Faster Ray Tracing Using Adaptive Grids

Krzysztof S. Klimaszewski
Cimetrix

Thomas W. Sederberg
Brigham Young University

A new hybrid approach outperforms the regular grid technique in scenes with highly irregular object distributions by a factor of hundreds, and combined with an area interpolator, by a factor of thousands.

Efficient ray tracing has been a contradiction in terms since its introduction as a computer version of ideas found in Dürer's *Underweysung der Messung* (1525) and Descartes' *La Dioptrique* (1637). The most effective acceleration techniques developed to reduce ray tracing's high computational cost are based on space coherence: bounding box hierarchies and space subdivision.[1] During preprocessing, a space subdivision algorithm associates objects with the space elements in which they reside. During ray runtime, a ray traverses the volume elements and is tested for intersection only with the objects inhabiting those voxels.

Partitioning 3D space into a regular mesh of voxels has attracted much attention. Fujimoto et al.[2] showed that in realistic scenes the method combined with an incremental ray traversal outpaces the adaptive octree approach by an order of magnitude. While octrees avoid excessive space subdivision in empty and sparsely populated regions, they require vertical traversal for detecting neighboring cells—a costly process even with improved methods, such as the Digital Differential Analyzer (DDA) octree traversal algorithm. The regular grid subdivision does not require vertical traversal, but incurs an unnecessary overhead of voxel-to-voxel steps in empty regions.

To deal with the problem, Devillers[3] introduced *macroregions*, collections of axis-aligned boxes that replace empty voxel sets. Each empty voxel, possibly residing in several macroregions, points to the optimal macroregion, enabling the ray entering the voxel to leap over the largest number of empty voxels quickly.

Yagel et al.[4] proposed a 3D raster ray tracer (RRT) that operates on a set of unit voxels associated with at most one object. To achieve a reasonable image quality, the voxel footprint has to be approximately pixel size. Even though now a ray-object intersection is found as soon as a ray enters a nonempty voxel, the method requires more than one billion voxels to achieve 1K resolution. This results in staggering memory requirements and an increased ray traversal cost.

Cohen and Sheffer[5] mitigated the latter problem with *proximity clouds*—a method that stores in each subdivision cell the distance to the nearest nonempty cell—improving the RRT's performance by 30 percent in simple sparse scenes.

The regular grid is conceptually simple, efficient, and easy to implement. However, it cannot meet the contradictory requirements of sufficient space decomposition in the densely populated areas and the need to avoid excessive partitioning in a scene's void regions. Although alternative solutions that mix space subdivision types accelerate ray tracing to some degree, they might present challenges such as maintenance, readability, reliability, and code size problems. In this article we attempt to alleviate the uniform grid's weaknesses while capitalizing on its strengths.

Best of both worlds

Our basic idea nests regular grids hierarchically. Thus, when a ray enters a voxel, the algorithm first computes intersections between the ray and all objects in the voxel. If an object is a grid, the ray proceeds into that grid to check for additional intersections. Figure 1 illustrates this concept.

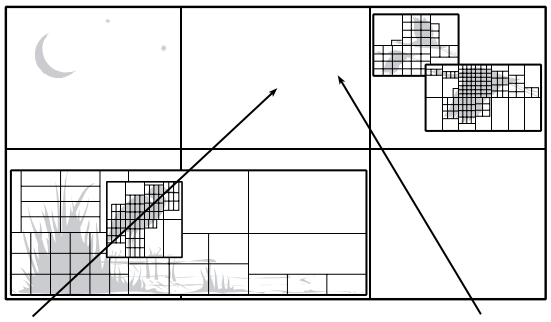
Snyder and Barr[6] were first to introduce a ray-tracing uniform grids hierarchy. Although some of their improvements became standard ray-tracing systems components, the hierarchical grids concept passed unnoticed because no one investigated the actual impact of hierarchical grids on rendering times. Snyder and Barr's work presented several questions, such as how to decide each grid resolution or how to organize complex models for efficient ray tracing. In this article, we try to fill these gaps and expand upon the multiple grids concept.

The structure in Figure 1 enables the ray on the right to travel rapidly through a scene's empty areas. If a ray, such as the one on the left, enters more complex scene regions, an acceleration structure—a local grid bounding an object or a group of objects—aids it. But is this

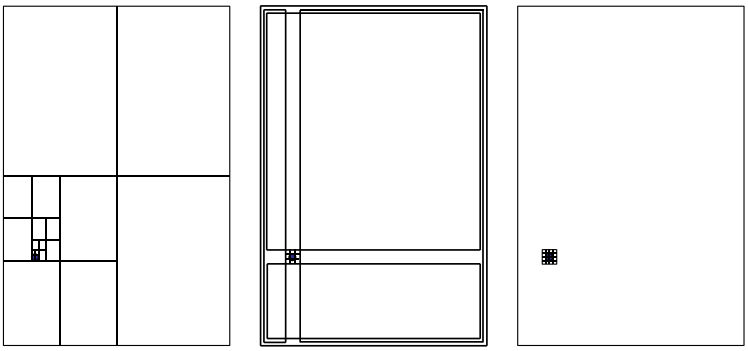
0272-1716/97/$10.00 © 1997 IEEE

grid type any better than an octree or a macroregion (see Figure 2)?

An octree might require many subdivisions to isolate an object. This results in a deep hierarchy and an expensive vertical ray traversal. Moreover, octrees suffer from the "teapot in a stadium" problem, in which large size disproportions between rendered objects result in deep trees and insufficient space decomposition. The macroregions' strength apparently stems from the lack of hierarchy. Nevertheless, macroregions still have to be entered and exited. Since they can overlap, you have to make a correct decision regarding which one to use as a vehicle for passing through the scene. The uniform grid applied locally in Figure 2 acts as both a bounding box isolating objects from a scene's void and a means of space decomposition in the object's vicinity (which might contain many smaller primitive objects). The grid's resolution can be set as the local conditions demand, while macroregions preserve the original grid's resolution. Further, a local grid bounding volume is the minimum, thus a more efficient, axis-aligned bounding box. Again, macroregions produce larger bounding boxes based on the underlying grid.

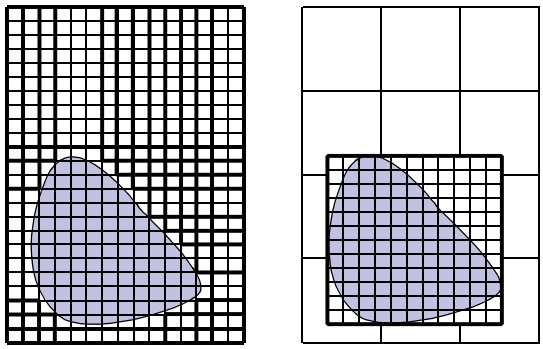


1 Adaptive grids.



2 Comparing (a) the octree, (b) macroregions, and (c) adaptive grids.

As Figure 3 illustrates, macroregions applied to a scene with slanted surfaces often result in an excessive number of empty overlapping areas. In our method, local uniform grids can also overlap. However, the number of grids does not depend on an object's shape or orientation. Local grids thus combine the octrees' adaptability with the macroregions' efficiency in empty areas. They also increase the uniform grid's efficiency in densely populated areas.



3 Overlapping of macroregions and grids.

Spatial associations

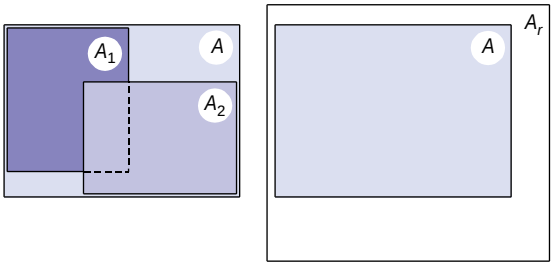
Because local grids act as bounding boxes, structuring them hierarchically is advisable. To create efficient hierarchy grids, we would put them in densely populated areas during preprocessing. Devillers[3] proposed a method to detect such areas. The technique, however, is difficult and expensive, and as we said before, does not produce the most efficient bounding boxes. Many design systems ensure that modeler grouping and hierarchies are based on spatial relationships and proximity, which aids efficient ray tracing. However, while tree nodes usually contain spatially associated objects, the hierarchy itself can be unsuitable for fast ray tracing, having been generated primarily to simplify modeling of the scene. Therefore, we need to organize these nodes into a new hierarchy that lends itself to fast ray tracing.

Grid hierarchy and its serendipity

To ensure that the local grids' bounding box function performs well, we apply a grids hierarchy by using Goldsmith and Salmon's algorithm.[7] In this approach, the criterion of the minimum total bounding volume surface area guides the structure's construction. The algorithm addresses the question of generating an efficient bounding volumes hierarchy, given a set of objects or their clusters. The problem of regrouping and changing the number of object clusters obtained from a modeler to improve performance has not been solved. Since the number of possible hierarchical arrangements grows exponentially with the number of objects, exhaustive search is usually impractical. This issue becomes even more complex when we attempt to convert the bounding boxes surrounding the clusters into uniform grids.

After finding a low-cost bounding box hierarchy, we can increase the speed of finding intersections with the objects by voxelizing the boxes. This leads to a uniform grids hierarchy in which grids are treated as regular primitive objects and appear on the lists of objects residing in the parent grids' voxels. The objects enclosed in a grid are listed only within this immediate grid and are not visible to the parent grid.

For most scenes, this approach gives significantly bet-



4 Bounding box merging. (a) Replacing two candidate bounding boxes with (b) a more efficient bounding box for optimal voxelization.

ter timings than the standard regular grid algorithm. Nonetheless, through simple experiments we find that different object groupings in the same scene—consequently, different grid numbers, sizes, and resolutions—result in different speedups. We made a serendipitous discovery that substituting grids containing object clusters close to one another with one larger grid usually improves the rendering speed. Our goal is to avoid large disparities among voxel populations, or reduce a scene's nonuniformity, which is a ratio of standard deviation over the average voxel occupancy.

Grid merging

To make merging close grids efficient, we conditionally replace two candidate bounding boxes with a new one before voxelization (hence, grid merging is merely box merging). The new bounding box surrounds the two dependent boxes tightly. To qualify the candidates, we require that the proposed new bounding box's surface area, A, be smaller than a prescribed fraction, f, of the sum of the surface areas of the two boxes to be merged, A_1 and A_2 (Figure 4a):

$$\frac{A}{\left(A_1 + A_2\right)} < f \qquad \textbf{(1)}$$

A bounding box with only a small void area between itself and another bounding box causes most rays entering the parent box to also hit the child box. Thus, the inner box intersection calculations are wasted. Accordingly, we ensured that the new grid is sufficiently smaller than the root grid of area A_r (Figure 4b):

$$\frac{A}{A_r} < m \qquad \textbf{(2)}$$

The f and m factors may have to be found experimentally for implementations supporting different primitive types or utilizing different ray traversal schemes. Since optimum grid formulas involve times to perform certain runtime operations, optimum values of f and m may vary from machine to machine.

We ensured that individual bounding boxes were efficient by removing the underpopulated ones. Otherwise, the cost of intersecting a ray with a box would be close to the cost of intersecting the object inside, and the box would reduce processing speed. Although we removed the boxes containing one element, systems with a rich variety of primitive objects might rate them according to their individual costs and declare "underpopulation" at different thresholds.

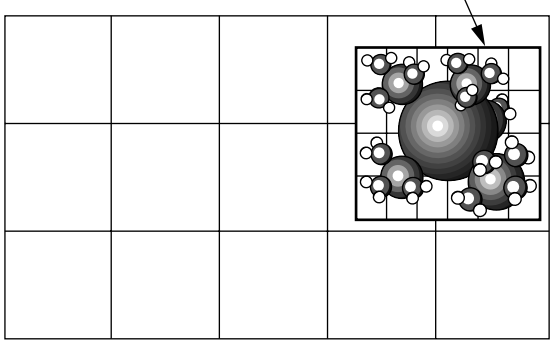
In certain instances, underpopulated bounding boxes prevail, while their contents en masse form spatially related and densely populated clusters. Removing all bounding boxes in this instance would result in a structure as inefficient as the object arrangement the process began with. Therefore, we mark the objects whose bounding boxes were deleted as "orphans." The orphan objects are then placed in an additional bounding box, which later will be fed into the box merging process along with all other bounding boxes. The resulting *orphanage grid* (Figure 5) is usually more efficient than the larger existing grids to which the orphan objects would otherwise have to be added.

After merging all close bounding boxes, we tested the surviving ones to check if they contained any other bounding boxes. If a box tests positively, we verify that its potential child is sufficiently smaller than the box. In general, we want to preserve boxes that are small compared to the surrounding box and get rid of larger ones. This means that the parent bounding box with surface area A might have to be merged with its child, if

$$\frac{A_c}{A} > m \qquad \textbf{(3)}$$

where A_c is the child box's surface area and m is the embedding factor from Equation (2). Every box removal makes the structure less complex and potentially less deep. Since the calculations of the area of an x_1 by x_2 by x_3 bounding box may be repeated many times during the hierarchy's construction, the area ratios in equations 1 through 3 can be computed most efficiently if we use the half area formula $x_1x_2 + (x_1 + x_2)x_3$.

The next step is connecting the remaining bounding boxes into a hierarchy through Goldsmith Salmon's algorithm, after which individual boxes are voxelized. The result is a uniform grids hierarchy. Each grid is now well adapted to local conditions, since its size is



5 Orphanage grid. Many objects left without bounding boxes after removing underpopulated extents "coagulate" into spatially related clusters. A new common bounding box should be made for them and later undergo the merging process with all other bounding boxes.

determined by the objects' extent and its resolution by the number of objects enclosed by the grid. A leaf grid usually contains many objects—none of them visible to any other grid. The objects are distributed more evenly, which speeds up ray-object intersections. Because objects are hidden in subgrids, grids higher in the hierarchy split into few voxels. This expedites fast ray traversal through empty scene regions. The whole process—performed during the preprocessing phase—is view independent. Therefore, as long as the objects remain unmodified, the same grid hierarchy can be used for multiple renderings. A hierarchy thus created usually has very little in common with the structure built by the user during the modeling process; it is suitable for rapid ray tracing. Figure 6 outlines the algorithm. A few passes through the list of bounding boxes are necessary, but the cost of the operation is not too high because the list shortens significantly after merging boxes that are in close spatial proximity to one another.

```
FOR all nodes of modeler hierachy
     surround node with bounding box
        /*Create unstructured grids*/
FOR all bounding boxes
     merge close boxes
        /*Create structured grids*/
FOR all remaining bounding boxes
     insert box into tree using minimum
        surface criterion (Goldsmith
        and Salomon 1987)
     IF box surface area is too large OR
        box is underpopulated merge box
        with its parent
FOR all bounding boxes in hierarchy
     voxelize box
```

6 Pseudocode for creating an adaptive grids hierarchy.

Heterogeneous grids

Since tessellation speeds up ray tracing procedural surfaces, we can assume that distributing polygons over a surface will be relatively uniform and that the polygon sizes will be similar. Consequently, distributing the polygons will cause most of them to face the largest sides of a bounding box. More voxels will be needed along the longer edges of the bounding box, as shown in Figure 7.

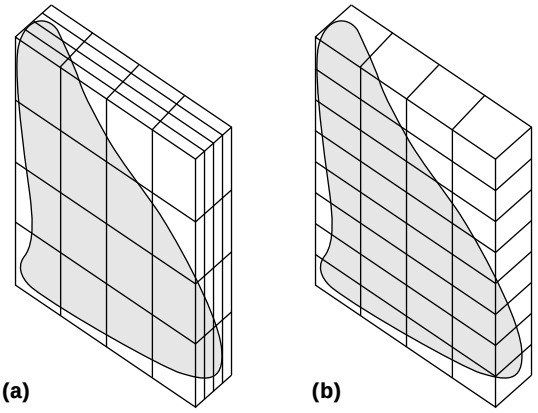
This tendency is ignored by the popular formula

$$N = \sqrt[3]{n} \qquad \textbf{(4)}$$

to calculate the total number of voxel rows, N, in a grid containing n objects. Let us assume that the voxel allocation is proportional to the lengths of the edges, x_1, x_2, and x_3 of the grid. Under such conditions, the respective numbers of voxels along these edges, N_1, N_2, and N_3, are computed as

$$N_3 = \left\lceil \sqrt[3]{\frac{nx_3^2}{x_1 x_2}} \right\rceil$$
$$N_2 = \left\lceil \sqrt{\frac{nx_2}{N_3 x_1}} \right\rceil$$
$$N_1 = \left\lceil \frac{n}{N_2 N_3} \right\rceil \qquad \textbf{(5)}$$

where square brackets denote a round-up operation. These heterogeneous grids also help avoid numerical problems due to excessively subdividing very thin local grids and often reduce ray-tracing time. Rarely can so much be gained in the rendering realm for so little. Replacing regular grids with heterogeneous grids affects only the ray initialization and traversal routines with minor modifications. The effect that heterogeneous grids have on rendering selected scenes is shown in Table 1 (next page), in which heterogeneous grid performance is compared with the performance of an identical local grid hierarchy of homogenous grids.



7 (a) Homogeneous grid and (b) heterogeneous grid.

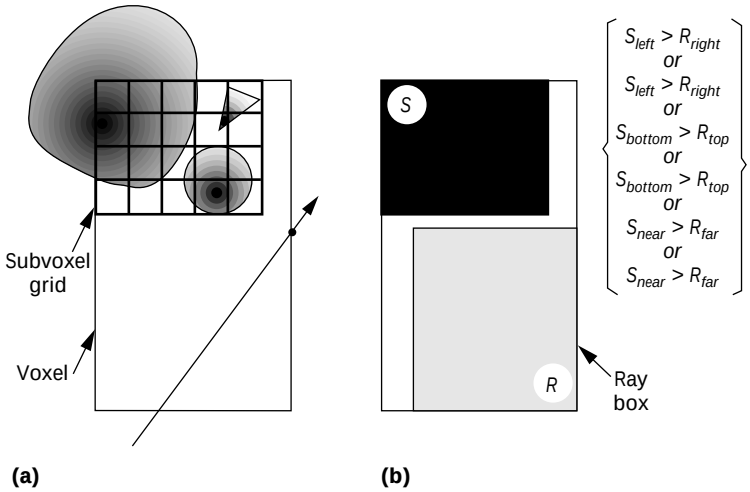
Subvoxel grids

Even with grids applied locally in a hierarchical fashion and using heterogeneous grid subdivision, some voxels may remain overpopulated. Rays in such voxels will spend too much time searching for only one visible intersection. To equalize the object distributions there, we insert *subvoxel grids*, an improved version of the adaptive voxel subdivision proposed by Jevans and Wyvill.[8] Instead of stretching an overpopulated voxel's grid limits to the boundaries, we can convert only the bounding box of the objects residing in the voxel into a grid, as illustrated in Figure 8 (next page). Subdividing only the box in lieu of the whole voxel increases the ratio of nonempty voxels to empty voxels and improves the whole grid structure's adaptability. The ray box defined by the two points at which a ray pierces the current voxel quickly determines if a ray misses a subvoxel grid. If one of the six comparisons shown in Figure 8b gives a positive result, all the objects inside the subvoxel grid are trivially rejected. Otherwise, a more accurate ray-box

Table 1. The influence of heterogeneous grids on ray-tracing time.

	Heterogeneous Grids	Empty Voxels (%)	Nonuniformity	Tracing Time (mm:ss)	Speedup of Previous Row
Bezier patch*	No	86.2	2.8	1:54	—-
	Yes	25.7	0.7	0:49	2:31
F15	No	91.5	6.1	2:42	—-
	Yes	90.0	5.5	2:21	1.15
Teapot	No	81.4	3.9	1:28	—-
	Yes	79.9	3.5	1:19	1.12
Mirrors	No	89.8	6.1	29:14	—-
	Yes	79.8	5.2	22:43	1.29

*Bezier patch is a flat tensor product patch parallel to the projection plane and tesselated into 1,922 triangles. The other test scenes contain very few axis-aligned elements.



8 (a) Subvoxel grid and its culling using (b) a ray box.

intersection test is performed. The ray box can be dynamically shrunk to the box defined by the ray's entry point and the nearest intersection point each time the latter has been found inside the voxel box. The technique is applicable to local grids as well.

```
FOR each grid in the local hierarchy
     recursion_level = 0
     GenerateSubvoxelGrids
       (grid, recursion_level)

procedure GenerateSubvoxelGrids (grid, level)
level = level + 1
FOR each voxel of grid
     IF the number of objects in the voxel
          > Max_Objects
        determine the extent of the objects in
          the voxel
        subdivide the object extent into a
        heterogeneous voxel_grid (Eq. (5))
        IF level < Max_Levels
           GenerateSubvoxelGrids
             (voxel_grid, level)
```

9 Pseudocode for recursive subvoxel grid decomposition. Max_Objects is the maximum number of objects per voxel. Max_Level is the maximum depth of the voxel grid tree.

Obviously, a subvoxel grid is a ray-tracing microcosm in its own right. All that we have said about tracing rays in regular grids holds here too, including local, heterogeneous, and subvoxel grids—all of which could be generated recursively. However, the enhancements applied on the hierarchy's higher levels have already alleviated the problems that might call for similar measures within subvoxel grids. Experiments show that one subvoxel grid generation accelerates ray tracing in the local grids' environment up to 50 percent. Next generations, however, contribute much less and incur severe memory and preprocessing time penalties. Figure 9 outlines the subvoxel grid enhancement.

Caspary[9] concluded that for the octree, five objects per cell was the optimum. The results shown in Table 2 indicate that adaptive grids are more memory-efficient, since as larger speedups are achievable when more than five objects populate a voxel. The rows in Table 2 contain the results obtained for the same local grids' hierarchy, each with a different maximum number of objects per voxel and/or a different number of subvoxel grid generations (Max_Objects and Max_Levels in Figure 9, respectively). The first row contains the results obtained when no subvoxel grids were generated. The gain factor in the last column was computed with respect to the first row's timings.

Results

The adaptive grid algorithm was tested on a Hewlett-Packard 730CRX24Z workstation with a 66-MHz clock, and 96 Mbytes of main memory, rated at 23.7 Linpack dou-

Table 2. The influence of sobvoxel grids on ray tracing. The scene is Teapot.

Max Object Per Voxel on Level:			Voxel Grids on Level:			Total Number of Voxels	Nonempty Voxels (%)	CPU Time (seconds)		Gain Factor With Respect to First Row
1	2	3	1	2	3			Preprocessing	Tracing	
—	—	—	—	—	—	7,592	20	1.1	77.4	—
8	—	—	814	—	—	24,709	51	1.8	54.7	1.42
8	8	—	814	3,255	—	57,445	69	6.5	53.3	1.45
8	8	8	814	3,255	5,090	104,692	77	20.0	52.5	1.47
5	—	—	954	—	—	25,491	51	1.8	57.2	1.34
2	—	—	1,415	—	—	26,567	52	2.0	65.8	1.18

Table 3. The efficiency of the adaptive grid algorithm.

	Number of Primitives n	Regular ($n^{1/3} \times n^{1/3} \times n^{1/3}$) grid			Adaptive grids			Total Gain Factor	Runtime Gain Factor
		Non-uniformity	Pre-processing	Ray Tracing	Non-uniformity	Pre-processing	Ray Tracing		
Flake	7,382	19.5	0:02	45:14	1.2	1:32*	1:14	16.4	36.7
F15	7,021	22.5	0:01	29:05	1.9	0:02	0:38	43.7	46.0
Teapot	7,742	11.6	0:01	15:47	1.5	0:02	0:55	16.6	17.2
Museum	11,512	66.5	0:02	35:24	1.7	0:04	2:38	13.1	13.4
Mirrors	123,008	14.0	0:16	54:48	1.7	0:43	10:47	4.8	5.1
Car	86,155	32.5	0:11	1:19:08	1.5	0:24	2:13	30.3	35.7
Eagles	85,412	104.1	0:18	15:51:08	4.5	0:26	1:25	514.3	671.4
Space	103,699	125.2	0:13	13:24:35	3.0	0:27	3:33	201.2	226.6

*Worst case—reduces to 0:04 for best case. CPU time shown in the hh:mm:ss format.

ble Mflops, or 76.0 Dhrystone MIPS. To compare our method with other space subdivision methods, we computed a gain factor as a ratio of a timing for our method and a timing for the regular grid method. We chose regular grid subdivision as a benchmark for the new algorithm because numerous comparisons between the regular grid and other methods existed, making an indirect comparison possible. Both algorithms incorporated exactly the same enhancements wherever applicable. The regular grid method was derived from an optimized production code; the two algorithms access the same traversal code, requiring two additions and two compares to step from one voxel to another. All pictures were generated at 512 by 512 pixel resolution.

The eight test images shown in Figures 10 through 17 on the following pages represent a large variety of scene complexity, both in terms of the number of objects and their spatial distribution. For example, if compiled into a 10 by 10 by 10 grid, Teapot (Figure 12) and Space (Figure 17) differ in their object "dispersion" over the voxels by a factor of 1 to 442. With the exception of Space, the scenes are characterized by full screen coverage, that is, the images contain no background pixels.

The results are presented in Table 3. Compared to the regular grid, adaptive grids equalize voxel occupancy, even though they slightly increase (double, on the average) the mean voxel population. At the same time, however, the algorithm reduces the dispersion, symmetry, and peakedness of the object distribution, mathematically characterized by variance, skewness, and kurtosis, respectively. More important, the distribution's nonuniformity diminishes dramatically.

The same arbitrarily chosen conditions were applied to all the scenes. Local grid merging was performed in accordance with Inequalities 1 through 3, at $f = 2.0$ and $m = 0.1$. One level of subvoxel grids was created inside the voxels containing more than 12 objects (Max_Levels = 1 and Max_Objects = 12 in Figure 9).

The adaptive grid algorithm resulted in a significant speedup. The most sparse scenes, Space (Figure 17) and Eagles (Figure 16), manifested the largest improvement. Now they can be rendered nearly as fast as the scenes with ten times less objects (for example, Museum (Figure 13) and Flake (Figure 10)). The speedup is smaller for the more compact scenes, since they contain a large number of equally sized and relatively uniformly distributed objects in close proximity to one another. An example of such a scene is Mirrors (Figure 14), with its finely triangulated parametric patches congregating closely in a tight space.

Results in perspective

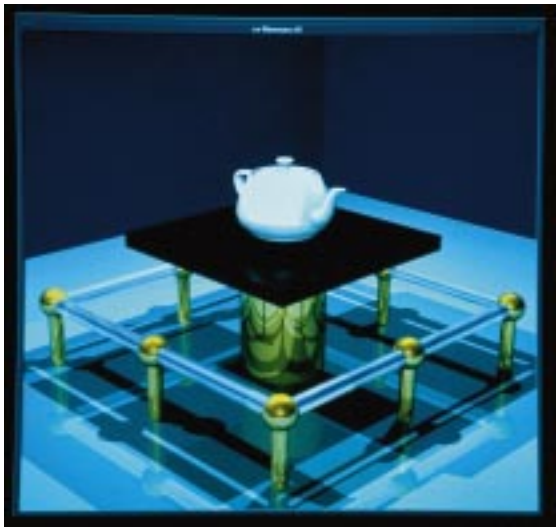
Flake enjoys a special place among the data sets used to test various programs and hardware configurations supporting ray tracing. Along with the Utah teapot, it is the image seen most frequently in ray-tracing publications. By comparing the adaptive grids' efficiency with Flake's best results, we will be able to see our own results in a better perspective.

Flake is an example of a scene with a large preprocessing expense (see Table 3). In this scene, each of the 7,381 spheres is put into a separate node in the model-

10 Flake.

11 F15.

12 Teapot.



13 Museum.

ing hierarchy. Our algorithm merges most of the nodal bounding boxes while testing if each new larger box should be merged with the other boxes. The process is time consuming. However, the modeling structure used as the starting point is an unlikely arrangement, since all the spheres form one cluster. As we pointed out earlier, most modelers would place the objects in a few nodes, or even in a single node. Therefore, Flake's preprocessing time is a worst-case timing. When all the spheres are placed in one node, the scene's spatial decomposition takes only 4.5 seconds. Although we have succeeded in making the adaptive grids' organization largely independent of the starting object hierarchy, the time it takes to create it remains the original structure's function.

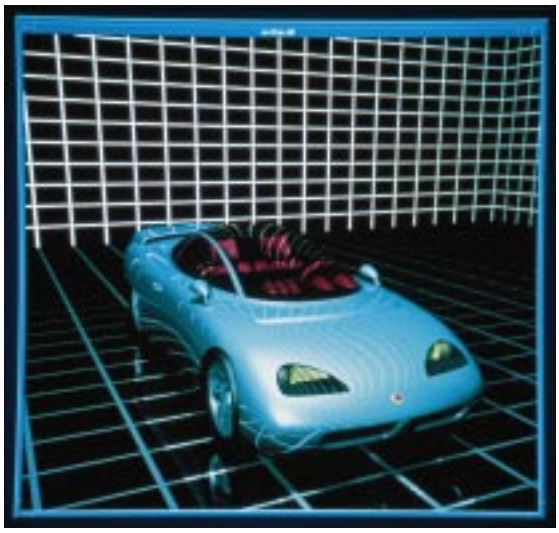
Extended preprocessing times are not unusual for accelerated ray tracers. Yagel[4] found that the preprocessing time in RRT was linear in the number of objects. Yagel's example lets us compute the time needed to voxelize Flake using the RRT method for the required 512^3 resolution as up to five times (424 seconds) longer than the time required for the adaptive grid method's worst case.

Unlike Devillers[3] and Yagel,[4] we did not tweak the data in Flake. Devillers explained the dramatic speed improvement he achieved for this scene—from more than 16 hours to 43 minutes—solely by making the background plane smaller. This is an expected result in light of what we have said about the detrimental influence scene sparseness has on ray tracing's efficiency. Reducing the background plane in Flake or removing it completely (Yagel) is equivalent to decreasing the scene's sparseness. Nevertheless, we attempted to render Flake under conditions similar to those in Yagel's paper. The results appear in Table 4.

The 91-sphereflake (Figure 18) and 820-sphereflake (Figure 19) duplicate the scenes shown by Yagel. We simulated the case of the 7,381-sphereflake without the support plane using the recursive structure in Figure 10. The computing platform we used was 3.8 times faster than the

14 Mirrors.

16 Eagles.



15 Car. Designed by Gary Morales using Evans & Sutherland's CDRS software.

17 Space.

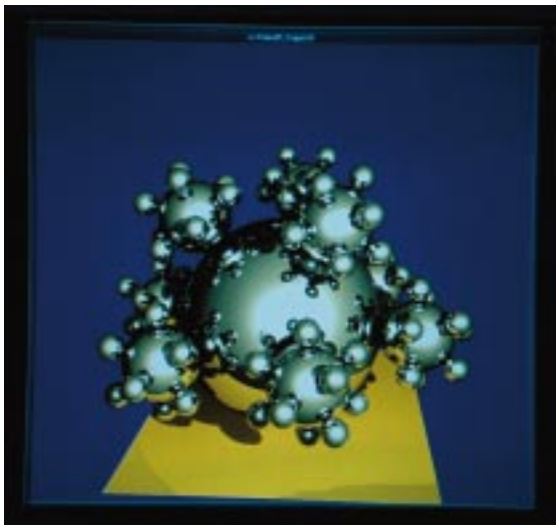
computer on which RRT was tested, as reflected in the "Scaled" column of Table 4. The gain factor resulted from comparing the scaled results with the RRT's speed. In all the test cases the preprocessing was much shorter with adaptive grids than with RRT. The three images contain many background pixels and only one compact object cluster. Such scenes are seldom found in the real world, and the adaptive grid approach is not designed for them. In spite of that, our results suggest that adaptive grids outperform RRT by a substantial margin even in an unfavorable environment. Additionally, the new algorithm exhibits a higher degree of scene independence: While the image in Figure 19 on the next page required some 40 percent more time than the image in Figure 18, the same difference in the case of RRT exceeded 300 percent.

Proximity clouds[5] intended to accelerate RRT by skipping a 3D grid's empty regions would not have had much impact on the comparison's outcome. The 30-percent speedup reported for proximity clouds was achieved with scenes whose volumetric sparseness resembles that of

Table 4. The efficiency of raster subdivision and adaptive grids in a compact Flake.

Scene	Image	Screen Resolution	Rendering Time (seconds) Raster Subdivision	Adaptive Grids Unscaled	Adaptive Grids Scaled	Gain Factor
91-sphereflake	Figure 18	320 × 320	85.0	10.7	40.7	2.1
820-sphereflake	Figure 19	320 × 320	352.0	15.0	57.0	6.2
7,381-sphereflake	Figure 10*	256 × 256	38.4	9.1	34.6	1.1

*Modified—without the background plane.



18 91-sphere-flake.

19 802-sphere-flake.

Car and other test scenes in this article rather than any of the images in Yagel's original article.[4]

A similar comparison of the results achieved for Flake on the Pixel Machine[10] showed that adaptive grids performed roughly 7.1 times better than the (unidentified) algorithm running on the AT&T supercomputer. Reportedly, that algorithm improved on a hierarchical bounding box approximately 1,000-fold.

Museum attempts to duplicate Snyder's Teapot Museum Piece.[6] The image, containing 15 percent more elements than the original, was antialiased and rendered approximately 10 times faster than the manually tuned search structure used in the adaptive grids' 1987 predecessor.

Using adaptive grids does not preclude applying any acceleration techniques devised either for the uniform grid or for the ray tracer built around a routine tracing a single ray. To verify that, we combined the algorithm with an adaptive area sampling scheme.[11] The scheme takes advantage of the human eye's low keenness and samples the image space sparsely. This reduces the number of calls to the mentioned single ray routine. Even without selecting optimal parameters controlling area sampling, rendering time for each of the scenes increased by 50 percent. For example, the runtime gain factor from Table 3 for Eagles rose from 671 to 1,927.

Table 5. Memory requirements.

	Regular Grid (Mb)	Adaptive Grids (Mb)
Flake	0.10	0.39
F15	0.13	0.50
Teapot	0.12	0.53
Museum	0.16	1.14
Mirrors	2.21	7.24
Car	2.40	4.39
Eagles	1.04	3.02
Space	1.24	4.79

Memory considerations

The memory requirements for regular grids are relatively high. Admittedly, developing the adaptive grid algorithm was driven mainly by the need to improve ray tracing's efficiency. Each new voxel requires storage space, and subvoxel grids can quickly multiply the number of voxels. Additionally, each grid (excluding voxels) occupies some space in memory. Almost as a rule, the number of subvoxel grids dwarfs the number of local grids.

However, adaptive grids do not require more than one or two subvoxel grid levels. The memory cost then has never been prohibitive. Table 5 summarizes the memory requirements for spatial decompositions of the test scenes using adaptive grids and compares them with the uniform grid's memory requirements. The maximum cost recorded during preprocessing is shown in megabytes. Since some auxiliary arrays are not used during runtime any longer, small amounts of memory (4 to 11 percent in the case of the tested scenes) are released. Further savings are made using hashing and bit-encoding techniques.

Future directions

Our new decomposition technique is a hybrid of a few known methods. This article illustrates the still untapped resources lurking in previous work. It is not difficult to envision other potentially successful mutant systems. For example, Devillers' macroregions[3] might adopt our adaptive grids in the nonempty areas to avoid focusing solely on empty areas. In RRT, the discrete ray traversal algorithm consumes up to 90 percent of the ray tracer execution time,[4] and the entire approach normally requires a huge amount of memory.[5] Adaptive grids reduce the ray traversal time considerably in a scene's empty regions and have moderate memory requirements. It would be extremely interesting to find out how raster voxelization within adaptive grids would compare with each of the techniques individually.

Axis-aligned rectangular grids have a few nice features that simplify the algorithm and its implementation, but their bounds do not provide the tightest fit possible. To achieve a better fit, arbitrarily oriented grids

should be used. Preliminary tests showed that Car, for example, is rendered 26 percent slower if the vehicle is oriented so that the empty space in the axis-aligned grids is maximized. This does not mean that arbitrarily oriented grids would speed up a scene's rendering by the converse 26 percent because arbitrary orientation requires that each ray be converted to the local coordinate systems of all the grids it visits. Moreover, the optimizations relying on the common orientation of various boxes, as shown in Figure 8, would no longer be possible. Wu[12] suggested that a near-optimal orientation of such grids could be determined using either multivariate calculus or interval arithmetic. We found that a method based on an inertia tensor is considerably less expensive[11] and plan to apply it when exploring arbitrarily oriented grids concepts.

Voxelization in our implementation is based on extent overlapping—a fast yet very inaccurate method that can mark all voxels in a grid containing a tesselated sphere as populated with no regard to the subdivision's resolution. A more accurate procedure might treat a voxel as an interval and evaluate a surface's implicit equation at each voxel according to interval analysis rules. The interval test would have to be performed only for those voxels that overlap an object's bounding box. We are currently implementing this technique.

As this article was going to press, it came to our attention that an alternate algorithm for determining a uniform grids hierarchy was proposed.[13] A comparison of the two methods is in order.

Conclusion

Much has been said about scene independence of different acceleration techniques and alleged superiority of one approach over another. Several theoretical and practical studies conducted in the past have lead to the same conclusion: a space partitioning method that allows the fastest rendering of one scene often fails with another. Specialization may be the answer. This has always been pursued, consciously or not, in developing various ray-tracing systems. Despite our new algorithm's impressive efficiency, we don't interpret the new method as the fastest ray-tracing scene decomposition possible. This is because our recent groundwork experiments with a derivative method produced in some of the test scenes presented here produced timings better by approximately 50 percent. ■

References

1. A.S. Glassner, *Introduction to Ray Tracing*, Academic Press, London, 1989.
2. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE CG&A*, Vol. 6, No. 4, Apr. 1986, pp. 16-26.
3. O. Devillers, "The Macroregions: An Efficient Space Subdivision Structure for Ray Tracing," *Proc. Eurographics 89*, Elsevier North-Holland, Amsterdam, 1989, pp. 27-38.
4. R. Yagel, D. Cohen, and A. Kaufman, "Discrete Ray Tracing," *IEEE CG&A*, Vol. 12, No. 5, Sep. 1992, pp. 19-28.
5. D. Cohen and Z. Sheffer, "Proximity Clouds: An Acceleration Technique for 3D Grid Traversal," *The Visual Computer*, Vol. 11, 1994, pp. 27-38.
6. J.M. Snyder and A.H. Barr, "Ray Tracing Complex Models Containing Surface Tesselations," *ACM Computer Graphics* (*Proc. Siggraph*), Vol. 21, No. 4, Jul. 1987, pp. 119-128.
7. J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE CG&A*, Vol. 7, No. 5, May 1987, pp. 14-20.
8. D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Proc. Graphics Interface 89*, Nat'l. Research Council of Canada, Ottawa, Ontario, 1989, pp. 164-172.
9. E. Caspary, *Sequential and Parallel Algorithms for Ray Tracing Complex Scenes*, doctoral dissertation, Univ. of California, Santa Barbara, Dept. of Electrical and Computer Engineering, 1988.
10. M. Potmesil et al., "A Parallel Image Computer with a Distributed Frame Buffer System Architecture and Programming," *Proc. Eurographics 89*, Elsevier North-Holland, Amsterdam, 1989, pp. 197-208.
11. K.S. Klimaszewski, *Faster Ray Tracing Using Adaptive Grids and Area Sampling*, doctoral dissertation, Brigham Young Univ., Provo, Utah, Dept. of Civil and Environmental Engineering, 1994.
12. X. Wu, "A Linear-Time Simple Bounding Volume Algorithm," *Graphics Gems III*, Academic Press, San Diego, Calif., 1992, pp. 301-306.
13. F. Cazals, G. Drettakis, and C. Puech, "Filtering, Clustering and Hierarchy Construction: A New Solution for Ray-Tracing Complex Scenes," *Proc. Eurographics 95*, Blackwell Publishers, Cambridge, Mass., 1995, pp. C371-C382.

Kris Klimaszewski *is a senior software engineer at Cimetrix in Utah. His current interests include computer graphics, scientific visualization, software engineering, and human-computer interaction. Klimaszewski received an MS in mechanical engineering from Warsaw Technical University and a PhD in civil engineering from Brigham Young University.*

Thomas W. Sederberg *is a professor in the Department of Computer Science at Brigham Young University, with research interests in computer graphics and computer-aided geometric design. He received a PhD from Purdue University in 1983. He serves on the editorial boards for* Computer-Aided Design *and* Computer Aided Geometric Design *and was papers chair for Siggraph 91.*

Contact Sederberg at Dept. of Computer Science, Brigham Young University, Provo, Utah, 84602, e-mail tom@byu.edu.