plane coherency test plus the TR test combination gave the best performance. For other types of navigation the plane coherency test plus the octant test combination was preferable.

2.2 Parallel View Frustum Culling and Collision Detection

Hierarchical VFC of complex scenes with large BV-trees requires high computing performance if a high frame rate should be achieved. Using multiprocessor systems is one way to increase the available computing power. The work load must be balanced between the processors in order to utilize the system efficiently. Load distribution cannot be done statically (once for all frames) since the actual work, i.e., the traversed and tested BVs, varies between different frames as the viewer or objects move.

The difficulty in a load distribution scheme for view frustum culling is that the computation cost at each node is very small (at the size of hundreds of cycles). In our target multiprocessor systems, the approximate time for the inter-processor communication cost is 100 clock cycles. This means that the penalty of communication is very high and that it is important to avoid all unnecessary communication, such as communication for synchronization of the sending and receiving processors. The cost of distributing work to another (typically less loaded) processor must be lower than the cost of performing the work itself, in order to benefit from dynamic load balancing.

A comparative evaluation of several previously documented load distribution strategies is presented in **II** in order to determine if they are suitable for parallel hierarchical VFC algorithms. There are lots of previous research on parallel tree traversals [9, 14, 15], but none has explicitly considered VFC. The kind of work done at each node strongly affects the characteristics of the parallel tree traversal, like cache-miss behavior and load-balance. Thus it is important to make an investigation particularly for view frustum culling.

The same three industrial models used in **I** were used as test scenes – all highly unbalanced. Balanced scenes would be easier to traverse in parallel, but such scenes cannot be expected by a real visual application. The camera was moved along one specific path – sampled from user navigation – for each model, involving both rotation and translation between many frames. The VFC algorithm utilizes the plane-coherency test and the octant test as this was one of the best combinations of optimizations from **I**.

The algorithms were implemented on a Sun Enterprise 4000 shared-memory multiprocessor system. This machine is equipped with 14 UltraSPARC-II CPUs running at 248 MHz. The execution times of the parallel experiments were measured and compared with the execution times of the algorithms when run on only one processor.

The evaluated previously documented load distribution schemes were found incapable of providing meaningful speedup when using multiple processors compared to using one processor. Paper **II** presents a modified scheme for which no synchronization is required when load balancing. On many multiprocessor systems all communication is performed at the so-called cache-block level, i.e., a whole block of data (typically 32 bytes) is sent in just a little bit more time it would take to send just one byte. The scheme utilizes this to lower the average distribution cost of a job to further reduce the communication cost. With the proposed scheme, up to about a four-fold speedup on eight processors was achieved.

As a final contribution, the success of the suggested load distribution scheme applied on collision detection is tested. Collision detection is a problem instance similar to VFC, where the BV-trees of two objects are tested mutually for intersection. A speedup of three times was achieved for seven processors.

Paper **II** shows that it is possible to get significant speedups with parallel VFC and collision detection in real applications, if a suitable load distribution strategy is used.

3 BART: A Benchmark for Animated Ray Tracing

Ray tracing is an alternative to rendering objects one by one for creating images. Typically it is slower but able to generate images of higher quality. Relatively simple algorithms are capable of rendering phenomena such as shadows and reflections.

For each picture element (pixel) of an image, a virtual ray of light is traced into the scene in order to find the closest hit object (the visible object at that spot). Acceleration data structures, built from the scene data, are commonly used to speed up this process. It could for instance be a BV hierarchy, where only the members of BVs penetrated by the ray, are searched.

For static scenes, the acceleration structures are usually built in a preprocessing step. For an animated scene, the data structures have to be rebuilt or updated between successive frames.

In the last few years real-time ray tracing has become a reality due to faster processors and improved algorithms, but still only fairly simple scenes can be rendered. When striving for ray tracing algorithms with higher performance and to help pushing research, it is essential to be able to compare the performance of different algorithms.

The goal of paper **III** is to identify the important criteria when comparing algorithms for animated ray tracing and implement a benchmark based on the results. The benchmark must consider features that stresses the algorithms. It also needs to specify a methodology for comparing the performance of the rendering. Since it is common to trade image quality for speed by using approximations it is also necessary to be able to measure and compare the differences in image quality due to approximation errors.

A number of benchmarks and test scenes for image rendering exist [1, 5, 7, 11], but none of them is appropriate for comparing algorithms of animated ray tracing.

The circumstances that tend to stress existing ray tracing algorithms, i.e., lowering the speed, were identified and categorized into eight groups. All eight types of stresses were implemented into the benchmark in three test scenes – *kitchen*, *robots*, and *museum*.

The benchmark can be used in two modes: *predetermined* and *interactive* mode. In predetermined mode it is allowed to utilize information about the future, i.e., how the objects and camera are going to move in the following frames. In interactive mode this is prohibited to simulate a non-predictable future, for example if a viewer is navigating in real time and objects are moved unpredictably by for instance artificial intelligence algorithms.

The following types of potential stresses were identified: 1) *Hierarchical animation using translation, rotation and scaling.* An animated object may have a tree structure (for instance a BV-tree), where nodes within the tree are animated and requires the acceleration structure to be updated between frames. This costs computation time. 2) *Unorganized animation of objects*, like waves on a surface or a squeezed ball, also requires the data structure to be updated. 3) *"Teapot in the stadium" problem.* The distribution of objects or details is highly unbalanced, which severely lowers the efficiency of certain acceleration structures. 4) *Low frame-to-frame coherency.* Some ray tracing algorithms utilizes similarities between adjacent frames. 5) *Large working sets.* Reading data from the main memory is typically slow. If scenes are used that do not fit into the memory caches of the processor(s), this may induce overhead. 6) *Bounding volume overlap.* It is common to test a ray for intersection of the bounding volume of several objects/entities before testing the ray against each internal element. If a ray penetrates several overlapping BVs, all the members of those BVs have to be tested, which increases the amount of work that has to be done. 7) *Changing object distribution.* The acceleration structure that is most efficient often varies with the object distribution. When the object distribution changes, the chosen acceleration structure may become inappropriate. Computing a new acceleration structure costs CPU time. 8) *The number of light sources* often affects the rendering time of ray tracing algorithms. The more light sources, the more work needs to be done to consider shadows correctly.

It is desirable to be able to compare different algorithms implemented by different researchers on different computers. To do this objectively, paper **III** suggests that a number of parameters and measures should be presented, of which some are: image resolution, number of frames the animation is divided into, processor speed, available memory, total rendering time, average frame time, worst frame time, preprocessing time. Moreover, a graph showing the rendering time as a function of the frame number should be presented.

Hopefully, the benchmark will be used extensively in future research, and evolve over time.

The authors of paper **III** have equally contributed to the work described here.

Acknowledgements

My first thanks go to my supervisor Professor Per Stenström for guiding me and teaching me how to be a good Ph.D student and perform quality research.

Tomas Möller deserves a whole page of thanks. He was the one that opened the door for me into the world of computer graphics research. He made me realize it is possible to do Ph.D studies in this interesting field. He found the perfect professor – Per Stenström – and also gave me an excellent start with his idea of writing a VFC-paper together. I look forward to many further exiting and mind thrilling projects and papers together in the future.

I want to thank Jonas Lext for inspiring discussions, exchange of ideas and for being a great fellow traveller to conferences. A special thanks to Jonas and Henrik Holmdahl for suggesting interesting courses and forcing me to read them. Thank you Björn Andersson for interesting discussions of computer science and graphics algorithms, and thanks to all the nice people at the institution.

Several co-workers at ABB deserve mentioning. Acknowledgement to Nabbe, Påfvel, Johannes, Robert, Anders, Daniel, Carina for great gaming of 3D intensive creations, to Gregers for spreading information, and to the Henriks.

Finally I also want to thank ABB Robotics Product AB for the financial support.

References

[1] 3DMark, *http://www.madonion.com/entry.shtml*

[2] Ulf Assarsson and Tomas Möller, "*Optimized View Frustum Culling Algorithms*," Technical Report 99-3, Department of Computer Engineering, Chalmers University of Technology, http://www.ce.chalmers.se/staff/uffe/, March 1999.

[3] M. de berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, "*Computational Geometry – Algorithms and Applications*," Springer-Verlag, Berlin, 1997.

[4] James D. Foley, Andries Van Dam, Steven K. Feiner, John F. Hughes, "*Computer Graphics Principles and Practice*," Addison-Wesley, ISBN 0-201-84840-6, pages 229-235.

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

[6] Daniel Green, Don Hatch, "Fast Polygon-Cube Intersection Testing", *Graphics Gems V, Heckbert*, pp. 375–379, 1995.

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

[8] James L. Helman, "Architecture and Performance of Entertainment Systems, Appendix A," *ACM SIGGRAPH 94 Course Notes – Designing Real-Time Graphics for Entertainment*, vol 23, pages 1.19 – 1.32, July, 1994

[9] V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search on Multiprocessors — Part I: Implementation; and Part II—analysis", *International Journal of Parallel Programming*, vol. 16, no. 6, 1987.

[10] Tomas Möller and Eric Haines, "*Real-Time Rendering*," A.K. Peters Ltd., ISBN 1-56881-101-2, p 1.

[11] Peter Shirley, *http://www.radsite.lbl.gov/mgf/scenes.html*.

[12] Peter Shirley, "*Realistic Ray Tracing*," A K Peters Ltd, ISBN: 1568811101, June 2000.

[13] Turner Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, 23 (6), pp. 343-349, June 1980.

[14] C. Xu, S. Tschoke, and B. Monien, ”Performance Evaluation of Load Distribution Strategies in Parallel Branch and bound Computations”, *Proc. of the 7th IEEE Symposium of Parallel and Distributed Processing (SPDP95)*, Oct. 1995.

[15] Myung K. Yang, Chita R. Das, ”Evaluation of a Parallel Branch-and-Bound Algorithm on a Class of Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 1, January, 1994.

Paper I

Optimized View Frustum Culling Algorithms for Bounding Boxes

Reprinted from

Journal of Graphics Tools, 5(1), Pages 9-22, 2000.

Optimized View Frustum Culling Algorithms for Bounding Boxes

Ulf Assarsson and Tomas Möller
Department of Computer Engineering
Chalmers University of Technology, Sweden

Accepted for publication in *journals of graphics tools*

March 1999, revised February 2000

Abstract

This paper presents optimizations for faster view frustum culling (VFC) for axis aligned bounding box (AABB) and oriented bounding box (OBB) hierarchies. We exploit frame-to-frame coherency by caching and by comparing against previous distances and rotation angles. By using an octant test, we potentially halve the number of plane tests needed, and we also evaluate masking, which is a well-known technique. The optimizations can be used for arbitrary bounding volumes, but we only present results for AABBs and OBBs. In particular, we provide solutions which is $2 - 11$ times faster than other VFC algorithms for AABBs and OBBs, depending on the circumstances.

1 Introduction

Bounding volume hierarchies are commonly used to speed up the rendering of a scene by using a view frustum culling (VFC) algorithm on the hierarchy [Clark76]. Each node in the hierarchy has a bounding volume (BV) that encloses a part of the scene. The hierarchy is traversed from the root, and if a BV is found to be outside the frustum during the traversal, then the contents of that BV need not be processed further, and performance is gained. Reducing the time for view frustum culling will increase performance of a single processor system and free processor time to other tasks. With view frustum culling taking 6 ms before speedup and 1.2 ms after speedup[1] and with 33 frames per second (30 ms/frame, no synchronization to monitor frequency), the view frustum culling goes from taking 20% of total execution time to only 4%, thus saving 16%.

We present several optimizations for culling axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs) against a frustum used for perspective viewing. Frame-to-frame coherency is exploited by caching and by comparing against previous distances and rotation angles. An octant test is introduced, which potentially halves the number of plane tests needed, and we also evaluate masking [Bishop98]. All these optimizations can be used for arbitrary bounding volumes which we show in our technical report [Assarsson99], where we also investigate bounding spheres, with speedups[2] from 1.2 up to 1.4 times. That report also provides more details on the results for AABBs.

2 Related Work

When reviewing existing view frustum culling algorithms [Bishop98, DirectModel, Hoff96a, Hoff96b, Hoff97, Green95, Greene94], we found that there are two common ways to approach the view frustum culling problem.

[1]These are our figures for path 1, model 3 with the plane-coherency and octant test optimization (see section 4).
[2]In this paper we define speedup as *time1/time2*, which means that a speedup of 1.0 is no speedup at all.

One approach is to perspective transform the bounding volume of a node to be tested and the view frustum, to the perspective coordinate system and perform the testing there. This is popular when the bounding volumes are axis aligned bounding boxes, since this results in testing two AABBs (if the perspective transformed AABB is bounded by a minimal AABB, see figure 1) against each other, which can be done with only six comparisons after the transformation has been done. The disadvantage is that we must transform the bounding volume to the perspective coordinate system. This means that all eight vertices of the AABB must be multiplied with the view- and projection (perspective) matrix, which includes at least 72 multiplications. The view frustum culler in DirectModel is based on this method [DirectModel].

The other approach is to test the bounding volume against the six planes defining the view frustum [Hoff96a, Hoff96b, Hoff97, Green95, Greene94]. This is also the approach that we choose. This has the advantage that trivial rejection or acceptance tests can be made [Greene94, Green95]. Should these fast tests fail, traditionally the more expensive intersection tests necessary to find exact intersection between a box and a frustum are computed. Instead of that, we recursively continue testing the planes of the sub-boxes, in order to get higher performance.

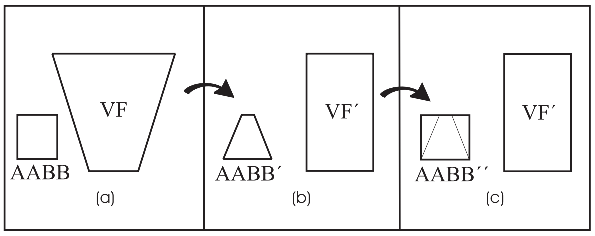


Figure 1: (a) View frustum and an AABB. (b) The same view frustum and AABB perspective transformed. (c) The perspective transformed AABB is bounded by a minimal AABB in the perspective coordinate system, which is tested for intersection with the view frustum.

VFCs can be based on BSP-trees to gain speed [Chin95] with the drawback that BSP-trees only represent static environments.

Slater et al. [Slater97] present a VFC based on a probabilistic caching scheme using ellipsoids, which according to their results provides comparable speedups to our methods[3]. It may, however, erroneously cull objects that should be visible.

3 Frustum-AABB/OBB Intersection

Our algorithms for view frustum culling of hierarchies with AABBs or OBBs are all based on a basic intersection test. Four different optimizations can be added on top of this, so we have partitioned our main algorithm into five steps:

- *the basic intersection test*, sped up to test just two box corners (section 3.1)
- *the plane-coherency test*, taking advantage of frame-to-frame coherency by using previous test results (section 3.2)
- *the octant test*, allowing half of the plane/bounding-volume tests to be avoided while using symmetric frustums (section 3.3)
- *masking*, in which bounding-box/plane test results are passed on and reused by children bounding-boxes (section 3.4)

[3]The most fair way is probably to compare their reported speedup of about 1.7 with the speedups provided by the combinations of our optimizations compared to our basic intersection test. See section 4 or our technical report for more details [Assarsson99].

- *TR*[4] *coherency test*, which allows reuse of previous frame test results when the view changes in limited ways (section 3.5)

The optimizations can be utilized in a VFC independently of each other. That is, we can choose and combine the steps anyway we like. A view frustum (VF) is defined by six planes:

$$\pi_i : \mathbf{n}_i \cdot \mathbf{x} + d_i = 0 \tag{1}$$

$i = 0 \ldots 5$, where $\mathbf{n}_i$ is the normal and d_i is the offset of plane π_i, and $\mathbf{x}$ is an arbitrary point on the plane. We say that a point $\mathbf{x}$ is outside a plane π_i if $\mathbf{n}_i \cdot \mathbf{x} + d_i > 0$. If the point is inside all planes then the point is inside the view frustum.

3.1 Basic Intersection Test

An exact intersection test between a box and a frustum may be expensive to compute. Therefore we use the following strategy: for each frustum plane, test if the box is outside, inside or intersecting the plane. If outside, terminate and report `outside`. If the box is inside all planes, return `inside`, else return `intersecting`. Note that this is an approximation; sometimes when we report `intersecting` the box may be outside (see figure 2). For intersecting boxes we continue the hierarchy traversal, so the end result is still correct. If a node is a leaf

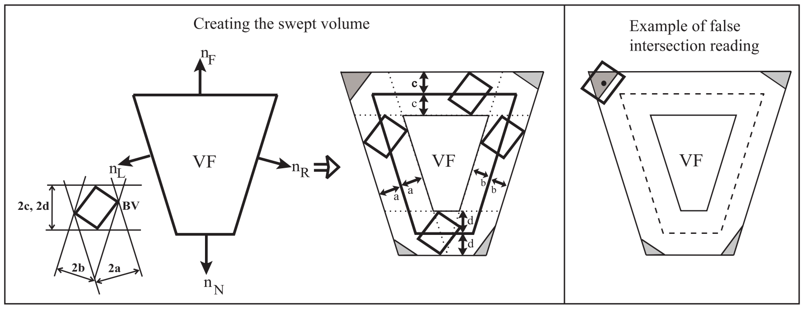

Creating the swept volume
Example of false intersection reading
n_F
VF
n_L
n_R
n_N
2c, 2d
BV
2b
2a
c
a
b
d
VF
VF

Figure 2: If the box center is within the grayed sections, our approximation reports `intersection` even though the box is completely outside the view frustum (VF). This is explained by the following; an intersection test can be conducted by testing the box center against the volume obtained by sweeping the box along the planes of the VF, keeping the box center on the planes. Our approximation is the same as testing the box center against the resulting inner and outer planes parallel with, and at different offsets from the VF planes. In three dimensions the corresponding gray sections are located at the corners and the edges. If the center is outside the planes, the box is outside. If the center is between the planes, we have intersection, and if it is inside, the box is inside. See our technical report for details and a generalization to any bounding volumes [Assarsson99].

then we can choose to do more accurate tests. However, since we found no observable penalty in the rendering performance, we skipped those tests.

We first present how to determine if a box intersect a plane. A naive method is to test all eight points against the plane. However, only two points need to be tested [Haines94, Greene94, Green95], namely those that forms the diagonal that is most closely aligned with the plane's normal, and that passes through the box center. These points are called the *n- and p-vertices*, where the p-vertex has a greater signed distance from the plane than the n-vertex.

First, the n-vertex is inserted in the plane equation. If the n-vertex is outside then the box is outside (both the plane and the frustum) and the test terminates. Then the p-vertex is tested, and if the p-vertex is inside then the the box is inside the plane. Otherwise the box intersects the plane. This is illustrated in figure 3. Finding the two *n- and p-vertices* can be done in 9 multiplications and 3 comparisons by projecting the normal of the view frustum plane

[4]TR stands for Translation and Rotation

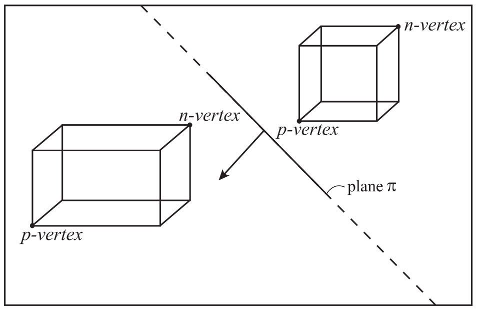


Figure 3: The negative far point (n-vertex) and positive far point (p-vertex) of a bounding box corresponding to plane π and its normal.

```
bool $intersect = false$
for i in [all view frustum planes $\pi_i$] do
     $\mathbf{v}_n \leftarrow$ negative far point (n-vertex) in world
           coordinates of box relative to $\pi_i$
     $a \leftarrow \mathbf{v}_n \cdot \mathbf{n}_i + d_i$
     if $a > 0$ then return Outside
     $\mathbf{v}_p \leftarrow$ positive far point (p-vertex) in world
           coordinates of box relative to $\pi_i$
     $b \leftarrow \mathbf{v}_p \cdot \mathbf{n}_i + d_i$
     if $b > 0$ then $intersect = true$
end loop
if $intersect$ then return Intersecting
else return Inside
```

Figure 4: Pseudo code of general algorithm for culling AABBs or OBBs

on to the box's axes and test the signs of the x-, y- and z-components of the projection. Since all AABBs are given in the world coordinate system (aligned to the world x-, y- and z-axes) and we transform all view frustum plane equations (i.e. plane normals and offsets) to world coordinates at the beginning of each frame, we have the AABBs and the normal of the planes in the same coordinate system, which makes a projection unnecessary. We can use the signs of the x-, y- and z-components of the plane normal immediately, leaving us with only three comparisons. If we create a bitfield of the signs, letting for instance a negative sign be represented by a '0' and a positive sign be represented by a '1', we can use this bitfield to get the p-vertex from a Look Up Table (LUT). In this way we avoid the conditional branches caused by if-statements, which can lead to expensive processor pipeline prediction misses. This idea is used by Donovan et al. to accelerate clipping [Donovan94]. If we order the LUT properly, we can invert the bitfield to get the n-vertex.

If we are going to test multiple AABBs against the view frustum (which generally is the case) and since all AABBs have the same orientation, it is a good idea to precompute the bitfields (indices to the n- and p-vertices) for each view frustum plane once each frame [Haines94].

A listing of the algorithm for testing a box against a frustum is given in figure 4.

3.2 The Plane-Coherency Test

The goal of this test is to exploit temporal coherence. Assume that a BV of a node was outside one of the view frustum planes last time it was tested for intersection (previous frame). For small movements of the view frustum there is a high probability that the node is outside the same plane this time, which means that we should start testing against that plane hoping for fast rejection of the BV. If the BV was outside a plane last frame, then an index to this plane is cached in the BV structure. For each intersection test, we start testing against that plane and test the others afterwards if necessary.

We test the planes in the order: $left$, $right$, $near$, far, up, and $down$. No experiments in finding an "optimal" order has been conducted.

3.3 The Octant Test

Assume that we split the view frustum in half along each axes, resulting in eight parts, like the first subdivision of an octree. We call each part an *octant* of the view frustum.

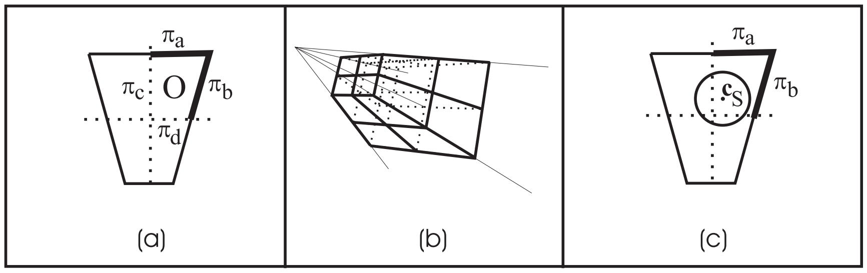


Figure 5: (a) 2D-view of a symmetric view frustum divided in half along each axis. π_a and π_b are the outer planes of octant O. π_c and π_d are the inner planes. (b) View frustum divided in octants. (c) For a symmetrical VF it is sufficent to test for intersection against the outer planes of the octant in which the center ($\mathbf{c}_S$) of the bounding sphere (of the box) lies.

If we have a symmetrical view frustum, which is the most common case (a CAVE [CrusNeira93] is one exception), and a bounding sphere, it is sufficient to test for culling against the outer three planes of the octant in which the center of the bounding sphere lies (see figure 5). This means that if the bounding sphere is inside the three nearest (outer) planes, it must also be inside all planes of the view frustum. If it is outside any of the planes, we know it is totally outside the view frustum, and otherwise it is intersecting.

This can be extended to general bounding volumes; see our report [Assarsson99]. To be able to use the octant test for boxes, the distance from the box center to a box corner must be smaller than the smallest distance from the view frustum center[5] to the frustum planes (see figure 6). This is true, since an arbitrary BV cannot intersect the planes of another octant without intersecting the planes of the selected octant, if the above holds. The distance between the center of the view frustum and its nearest plane can be precomputed once for each frame.

The cost of locating the octant was found to be approximately equal to one plane/box test.

3.4 Masking

Assume that a node's BV is completely inside one of the planes of the view frustum. Then, as pointed out by Bishop et al. [Bishop98], we know that the BVs of the node's children also lie completely inside that plane, and that plane can be eliminated (masked off) from further testing in the subtree of the node.

When traversing the scene graph, a mask (implemented as a bitfield) is sent from the parent to the children. This mask is used to store a bit for each frustum plane, which indicates whether the parent is inside that plane. Before each plane test, we check if that plane is masked off or not. In this way, plane tests can be avoided if the parent of a node is inside one or more planes.

If we can eliminate the low-level polygon clipping against the window border corresponding to a view frustum plane, for all nodes that are totally inside that plane, then maybe masking could pay off a lot [Bishop98]. Low

[5]The center of the frustum is the sum of the eight frustum corners divided by eight.

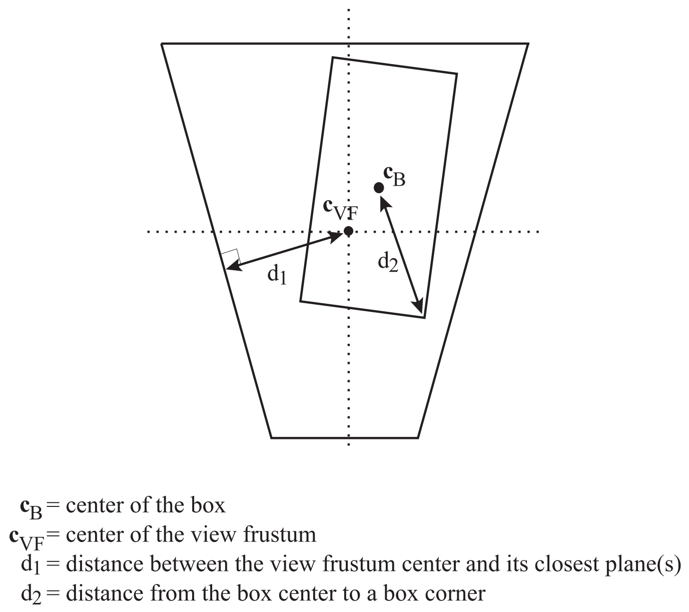

$\mathbf{c}_B$
$\mathbf{c}_{VF}$
d_1
d_2
$\mathbf{c}_B$ = center of the box
$\mathbf{c}_{VF}$= center of the view frustum
d_1 = distance between the view frustum center and its closest plane(s)
d_2 = distance from the box center to a box corner

Figure 6: If $d_2 <= d_1$ we can use the octant test for bounding boxes as well.

level clipping of polygons is usually done against each view frustum plane for each polygon sent for rendering. For nodes that are totally inside the view frustum, all clipping could be disabled and then potentially provide speedups.

3.5 The TR Coherency Test

TR coherency stands for translation and rotation coherency. In this optimization step, we exploit the fact that when navigating in a three dimensional world, you sometimes only rotate around one axis or translate, or you might even be standing still. For objects that have not moved since the last frame the following applies:

1. If, for instance, a BV was outside the left plane of the view frustum last frame, and the view frustum only has rotated to the right since then, we know that the BV still is outside the left plane (assuming that the rotation is smaller than $180° - angle\ between\ left\ and\ right\ plane$). In general this means that if only view frustum rotations have been done around either the x-axis, y-axis or the z-axis of the view frustum since last culling invocation, we can return `outside` for BVs if they were outside the plane last frame and if the distance to the plane must have increased (see figure 7a).

2. If the view frustum only has done a pure translation since last frame, the distances from all BVs to the same view frustum plane have increased or decreased by the same fixed amount Δd (see figure 7b). This Δd is possible to precompute once for all intersection tests against the corresponding plane. If only a translation (in any direction) has been done since last view frustum culling invocation, we precompute Δd_i for each view frustum plane π_i by projecting the translation on the normal of the planes. For each BV and view frustum plane to be tested, we compare the corresponding Δd_i with the distance between the BV and the plane last frame.

For (1) we precompute the plane that can use this optimization (if any), and for each BV which was outside this plane last frame, we return `outside`. Let us assume the view frustum axes are arranged according to figure 7c. If the view frustum, since last frame, has done a pure rotation around the y-axis in the positive direction, we can do quick rejection against the right plane. If instead the rotation was negative, we can do quick rejection against the

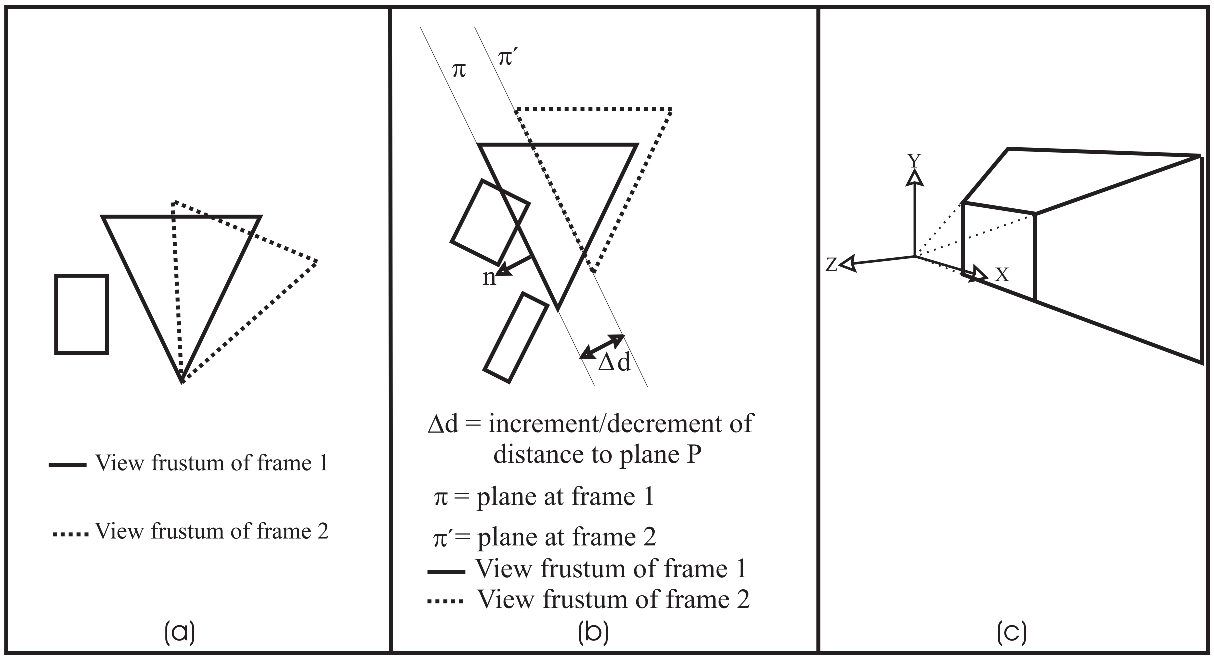


Figure 7: (a) Rotations of the view frustum. If the BV of a non-moving object was outside the view frustum at frame 1, we know that, because of the direction of rotation, it is also outside in frame 2. (b) Translations of the view frustum. (c) A view frustum and its frustum coordinate axes.

left plane. We have to keep track of the accumulated rotations to be able to invalidate any quick rejections when the total rotation around the axis exceeds $180^o - angle\ between\ left\ and\ right\ plane$. The x-axis and the up- and down planes are treated similarly. If rotations only occured around the z-axis, objects outside the near- and far plane will remain outside.

For (2) we have to add members to the BV structure holding the distances from the BV to the different planes of the view frustum, and we must also add a member indicating whether or not the BV was explicitly tested last time (otherwise the interesting distances is not calculated, and we must perform our test with another method).

4 Results

Each optimization and combinations of optimizations have been thoroughly tested to determine whether its possibly introduced overhead has paid off in shorter average execution times. The presented figures are speedups of the VFC-algorithms - not of total rendering time - and are compared against a view frustum culler testing AABBs in the perspective coordinate system (see section 2).

The implementation was done on a double PentiumII 200 MHz with 128 Mb RAM and were compared with the VFC in DirectModel on the same machine. All algorithms were tested on three virtual environments:

- Model 1: a car factory shop floor (184,000 polygons, 3800 graph nodes)
- Model 2: a factory cell (167,000 polygons, 188 graph nodes)
- Model 3: a factory shop floor (52,000 polygons, 1274 graph nodes)

We used four paths in our tests. For each path, about a million box/intersection tests were computed.

path	1	1	1	2	2	2	3	3	3	4	4	4
model	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*
only Basic intersection test	2.8	1.9	3.9	2.2	2.0	3.1	3.9	2.5	4.3	3.1	2.2	3.7
Plane-coherency + octant test	**4.0**	**2.4**	**5.1**	**2.8**	**2.6**	**3.9**	4.8	**3.5**	**5.6**	3.3	3.0	5.1
Plane-coherency + TR coherency	3.8	2.0	4.0	2.5	2.2	3.0	5.0	2.8	4.4	**8.3**	3.1	**11.0**
Plane-coherency + octant test + TR coherency	3.7	2.2	4.5	2.6	2.4	3.6	**5.1**	3.0	4.8	8.0	**3.3**	9.0

Table 1: Speedup for the most promising combinations of optimizations, except for the 'only Basic intersection test'-row which is there for comparison issues. The figures are speedup compared to a view frustum culler testing AABBs in the perspective coordinate system (see section 2). The speedup figure of the best algorithm in each test case is marked with bold text. Each figure corresponds to a path, model and an algorithm.

- Path 1: sampled from a user navigating through our scenes
- Path 2: constructed with both a translation and a rotation each frame
- Path 3: pure rotations only
- Path 4: pure translations only

Table 1 presents the results of the most promising optimizations. For statistical data about other combinations, see our report [Assarsson99].

The speedup numbers were obtained using AABBs. For OBBs we did not create any separate OBB-tree. Instead we treated the AABBs in the AABB-hierarchy as OBBs, i.e added the necessary 9 multiplications and 6 additions in each intersection test and continued comparing against the AABB-algorithm of DirectModel. The penalty for OBBs showed to be about 10% more computation time for all test cases. Since OBBs provide better fits, they might give better overall performance.

If we for AABBs precompute the indices to the n- and p-vertices once each frame, instead of calculating them for each box (see section 3.1), an additional speedup of $5 - 10\%$ will be achieved. This optimization cannot be used for OBBs.

The plane-coherency + octant test is best in all test cases except for pure translations, where the plane-coherency + TR coherency test is superior. For symmetrical frustums (which is most common except for CAVEs [CrusNeira93]) we recommend the plane-coherency + octant test, and if we expect many pure translations also the TR coherency test to get the best of both worlds.

For asymmetric frustums, we recommend that the plane-coherency test and the TR coherency test are combined and used[6].

Masking was not found to be competitive with the algorithms above.

Our recommended combinations of optimizations boost the basic intersection test up to 3.0 times with an average of 1.4 times.

For individual intersection tests between a BV and the view frustum, the AABB implementation of DirectModel was sometimes faster than our implementations. This occurred in average for $\approx 0.2\%$ of all intersection tests, which means that for $\approx 99.8\%$ of all cases, our algorithms were faster. This figure is based on timing the intersection tests with the CPU clock, which means that anomalies due to cache misses and page faults are included.

[6]For asymmetric frustums the octant test cannot be used.

5 Discussion

All our optimizations, including the basic intersection test, can be used on any kind of bounding volume, but we have only gathered statistics for AABBs, OBBs and bounding spheres. Since the sphere/plane test is so short, there was little gain in using our optimizations. For details, see our report [Assarsson99].

We have not made use of the fact that in general, the near clip plane and far clip plane are parallel, in any other cases than for the *octant test* and the *TR coherency test.* We could add this to our *basic intersection test* as well. We should then treat the near- and far clip planes as a pair of parallel planes instead of two individual, saving at least 6 multiplications and 2 evaluations of the *n- and p-vertices.* The reason why we did not include this is that we did not find an easy way to insert this into the algorithm, without slowing down other parts or make the code ugly.

If we find that the bounding volume is neither completely outside any plane, nor completely inside all planes, we might want to store one of the intersecting planes so that we can start checking against that plane in the next round, hoping that the bounding volume would have moved outside the plane and the view frustum.

It would also be interesting to modify our algorithm to handle k-DOPs[7] [Klosowski97] as bounding volumes. For DOPs we should probably take advantage of that they consist of pairs of parallel planes. Finding the *n- and p-vertices* or maximum extension in a specific direction from the center of a DOP is not trivial. Look up tables could perhaps be used in some cases, or we might have to approach the problem in a totally different way.

Since the difference in cost of using AABBs and OBBs is small, it would be interesting to investigate whether OBB hierarchies are faster than AABB hierarchies.

6 Acknowledgements

We would like to thank professor Per Stenström for his extensive guidance in improving the quality of this paper. Thanks to Lars Östlund, manager of research and development at Digital Plant Technologies AB, ABB, for the financial support. We also thank Eric Haines for pointing us at his and Wallace's paper [Haines94].

References

[Assarsson99] Ulf Assarsson and Tomas Möller, "Optimized View Frustum Culling Algorithms", Technical Report 99-3, Department of Computer Engineering, Chalmers University of Technology, *http://www.ce.chalmers.se/staff/uffe/* March 1999.

[Bishop98] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, Michael Shantz, "Designing a PC Game Engine", *Computer Graphics in Entertainment*, pp. 46–53, January/february 1998.

[Chin95] Norman Chin, "A Walk through BSP Trees", *Graphics Gems V, Heckbert*, pp. 121–138, 1995.

[Clark76] James H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithm", *Communications of the ACM*, vol. 19, no. 10, pp. 547–554, October 1976.

[CrusNeira93] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, "Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE", *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp 135-142, volume 27, aug, 1993.

[Donovan94] Walt Donovan, Tim van Hook, "Direct Outcode Calculation for Faster Clip Testing", *Graphics Gems IV, Heckbert*, pp. 125–131, 1994.

[DirectModel] *DirectModel 1.0 Specification*, Hewlett Packard Company, Corvalis, 1998

[7]A k-DOP (discrete oriented polytope) is made up of k pairs of parallel planes, the intersection of which forms a bounding volume. A bounding box can be thought of as a k-DOP where k is three and all planes are orthogonal.

[Greene94] Ned Greene, “Detecting Intersection of a Rectangular Solid and a Convex Polyhedron”, *Graphics Gems IV, Heckbert*, pp. 74–82, 1994.

[Green95] Daniel Green, Don Hatch, “Fast Polygon-Cube Intersection Testing”, *Graphics Gems V, Heckbert*, pp. 375–379, 1995.

[Haines94] “Shaft Culling for Efficient Ray-Traced Radiosity”, Eric A. Haines and John R. Wallace, *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, Springer-Verlag, New York, pp.122–138, 1994, also in *SIGGRAPH ’91 Frontiers in Rendering course notes*.

[Hoff96a] K. Hoff, “A Fast Method for Culling of Oriented-Bounding Boxes (OBBs) Against a Perspective Viewing Frustum in Large ”Walktrough” Models”, *http://www.cs.unc.edu/ hoff/research/index.html*, 1996.

[Hoff96b] K. Hoff, “A Faster Overlap Test for a Plane and a Bounding Box”, *http://www.cs.unc.edu/ hoff/research/index.html*, 07/08/96, 1996.

[Hoff97] K. Hoff, “Fast AABB/View-Frustum Overlap Test”, *http://www.cs.unc.edu/ hoff/research/index.html*, 1997.

[Klosowski97] J.T. Klosowski, M. Held, J.S.B. Mitchell,H. Sowizral, K. Zikan “Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs” , *http://www.ams.sunysb.edu/∼jklosow/projects/coll_det/collision.html*, 1997.

[Slater97] Mel Slater, Yiorgos Chrysanthou, Department of Computer Science, University College London, “View Volume Culling Using a Probabilistic Caching Scheme” *ACM VRST ’97 Lausanne Switzerland*, 1997.

Paper II

A Case Study of Load Distribution in Parallel View Frustum Culling and Collision Detection

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden, January 2001
Submitted for publication.

A Case Study of Load Distribution in Parallel View Frustum Culling and Collision Detection

Ulf Assarsson[1] and Per Stenström[2]

[1] ABB Robotics, Drakegatan 6, SE-412 50 Göteborg, Sweden
uffe@ce.chalmers.se,
[2] Department of Computer Engineering Chalmers University of Technology SE-412 96, Göteborg, Sweden
pers@ce.chalmers.se

Abstract. When parallelizing hierarchical view frustum culling and collision detection, the low computation cost per node and the fact that the traversal path through the tree structure is not known à priori make the classical load-balance versus communication tradeoff very challenging. In this paper, a comparative performance evaluation of a number of load distribution strategies is conducted. We show that several strategies suffer from a too high an orchestration overhead to provide any meaningful speedup. However, by applying some straightforward tricks to get rid of most of the locking needed, it is possible to achieve interesting speedups. For our industrially related test scenes, we get about a four-fold speedup on eight processors for view frustum culling and three times speedup for collision detection.

1 Introduction

View frustum culling (VFC) and collision detection are two very common components of real time computer graphics applications. VFC aims at reducing the computational complexity of a succeeding rendering pass by extracting the graphics objects that are in the view frustum. For hierarchical VFC, a hierarchy is built up as a tree structure from the bounding volume of each object. Each node in the tree has a bounding volume enclosing a part of the scene. The tree is traversed from the root in a depth-first manner, and if a bounding volume is found to be outside the frustum during the traversal, the contents of that subtree can be culled from rendering. The typically low computation cost makes the load distribution in a parallel implementation extremely challenging.

In this paper we evaluate the effectiveness of a set of load distribution strategies on parallel implementations of hierarchical view frustum culling with scenes from an industrial application. We also examine the capability of the most promising scheme applied on collision detection. For VFC we use axis aligned bounding box (AABB) trees [8], while for collision detection we use both AABB- and oriented bounding box (OBB) trees [4].

The load distribution schemes we select are a global task queue, and a number of distributed task queue schemes well-known from the literature. We evaluate the speedup of the parallel implementations using these strategies on a 13-node Sun Enterprise shared-memory multiprocessor and on a dual PentiumIII 500 MHz personal computer.

We find that while some of the schemes were expected to provide a reasonable speedup, they performed inferior owing to the high communication and synchronization cost. Our results show that due to the low computation cost per node compared to the distribution cost, only the more sophisticated lock-free scheme provides interesting speedup numbers. By considering a number of optimizations – especially by getting rid of the synchronizations – we managed to get promising results, even for highly unbalanced industrial scenes. For our scenes, we achieve a speedup of around four on eight processors for view frustum culling and about three on seven processors for collision detection with real test cases from an industrial case study.

2 Experimental Set-Up

The code for testing a bounding volume against the view frustum is the one of a previously proposed optimized algorithm [1]. This implements many optimizations such as caching of previous computations, implying little computation cost per node in many cases. Other optimizations include plane-coherency, octant, and translation and rotation coherency tests (see [1] for details).

We use three trees that are the hierarchical scene graph representations of three 3D models - all of real environments and all used in industrial applications. The three highly unbalanced trees used in the tests are: a car factory shop floor in $3,932$ graph nodes, a factory shop floor in $1,137$ graph nodes and a factory cell in 254 graph nodes. We refer to them as the *large model*, the *medium model*, and the *small model,* respectively.

The camera–or view frustum–used in the view frustum culling computations is moved along one specific path for each model, each sampled from a user walk through in the model. The presented traversal times and speedups are the average times and average speedups of all traversals during the walk through.

The experiments are carried out on a Sun Enterprise 4000 shared-memory multiprocessor. This machine is equipped with 14 UltraSPARC-II CPUs running at 248 MHz. Each CPU is attached to a 16-Kbyte L1 data cache and a 1-Mbyte L2 cache, both using a line size of 32 bytes. The locks used have been implemented using the SPARC-instruction `ldstub` which loads a byte followed by a store that sets all bits in that byte atomically. We only show results for up to 13 processors. One processor is left for the operating system to avoid the perturbation it would cause when it is invoked every millisecond.

3 Evaluation of Load-Distribution Schemes

In this section, we consider the effectiveness of load distribution strategies that seem adequate for the dynamic behavior of our workload. As a reference, we use the classical global task queue scheme which we consider first.

3.1 Global Task Queue

In this approach, each processor removes and add tasks (tree-nodes) using a global task queue. The virtue is good load balance while the overhead associated with orchestrating the global task queue is known to be high.

Results from the experiments of parallel VFC are presented in Figure 1. Figure 1.a-1.c show the average speedup, and Figure 1.e-1.f show the average execution time for VFC of one frame.

For the global task queue, the maximum number of processors that can provide speed-up, before the global task queue becomes the bottle-neck, is limited to the total time for processing a node divided by the time for accessing the global queue (*node_cost/ access_cost*). We see that we get a maximum speedup of only 1.5, with only three processors on the small model. Moreover, when we increase the number of processors, the speedup goes down owing to serialization effects, as expected.

3.2 The Global Counter Scheme

A more scalable strategy is to associate a local task queue with each processor. Each processor adds tasks to the local queue pointed to by a global counter that is incremented after each insertion by any processor and protected by a lock[1] according to [11]. In this way the load will be nearly optimally balanced if all processors can process nodes equally fast. The serialization of accesses to one single queue is replaced by the serialization of reading and incrementing the global counter, which is usually faster. However, the lock mechanism around the counter can potentially become a new bottleneck when we increase the number of processors. In addition, the locks that synchronize the accesses to the queue attached to each processor is another potential bottleneck.

As can be seen in Figure 1a, compared to the global task queue algorithm, the stagnation in speed-up which peaks at about 1.9, comes later – at more than eight processors instead of three, which is expected since incrementing a counter is quicker than inserting or removing a task (which in our implementation basically consists of changing an array index and reading the contents of the array element, i.e about twice the cost). The stagnation comes from the global lock which gives a high cost and introduces serialization.

3.3 The Hybrid Scheme

To further reduce the orchestration overhead and contention due to locking and shared memory access, we considered two optimizations of the global counter scheme. The resulting scheme is referred to as *hybrid*.

– **The skip-pointer tree optimization:** A common optimization in raytracing is to represent the tree in depth-first order in an array [16], with a skip

[1] For some processors it is possible to atomically read and increment a variable with just one or two assembler instructions instead of using a lock.

index for each node that points out the next element to access if the underlying sub-graph should be skipped during the traversal. Then a full tree traversal can be performed by simply accessing the array sequentially from start to end. Every subtree will be represented in the array as a consecutive chunk of elements, so instead of distributing a node (subtree), we send the start-index and the stop-index of the array. While it provides good cache-locality in the sequential single processor case, it can also give better locality in the parallel case.

– **Trading off larger tasks for less load balance:** This straightforward optimization uses the observation that at a certain depth, when the underlying subtree only contains a few nodes, it will be faster to process the nodes rather than distributing them, if the computation-cost is smaller than the distribution-cost [13].

Since the size of each subtree is not known beforehand, the heuristic we have tried is to distribute tasks at the node-level until a certain level after which the rest of the subtrees are considered as tasks. The first phase uses the global counter scheme according to Section 3.2, whereas the second phase serially executes the tree traversal algorithm with no further balancing of the load. Both phases use the skip pointer optimization and thus will enjoy the increased locality it provides. A counter keeps track of how many nodes that so far have been processed by the distribution algorithm. If a threshold number is exceeded, all processors finish the computations and distribution of children for the node it is currently working on, and enter the serial phase. We found empirically that a threshold of six times the number of processors gave the best performance for our models with a difference in load of less than 2% for the large model.

The skip-pointer tree optimization contributed with an overall speedup of $15 - 40\%$ compared to the global counter scheme. Despite the possibility to also trade between load balance and larger tasks, the total speedup for both optimizations together peaks at only 2.2 times (for 10 processors).

We also made measurements showing that if the cost of the VFC computations at each node were virtually zero, we would get a huge slowdown using more than one processor. The reason is the high distribution cost compared to the cost of the serial traversal of the skip-pointer tree. Skipping the distribution phase, resulting in a serial single processor algorithm, would actually have been optimal for this case.

The schemes used so far suffer from too much overhead, especially concering lock accesses. This motivated us to seek for a lock-free approach which we study in the next section.

4 A Lock-free Scheme

The Lock-Free scheme distributes the load without requiring locks or any synchronization. The way we adapted the original scheme to avoid locking is as follows.

Each processor has one local-queue and some in-queues. A processor removes tasks from its local-queue and its in-queues, and adds new tasks to its local queue and dedicated in-queues of neighboring processors. The in-queues are created such that one processor can insert tasks at one end of the queue and another processor can remove tasks from the other end of the queue, without any need for synchronization between the two. There is one dedicated in-queue for each sender/receiver pair. We use a ring buffer with two indices to point out the start and the end of the buffer.

The `insert()` method only needs to affect the start-index, and the `remove()` method only needs to affect the end-index. It is easy to assure that the `insert()` and `remove()` operations never can access the same memory location simultaneously.

The `remove()` operation needs to check if the queue is empty before allowing removal of a task, and because the `insert()` operation always inserts a task into the array before incrementing the end-index, computing `end - start` will always give a safe result. The same safe situation holds for the `insert()` method, when checking if the queue has room for more elements before inserting a task. The array simulates a ring and the indices will wrap around to the first element after passing the last element of the array, but this is easy to adjust for.

Since we want to avoid locks completely, we only allow a processor to either insert or remove jobs from an in-queue - not both. The opposite could be interesting to try, since there are ways to implement this such that the locks, with a high probability, seldom will be used [3].

4.1 Topology

In order to easily change the number of processor connections in the topology, we first order the processors virtually in a ring, where each processor distributes tasks to its successor's in-queue. When increasing the connectivity and wanting every processor to send tasks to n receivers, with p processors in the ring, we add connections to every $(\frac{p}{n} + 1)$:th successor. When inserting a connection between two processors, we assign an in-queue for the receiver and let the sender send tasks to this queue. Figure 1.h) shows an example of 6 processors, each distributing to 3 receivers.

Load Balancing For Adaptive Contracting within neighborhood (ACWN), the least loaded nearest neighbor is always selected as the receiver of a newly generated job. It is known that local averaging strategies generally outperforms methods such as the randomized allocation and the ACWN algorithm significantly in large scale system [17]. Since our shared memory system is a so called one-port communication system (i.e at most one neighbor can receive a message in a communication step) with one central data bus, we use the Local Averaging Dimension Exchange (LADE) policy. Generally it is better than the diffusion method (LADF) on such a system [18]. In LADF, load balancing is done with all neighbors, while in LADE load balancing is only done with one of the neighbors, or one at a time with the new load-balance successively considered.

Our approach is to use a sender-induced rather than a receiver-induced load distribution strategy. An advantage of the receiver-induced approach is that tasks are only distributed on demand which potentially reduces the overall cost of distribution. A disadvantage, however, is that processors may sit idle to wait for tasks to be available which may waste computing cycles. We briefly tried some receiver-induced approaches for the lock-based schemes, but they were inferior to the sender-induced, and thus we decided to try the sender-induced policy first for the lock-free schemes. Cilk-5 [3] is a parallel development system that uses the other approach (see section 6).

A high degree of connections between processors in the virtual topology enables better load-balancing. Since the communication is the bottle-neck and the computation cost at each node in the tree is low, we need a simple/fast load-balancing scheme. Only sending newly generated jobs to each receiver and to the local queue in a round-robin fashion, was found to be insufficient to maintain good load-balance. We needed to consider the load difference between processors, which costs computation and communication. If a processor has more jobs than the receiver, it sends half the difference of the load. However, empirically we found that it was enough to even out the load balance this way with only one of the receivers and send blindly to the rest, to get similar load balance as the global task queue. We chose to consider the load balance difference only with the successor in the main ring. If n jobs are transferred in this step, we wait at least n traversed nodes before trying to load-balance carefully again, since load balancing is expensive and the successor probably will have work to do at least the corresponding time. In the final algorithm, after every processed node we distribute the newly generated jobs to the local-queue and the receiving processors in a round-robin fashion. If $n = 0$, where n is a variable set to the number of tasks sent to the successor last time and decreased after every traversed node, we also do the extra load-balancing with the successor. Every time we distribute jobs to the successor we may increment n.

If we have a topology with many connections for each processor, we potentially risk lowering cache-locality when we spread the jobs over many queues. In the shared memory system, the jobs are physically sent when the receiving processor reads its in-queues and the corresponding cache-blocks are transferred from the sending processor to the receiving. In order to minimize the number of cache-block reads, the receiving processor selects one in-queue for reading until it is empty, before selecting a new in-queue. We could also avoid using an in-queue for reading that does not fill up an entire cache-block, if there are others that do, but we did not implement this.

In general, a high number of connections between processors in the virtual topology seemed to be preferred (see tables at the side of Figure 1.a-1.c).

4.2 Experimental results

For this scheme, the speedup is substantially better for the large and medium model, with 4.3 and 3.1 times respectively. For the small model it is only 1.7, but this model provides poor speedup for all the schemes. Load balance is similar to what that of the other schemes.

It was found that the time for just traversing the trees in parallel, not doing any VFC-computations, was fairly constant independently of the number of processors used. This means that we can decompose the total execution time as:

$$time_{total} = time_{traversal} + time_{VFC} \tag{1}$$

where $time_{VFC}$ is the only term that enjoys speedup from the parallelism in VFC. This speedup, however, is basically optimal with respect to the possible parallelism provided by the traversed paths.

Depending on which parts of the scene-graph that are visible in a frame, the maximum of possible parallelism can vary, since there is a limited amount of parallel paths in the traversed graph. We found that if the whole tree is traversed, with each child selected for continued traversal disregarding the result of the VFC computations, the speedup peaks at 5.1, which is slightly higher than the average speedup. This indicates that the speedup is limited by the appearance of the scene graph. Since it represents a bounding box hierarchy, we cannot rearrange the graph without caution.

We also tested the Lock-Free scheme on a 2-processor PentiumIII 500MHz, with 256 Mb RAM, with a simpler load distribution policy that just keeps every 2:nd child and distributes the other to the other processor. The topology is a virtual ring of 2 nodes. With this approach we got 1.7, 1.5 and 1.3 times speedup for the large-, medium- and small model respectively. The load balance was practically perfect.

5 Collision Detection

Since the lock-free scheme was pretty successful in parallelizing VFC we tested it on hierarchical collision detection to see how it performs on this similar type of problem. We kept the same load balancing strategy. Collision detection is known as non-trivial to parallelize [14].

To find collision between two objects, their bounding box hierarchies are tested against each other for overlap. If any of the leaves between the two trees intersect, the objects are considered colliding. The algorithm starts with the root boxes of both trees. If intersection occurs, the algorithm continues recursively by testing the smallest of the two boxes (or the one that is not a leaf) against the children of the larger box respectively. If both boxes are leaves, a collision is found and the algorithm terminates. In this way a virtual graph is traversed.

A hierarchical AABB-tree of a small industry-robot with 102 nodes and a tree-depth of 11, was tested for intersection against the large model (a car factory). The robot was spatially placed such that the algorithm is forced to traverse deep down in both trees to verify that collision (in this case) not occurs.

Testing two AABBs against each other for overlap is extremely fast and basically consists of just 6 compares, while testing two arbitrarily oriented bounding boxes (OBBs) costs about 200 flops in average [4]. OBBs, however, can be more tight fitting and are thus often preferred. We wanted to test both cases. In the OBB-case, for simplicity, the AABBs were treated as OBBs in the overlap-computation with the orientation incidentally coinciding with the x,y,x-axes.

We found that for collision detection as well as for VFC, the traversal time without collision computations was nearly independent of the number of processors used. Consequently, since AABBs are very fast to test for overlap, we only got very limited speedup - 30% with 4 processors. For OBBs, however, the speedup peaks at 3.2 as can be seen in Figure 1.d).

6 Related Work and Discussion

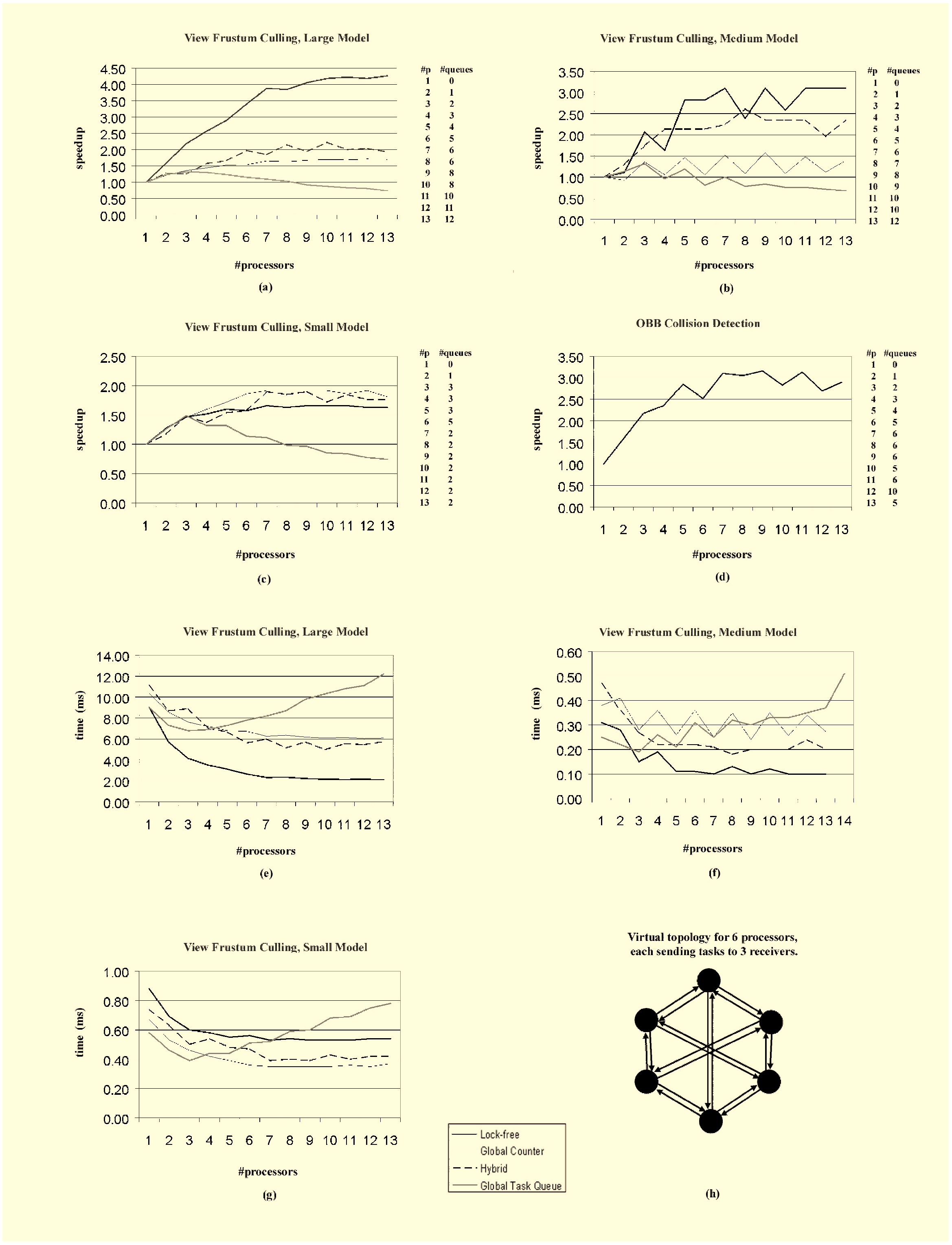
Several older parallel branch-and-bound techniques [2, 5, 7, 9, 10, 19] and depth-first search algorithms like backtracking [11–13] seem at a first glance to be applicable to the applications we have at hand. Our results indicate, however, that the load distribution strategies in these algorithms do not apply very well to tree traversals found in VFC and collision detection because of the low computation cost per node compared to the distribution cost.

In this paper we have focused on sender-induced schemes since this seemed most promising for the lock-based approaches. However, Cilk-5, which has been available for a short time, uses task-stealing in a way that looks promising. It requires the use of locks, but there are convincing arguments that they seldom will cause contention or significantly increased communication. Two of the main features of Cilk-5 is 1) that it compiles two versions of the code: one serial and one parallel, and can switch in run-time when load-balancing requests are issued, and 2) that load-balancing can occur efficiently through queues similar to those we use in our lock-free schemes.

Other related work that aims at reducing the orchestration overhead in tree traversals includes using prefetching techniques to tolerate communication latencies in the system. Karlsson et al. [6] studied how annotation of prefetch instructions can speed up tree traversals to tolerate the latency of cache misses. They especially considered the class of tree traversals where the traversal path is not known beforehand and obtained encouraging results. While they studied only sequential tree traversals it would be interesting to study the potential for parallel tree traversals.

7 Conclusion

In this paper we have presented a comparative evaluation of load distribution strategies based on a real application case study including two important computer graphics algorithms used in virtual reality. The low computation-to-communication ratio in these algorithms make load distribution particularly challenging. Based on some minor – but important – adaptations of well-known load distribution schemes in the literature, we managed to demonstrate reasonable speedups on a symmetric multiprocessor. Since multiprocessors of this scale are now being used in personal computers, and are seriously considered to



(a)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	6
9	8
10	8
11	10
12	11
13	12

(b)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	10
13	12

(c)

#p	#queues
1	0
2	1
3	3
4	3
5	3
6	5
7	2
8	2
9	2
10	2
11	2
12	2
13	2

(d)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	6
9	6
10	5
11	6
12	10
13	5

Fig. 1. (a-c) Speedup with 1 to 13 processors for the large, medium and small model. For the lock-free scheme, the figures are for the best topology, with the number of connections (in-queues) per processor marked at the side. (d) Speedup for collision detection with an OBB-algorithm with the lock-free scheme. The jaggedness comes from the difference in topology and number of optimal connections. (e-g) Corresponding execution time for the algorithms. (h) Virtual topology for 6 processors where each processor distributes load to 3 other processors. Note that depending on the camera position, a larger tree can be faster to traverse than a smaller. This is the case for the small vs. medium model, where the small offers more immerse navigation.

migrate to the chip-level, our results are indeed encouraging. They show that multiprocessors can be exploited for an emerging class of real-time computer graphics applications.

Acknowledgments

We would like to thank ABB Robotics Products, for the financial support of Ulf Assarsson's research. This research has also been supported by the Swedish Foundation of Strategic Research (SSF) financed ARTES/PAMP program, Sun Microsystems Inc, and by an equipment grant from the Swedish Council for the Planning and Coordination of Research (FRN) under contract 96238.

References

1. Ulf Assarsson and Tomas Möller, "Optimized View Frustum Culling Algorithms for Bounding Boxes", Journal of Graphics Tools, 5(1), Pages 9-22, 2000.
2. E. W. Felten, "Best-first Branch-and Bound on a Hypercube", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, (Vol. 2), Pages 1500-1504, 1988.
3. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, "The Implementation of the Cilk-5 Multithreaded Language", ACM SIGPLAN Conference on Programming Language, 1998.
4. S. Gottschalk, M.C Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection", Proc. of ACM Siggraph, Pages 171-180, 1996.
5. V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, "A Randomized Parallel Branch-and-Bound Algorithm", in Proc. Int. Conf. Parallel Process., Pages 69-75., Aug. 1988.
6. M. Karlsson, F. Dahlgren, and P. Stenström, "A Prefetching Technique for Irregular Accesses to Linked Data Structures", Proc. of 6th Int. Symp. on High Performance Computer Architecture, Pages 206-217, Jan. 2000.
7. Richard M. Karp, Yanjun Zhang, "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation", Journal of the ACM, Volume 40, Pages 765-789, Issue 3, 1993.
8. Tomas Möller and Eric Haines, "Real-Time Rendering", A.K. Peters Ltd, ISBN 1-56881-101-2, 1999.
9. Roy P. Pargas and E. Daniels Wooster, "Branch-and-Bound Algorithms on a Hypercube", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, (Vol. 2), Pages 1514 - 1519, 1988.
10. Michael J. Quinn, "Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer", IEEE Transactions on Computers, vol. C-39, Pages 384-387, no. 3, March, 1990.
11. V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search on Multiprocessors — Part I: Implementation; and Part II—analysis", International Journal of Parallel Programming, vol. 16, no. 6, 1987.
12. V. Nageshwara Rao, Vipin Kumar, "On the Efficiency of Parallel Backtracking", IEEE Transactions on Parallel and Distributed Systems, vol 4, no. 4, Pages 427–437, April, 1993.

13. A. Reinefeld, V. Schnecke, "Work-Load Balancing in Highly Parallel Depth-First Search", Proc. Scalable High Performance Computing Conf. SHPCC'94, IEEE Comp. Sc. Press, Pages 773-780, 1994.
14. Peter Rundberg, "An Optimized Collision Detection Algorithm", http://www.ce.chalmers.se/staff/biff/exjobb, 1998.
15. A. Saulsbury, F. Pong, and A. Novatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration" Proc. of 23rd Int. Symp. on Computer Architecture, Pages 90-101, June, 1996.
16. Brian Smits, "Efficiency Issues for Ray Tracing", A K Peters, Ltd, Journal of Graphics Tools, vol 3, no 2, Pages 1-14, 1999.
17. C. Xu, S. Tschoke, and B. Monien, "Performance Evaluation of Load Distribution Strategies in Parallel Branch and bound Computations", Proc. of the 7th IEEE Symposium of Parallel and Distributed Processing (SPDP95), Oct. 1995.
18. C. Xu and R. Lüling and B. Monien and F. Lau, "An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers", Proceedings of 9th International Parallel Processing Symposium, 1995.
19. Myung K. Yang, Chita R. Das, "Evaluation of a Parallel Branch-and-Bound Algorithm on a Class of Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 1, January, 1994.

Paper III

BART: A Benchmark for Animated Ray Tracing

Reprinted from

IEEE Computer Graphics and Applications, March/April, 2001.

A Benchmark for Animated Ray Tracing

Jonas Lext, Ulf Assarsson, and Tomas Möller
Chalmers University of Technology, Sweden

Benchmarks let people accurately and objectively compare performance. In computer graphics, we need to measure and compare performance and quality to find effective, good algorithms. Currently, however, benchmarks only exist in a few computer graphics areas, but there's a need for them in a variety of areas, such as radiosity, global illumination, collision detection, animation, image-based rendering, and polygon rendering. (See the "Related Work" sidebar for more information.) Our effort is an attempt to bridge that gap.

Our suite of test scenes, BART, is designed to accurately measure and objectively compare performance and quality of ray traced, animated scenes.

We saw the need for our benchmark for animated ray tracing (BART), because no benchmark exists in this area and because at least two groups have been ray tracing fairly complex and realistic scenes at interactive speeds[1,2]—at rates 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. BART's main contribution is three parametrically animated test scenes that we designed to stress ray-tracing algorithms and a set of reliable performance measurements that let BART users compare performance of different ray-tracing algorithms. For approximating algorithms (that is, algorithms that may produce approximate pixel values), we also define how to measure the quality of the approximated images.

Potential stresses

To construct a benchmark with a relatively long lifetime, we first identified 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 (combinations other than translation, rotation, and scaling)
3. the "teapot in the stadium" problem
4. low frame-to-frame coherency
5. large working-set sizes
6. overlap of bounding volumes or their projections
7. changing object distribution
8. the number of light sources

Stress 1: Hierarchical animation

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

When adding animation to a scene, we'll most likely have to reconstruct whole or parts of the acceleration data structures between frames. Depending on the amount of scene changes, this could seriously stress the reconstruction phase when using uniform grids, recursive grids, hierarchical grids,[3] octrees, binary space partitioning trees, and bounding volume hierarchies (BVHs).[4] In our benchmark, we excluded light-source animation, because it simplified our animations, and animated objects can achieve the same stress. Therefore, this is also a serious stress for some acceleration data structures, such as light buffers.[4]

Stress 2: Unorganized animation

To cope with transforms, ray tracers often transform the ray with the inverse transform instead of transforming the object and its efficiency data structures. Thus, for some acceleration schemes (such as a static grid or a BVH around an object), we don't 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 prohibit use of this approach. Because all currently available types of acceleration schemes for ray tracing must rebuild their efficiency structures for such animations,[3,4] this will be a serious stress on all ray-tracing algorithms. Note, however, that there exists an algorithm with $O(1)$ complexity for inserting and deleting objects in an octree that might solve this problem.[5]

0272-1716/01/$10.00 © 2001 IEEE

Related Work

Graphics hardware vendors and implementers have many different benchmarks. The Standard Performance Evaluation Corporation (SPEC) has a subgroup called the Graphics Performance Characterization Group (GPC); see http://www.specbench.org/gpc, which has a set of graphics benchmarks. Their SPECglperf measures the performance of rendering low-level primitives, such as points, lines, triangles, and so on with OpenGL. In contrast, GPC's SPECviewperf for OpenGL measures the rendering performance of a set of real models. Both SPECglperf and SPECviewperf target graphics hardware vendors and implementers, because benchmarks aren't allowed to alter the program (for example, implement an occlusion algorithm) to make the execution more efficient. GPC also has benchmarks for a few commercial programs, but these require fully licensed versions of the programs and thus aren't available for everyone.

For PCs, several other benchmarks for measuring a graphics subsystem's performance exist—3D WinBench, 3DMark, and QuakeIII, to mention a few. We believe that it would be unsuitable, or even impossible in some cases, to use the scenes from these benchmarks (which are targeted toward graphics hardware systems) to evaluate ray-tracing algorithms' performance. One reason is that the scenes in SPEC, for example, are surrounded by license agreements that limit their use for purposes other than those originally intended. Plus, you must report measured results strictly in accordance with the rules SPEC publishes. Because other performance parameters are interesting for ray tracing, we can't use SPEC. Naturally, these benchmarks are also constructed for the single purpose of testing graphics hardware-specific features or specific applications (such as 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, we 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. The scenes in our benchmark are specifically designed to stress ray-tracing algorithms and have a simple, readable format for greater flexibility.

Yuan et al.[1] described a framework for a performance evaluation system for real-time rendering algorithms in virtual reality. They also targeted this benchmark against polygon rendering hardware, and there doesn't seem to be any scenes available on the Internet.

To our knowledge, there's only one recognized benchmark for ray tracing: the standard procedural database by Haines.[2] The SPD targets 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 camera's view frustum. Although the SPD is perfectly valid and has been widely used for more than a decade, progress in computer architecture and algorithms has advanced beyond what it was intended for.

The work by Bala et al.[3] also shows promise for interactive ray tracing. A vast number of ray-tracing algorithms and variants have been developed over the years. We refer to Glassner's book[4] and some recent papers[3,5] for a review of many ray-tracing algorithms. For more information on BART, see our Web site (http://www.ce.chalmers.se/BART), where you can also find our technical report on BART.

References

1. P. Yuan, M. Green, and R. Lau, "A Framework of Performance Evaluation of Real-Time Rendering Algorithms in Virtual Reality," *ACM Symp. Virtual Reality Software and Technology*, 1997, pp. 51-58.
2. E. Haines, "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Applications*, vol. 7, no. 11, Nov. 1987, pp. 3-5.
3. K. Bala, J. Dorsey, and S. Teller, "Radiance Interpolants for Accelerated Bounded-Error Ray Tracing," *ACM Trans. Graphics*, vol. 18, no. 3, Aug. 1999, pp. 213-256.
4. A. Glassner, *An Introduction to Ray Tracing*, Academic Press, London, 1989.
5. K.S. Klimaszewski and T.W. Sederberg, "Faster Ray Tracing Using Adaptive Grids," *IEEE Computer Graphics and Applications*, vol. 17, no. 1, Jan./Feb. 1997, pp. 42-51.

Stress 3: Teapot in the stadium

The "teapot in the stadium" problem[6] refers to when a small, detailed object (teapot) is in a relatively large, surrounding object (stadium). This tends to stress uniform grid-based algorithms and octree-based schemes, because the uniform grid has finite-sized voxels and the octree has a finite depth. Therefore, the teapot will be in one or only a few voxels or octree nodes. For example, if the viewer is looking at the teapot so 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, enormously degrading performance.

Stress 4: Low frame-to-frame coherency

Situations where the frame-to-frame coherency is low tend to stress reprojection algorithms,[7] because they use information from previous frames. If the difference between two frames is too big, the performance of such algorithms is worse or the quality of the rendered images gets worse. Similarly, frameless rendering techniques[8] will produce images of poorer quality when the frame-to-frame coherency is low. In this article, we use the name approximating algorithms for all algorithms that can generate images that aren't 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 processor's computational speed and the speed with which the memory system can feed the processor with data. The conventional solution to this problem calls for using a cache hierarchy between

In ray tracing, the rendering time often increases as the number of light sources in a scene increases. The number of light sources can, therefore, be a serious stress on the rendering engine.

the processor and memory and relying on spatial and temporal coherence in the data access patterns. Current computer architectures are usually equipped with two levels of caches (L1 and L2) with typical sizes ranging between 16 to 64 Kbytes and 128 Kbytes to 2 Mbytes, respectively.

Due to reflecting objects in the scene, we can use a large portion of the scene data when rendering a single frame in an animation. Often, because of frame-to-frame coherence, we'll reuse a lot of this scene data in the following frames. However, if the working-set size exceeds the L2 cache, not all of the required scene data will fit in the cache at the same time, so it must be reread between frames. Thus, the L2 cache miss ratio will be high, and the calculations will be slowed down. The scenes in BART should be able to generate such large working sets and, therefore, stress the memory hierarchy of contemporary computer systems.

Stress 6: Bounding volume overlap

A problem might occur when bounding volumes or local grids overlap (perhaps due to animation). If a ray penetrates all the overlapping volumes, it isn't necessarily the first one reached that contains the closest object intersection. Therefore, a number of bounding volumes or grids may have to be traversed by the ray before encountering the true intersection point. The bounding volumes or grids don't necessarily have to overlap to get this effect. If a ray intersects several bounding volumes or grids that contain a lot of empty space around the object they cover, again, a number of rays might have to traverse a large number of bounding volumes 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's 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 object distribution in the scene changes over time, the most suitable data structure might also change. A solution might be to recreate or update the data structure in each frame.

Stress 8: Number of light sources

In ray tracing, the rendering time often increases as the number of light sources in a scene increases. The number of light sources can, therefore, be a serious stress on the rendering engine.

Animated test scenes

Here, we present the three test scenes. (Table 1 gives brief descriptions of each.) All the test scenes are parametrically animated—that is, the BART user can easily vary the number of frames in an animation. One use of this feature is to test 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.

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, then follows it on its path through the room. Figure 1 shows images taken from this scene.

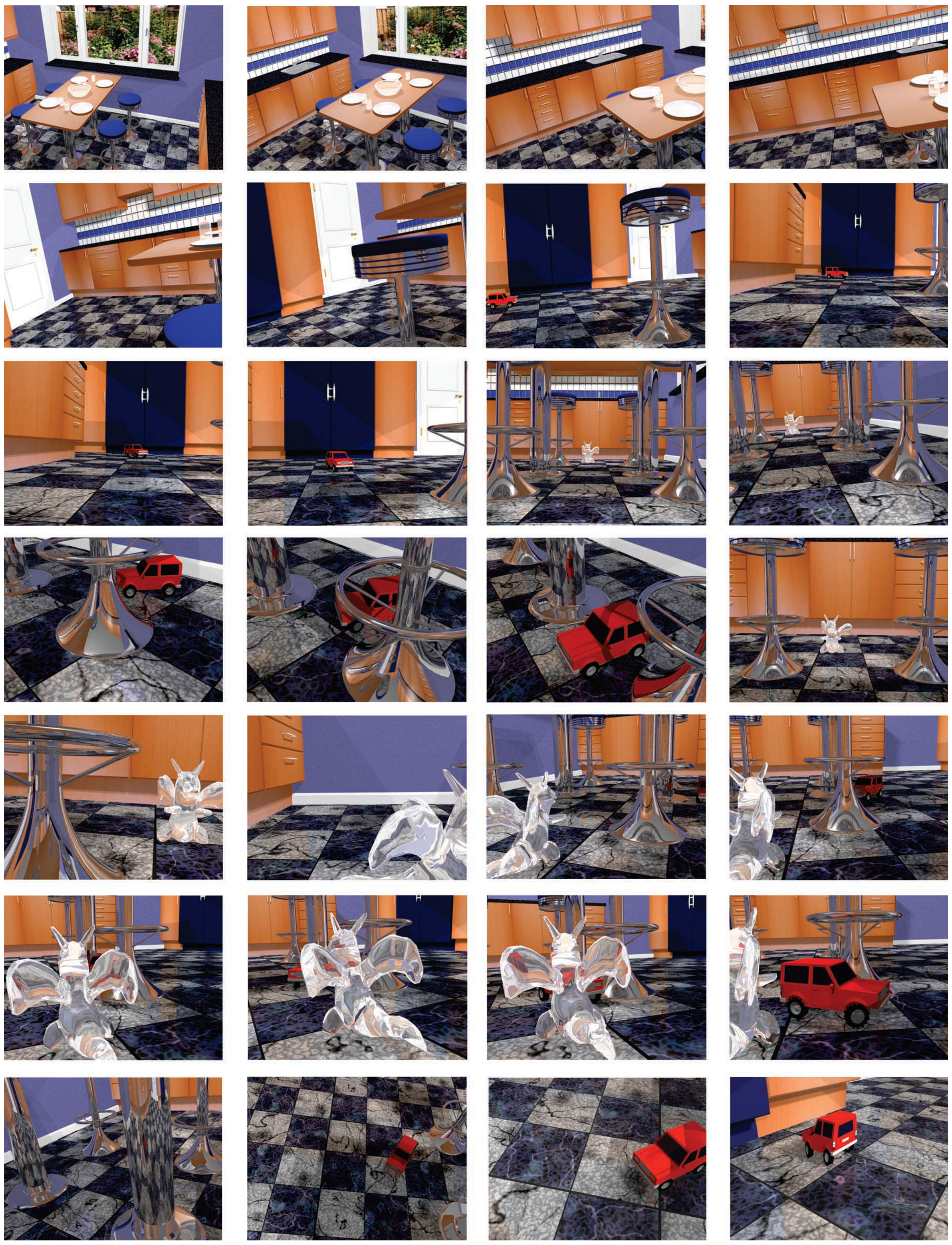
The toy car is hierarchically animated using translations, rotations, and scalings (stress 1). The hierarchy is at most three levels deep, and the scaling occurs at the end of the animation when the car crashes into a cupboard and is thus scaled along the current driving direction. The kitchen scene should be subject to the teapot-in-the-stadium problem (stress 3)—we modeled the walls, roof, and floor using only a few large triangles. However, the scene also contains many complex objects, which vary from small (such as a door knob) to medium (such as chairs).

Low frame-to-frame coherency (stress 4) is 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 occurs when the camera moves rapidly, 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 low.

The kitchen scene can use 1 to 10 light sources (stress 8) and is

Table 1. Short summary of test scenes.

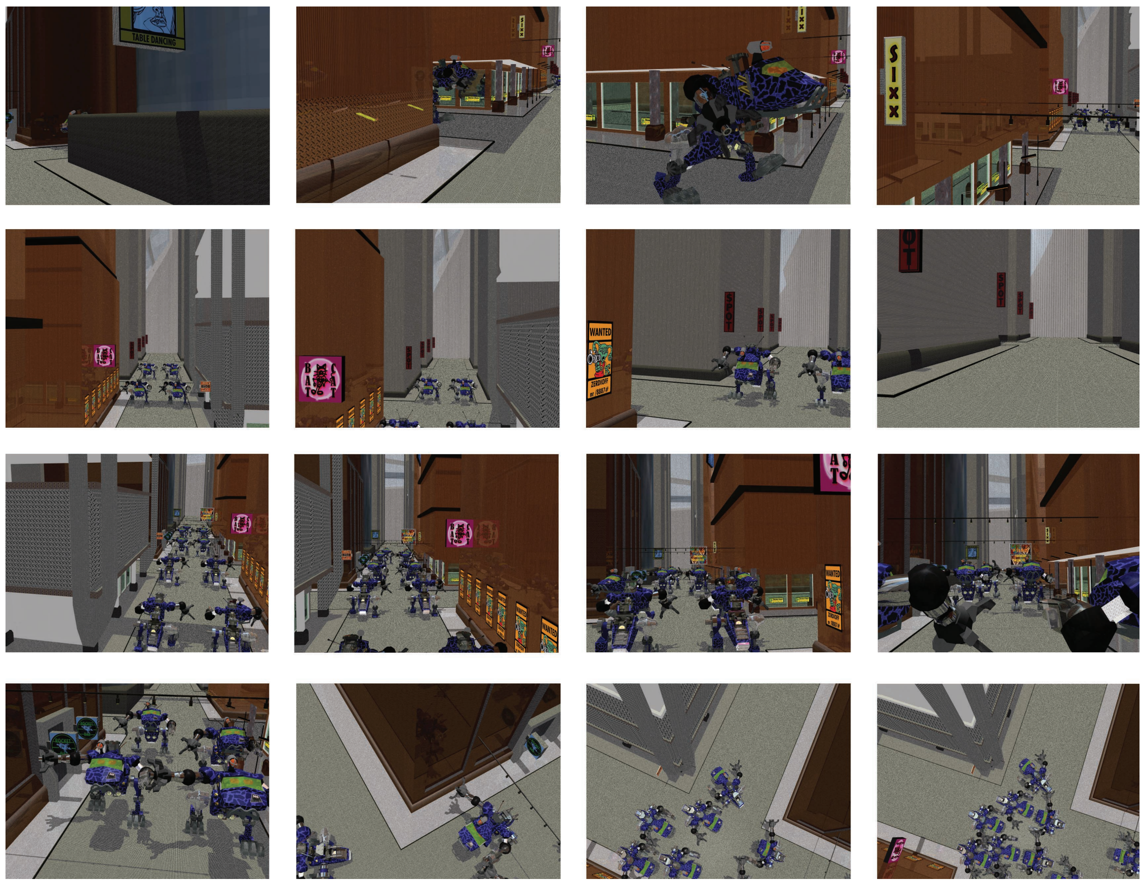
Scenes	Primitives	Texture Memory (Mbytes)	Light(s)	Stresses
Kitchen	110,561	7.5	1 to 10	1,3,4,5,8
Robots	71,708	9.5	1	1,3,5,6,7
Museum	10,215 to 75,687	3.9	2	2,5



1 Frames from the kitchen test scene animation.

modeled using 110,561 polygons. These require more than 15 Mbytes of memory to store. (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 ≈ 15 Mbytes.) Furthermore, we used eight texture maps, ranging in size from 96 Kbytes to 3.1 Mbytes and requiring 7.5 Mbytes of memory.

In all, the kitchen scene might require more than 22 Mbytes 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, there should be the possibility of bounding volume overlap (stress 6). However, this depends on the acceleration data structure and how users apply it to scene objects.



2 Frames from the robots test scene animation.

Robots

The robot scene consists of 10 animated warrior robots and a static downtown environment with skyscrapers, all equaling 71,708 polygons. Each robot consists of 6,249 polygons with 18 moving parts. The city is 9,218 polygons. One light source exists: the Sun. We implemented background lighting by using an ambient contribution. Figure 2 shows snapshots from the robots test scene.

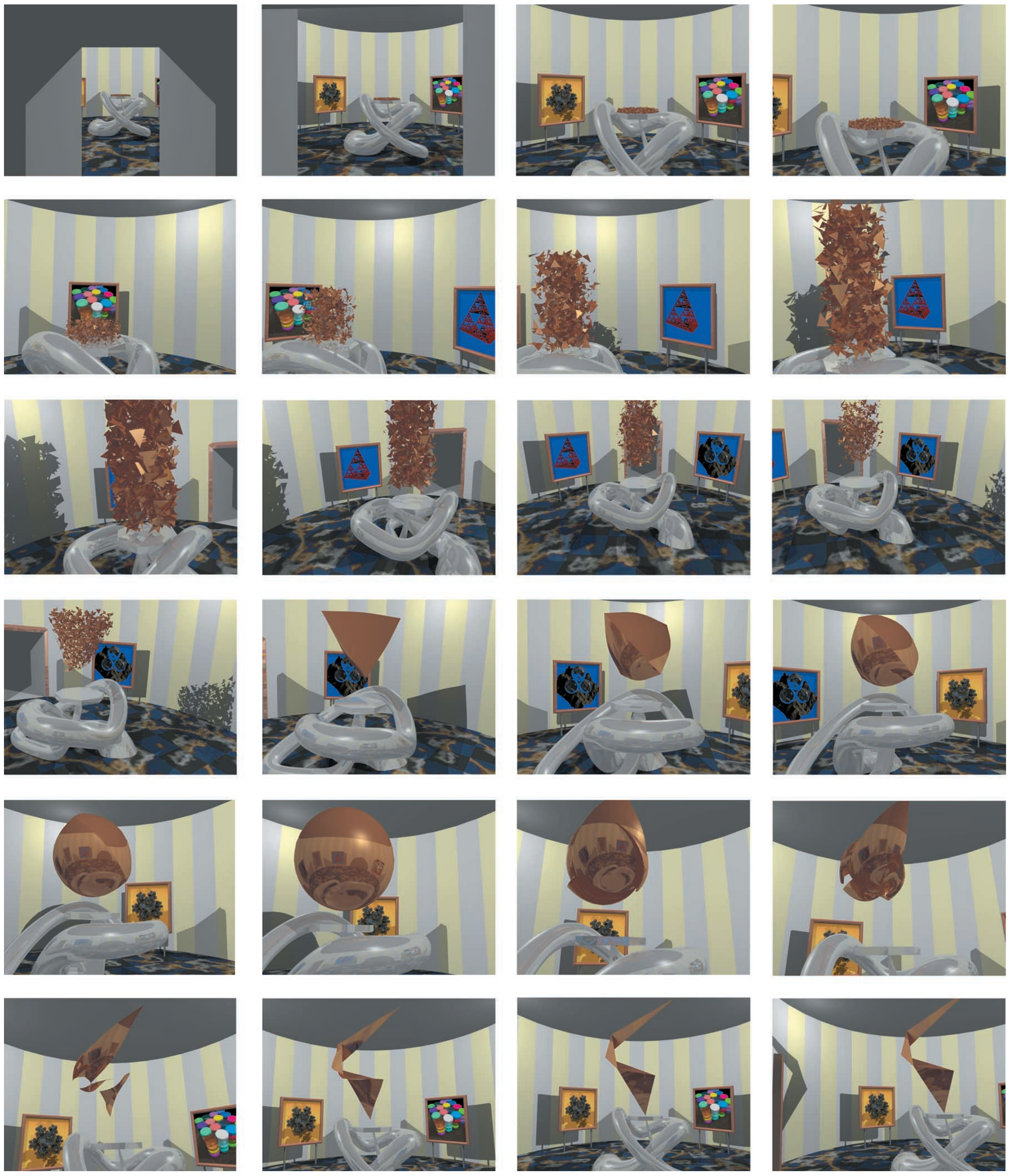
The robots are spread out in the city at start of the animation and walk down the streets to finally gather in the middle of the scene. We implemented stress 7 in this scene by letting the distribution of robots change drastically from fairly balanced at the start to highly unbalanced at the end. This also gives the teapot-in-the-stadium problem (stress 3). The robots' hierarchical animation 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. We implemented stress 6 by the robots' moving parts, because 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 tight fitting, overlap will also occur between different robots when they're clustered at the end of the animation and when we view several robots head on as they march down the street.

Each robot of 6,249 polygons will fit in most L2 caches, but all 10 robots together probably will not. Therefore, in the frames where many or all robots are visible simultaneously, stress 5 should occur. On the other hand, since all 10 robots are identical except for the positions and rotations of their parts, this fact could be used to save memory. Only the information about the transforms must be handled separately. In this way, large scenes can still fit in the caches, which can be essential for speed.

Museum

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

Therefore, the museum scene consists of a small room. The main subject in this room is an animated piece of abstract art, which features several triangle patches 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. Figure 3 shows snapshots of the scene.



3 Frames from the museum test scene animation.

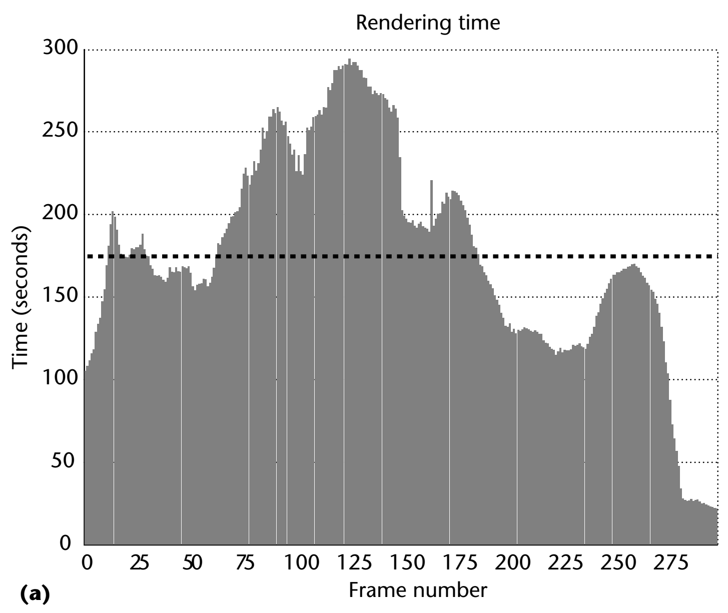
This scene has 10,143 polygons and 8 cones without the animated art in the middle. There are five 512 × 512 RGB textures, which together occupy 3.9 Mbytes of memory. This is a small scene in terms of the number of primitives. To test scenes with different numbers of primitives, we provide different complexity levels of the animated art in the scene. More specifically, six different versions exist, consisting of 2^{2k} animated triangle patches, where $k = 3, \ldots, 8$. This means that the lowest complexity level has 64 triangle patches and the highest 65,536. All these triangles exist at five different times—they're interpolated into five different constellations. The memory usage for the highest complexity level is 45 Mbytes. So, all in all, the highest complexity level of this scene occupies at least 50 Mbytes. Because the complex and highly reflective abstract art and pedestal are fully visible in most of the frames in this animation, stress 5 might occur.

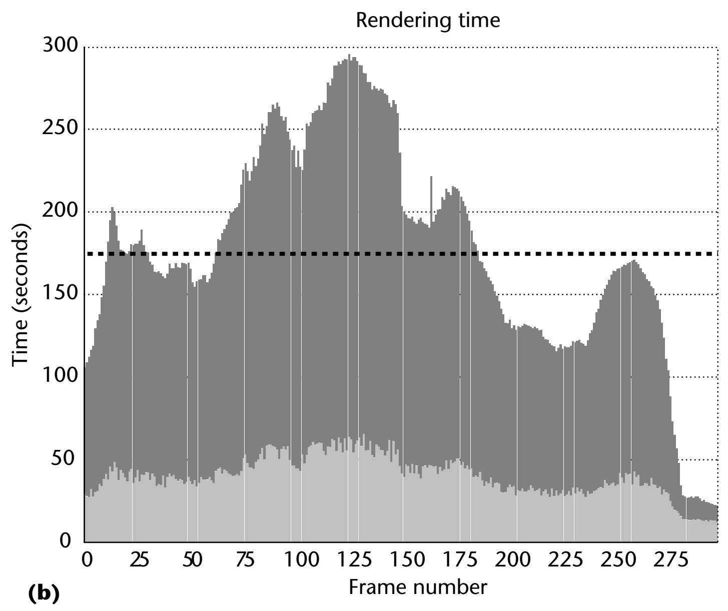
Performance measurements

The main reason for using a benchmark is so people can objectively compare different algorithms' performance. To do that, a benchmark's users must report the same performance measurements, in the same manner.

Measurement report proposal

Here, we give the parameters and some measurements that we propose, together with actual numbers and parameters from rendering of the animated museum test scene.





4 Rendering times as a function of frame number for the museum test scene. We rendered the 300 images of this animation using the publicly available Rayshade ray tracer and measured the time using the Unix time command. Figure 4a shows only the total rendering time per frame. Figure 4b shows what it might look like if we divided it into shading time (light gray) and visibility time (dark gray). The dashed horizontal lines mark the average rendering time.

- Model: museum
- Number of animated frames in the scene: 300
- Number of primitives in the scene: 11,175
- Complexity level: 5
- Resolution of the rendered image (dpi): 800 × 600
- Mode: interactive
- Average time it takes to render a frame: 176.6 seconds
- Worst time it takes to render a frame: 294 seconds
- Deviation: 0.37
- Continuity (optional): 0.18
- Total time it takes to render all frames: 52,966.0 seconds
- Preprocessing time: 0 seconds
- Machine: 333-MHz Sun UltraSparc 10, 128 Mbytes of memory
- Scene memory: 8.5 Mbytes
- Efficiency memory: 0.5 Mbytes

We rendered the scene using the publicly available Rayshade ray tracer (http://www-graphics.stanford.edu/~cek/rayshade), adapted to read our file format, and used a uniform grid with fixed size, which was rebuilt each frame. (Please note that these measurements aren't a serious attempt to achieve good rendering time.) In an interactive or near interactive ray tracer, the worse time parameter would be useful to get an absolute measure of the worst possible frame rate we can expect. Preprocessing time is done once before the rendering of the entire animation starts.

In addition to this information, we can supply a graph that shows the number of rays as a function of the frame number. We can divide this graph into the following classes of rays: eye, reflection, refraction, and shadow rays.

Also, users should give a rendering time diagram as a function of the frame number. As an option, we can divide the rendering time into shading time (the rendering time spent shading and lighting a frame), visibility time (the rendering time spent finding the intersection with the closest object), and rebuild time (the time it takes to rebuild the efficiency data structures each frame). Figure 4 gives example diagrams. Note that the rendering time is the sum of the visibility, shading, and rebuild times. The main reason to include those is mostly for the algorithm developers, who can gain insight about where the bottleneck in their algorithm lies and where best to optimize.

If a BART user compares algorithms, we recommend giving a speed-up diagram of $t_{\text{new}}(k)/t_{\text{old}}(k)$. Here, $t_{\text{new}}(k)$ is the time it took for the new algorithm to render frame k, and $t_{\text{old}}(k)$ 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 peak noise to signal ratio (PNSR) diagram is a function of the frame number, and the average peak noise to signal ratio (APNSR) should also be reported (see the Approximating Algorithms section).

Predetermined versus interactive mode

We can use the benchmark in two different modes: interactive and predetermined. In predetermined mode, the benchmark's users can look into the future—that is, they retrieve information about frames that have not been rendered. For example, assume that frame k is about to be rendered. In predetermined mode, we can investigate the position and the orientation of the camera for, say, frame $k + 1$, $k + 2$, and $k + 3$ (to see if that information can be exploited for faster rendering).

Because of the difficulties in specifying a truly interactive animation (where things usually change based on the input from the system user) that gives the same amount of work each time the benchmark is used, we included a "fake" interactive mode. In this interactive mode, users can't retrieve information about frames that appear after the frame that is currently being rendered. No other differences between the modes exist.

Deviation and continuity

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

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

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

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

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.

We computed the continuity measurement as

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

which 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. (We experimentally verified this by computing both the deviation and continuity for 167-MHz and 333-MHz machines. The observed values were close—the deviation differed by 2 percent and the continuity by 5 percent.) This implies that a researcher can compare deviations and continuity by reading another researcher's paper. In a perfect world, all measurement would be such, which would lessen researchers' burden. Instead, researchers must implement algorithms to compare performance. Unfortunately, all timings—such as average frame time, total time, and so on—can't be made such without losing their meaning.

We recommend that users report both the deviation and continuity. The deviation reports a deviation globally, while the continuity is good because it catches frame-to-frame anomalies, which for frame rates higher than one per second, are distracting to the human visual system. Therefore, if we don't achieve interactive rates, there's probably no use in reporting the continuity.

Approximating algorithms

To render images rapidly or to maintain a constant frame rate, we can use approximating techniques. Examples include reprojection methods[7] and frameless rendering techniques.[8] More algorithms along this line are likely to be developed to meet the need for speed. However, errors can be introduced in the rendered images, so we recommend reporting certain error values when using BART with such algorithms. To do this, we assume that BART users render a reference set of images of the animation without approximating techniques and with the highest possible quality (high antialiasing, high ray depth, and so on) or at least state how they rendered the reference set. Because of differences in shading, antialiasing, and other items in different renderers, we strongly believe that users should generate their own set of reference images. In image analysis and compression, it's common to measure differences between two images using the 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

$$\text{MSE} = \frac{1}{3wh}\sum_{x=0}^{w-1}\sum_{y=0}^{h-1}\left(\begin{array}{l}\left(\mathbf{a}(x,y)_r - \mathbf{c}(x,y)_r\right)^2 \\ +\left(\mathbf{a}(x,y)_g - \mathbf{c}(x,y)_g\right)^2 \\ +\left(\mathbf{a}(x,y)_b - \mathbf{c}(x,y)_b\right)^2\end{array}\right) \tag{4}$$

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. We assume a pixel's RGB components to be in the interval [0,1] and thus that MSE $\in$ [0,1]. The individual color components are accessed as $\mathbf{c}(x, y)_c$, where c could be r, g, or 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

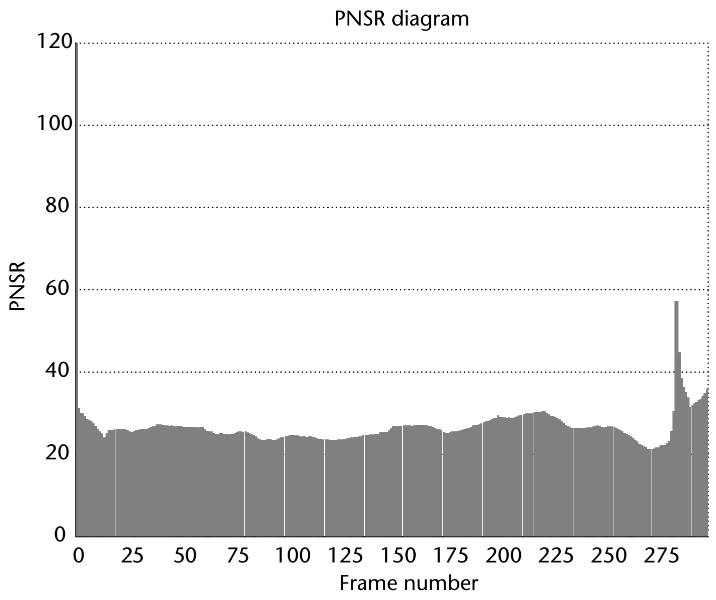
$$\text{PNSR} = 10 \log_{10}(1.0^2/\text{MSE}) = -10 \log_{10} (\text{MSE}) \tag{5}$$

Note that the lower the PNSR, the worse the approximation. There might be some other measure that rewards rendered images that are perceptually of better quality. For example, if an image shifts one pixel to the left, then the 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 PNSR's average, or APNSR, is the PNSR of the average of the MSE of all images in an animation:

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

where n is the number of images in the animation, and MSE_i is the mean square error for image i.

A 0.0 PNSR means the maximum possible error, which might occur when all pixels in an image are black when they should be white. A totally correct image implies a PNSR value of infinity, which is unreasonable to draw in a diagram. Assume, therefore, that we render an image of 1280×1024 pixels with 8 bits per color



5 A PNSR diagram for a rendering of the museum scene. We rendered the 300 images of this animation using the publicly available Rayshade ray tracer and calculated a PNSR value for each frame, reusing every fourth pixel from the previous frame. This made for faster rendering, but worse image quality. Note that only the first image has a PNSR equal to 120, which means that this is the only image that was 100 percent correct.

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 PNSR $\in$ [0,120], where we assume that every value above 120 represents an error-free image. The value 120 can be adjusted for other resolutions.

When using approximating algorithms, we recommend reporting APNSR and a diagram of PNSR$_i$ as a function of the frame number i, where $i \in$ [startframe, stopframe] to see how good the approximation really is, which is something the computer graphics community has previously neglected. Thus, when comparing algorithms fairly, these are important measurements. See Figure 5 for an example of a PNSR diagram.

Implementation notes

We placed the three BART test scenes in the public domain. We wanted to keep the file format as simple as possible and yet reasonably flexible. Therefore, we enhanced the neutral file format[9] in several ways to handle animated scenes. We describe the new format—Animated File Format (AFF)—on our Web site, http://www.ce.chalmers.se/BART. We used Kochanek–Bartels splines to animate the viewer and the objects. (We prefer this approach to storing a separate matrix for each transform for each frame, because it lets us easily change the number of frames in an animation.) This included rotations, translations, and scalings hierarchically organized. As a result, there's a notation of objects in AFF, because an object is usually grouped under a transform (static or animated). We used Eberly's implementation (from http://www.magic-software.com). We chose this approach, because it's expensive for interactive programs to read files between frames and it is simple.

In addition to the test scene files, we provide BART users with

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

Future work

It's 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 include parametric patches (such as 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—that is, scenes that are a few magnitudes larger than those in BART. Furthermore, some of the interactive ray tracers[1,2] require powerful parallel computer systems to achieve interactive rendering speeds. Since parallel computer systems most likely will be used in the future, we could add special scenes that stress multiprocessor ray-tracing algorithms.

The goal for BART users is real-time ray tracing with a constant frame rate, and it will be exciting to see when this will happen and what kind of algorithms they will use. Finally, we encourage everyone to participate in BART's usage and development, which we hope will grow and evolve over time. ■

Acknowledgments

We thank Joachim Helenklaken and Jens Larsson for help with modeling and Eric Haines and Per Stenström for support during this project.

References

1. M.J. Muuss, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models," *Proc. Ballistic Research Lab Computer-Aided Design Symp. 95* (BRL-CAD), 1995.
2. S. Parker et al., "Interactive Ray Tracing," *Symp. Interactive 3D Graphics*, ACM Press, New York, 1999, pp. 119-126.
3. K.S. Klimaszewski and T.W. Sederberg, "Faster Ray Tracing Using Adaptive Grids," *IEEE Computer Graphics and Applications*, vol. 17, no. 1, Jan./Feb. 1997, pp. 42-51.
4. A. Glassner, *An Introduction to Ray Tracing*, Academic Press, London, 1989.
5. E. Reinhard, B. Smits, and C. Hansen, "Dynamic Acceleration Structures for Interactive Ray Tracing," *Proc. 11th Eurographics Workshop on Rendering*, Springer-Verlag, New York, 2000, pp. 299-306.
6. E. Haines, "Spline Surface Rendering, and What's Wrong with Octrees," *Ray Tracing News*, vol. 1, no. 2, Jan. 1988, http://www.raytracingnews.org.
7. S. Badt, Jr, "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing," *The Visual Computer*, vol. 4, no. 3, Sept. 1988, pp. 123-132.

8. G. Bishop et al., "Frameless Rendering: Double Buffering Considered Harmful," *Computer Graphics Proc.* (Siggraph 94), ACM Press, New York, 1994, pp. 175-176.
9. E. Haines, "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Applications*, vol. 7, no. 11, Nov. 1987, pp. 3-5.

Jonas Lext *is a PhD student in computer graphics at Chalmers University of Technology, Sweden. He has an MSc in engineering physics from Chalmers University of Technology. His research interests include interactive ray tracing, parallelization of computer graphics algorithms, and hardware–software interaction.*

Ulf Assarsson *is a PhD student in computer graphics at Chalmers University of Technology, Sweden. He has an MSc in engineering physics from Chalmers University of Technology. His research interests include real-time computer graphics, ray tracing, and parallel algorithms.*

Tomas Möller *is an assistant professor at Chalmers University of Technology, Sweden. He has an MSc in computer science from Lund Institute of Technology, Sweden, and a PhD from Chalmers University of Technology. He is the coauthor of Real-Time Rendering (AK Peters, 1999) with Eric Haines. His research interests include realistic and rapid real-time rendering, interactive ray tracing, and algorithms for future graphics hardware.*

Readers may contact Lext at Chalmers University of Technology, Dept. of Computer Eng., SE-412 96, Göteborg, Sweden, email lext@ce.chalmers.se.