Fast Rendering of Complex Environments Using a Spatial Hierarchy

Bradford Chamberlain   Tony DeRose*   Dani Lischinski   David Salesin   John Snyder†

University of Washington
Box 352350
Seattle, WA 98195-2350
brad,derose,danix,salesin@cs.washington.edu

†Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
johnsny@microsoft.com

Abstract

We present a new method for accelerating the rendering of complex static scenes. The technique is applicable to unstructured scenes containing arbitrary geometric primitives and has sublinear asymptotic complexity. Our approach is to construct a spatial hierarchy of cells over the scene and to associate with each cell a simplified representation of its contents. The scene is then rendered using a traversal of the hierarchy in which a cell's approximation is drawn instead of its contents if the approximation is sufficiently accurate. We apply the method to several different scenes and demonstrate significant speedups with little image degradation. We also exhibit and discuss some of the artifacts that our approximation may cause.

Keywords: level-of-detail (LOD), multiresolution, octree, rendering, spatial hierarchy, walkthrough.

1 Introduction

Advances in the throughput of rendering hardware are continually offset by the need to render more and more complex scenes. Virtual environments consisting of tens or even hundreds of millions of polygons are becoming increasingly common — for example, the Boeing 777 database contains over 500 million polygons (Brechner 1995). Environments of such extreme complexity cannot be rendered in real-time using traditional means. Another potential difficulty with rendering large databases is that even the *working set* of data can potentially outgrow the capacity of main memory or the local disk, so that network bandwidth becomes the major bottleneck. To display truly complex scenes at interactive rates will require rendering algorithms whose running times and working-set memory requirements grow sub-linearly in the complexity of the scene.

This paper describes a hierarchical method that accelerates the rendering process without greatly sacrificing image quality. Each node in the hierarchy represents a region of the scene, or *cell*. Associated with each cell is an approximation to the distant appearance of the geometry contained within the cell. This approximation can be rendered in less time than the geometry within the cell, and is used in place of the geometry when the projected size of the cell on the image plane is sufficiently small.

More specifically, our approach consists of subdividing the input scene using an octree. The appearance of each cell in the octree is approximated using a *color-cube* — a cube with a color and opacity associated with each of its six faces. The hierarchy is constructed as a preprocessing step and serves as a multiresolution volumetric approximation to the original scene. At display time, regions of the scene in the *near field* are drawn using actual geometry. Regions further from the viewer are drawn by rendering the faces of their associated color-cubes, with each cube selected from the hierarchy so that it projects to no more than a pixel on the display.

The major advantages of this approach are that it is fully automatic, that it can take a completely unstructured database of arbitrary geometric primitives as input, and that it requires rendering time (and working-set memory usage) that is asymptotically logarithmic in the number of primitives. The approach is also designed to perform particularly well on relatively *unoccluded* environments, such as outdoor scenes, in which a large fraction of the primitives are at least partially visible.

In the next section, we briefly compare our approach to previous work. In Section 3 we present our algorithm, addressing both its motivation and characteristics. Details regarding the construction of the hierarchy are discussed in Section 4. In Section 5, we present results of several experiments using the method. Finally, we conclude and offer directions for future work in Section 6.

2 Related Work

A number of techniques have been explored for accelerating the rendering of complex environments.

One approach that has been extensively studied is to use *visibility culling* to avoid displaying objects that are completely occluded. This approach was first investi-

*Currently at Pixar Animation Studios, 1001 West Cutting Blvd., Richmond, CA 94804. E-mail: derose@pixar.com

gated by Clark (1976), who used an object hierarchy to rapidly cull surfaces that lie outside the viewing frustum. Airey *et al.* (1990) and Teller (1992) extended this idea, allowing rapid culling of surfaces that lie within the viewing volume but are occluded by other objects in the scene. These approaches work well for scenes with large occluders, such as the walls in a building. The hierarchical Z-buffer (Greene, Kass, and Miller 1993) is another approach to fast visibility culling that allows a region of the scene to be culled whenever its closest depth value is greater than those of the pixels that have already been drawn at its projected screen location. Like previous approaches, this method can achieve dramatic speed-ups for environments with significant occlusion but is less effective for largely unoccluded environments with high visible complexity, such as a model of a tree with thousands of branches and leaves.

Another approach for accelerating rendering is the use of multiresolution or *level-of-detail* (LOD) modeling. The idea is to use progressively coarse representations of a model as it moves away from the viewer. Such an approach has been used since the early days of flight simulators, and has more recently been incorporated in "walk through" systems for complex environments by Funkhouser and Séquin (1993), and Maciel and Shirley (1995). Our work can be thought of as a kind of LOD modeling, which uses a hierarchical representation of the *space* in which the model is embedded, rather than a hierarchical representation of the model itself. An advantage of this approach is that it is extremely general, allowing for unstructured databases, as well as for clusters of objects that overlap spatially.

One of the chief difficulties with the LOD approach is the problem of generating the various coarse-level representations of a model. Funkhouser and Séquin (1993) created the different LOD models manually. Lounsbery *et al.* (1994) and Eck *et al.* (1995) describe methods based on wavelet analysis that can be used to automatically create reasonably accurate low-detail models of surfaces. Maciel and Shirley (1995) used a number of LOD representations, including geometric simplifications created by *Iris Performer* (Rohlf and Helman 1994), texture maps, and colored bounding boxes. Another approach to creating LOD models is described by Rossignac and Borrel (1992), in which objects of arbitrary topology are simplified by collapsing groups of nearby vertices into a single representative vertex, regardless of whether they belong to the same logical part. Our approach can be thought of as a technique for automatically creating LOD models for regions of a scene, rather than for individual objects in the scene.

A different approach for interactive scene display is based on the idea of *view interpolation*, in which different views of a scene are rendered as a preprocessing step, and intermediate views are generated by performing image morphing on the source images in real time. Chen and Williams (1993) and McMillan and Bishop (1995) have demonstrated two variants of this approach for restricted movement in three-dimensional environments. Another image-based approach, described by Regan and Pose (1994), uses multiple display memories and image compositing with depth to allow different parts of an environment to be updated at different rates. Only parts of the environment that change or move significantly are rerendered from one frame to the next, resulting in the majority of objects being rendered infrequently.

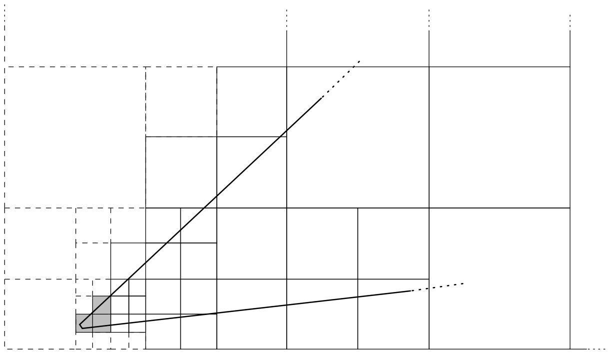

Figure 1: *An example of how a scene is displayed using the color-cube hierarchy. Shaded squares represent leaf cells whose contained geometry is rendered. Dashed squares indicate cells that are culled because they lie completely outside the viewing frustum. Solid squares represent cells that can be drawn with their color-cube approximation.*

Our approach is also related to volume rendering — in particular to the *hierarchical splatting* idea developed by Laur and Hanrahan (1991). Their algorithm renders an octree-encoded volume model to a given error tolerance by rendering splats at different levels according to the estimated error of rendering the level. Our work uses a similar octree-based approach but is tailored to rendering surface (polygonal) geometry rather than volume density data. Specifically, we use a view-dependent representation of color and transparency in each octree cell, provide a method for converting geometry to this representation, and render actual geometry, rather than an approximation, when the geometry is close to the viewer.

3 The Algorithm

We start by constructing an octree subdivision of the input scene. With each cell in the octree we store a *color-cube* — a cube bounding the cell whose faces have an associated color and opacity. The color and opacity of each cube

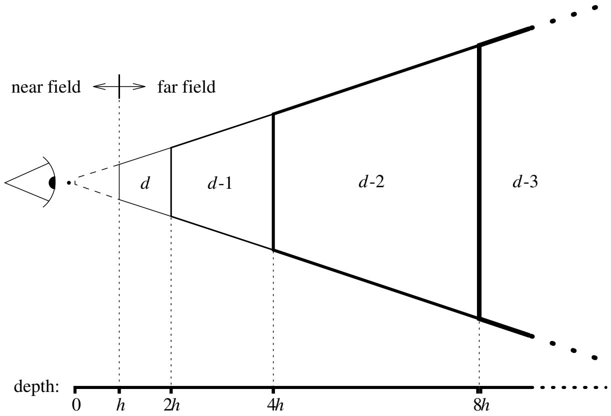

Figure 2: *Our algorithm decomposes the viewing frustum into a series of adjacent sub-frusta $i = d, d-1, \ldots, 1$. The i-th sub-frustum corresponds to the region in which cells at level i of the hierarchy form a valid far-field approximation.*

face approximate the appearance of the cell's geometry as seen through an orthographic projection in the direction of the face's normal. This computation is performed as a preprocessing step and is described in more detail in Section 4.

To render the scene we recursively traverse the octree, starting at the root:

<u>*Render*(*cell*)</u>:
if no part of *cell* is in the view frustum **return**
if projected size of *cell* $< \epsilon$ **then**
  Draw *cell*'s color-cube using a Z-buffer
else if *cell* is a leaf **then**
  Draw *cell*'s geometry using a Z-buffer
else
  for each *child* of *cell* (in back to front order) **do**
    Render(*child*)
  end for
end if

Note that although back-to-front cell traversal would seem to obviate the need for a Z-buffer, its use is nevertheless required to correctly render cells drawn as actual geometry.

As the distance from the viewer increases, larger and larger octree cells project to a screen size smaller than ϵ, allowing increasingly large portions of the scene to be rendered using the color-cube approximation rather than geometry. See Figure 1. We typically set the threshold ϵ so that color-cubes are drawn only for cells whose projected screen image is no larger than a pixel. Larger values of ϵ can also be used to render lower-quality images in less time.

Since a primitive may span multiple cells, one method for drawing a cell's geometry is to render each of its primitives clipped to the cell boundaries. The same primitive may then be rendered multiple times, once for each containing cell. To avoid drawing a primitive more than once, we render each primitive without any clipping and set a flag indicating that it has already been drawn in the current frame.

3.1 Complexity Analysis

In our analysis of the algorithm's asymptotic complexity, we make the following simplifying assumptions:

1. The depth of the octree is $O(\log n)$, where n is the number of primitives.
2. The maximum number of geometric primitives per leaf cell of the octree is constant.

These assumptions are true, for example, in the case of a full octree decomposition of a uniform distribution of primitives. These assumptions have also been empirically verified for the test databases used in Section 5.

Consider a scene satisfying the above assumptions, and let d denote the depth of the octree. We refer to the part of the view frustum in which geometry must be rendered as the *near field*, and the part in which color-cube approximations can be used as the *far field*. A far-field *sub-frustum* i is the part of the viewing frustum in which cells at depth i in the octree have projected size less than ϵ (see Figure 2). There are d such sub-frusta: one for each level of the octree. Note that the ratio of heights of two adjacent far-field sub-frusta is 1:2, and therefore the ratio of their volumes is 1:8.

Since the volume of the sub-frusta grows commensurately with the volume of the cells whose projected size is less than ϵ, the number of such cells per sub-frustum remains constant. Thus, a constant number of cells needs to be rendered for each of the $d = O(\log n)$ sub-frusta, or $O(\log n)$ cells altogether. Furthermore, each far-field cell is rendered in constant time, so $O(\log n)$ time is required to render these cells.

We must also count the time it takes to process the cells in the near field, where the geometric primitives are drawn explicitly. Under our assumptions, the number of leaf cells in the near field is also bounded by a constant, independent of the number of primitives n. Since each leaf cell also contains a constant number of primitives, the total rendering time for the near-field cells is $O(1)$.

Finally, we must count the time spent recursively checking whether a cell is inside the view frustum. Let F_k denote the part of the view frustum containing the near field and sub-frusta $d, d-1, \ldots, k$ (see Figure 2). Now consider a set of eight sibling cells at level k of the octree that are examined by the *Render* routine. At least one of these siblings must lie inside F_k; otherwise, their parent cell would have either been culled or drawn as a color-cube, and these cells would not be tested at all. The volume F_k can only contain a constant number of level k cells, so the total number of level k cells examined by *Render* is constant. Since there are $O(\log n)$ levels in the octree, the total number of tested cells is also $O(\log n)$.

In summary, it takes $O(\log n)$ time to render the cells in the far field, $O(1)$ time to render the geometry in the near field, and $O(\log n)$ time to cull cells outside the viewing frustum. The total time complexity is therefore $O(\log n)$.

The same analysis also applies to the space complexity of the algorithm. The *working set* of the algorithm is defined as all the data required to render the scene from a given view. As we have argued above, the octree traversal for a given view requires an examination of $O(\log n)$ cells. Each cell contains either a fixed-size representation for a color-cube, or a constant number of geometric primitives. Thus the space complexity of the working set is also $O(\log n)$.

Another statistic of interest is the memory required to store the entire scene database after preprocessing. Our octree data structure stores all the original geometry using $O(n)$ pointers, assuming each primitive is contained in $O(1)$ leaf cells. In addition to $O(n)$ leaf cells there are $O(n)$ ancestor cells in the octree. Since each cell's color-cube requires a constant amount of storage, the *total memory* required is $O(n)$.

3.2 Discussion

Our method is similar in spirit to the Barnes-Hut algorithm for the n-body problem (Barnes and Hut 1986). This algorithm approximates the gravitational field due to n point masses in $O(\log n)$ time by replacing large but distant clusters of points with a single point at their center of mass.

One primary respect in which our algorithm differs from Barnes-Hut is that our far-field approximation does not become arbitrarily accurate as a cluster of geometric primitives is moved sufficiently far away. In our case, the appearance of such a cluster can be extremely sensitive to the direction from which it is viewed. For instance, consider a cluster of closely-spaced parallel polygons. As a viewer moves around this cluster, the polygons act as a "venetian blind": a small change in viewing direction can lead to very drastic changes in the way the polygons occlude the scene behind them. Clearly, an approximation based on six directional samples of the cluster's distant appearance is not adequate in such a case; indeed, it is difficult to imagine that any constant-time representation would be able to approximate the space of all possible cell geometries arbitrarily well.

Our algorithm is capable of limiting the spatial extent of the far-field approximation's error to arbitrarily small regions of the image plane, but the cumulative effect of the errors in adjacent approximations can still cause large scale errors and artifacts. We will discuss in detail a specific example exhibiting artifacts due to our approximation in Section 5.

4 Creating the Hierarchy

In our implementation, the hierarchy is built adaptively as a preprocessing step to the display algorithm. We strive for an octree whose depth at any point is logarithmic in that region's geometric complexity. To this end, a cost is associated with each primitive that is proportional to its rendering time. Similarly, we ascribe a cost to each color-cube that will be used to compare its rendering time with that of the original geometry. The implementation uses triangles as its unit of cost; the cost of a color-cube is deemed equivalent to 12 triangles (two triangles times six faces).

The creation of the hierarchy begins with a single *root cell* that contains all the scene's primitives and whose extent spans the scene's bounding volume. We then apply a recursive process, called *deepening*, that considers the merits of pushing the root cell's primitives down one octree level. If the total cost of the cell's primitives exceeds the cost of the color-cubes that would be created for its children, each primitive is pushed down into all of the child cells that contain it, thereby deepening the octree. When a primitive is split between multiple children, its cost is divided evenly between them. This process is repeated for the new cells recursively until no further deepening is warranted. Note that the resulting tree has primitives only at its leaves.

Once the octree construction is complete, we create a color-cube representation for every node in the hierarchy by using post-order traversal of the octree. At each leaf cell a color-cube is created by rendering its list of associated primitives to the screen, using an orthographic projection through each of its six faces. The resulting image from each projection is used to compute the color of the corresponding color-cube face, as described below. In addition to color values, we store a per-face *opacity* (alpha) value (Porter and Duff 1984) to indicate the fraction of the cell face that is covered by geometry.

Several different methods can be used to convert the projected cell's image to a single color value. The method we use is to project the cell's contained geometry to a small screen image — typically, 4×4 pixels — and then average these pixels together to obtain a single $RGBA$ color value (where the resulting R, G, and B values are "pre-multiplied" by the coverage A). With antialiased polygon-drawing hardware (Akeley 1993) the $RGBA$ color-cube approximation may be adequately computed from just a single pixel. In some cases, it might also be

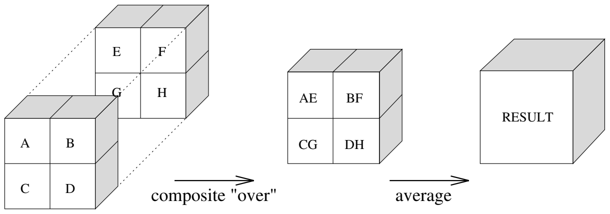

Figure 3: *Compositing and averaging child cells' color-cube faces.*

advantageous to compute the $RGBA$ value analytically, rather than using the sampling approach that we have implemented.

The color-cube approximation for each internal node is computed by combining the $RGBA$ colors of its eight children, as shown in Figure 3. First, the front four cube faces are composited over the back four, using an ordinary "over" compositing operation (Porter and Duff 1984). Then, the four resulting pixels are averaged to give a single $RGBA$ color for the internal node's color-cube face. This process is repeated for each of the six faces of the internal node's color cube. (Alternatively, the color-cube approximations for internal nodes could be computed directly in the same fashion as color-cubes of the leaf nodes; this would improve the internal-node color-cube approximations at the expense of some extra preprocessing time.)

Note that this procedure imposes very few requirements on the types of geometric primitives it can handle. All that is required is to be able to determine the bounding box of each primitive and to render it to the display. Thus, new types of scene databases can be added to the system with minimal effort.

5 Results

We used our method to render several databases, some of which are shown in Plates 1, 2, and 3. Statistics for these databases are given in Tables 1 and 2. All of the images and the statistics in this section were computed on a Silicon Graphics Onyx RealityEngine2 equipped with a 150MHz R4400 processor and 256MB of memory. All geometric primitives and color-cube faces were drawn using OpenGL with 2×2 samples per pixel.

Plate 1 illustrates our method using a tree model containing 52,470 triangles. The four columns show the tree rendered at four increasing distances from the viewer. The top row shows the model rendered using all of its original geometry; the middle row shows the model rendered approximately using our method; and the bottom row shows a magnified view of the approximation. In all views, the model is rotated 30 degrees about the vertical octree axis to give a sense of how well the approximation does when viewed from a different direction than the ones used to compute the color-cubes.

Plate 2 shows the same experiment for a model of a tumor necrosis factor protein and its receptor (Banner et al. 1993). The model consists of 6,534 atoms (spheres), each of which is rendered as 800 polygons, resulting in a total of 5.2 million polygons. In this series, the model is viewed 25 degrees from a sampled direction.

Plate 3 shows a view of our third model, an island landscape with trees. The ground is modeled with 33,000 polygons and the trees add an additional 3.6 million polygons.

Table 1 reports static data for these models. Note that the increase in memory required for the tree model is quite modest: approximately one-ninth the memory of the database itself. For the molecule database, we see an increase in the memory overhead. This is due to the fact that the original model, though complex, can be compactly represented by instancing a small set of atoms. Though we can save memory by using the sphere as the geometric primitive, the octree cannot take advantage of instancing as the geometry could and therefore is rather large. The island exhibits a larger memory overhead, due primarily to the large number of primitives it contains and the storage they require in the octree leaf cells.

Table 2 lists various statistics for the images in Plates 1 and 2. Note that the time to render the full geometry of the tree is roughly constant for all views. With our method, rendering time decreases dramatically as the model recedes from the viewer, yielding a speedup factor of 5.4 for the farthest view. For the molecule database, our gains over the original geometry are even more dramatic, yielding a speedup of 136.289 in the farthest view.

As is evident in these plates, the images produced by our method have a "softer", more anti-aliased look than the ones produced by simply rendering the geometry. This effect is not surprising, since our method effectively prefilters the geometry at a variety of resolutions as it creates the color-cube hierarchy, and then selects the appropriate resolution to render. This effect becomes even more apparent in dynamic sequences when geometric models demonstrate considerable aliasing artifacts.

The island database is different from the other models in that its groundplane and trees are drawn using triangles of vastly different sizes. The resulting octree therefore is roughly 15 levels deep at tree locations and only 5 levels deep in empty regions. Thus, for viewpoints within the scene, the ground and nearby trees are rendered as geometry while distant trees are drawn using color-cubes. Speedups for the viewpoint shown in Plate 5 ranged from 22.7 for a 64×64 image to 1.73 for a 1024×1024 image (shown in Plate 5).

	Database		
	Tree	Tumor Necrosis Factor Molecule	Island Landscape
Number of triangles	52,470	5,227,200 (6,534 spheres)	3,642,224
Required Memory	9,024,840 bytes	213,888 bytes	2,300,532 bytes
Levels in octree	9	25	15
Size of octree	6,967 nodes	343,483 nodes	238,321 nodes
Memory Overhead	1,017,250 bytes	24,492,482 bytes	52,561,676 bytes
Octree Creation Time	248 sec	1,099 sec	4,382 sec
Color-cube Creation Time	164 sec	3,979 sec	2,143 sec
Load from Disk	0.8 sec	20 sec	28 sec

Table 1: *Database statistics.* "Required Memory" *is the memory needed to store the original geometry.* "Memory Overhead" *is the additional memory needed to store the hierarchy (the octree nodes and their associated primitive lists).* "Octree Creation Time" *is the time to construct the octree's structure while* "Color-cube Creation Time" *is the time to create its color-cubes.* "Load from Disk" *is the time required to load the octree and color-cube specifications from disk on successive runs.*

	Tree (Color Plate 1a)			
	$\theta = -30$ degrees			
	$d = 1.75$	$d = 3.5$	$d = 7$	$d = 14$
Geometric Time	0.212 sec	0.211 sec	0.208 sec	0.207 sec
Hierarchical Time	0.503 sec	0.397 sec	0.156 sec	0.038 sec
Speedup	**0.4**×	**0.5**×	**1.3**×	**5.4**×
Level 4 cube faces	0	0	0	3,452
Level 5 cube faces	0	0	11,246	0
Level 6 cube faces	0	14,024	0	0
Level 7 cube faces	1,688	0	0	0
Original Triangles	42,568	26,585	5,985	632

	Tumor Necrosis Factor Molecule (Color Plate 1b)			
	$\theta = 25$ degrees			
	$d = 1.75$	$d = 3.5$	$d = 7$	$d = 14$
Geometric Time	5.774 sec	5.741 sec	5.526 sec	5.179 sec
Hierarchical Time	10.544 sec	1.251 sec	0.213 sec	0.038 sec
Speedup	**0.5**×	**4.6**×	**25.9**×	**136.3**×
Level 4 cube faces	0	0	0	5,384
Level 5 cube faces	0	0	30,328	0
Level 6 cube faces	0	176,919	0	0
Level 7 cube faces	748,634	0	0	0
Level 8 cube faces	4,083	0	0	0
Original Triangles	4,286,400	3,200	0	0

Table 2: *Statistics for the views shown in Plate 1. "θ" indicates the angle of view with respect to a sampled direction and "d" indicates the distance from the scene center in multiples of the object's radius. All views are rendered as a* 124×124 *image.* "Geometric Time" *is the amount of time to draw the model using its geometry, whereas* "Hierarchical Time" *is the time required using our method. Statistics are also given to indicate how many cube faces and triangles were rendered by our method.*

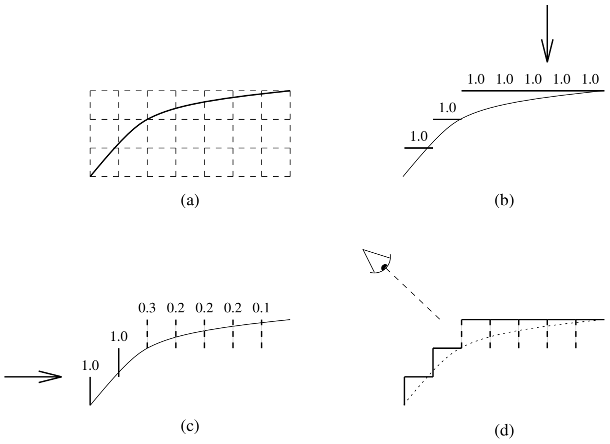

Figure 4: *(a) Original geometry and the leaf cell grid; (b),(c) color-cube faces for two different directions (the number next to each face indicates its opacity); (d) an off-axis view of the approximation.*

5.1 Artifacts

Since our algorithm only approximates the appearance of distant cells, it is important to understand when the approximation behaves well and when it behaves poorly. Our experience has been that the hierarchy of semi-transparent color-cubes generally does a good job of modeling the distant appearance of "suspension-like" distributions of primitives that are uncorrelated and small relative to the leaf cells of the octree. However, it does not generally do as well at representing the appearance of contiguous surfaces that are large relative to the leaf cells, such as the bust of Spock shown in Plate 4. The center image shows the image created by our algorithm for a viewing direction 45 degrees off axis. Notice in particular that some of the space behind the bust incorrectly shows through.

Figure 4 illustrates why this "tearing" artifact occurs. When the original geometry is projected onto cell faces in the preprocessing step, some cell faces (indicated by solid lines in Figure 4(b–d)) are nearly opaque, while others (indicated by dashed lines) are nearly transparent. When the viewer looks at the approximation from an off-axis direction, the background incorrectly shows through the nearly transparent faces.

Another display artifact in the algorithm is caused by our use of the Z-buffer. In cases where translucent color-cubes intersect geometry, the visibility of the geometry within the color-cube varies as the drawing order of the cells changes. Although this artifact has been observed while viewing models during debugging, its effects are slight enough that they have not been noticable in practice.

5.2 Observed asymptotic behavior

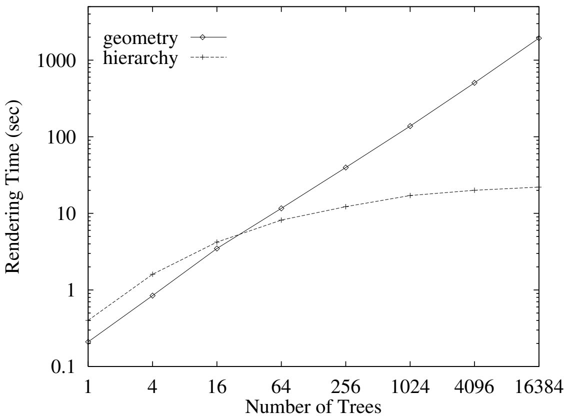

Figure 5: *A log-log plot of rendering times as a function of scene complexity.*

To demonstrate experimentally the logarithmic time complexity of our algorithm, we created a sequence of increasingly complex scenes by recursively instancing the tree model from Plate 1. Several of these scenes are shown in Plate 5. Rendering times for the sequence are plotted in Figure 5 using a log-log scale. The plot shows that there is good correspondence between the measured rendering times and the predicted logarithmic performance.

In this progression of databases, the crossover point where our method becomes more advantageous to use is around 20 trees (roughly 1 million polygons). For a database containing 16,384 trees (859 million polygons), our method gives a speedup factor of 88, becoming equivalent to 39 million "raw" polygons per second.

6 Conclusion

We have developed an algorithm that significantly improves rendering performance of complex environments, using a spatial hierarchy in which the distant appearance of each cell is approximated by a cube with semi-transparent colored faces.

Although visual artifacts can sometimes arise, the algorithm has the following desirable properties:

- The algorithm can be used to display unstructured databases consisting of arbitrary geometric primitives (polygons, spline surfaces, fractals, etc.).
- Under reasonably mild assumptions on the distribution of objects in the environment, rendering time and working-set space have been shown to grow logarithmically in the number of primitives.

- The algorithm is practical and fairly easy to implement on existing high-performance graphics workstations.
- The results of the algorithm animate well.

One drawback of our algorithm is that, while it handles "suspension-like" objects (such as leafy trees) well, it can produce noticeable artifacts when rendering continuous surfaces. On the other hand, many promising approaches already exist for speeding the rendering of continuous geometry, including multiresolution surfaces (Eck et al. 1995) and other LOD approaches for simplifying the geometry itself, and hierarchical Z-buffer algorithms (Greene et al. 1993) for speeding the visibility computations. One worthwhile direction for future work, therefore, would be to look at combining our approach with some of these other methods, employing various different techniques in rendering a given scene, according to the particular geometry being rendered.

Acknowledgements
We would like to thank Eric Brechner, Ka Chai, Michael Cohen, Hugues Hoppe, and Jack Tumblin for many useful discussions during the early stages of this project; and Jim Ahrens, Sean Anderson, Stuart Denman, Przemyslaw Prusinkiewicz, Jonathan Shade, Geoff Skillman, Eric Stollnitz, and the scientists at Los Alamos for supplying complex databases that we could use as test cases for our algorithm. This work was supported by an NSF Young Investigator award (CCR-9357790), an NSF Presidential Faculty Fellowship (CCR-9553199), an ONR Young Investigator award (N00014-95-1-0728), an Alfred P. Sloan Research Fellowship (BR-3495), and Microsoft Research.

References

Airey, J. M., J. H. Rohlf, and F. P. Brooks, Jr. (1990, March). Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics) 24*(2), 41–50.

Akeley, K. (1993, August). RealityEngine graphics. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 109–116.

Banner, D. W., A. D'Arcy, W. Janes, R. Gentz, H. J. Schoenfeld, C. Broger, H. Loetscher, and W. Lesslauer (1993). Crystal structure of the soluble human 55 kd tnf receptor — human tnf beta complex: implications for tnf receptor activation. *Cell 73*, 431–445.

Barnes, J. and P. Hut (1986, December). A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324*, 446–449.

Brechner, E. (1995). Personal communication.

Chen, S. E. and L. Williams (1993, August). View interpolation for image synthesis. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 279–288.

Clark, J. H. (1976, October). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*(10), 547–554.

Eck, M., T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle (1995, August). Multiresolution analysis for arbitrary meshes. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 173–182.

Funkhouser, T. A. and C. H. Séquin (1993, August). Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 247–254.

Greene, N., M. Kass, and G. Miller (1993, August). Hierarchical z-buffer visibility. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 231–238.

Laur, D. and P. Hanrahan (1991, July). Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Proceedings) 25*(4), 285–288.

Lounsbery, M., T. DeRose, and J. Warren (1994, January). Multiresolution surfaces of arbitrary topological type. Technical Report 93-10-05b, Department of Computer Science and Engineering, University of Washington. To appear in *ACM Transactions on Graphics*.

Maciel, P. W. C. and P. Shirley (1995, April). Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, pp. 95–102.

McMillan, L. and G. Bishop (1995, August). Plenoptic modeling: An image-based rendering system. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 39–46.

Porter, T. and T. Duff (1984, July). Compositing digital images. *Computer Graphics (SIGGRAPH '84 Proceedings) 18*(3), 253–259.

Regan, M. and R. Pose (1994, July). Priority rendering with a virtual reality address recalculation pipeline. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 155–162.

Rohlf, J. and J. Helman (1994, July). Iris Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 381–394.

Rossignac, J. and P. Borrel (1992). Multi-resolution 3D approximations for rendering complex scenes. Research Report RC 17697 (#77951), IBM, Yorktown Heights, New York 10598. Also appeared in the *IFIP TC 5.WG 5.10*.

Teller, S. J. (1992, October). *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph. D. thesis, Computer Science Division (EECS), UC Berkeley, Berkeley, California 94720. Available as Report No. UCB/CSD-92-708.