Fast Rendering of Complex Environments Using a Spatial Hierarchy

Bradford Chamberlain
Tony DeRose
Dani Lischinski
David Salesin
John Snyder

July 24, 1995

Abstract

This paper presents work in progress on an automated method for accelerating the rendering of complex static scenes. The technique is general in that it is applicable to unstructured scenes consisting of arbitrary geometric primitives without relying on the presence of specific occlusive qualities in the scene to achieve its speed. Our approach is to place a spatially hierarchical decomposition over the scene and to construct a simplified representation of each cell in the hierarchy that approximates the general appearance of its contents. The scene is then rendered using a traversal of the hierarchy in which a cell's approximation is drawn if it is sufficiently accurate. We apply the method to several scenes, present performance results, and discuss artifacts caused by the approximation. Additionally, we address limitations in the current implementation and present our plans to circumvent these problems.

CR Categories and Subject Descriptors: I.3.3—[**Computer Graphics**]: Picture/Image Generation; I.3.7—[**Computer Graphics**]: Three-Dimensional Graphics and Realism.

Additional Key Words and Phrases: level-of-detail (LOD), spatial hierarchy, walkthrough

1 Introduction

Advances that increase the throughput of rendering hardware are continually offset by the user's demand for the ability to render larger and more complex scenes. Virtual environments consisting of tens or hundreds of millions of polygons are becoming increasingly common — for example, the database for the Boeing 777 contains over 500 million polygons (Brechner 1995). As time passes, we can expect environments to continue growing in size at such a rate that they continually exceed the abilities of the available hardware. To display scenes of this magnitude at anything close to interactive rates, algorithmic techniques for achieving rendering times that grow sub-linearly in the complexity of the scene are necessary.

We are currently investigating a spatially hierarchical method of representing environments that accelerates the rendering process without greatly sacrificing the resulting image quality. Each node in the hierarchy represents a region of the scene, or *cell*, and has an associated simplified representation that approximates

the appearance of the cell as seen from a distance. The simplified representation can be rendered in less time than the cell's contained geometry and is a suitable approximation for the cell if it is beyond a certain depth in the viewer's field-of-view. This depth is dictated by the node's level in the hierarchy, the size of the scene's bounding cube, and the viewing parameters. For example, the root of the hierarchy represents the entire scene and is used when viewing the scene from great distances, while the leaves of the hierarchy represent the smallest subdivisions of the scene and are used when the viewer is located very close to them.

Our approach consists of subdividing the input scene using an *octree*. Each cell in the octree is approximated using a *color-cube* — a cube with an RGB value associated with each of its six faces. This hierarchy is constructed as a pre-processing step and serves as a *multiresolution* volumetric approximation to the original scene. At display time, only 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 and that it can take a completely unstructured database of arbitrary geometric primitives as input. The approach is also designed to work well with relatively *unoccluded* environments in which a large fraction of the scene's polygons are at least partially visible.

The rest of the paper is organized as follows. In the next section we present related work and discuss how our work differs from it. In Section 3 we present our algorithm, detailing both its motivation and characteristics. We discuss the details of our implementation in Section 4 and present the results we obtained with it in Section 5. Section 6 contains a discussion of these results, examines areas in which the implementation requires improvement, and discusses future directions. We conclude in Section 7.

2 Related Work

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

One approach that has been extensively studied is to use *visibility culling* to avoid displaying objects that are completely occluded. This approach was first investigated 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 a recent 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 not as 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 coarser representations of a model as it moves further from the viewer. This approach was first demonstrated by Funkhouser and Séquin (1993) for an architectural environment. Lounsbery *et al.* (1994) later showed how different wavelet approximations could be used to provide reasonably accurate renderings of a surface at different distances from a viewer. Recently, Maciel and Shirley (1995)

have shown how the LOD approach can be generalized to a user-specified hierarchical collection of objects. Our approach is similar, except that it employs a spatial decomposition of the environment rather than an object hierarchy. This allows us to take an unstructured database as input and also to handle clusters of objects with spatial overlap.

One of the chief difficulties with the LOD approach is in generating the various coarse-level representations of a model. Funkhouser and Séquin created the different LOD models manually. Eck *et al.* (1995) describe a complex method for creating a wavelet representation of a surface, suitable for display by Lounsbery's approach. Maciel and Shirley describe a number of methods for creating their LOD representations, including geometric simplifications created by *Iris Performer* (Rohlf and Helman 1994), the use of texture maps, and displaying colored bounding boxes for objects. 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 vertices into a single representative vertex, regardless of whether or not 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 quickly displaying scenes interactively is based on the idea of *view interpolation*, in which different views of a scene are rendered as a pre-processing step, and intermediate views are generated by performing image morphing on the source images in real time. Chen and Williams (1993) showed how this approach could be used for restricted movement in three-dimensional environments. Another image-based approach, described by Regan and Pose (1994), uses multiple display memories and image compositing to allow different parts of an environment to be updated at different rates. Only parts of the environment that change or move significantly are re-rendered from one frame to the next, resulting in the majority of the objects being rendered infrequently.

Our approach is closely related to ideas from volume rendering — in particular to the *hierarchical splatting* work developed by Laur and Hanrahan (1991). In hierarchical splatting, the volume is partitioned into an octree, with each node of the octree containing average color and transparency information for the volume it represents. The image is created by compositing two-dimensional Gaussian "splats" from back to front, with each splat corresponding to a cell in the octree. The algorithm strives to minimize the work required to create an image within a given error tolerance by employing splats at different levels in the octree according to the estimated error of rendering each cell. Our work differs from hierarchical splatting primarily in that we keep a view-dependent representation of the color and transparency of each volumetric cell. A second difference is that instead of using Gaussian splats, we render the faces of our octree's color-cube representations directly.

3 The Algorithm

3.1 Overview

Our spatially-based approach to rendering is motivated by the Barnes-Hut algorithm for approximating gravitational fields caused by a large number of mutually attracting bodies (Barnes and Hut 1986). Whereas the naive method for computing the exact gravitational field caused by N bodies at a point P requires $O(N)$ time, the Barnes-Hut algorithm is able to approximate the field arbitrarily well in $O(\log N)$ time. This algorithm capitalizes on the observation that the *far field* effect of a cluster of bodies is well approximated

by a single body located at the center of gravity of the cluster. However, the far field approximation can only be used if the ratio of the diameter of the cluster to its distance from P is sufficiently small — less than some constant ϵ. Barnes and Hut define the clusters and their far field approximations using an octree constructed in a pre-processing phase that is independent of the location of P. The approximate field for a given position of P is then computed recursively by starting at the root of the octree and using the following procedure:

```
ApproximateField(cell)
ratio ← cell size / distance from P to cell
if ratio < ε then
    return far field approximation for cell
else if cell is a leaf then
    return exact field due to bodies within cell
else
    field ← 0
    foreach child do
        field ← field + ApproximateField(child)
    return field
```

Rendering a complex environment consisting of N geometric primitives as seen from a particular viewpoint P can be thought of in similar terms. Like the Barnes-Hut algorithm, our approach approximates the influence (that is, the visual appearance) of distant objects on the scene's appearance using an appropriate far field representation that can be processed (that is, rendered) in constant time. Our far field representation, described in Section 3.3, is only used for cells that project to sufficiently small regions of the screen. Since the projected area of a cell is based on the ratio of its size to its depth in the viewer's field-of-view, the criterion for using the far field approximation is similar to that used in the Barnes-Hut algorithm, and an appropriate value of ϵ can be selected to restrict the size of the cell's projected area. The hierarchy is constructed in a view-independent manner as a pre-processing step. At runtime, the scene is rendered by recursively visiting the cells of the octree starting at the root and using the following procedure:

```
Render(cell)
if cell is not in view frustum return
ratio ← cell size / depth of cell in field-of-view
if ratio < ε then
    draw cell's far field representation
else if cell is a leaf then
    draw geometry contained within cell
else foreach child do
    Render(child)
```

One primary respect in which our algorithm differs from Barnes-Hut is that it cannot create arbitrarily good approximations in $O(\log N)$ time. Unlike the N-body problem, we cannot guarantee that the distant appearance of a cluster of objects is arbitrarily well approximated using our constant-cost far field representation. For instance, consider a cell containing a number of closely-spaced parallel polygons. As a viewer moves around these polygons, they act as a "venetian blind," gradually changing the degree to which they occlude their surroundings. This collective degree of occlusion is highly sensitive to angular variation, and

thus our far field approximation will not do a good job of representing their appearance from all angles. 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.

A second difference between our algorithm and the Barnes-Hut solver is that in Barnes-Hut, *ratio* is dependent on the *distance* between the cell and P, whereas ours is dependent on the *depth* of the cell in the field-of-view of the user at point P. Thus, our scheme effectively defines a sequence of disjoint frusta that collectively form the entire field-of-view frustum. This is illustrated in Figure 1. Each of these *sub-frusta* corresponds to a region of space in which cells at a particular depth in the octree are appropriate approximations of their contained geometry. Thus, the farthest sub-frustum defines the region in which the entire scene can be represented using the root cell's approximating representation, whereas the closest sub-frustum to P represents the region in which the scene's actual geometry needs to be drawn. The plane between this sub-frustum and the next furthest from P defines the boundary between near field and far field. The number of sub-frusta, and thus the location of this boundary, is determined by the number of levels in the hierarchy. The extent of each sub-frustum is determined by the field-of-view angle, screen size, and the scene's bounding cube.

3.2 Time Complexity

The intuition behind the logarithmic time complexity of our algorithm is illustrated in Figure 2. As the distance from the viewer increases, the far field approximation for larger cells in the octree can be used. Since each cell can contain an arbitrarily large number of geometric primitives and is approximated using a constant-cost representation, significant speedups can be achieved.

In studying the time complexity of an algorithm like ours, we make the following simplifying assumptions:

(a) The number of geometric primitives per leaf cell of the octree is constant.

(b) Each far field representation can be rendered in constant time.

(c) The octree is full and has a depth of $O(\log N)$.

Note that the combination of assumptions **(a)** and **(c)** imply that the geometric primitives are evenly distributed throughout the scene.

We will now argue that these conditions are sufficient to guarantee that a scene can be rendered in $O(\log N)$ time. The crux of the argument is to show that an octree of depth d can be traversed and rendered by our algorithm in time proportional to d. This is due to the fact that only a constant number of cells need to be processed at each level of the octree. Since each cell's color-cube can be rendered in constant time and since we assume that there are $O(\log N)$ levels in the octree, the $O(\log N)$ time bound follows immediately.

To show that a constant number of cells are rendered at each level, we consider the worst case location of the scene, in which it fills up as many of the closest sub-frusta as possible. This constitutes the worst case because those parts of the scene that are closest to the viewer are appropriately drawn using the leaf cells of the hierarchy, and deeper levels of the hierarchy contain more cells than shallower levels. If the scene were to move further from the viewer, the number of cells that need to be considered and rendered can only decrease, as the scene will lie further out in the view frustum where shallower cells in the hierarchy are appropriate.

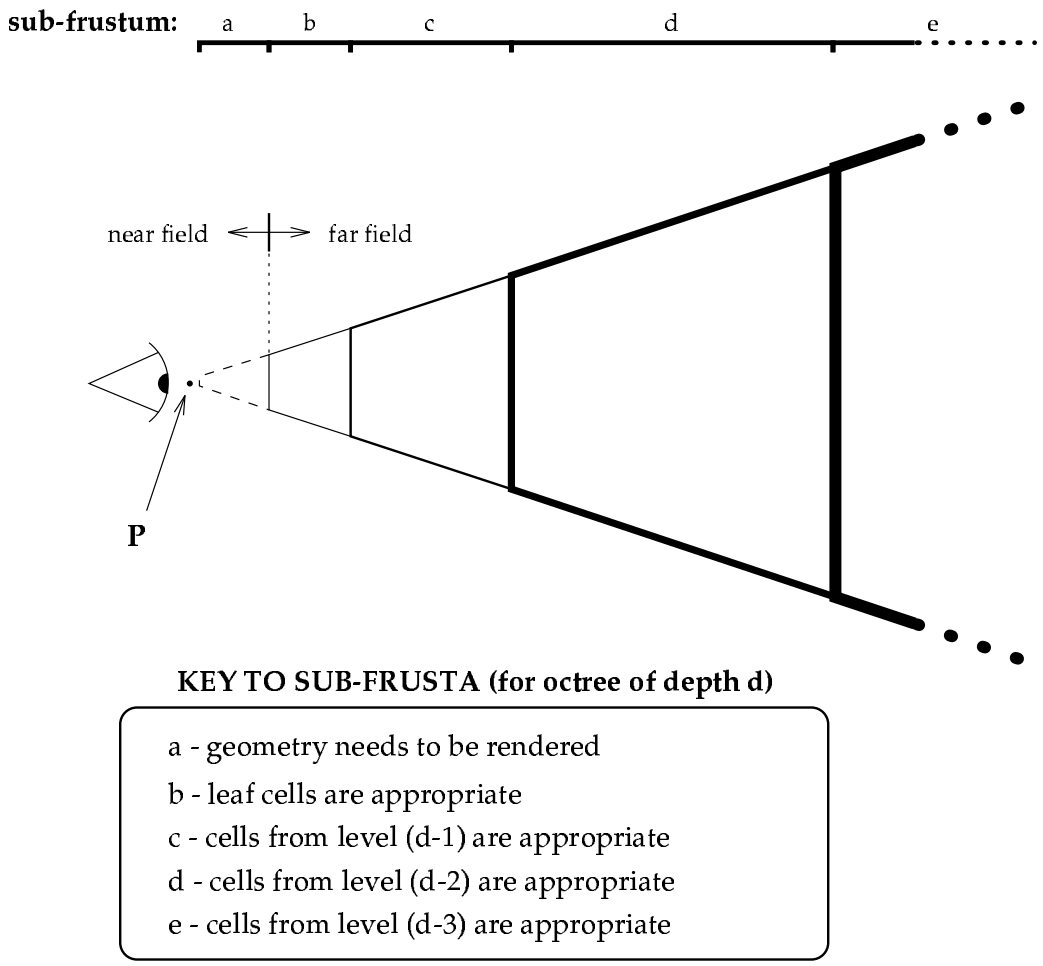

Figure 1: This picture illustrates how the viewing frustum can be thought of as a series of adjacent sub-frusta. Each sub-frustum corresponds to the region in which cells at a particular level of the hierarchy form a valid far field approximation. The viewpoint is at P, and the boundary between near field and far field is indicated.

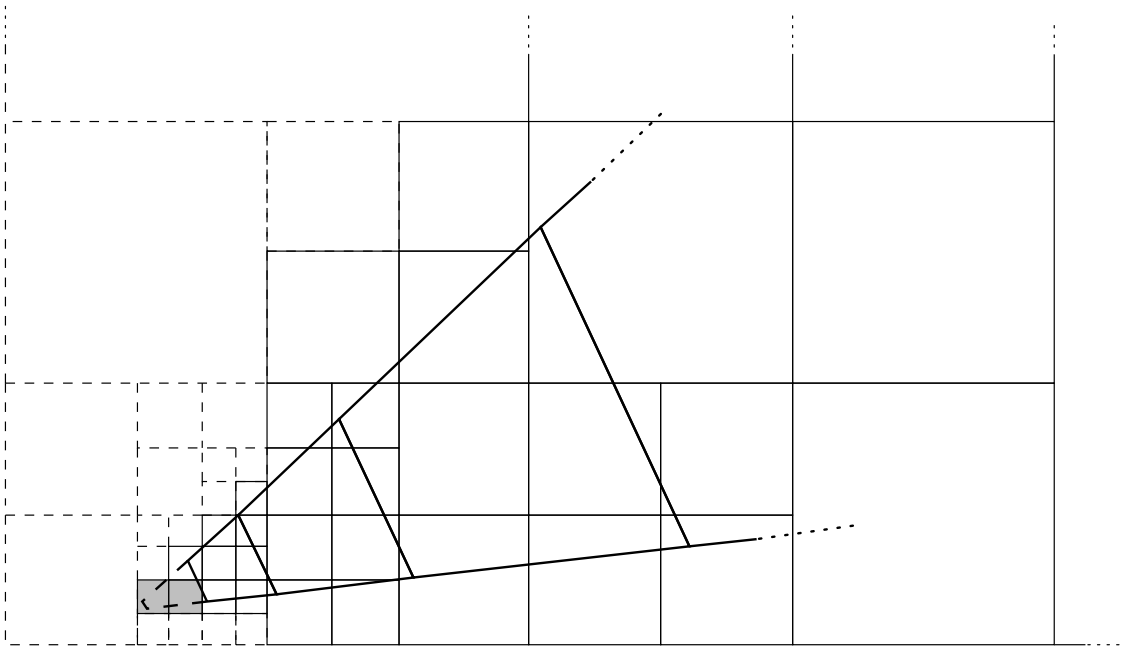

Figure 2: An example of how a scene is displayed using the color-cube hierarchy. The series of adjacent trapezoids represents the viewing frustum. Shaded squares indicate leaf cells that must have their contained geometry rendered. Solid squares represent cells that can be drawn using their far field approximation. Dashed squares indicate cells that are culled because they lie completely outside the viewing frustum.

For this argument, we will also rely on the fact that each level added to the hierarchy corresponds to moving the boundary between near field and far field half the distance to the viewpoint P. Thus, as illustrated in Figure 1, each sub-frustum has half the depth of the next largest sub-frustum. This is the case because the sub-frusta indicate distances required to double an object's projected screen image, and halving an object's distance from the viewer causes its image size to double in each dimension.

We next note that every cell in the octree down to some level i must be visited. This level i indicates the shallowest level in which cells will be rendered, and is dependent on ϵ and the position of P, but not on the depth of the octree. That is, for a given scene, regardless of the octree's depth this level i will remain the same, and is determined only by the viewing parameters. Figure 3 shows an example of a scene in the viewing frustum and indicates the position of level i. For this example, i is the level just below the root node, as it is the first level in which cells are appropriate for rendering. The number of cells processed in levels 1 through i is independent of the octree's depth and therefore constant, implying that all such cells can be visited in constant total time.

Next, let us define a level j that corresponds to the sub-frustum in which the scene overflows the viewing frustum. This is the point when our algorithm starts to benefit from culling cells that lie outside of the view frustum. At each level between i and j, all of the cells whose ancestors have not been rendered must be considered. The number of cells that must be processed at each level is once again independent of the tree's depth, depending only on the viewing parameters. Therefore a constant fraction of the cells at each level pass the far field test and can be rendered in constant total time.

Consider the number of cells that need to be rendered at level $j + 1$. This number is independent of

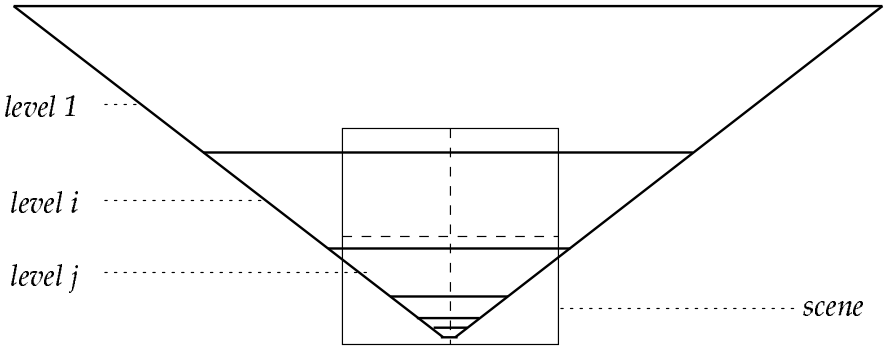

Figure 3: An illustration of where levels i and j fall for a particular scene location relative to the viewing frustum. Level i is the shallowest level whose cells are rendered. Level j is the shallowest level where the scene overflows the viewing frustum.

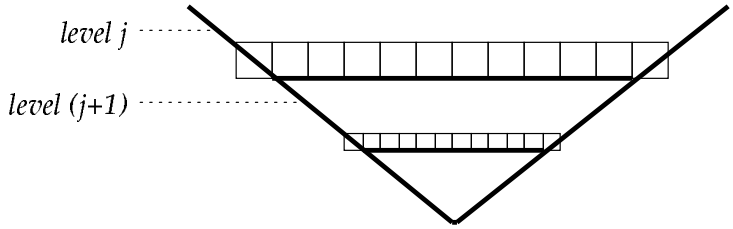

Figure 4: This is a detail of Figure 3, intended to demonstrate that once the scene has overflowed the view frustum, the number of cells along a given dimension of a sub-frustum remains constant for equivalent dimensions of all smaller sub-frusta.

the depth of the tree and can therefore be expressed by a constant c. We claim that the number of octree cells that need to be rendered at level $j+2$ will also be approximately c. The reason for this is that the cells in level $j+2$ are half as large in each dimension as the level $j+1$ cells. In addition, the sub-frustum corresponding to level $j+2$ is half as large in each dimension as the sub-frustum for level $j+1$. Therefore the number of cells rendered at each level past $j+1$ is roughly c. This behavior is illustrated in Figure 4 by demonstrating that for the scene in Figure 3 the number of cells that must be rendered on the boundary of the sub-frusta corresponding to levels j and $j+1$ is roughly equal to the number of cells that must be rendered on the boundary between the sub-frusta for $j+1$ and $j+2$. Similar illustrations can be generated for the other dimensions of the sub-frusta.

Thus, we have argued that at each level, the number of cells that need to be visited in our traversal routine is constant and that the time required to render the visible cells is also constant. In the worst case, we will have to visit cells at all d levels of the hierarchy, therefore the time required to render the hierarchy is $O(d)$. Since we assume that our octree has depth $O(\log N)$, the running time of our algorithm should be $O(\log N)$.

3.3 The Far Field Representation

3.3.1 Design Criteria

As explained above, logarithmic time complexity requires that the far field representation of a cell be rendered in constant-time. However, to be used in a practical algorithm, several additional criteria should be considered when designing a far field representation:

(i) Rendering the approximation for a cell should be cheaper than rendering the geometry contained in the cell.

(ii) The representation should be a reasonable approximation of the cell's contents as seen from arbitrary viewpoints.

(iii) Since the octree may split objects among multiple cells, the approximations of adjacent cells should mesh together well to make any split objects appear whole.

3.3.2 Color-Cubes

As mentioned, the far field approximation that we use is the color-cube — a cube whose faces each have an RGB value associated with them. The size of each color-cube is equal to that of the octree cell with which it is associated. Each cube face's color is computed so that it represents the general appearance of the cell's contained geometry as viewed using an orthogonal projection perpendicular to the face. If the cell appears to be empty when viewed from a particular direction — as a cell containing a single polygon would if the polygon were being viewed edgewise — the face corresponding to that direction is marked as being transparent and no color value is stored.

The color-cube approximation initially seems crude since it is very simple, yet is used to represent regions containing an arbitrary amount of geometry. However, we typically set the threshold ϵ so that color-cubes are used only for cells whose projected screen image is no larger than the size of a pixel. Under these conditions, the cell's fully rendered geometry would take up no more than a single pixel and therefore would be reduced to a single RGB value regardless of its complexity. Thus, our approximation no longer seems overly coarse. Larger values of ϵ can also be used to render lower-quality images in less time.

The color-cube approximation does a good job of fulfilling our representation design criteria. Drawing a color-cube requires only constant time, as no more than three flat-shaded quadrilaterals need to be drawn for any view. Due to the fact that the color-cubes collectively act as a volumetric approximation of the scene, they do a satisfactory job of approximating the scene, except when used with scenes which are particularly sensitive to slight angular variations. Lastly, the fact that the cubes fill their entire cell volume means that adjacent cubes are tightly packed together, keeping artifacts from occurring when objects are split among multiple octree cells.

4 Implementation

Our implementation of this technique is very faithful to the algorithm as presented in Section 3. The hierarchy is built in full to a user-specified depth as a pre-processing step, and then rendered during runtime

using the octree traversal routine *Render()* as presented in Section 3.1. During rendering, we consider child cells in front-to-back order, using the hardware Z-buffer to resolve hidden surfaces for geometry at leaf cells. Cubes are simply drawn to the screen, allowing the rendering pipeline to make decisions about how the sub-pixel interactions are handled. In our current implementation, the rendering is performed by simply sampling at the pixel centers.

At this point, the primary detail of the implementation that we have not addressed concerns the construction of the hierarchy and color-cubes. This is discussed in Section 4.1. We also store a number of cubes for each cell in an attempt to reduce the memory and time overhead required to store and traverse the octree. This technique is described in Section 4.2.

4.1 Creating the Hierarchy

To create the hierarchy, our implementation first reads the input scene and determines its bounding cube. This cube is then subdivided into the $2^d \times 2^d \times 2^d$ leaf cells, where d is the user-specified depth of the octree. Next, we consider each geometric primitive in the scene to determine which leaf cells contain it. Each primitive is associated with all leaf nodes in which it lies by storing a list at each node. After all the primitives have been considered, empty leaves are pruned from the tree, as are internal nodes whose descendents are all empty.

Next, we create a color-cube representation for each node in the hierarchy, using a post-order octree traversal routine. Each leaf node 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. In addition to color values, we store a per-face *coverage* value to indicate what fraction of the cell face is covered by geometry and what fraction is unoccluding.

There are several different methods that can be used to convert the projected cell's image to a single color value. The technique that we use is designed to work well with large surfaces that are continuous between adjacent cells. We project the cell's contained geometry to a small screen image — typically between 1×1 and 5×5 pixels — and then average all of the non-background pixels to obtain a single RGB color value. This oversampling is used to compensate for the single-point sample that each pixel corresponds to in a straightforward rendering, and may be less valuable if sub-pixel sampling or antialiasing were used during this rendering. If all pixels are background, we set the coverage value to be 0.0; otherwise it is set to 1.0.

Some alternative mapping techniques that are worthy of consideration are: to average both background and non-background pixels to arrive at a color value; to set the coverage value based on the fraction of the pixels that are non-background; or to use similar techniques on larger projected images. As another option, the notion of drawing cells to the screen and sampling the resulting image could be abandoned in favor of mathematical methods for calculating appropriate color and coverage values for the cube faces.

For each internal node, its eight child cells' cubes are combined to form color values for its cube, as shown in Figure 5. Specifically, for each face in the parent cube, the eight corresponding faces from the child cubes are composed pairwise, collapsing them into four faces. These faces are then averaged, resulting in the parent cube face's color and coverage values. Both the compositing and averaging steps are performed with regard to the child faces' coverage values. The composite operation is a variation on the *over* operation that was designed to prevent large opaque surfaces from becoming transparent at coarser levels in the hierarchy. This

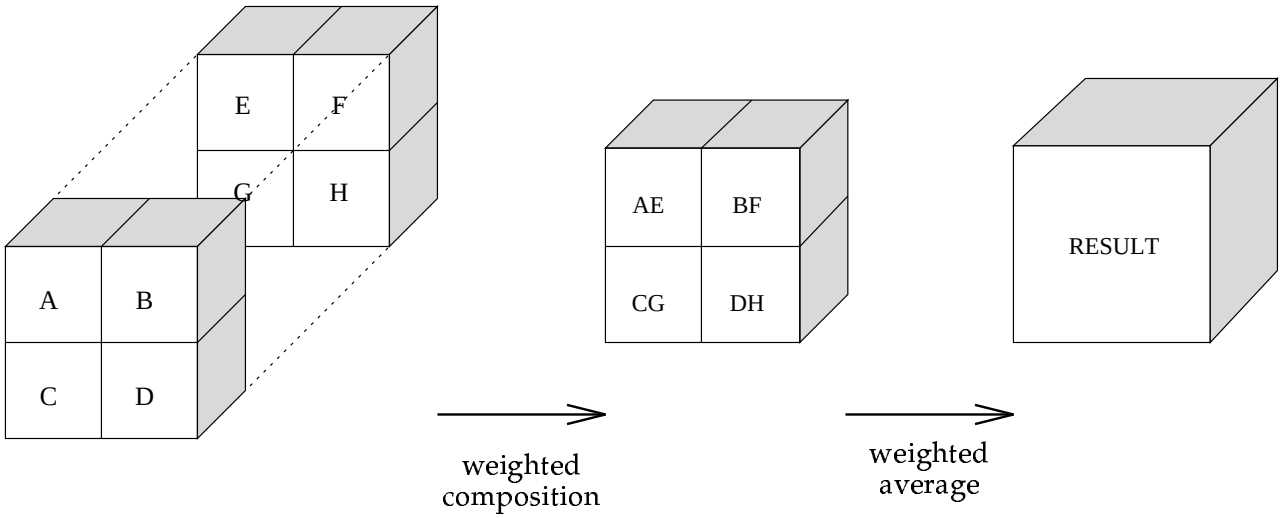

Figure 5: Compositing and averaging child cells' color-cube faces.

is achieved by assuming that the front and back faces' coverage values have minimal overlap. The averaging step uses a simple weighted average, using the faces' coverage values as weights.

Once the color-cubes are built, we determine whether or not each face is important enough to be drawn by comparing its coverage value to a user-specified threshold. If a coverage value exceeds the threshold, the face is drawn; otherwise it is omitted and no longer needs to be retained once the hierarchy is built. If this thresholding is not done and all faces whose coverage values are non-zero are drawn, small isolated details in the original scene are magnified to inappropriately large sizes at coarser levels in the octree.

Another way to avoid this problem would be to use the color-cube coverage values as transparency cues (e.g. alpha values) when rendering the faces. Initial experiments with this technique indicated that transparency may not be approximated as accurately by color-cubes as color is. In these experiments, opaque objects tended to become partially transparent when viewed from non-sampled directions. However, this method deserves to be studied further, as appropriate use of transparency would result in more realistic images for distant viewing of cells containing many small objects.

This completes the discussion of the construction of the hierarchy and color-cubes. Our implementation makes very few assumptions about the input scene format or the geometric primitives required to draw it. As a result, new types of scene databases can be added to the system with a minimal amount of work. The required routines must: read the input file format; determine the scene's bounding box; determine the bounding box of each geometric primitive; and draw each geometric primitive. As a testament to the system's flexibility, we were able to incorporate three different input scene formats in less than a day's work each.

4.2 Cube Clustering and Display Lists

Another feature of our implementation is to allow each cell to be represented not simply by a single color-cube, but by a cluster of color-cubes (e.g., $8 \times 8 \times 8$ or $16 \times 16 \times 16$ cubes per cell). The size of these clusters can be specified by the user at runtime. This representation achieves the same overall effect as a deeper octree, but results in less overhead. There are fewer pointers to store, and fewer cells must be visited during

the runtime traversal routine. Additionally, clustering makes it easier to render an entire bank of leaf cube faces, rather than a single cube face at a time. The disadvantage of clustering is that all cubes within a cluster will be drawn at the same size, even if some of them would have been more appropriately drawn at a coarser level in the hierarchy.

To represent this potentially sparse set of cube faces compactly, we use six display lists per cell — one for each of the six possible cube face orientations. In this way, only those faces that are drawn need to be stored in the display lists. However, it also requires storing each face's location within the cluster of cubes. At runtime, only those display lists that contain visible faces need to be rendered for a given cell.

5 Results

We ran our implementation on a 150MHz R4400 *Onyx RealityEngine2* with 256MB of memory. The five database formats that we used to test the implementation are: (i) Object File Format (OFF) files — a subset of the Object Oriented Graphics Library (OOGL) geometry description language which is used to specify polygonal objects (Phillips 1993); (ii) a restricted use of the rayshade file format designed for the *Rayshade* ray tracing program (Kolb 1992); (iii) raw particle position and energy data obtained in atomic simulations at Los Alamos National Lab; (iv) the Protein Data Bank (PDB) format used to describe all known protein structures (Bernstein, Koetzle, Williams, Meyer Jr., Brice, Rodgers, Kennard, Shimanouchi, and Tasumi 1977; Abola, Bernstein, Bryant, Koetzle, and Weng 1987); and (v) a home-grown format combining OFF and rayshade files to create more complex scenes. Some examples of scenes that we draw using these formats are: a 32,000 polygon model of Spock's head; a topiary modeled using L-systems (Prusinkiewicz, James, and Měch 1994) that consists of roughly 50,000 cylinders to model the branches and 14,000 leaves containing 16 triangles each; a model of an impacted thin plate composed of 1.2 million atoms; a model of a tumor necrosis factor protein and its receptor consisting of 6,500 atoms (Banner, D'Arcy, Janes, Gentz, Schoenfeld, Broger, Loetscher, and Lesslauer 1993); and a landscape composed of 20,000 texture-mapped polygons and five of the aforementioned topiaries.

5.1 Scene Statistics

The results of some of our experiments on these scenes are summarized in Table 1. It should be noted that in these experiments, octree cells that fell outside of the viewing frustum were not pruned from the traversal in *Render()*. However, this does not greatly affect our results, as the majority of each scene fits within the viewing frustum. Furthermore, our implementation currently renders all color-cube faces, whether or not they are facing the viewer. Faster rendering times for the hierarchy are expected once we draw only front-facing color-cube faces.

Times are presented to indicate the performance of our implementation as well as the amount of time required to perform the pre-processing phase of the algorithm. The pre-processing times are divided into two phases: in the first phase, we render each leaf cell and save its complete set of clustered cube face values to disk; in the second phase, we read the leaf cell values from disk and construct the color-cube hierarchy, converting the color-cubes to display lists as we go. Note that the first step only has to be done once per scene for a given octree depth, clustering factor, and leaf cell oversampling factor. Thereafter, the model can

Table 1: Statistics for our test cases. *Geometric time* refers to the time required to render the exact geometry, *Hierarchical time* is the time required by our algorithm, and *Speedup factor* is the ratio of the two. *Preprocessing* is the additional preprocessing time the algorithm requires for building the octree and is divided into two phases as discussed in Section 5. *Tree size* refers to the total number of nodes in the octree. *Cube faces in model* refers to the total number of faces used to describe the model, whereas *Cube faces used* is the number used to draw the presented image. *Hierarchy size* indicates how many levels of the hierarchy were required, *Clustering factor* indicates the number of cubes used per node of the hierarchy, and *Oversampling factor* indicates the size of the cell projection that was used to create color-cube face values.

	Database		
	Tumor molecule (A)	Sparsest tree (B)	Sparse Tree (C)
Number of polygons	2,613,600	38,893	77,807
Geometric time	5.51 sec	0.918 sec	1.71 sec
Hierarchical time	1.64 sec	0.070 sec	0.154 sec
Speedup factor	3.36	13.1	11.1
Preprocessing time (I \ II)	1180 \ 172 sec	212 \ 31.0 sec	309 \ 32.2 sec
Tree size	236 cells	186 cells	177 cells
Cube faces in model	1,628,110	39,611	85,603
Cube faces used	1,320,437	30,583	64,603
Hierarchy size	4 levels	4 levels	4 levels
Clustering factor	$20 \times 20 \times 20$	$18 \times 18 \times 18$	$18 \times 18 \times 18$
Oversampling factor	2×2	1×1	1×1

	Database		
	Medium Tree (D)	Full tree (E)	Landscape (F)
Number of polygons	157,822	479,836	2,419,180
Geometric time	3.35 sec	7.55 sec	37.9 sec
Hierarchical time	0.261 sec	0.823 sec	0.893 sec
Speedup factor	12.8	9.18	42.4
Preprocessing time (I \ II)	480 \ 37.3 sec	939 \ 53.6 sec	13,000 sec \ 85.1 sec
Tree size	180 cells	187 cells	476 cells
Cube faces in model	160,088	518,157	311,740
Cube faces used	120,106	398,046	235,196 (approx.)
Hierarchy size	4 levels	4 levels	5 levels
Clustering factor	$18 \times 18 \times 18$	$18 \times 18 \times 18$	$20 \times 20 \times 20$
Oversampling factor	1×1	1×1	2×2

be viewed by incurring only the second phase's preprocessing cost, which is typically much less expensive than the first phase's.

We present the number of cube faces stored for the model and the number of octree nodes in order to give an indication of the amount of memory used by the hierarchy. In the current implementation, the size of an octree node is 68 bytes and each cube face requires approximately 144 bytes of storage.

5.2 Color Plates

A set of images produced using our algorithm as compared with images produced by rendering the original geometry are shown in color Plate 1. All of these images were created at a twenty-five degree angle from the octree axes to illustrate how the models appear when viewed from a direction that does not correspond to the ones that we sampled in constructing the color-cubes. Table 1 has an entry that lists the number of cube faces used in drawing these images for comparison with the number of polygons rendered in the original models. This number includes both front- and back-facing faces. Note that in the landscape model, the viewpoint is located within the scene, and therefore a portion of the image was rendered using actual geometry. For this reason, the number of cube faces given in the table is an upper bound rather than an exact figure.

Plate 2 shows a set of images taken very close to our models. This reveals the geometry of the color-cubes used to create the images in Plate 1. These pictures are created by forcing the algorithm to draw the leaf node color-cubes when the scene's original geometry is more suitable. The same views are also presented in the geometric models for comparison.

In Plate 3 we present images designed to show the behavior of the algorithm when the scene is split between the geometry sub-frustum and the leaf cells' sub-frustum. Note that the transition is hardly noticeable. Finally, in Plate 4 we present a series of images designed to indicate the appearance of the coarser levels of the hierarchy for some of our scenes. Each sequence of images was created by repeatedly doubling the value of ϵ, allowing larger color-cubes to represent the scene. Some of these images were created using two adjacent levels to represent a scene. As with the junctions between leaf color-cubes and geometry, transitions between adjacent levels in the hierarchy are very smooth.

5.3 Other Results

In order to get a sense of how the algorithm behaves not only with dense, opaque models, but also with sparse, partially-occluding models, we ran it on several "pruned" versions of the full topiary to see how the artifacts changed. In addition to including these models in Table 1 and Plate 1 , we present a graph in Figure 6 showing the time required to render the geometric and hierarchical models versus the number of primitives contained in the original model. This graph indicates that the time required to render the hierarchical version of the model is an improvement, but does not exhibit sublinear behavior. For a discussion of this phenomenon, refer to Section 6.3.

Finally, although we are not able to demonstrate it in this paper, our color-cube models have proven to animate well without introducing dynamically-based artifacts.

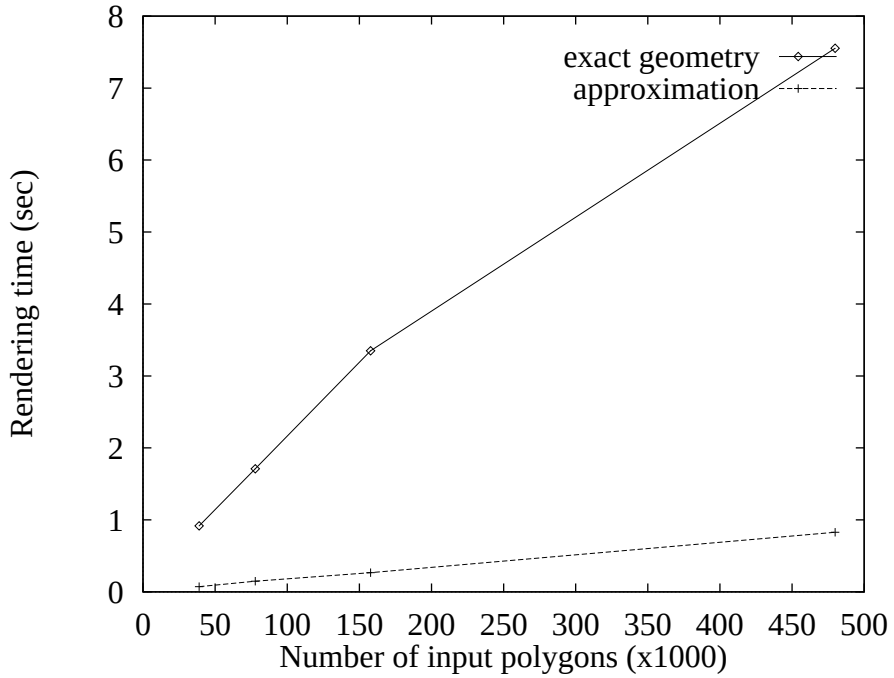

Figure 6: Comparison of exact and approximate rendering performance.

6 Discussion and Future Work

6.1 Artifacts

A few visual artifacts occur in the scenes that are depicted here. First, there is an artifact in which small gaps sometimes appear in solid, continuous surfaces. This can be seen in the landscape model, where a few blue background pixels show through the otherwise solid, grassy hills. This problem stems from the fact that the leaf cells' color values are created using a sampling technique. Due to aliasing artifacts in the sampling process, the representation of a surface across two faces of a cube may not correlate, causing a gap to occur when the color-cube version of the surface is viewed from a direction other than those that were sampled. This artifact is fixed to an extent by our oversampling technique, which effectively causes extra cube cells to be added, making the color-cube representation opaque. However, regardless of the amount of oversampling, a surface can always be constructed that suffers from this artifact. Other measures might combat the problem, such as: using better antialiasing techniques during leaf-cell construction; running gap-filling algorithms to find such problems in the color-cube models and fix them; or using geometric analysis to construct leaf cell values.

A second artifact is a slight inflation in the size of the scene's contents. This can best be seen in the molecular model and the sparsest version of the topiary. In case of the molecule, the color-cube representation makes the atoms appear larger than they do in the geometric version. In the topiary, the artifact manifests itself by making the object appear less sparse, thereby letting less of the background show through. This artifact is caused by the fact that we project the geometry contained within a cell onto the cell's surfaces when creating its color-cube. This has the effect of "inflating" the contained geometry to the cube's size when it is actually smaller. In the case of the molecule, the effect is not overly objectionable, and is difficult

to notice without making a direct comparison to the geometric model. However, in the sparse topiary model, the artifact is more intrusive since it alters the occlusive properties of the topiary. This not only changes its appearance, but also the appearance of everything that falls behind it. We postulate that the use of coverage values as transparency cues as discussed in Section 4.1 will help fix this problem.

A final artifact is seen in the full topiary model. The image created with the hierarchy appears to be darker than the image created by rendering the full geometry. This is an example of a model that is sensitive to slight angular variations. The overall tone of the topiary is based on the large number of leaves that are facing in many different directions. The leaf sides that face into the light source contribute a bright green color to the scene, whereas those that face away from the light are darker. For any given view direction, the geometric version will render only the leaf sides that are visible from the viewpoint P. However, since the color-cubes represent only six sampled directions, they effectively can represent surfaces that should be hidden from the viewpoint. This is illustrated in Figure 7. During the creation of the color-cubes, the left cube face approximates the lit side of the leaf, whereas the bottom face is created using the dark side of the leaf. In the illustrated view of the geometric model (a), only the lit side of the leaf is visible; however, from the same viewpoint in the hierarchical model (b), the left and bottom cube faces are visible, causing the back side of the leaf to contribute to the appearance of the leaf. This effect is what causes the topiary to appear darker from this angle. Conversely, when the topiary is viewed from the back, its overall appearance is lighter for the same reason.

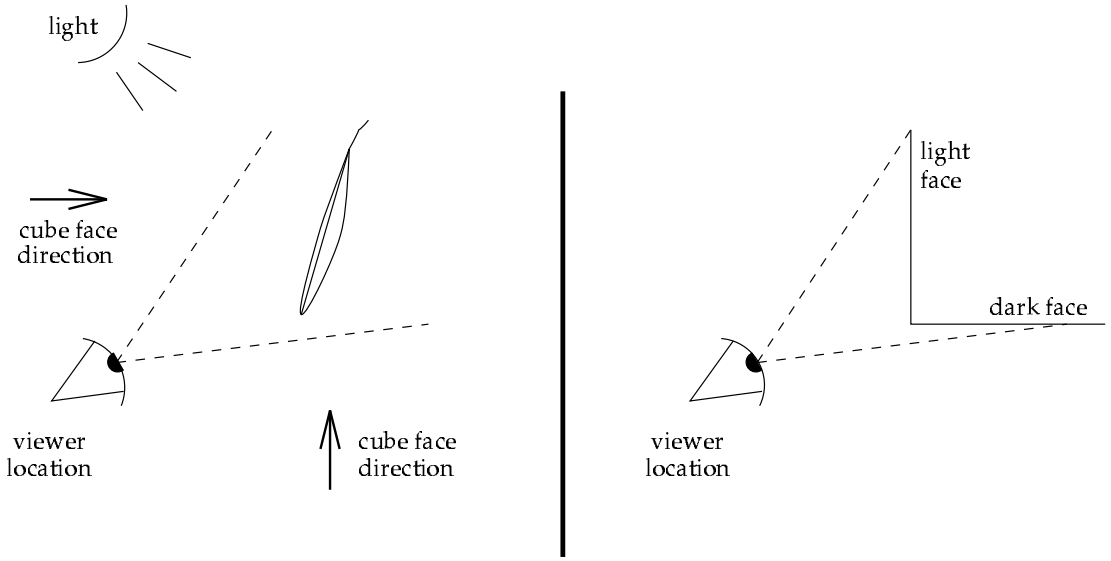

Figure 7: A demonstration of the tone artifact as seen in the topiary model.

6.2 Memory Issues

The current bottleneck in our implementation is a lack of the memory which is required to store deep hierarchies. Although the algorithm by nature consumes memory in order to save time, our current implementation is unnecessarily heavyweight. Our implementation uses *OpenGL*'s display lists for the cube face display lists

in order to maximize our face-drawing performance. The disadvantage is that in order to achieve this performance, *OpenGL* converts the display list items into a standard format, which causes them to take up more memory than is actually necessary. To solve this, we plan to implement our own lightweight display lists, which will reduce the amount of memory required per face from 144 bytes to 6 bytes. This savings should allow us to add a few more levels to our octree at the cost of possibly slowing down the rendering of the cube faces.

Another method that we plan to use to save memory is to construct the octree less blindly. Currently, we build the octree to a fixed depth, regardless of the complexity of the primitives contained within a leaf cell. In less complex regions of the scene, this representation might be wasteful, since drawing the leaf's contained geometry may be cheaper than its color-cube representation. For example, our landscape scene has regions of high complexity wherever there are trees, but the rest of the landscape is fairly simple by comparison. If the hierarchy is built to a depth that would be appropriate for rendering the trees quickly, it would wastefully use the same depth for cells that contain only grass. By pruning the octree so that the depth of each leaf is proportional to the complexity of the geometry contained within the leaf, we could construct an octree whose memory consumption would be more proportional to the scene's localized complexity.

In spite of these improvements, it seems that memory will still remain a bottleneck. Thus, we are contemplating a model in which the color-cube hierarchy will be stored on disk while the workstation's RAM is used as a cache. For a viewpoint moving smoothly through a scene, our algorithm exhibits good data locality since only those cells that are near the boundary between two sub-frusta are candidates for moving to and from disk. Thus, with an intelligent prefetching algorithm, we should be able to take advantage of the inherent locality properties in our algorithm.

Constructing a working version of this approach has already been discussed with members of the *Opal* research group, who are seeking applications that will demonstrate the benefits of using mapped files rather than virtual memory. Our color-cube algorithm is the sort of memory-bound application that they are trying to study, and thus the two projects are mutually beneficial. In order to benefit from their work, we will most likely create a distributed version of the algorithm in which a remote computer acts as a memory server, feeding the graphics front-end the octree nodes that it needs for the current frame.

At a glance, it may seem like the amount of memory required to run our algorithm is exponential, since the addition of a new level in the hierarchy could potentially multiply the number of cube faces by eight. However, a simple estimation shows that the amount of memory required for most scenes is linearly proportional to the storage required by the geometric version of the scene, and that the constants involved are not very big. See Appendix A for a presentation of this argument.

6.3 Achieving Sublinear Rendering Times

Our results show that while we achieve some remarkable speedups over the fully rendered geometry, our topiary timings indicate that we are not achieving sublinear speeds as the number of primitives used to draw the scene increases. The problem with this set of data is that although the number of primitives in the scene is increasing, neither the size of the scene nor the number of levels used to draw the scene's hierarchical representation is changing. As depicted, all of the topiary models are completely rendered using the leaf cells in the hierarchy and no actual geometry. Thus, the times that we report in rendering each model are effectively measuring the densities of the octree and the cube clusters used to represent the topiary. It is

therefore not surprising that the times grow linearly.

As more objects are added to the tree's description, new nodes are inserted to the hierarchy, and new cube faces are added to display lists to account for the new geometry. If geometry were added to the scene indefinitely, the octree and cube clusters would eventually become dense, and the time required to draw the hierarchy would flatten out. In contrast, the time required to draw the actual geometry would continue to rise linearly. Since the tree model was constructed with an eye for realism, self-intersections aren't likely. This explains why we have not yet seen the hierarchical version level off — each new primitive is filling up a relatively sparse region of space, rather than overlapping with existing geometry. The flattening off that we observe in the actual geometry's timings is due to the fact that an increasingly greater fraction of the leaves are occluded. However, this curve will never level off completely.

In order to demonstrate the sublinear behavior of the algorithm, one could repeatedly double the number of primitives while simultaneously incrementing the number of levels in the hierarchy. In addition, the viewing parameters must be chosen such that these new levels are utilized while rendering. An example of such an experiment would be to add more and more topiaries to our existing model in a regular pattern stretching away from the viewer. Every time the number of topiaries quadrupled, we would add another level to our hierarchy. The front topiary would always be modeled with the octree's leaf nodes, which would remain the same size. Meanwhile, the distant topiaries would require increasingly coarse levels of the hierarchy. Thus, each topiary added to the scene would result in another N primitives that the geometric version must draw, but a comparatively small amount of work in the hierarchical model. A brief argument for this experiment based on numbers that we collected during our research is presented in Appendix B

6.4 Other Future Work

As mentioned, our implementation does not cull octree cells that are completely outside of the view frustum, nor does it draw only those color-cube faces that are oriented toward the viewer. Both of these improvements will increase the performance of our implementation, and the former is crucial for sublinear behavior, as addressed in Section 3.2. Our implementation would further be aided by heuristically calculating an appropriate octree depth, clustering factor, and oversampling factor for a given input scene. Such values would most likely be computed using statistics such as the density of geometric primitives in the scene or the ratio of the average primitive's size to the scene's size.

Our scheme would interoperate nicely with the hierarchical Z-buffer culling algorithm for scenes with significant occlusion. Our approach already has an octree as its framework and is able to visit child cells in a front-to-back order, both of which are key elements of the hierarchical Z-buffer method. Therefore, the main thing that is required is to implement the hierarchical Z-buffer itself.

We also intend to experiment with localized dynamic elements in the scene. One way of doing this would be to have moving objects cause the cells in which they're located to be drawn using actual geometry. In this way, the moving object would interact properly with the scene in 3-D, and only a few additional cells in the hierarchy would need to be rendered in geometry. Of course if the movement is not localized or the scene itself changes, the hierarchy's power would be nullified.

Lastly, we will be doing a more quantitative analysis of the accuracy of the color-cube representation. In doing so, we plan to consider alternative cell representations which might better handle scenes that are badly approximated by color-cubes.

7 Conclusion

We have demonstrated that rendering performance can be significantly improved through the use of a spatial hierarchy in which each cell is represented by a cube with colored faces. Although we have argued that this scheme should be able to achieve sublinear rendering times with respect to the number of geometric primitives in the scene, our results do not support this claim. We are confident, however, that future experiments will demonstrate logarithmically growing times.

The simplicity of using a color-cube as our cell approximation is good in that it is a representation that is easy to compute in pre-processing and cheap to render at runtime. However, as we have shown, the color-cube can cause artifacts due to the small, finite number of view directions that it is able to represent.

If our current implementation is any indication, the primary drawback of this scheme is its high memory requirements. We feel, however, that this can be counteracted with a different perspective on our use of RAM, and that for horrendously complex scenes, such methods will be required by any algorithm that tries to accelerate rendering.

Acknowledgements

We would like to thank Eric Brechner, Ka Chai, and Hugues Hoppe, for all of their help in the early stages of the project. Additional thanks go out to Geoff Skillman, Jim Ahrens, Dr. Prusinkiewicz, and the scientists at Los Alamos for supplying us with complex databases to play with.

Simulation data for the thin-plate model generated by SPaSM, courtesy of: D. Beazley, N. Gronbech-Jensen, and P. Lomdahl Theoretical Division and Advanced Computing Lab, Los Alamos National Laboratory.

References

Abola, E. E., F. C. Bernstein, S. H. Bryant, T. F. Koetzle, and J. Weng (1987). Protein data bank. In F. H. Allen, G. Bergerhoff, and R. Sievers (Eds.), *Crystallographic Databases — Information Content, Software Systems, Scientific Applications*, Bonn/Cambridge/Chester, pp. 107–132. Data Commission of the International Union of Crystallography.

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.

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. Basel, Switzerland, pp. 431–445. La Roche Limited, Pharmaceutical Research — New Technologies.

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

Bernstein, F. C., T. F. Koetzle, G. J. B. Williams, E. F. Meyer Jr., M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi (1977). The protein data bank: A computer-based archival file for macromolecular structures. *J. Mol. Biol. 112*, 535–542.

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, January). Multiresolution analysis for arbitrary meshes. Technical Report 95-01-02, Department of Computer Science and Engineering, University of Washington. To appear in SIGGRAPH '95.

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.

Kolb, C. E. (1992). *Rayshade User's Guide and Reference Manual* (0.4 ed.). http://www-graphics.stanford.edu:80/~cek/rayshade/.

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. Submitted for publication.

Maciel, P. W. C. and P. Shirley (1995, April). Visual navigation of largely unoccluded environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*. ACM.

Phillips, M. (1993). *Geomview Manual* (1.4.2 ed.). the Geometry Center. ftp://geom.umn.edu/pub/software/geomview/.

Prusinkiewicz, P., M. James, and R. Měch (1994, July). Synthetic topiary. In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, pp. 351–358.

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.

A Linear Memory Consumption

Suppose that we are considering a scene with N primitives. Assume that the primitives for this scene are triangles with no associated normals or colors. The amount of memory required to store this database would be N polygons $\times$ 3 vertices per triangle $\times$ 3 coordinates per triangle $\times$ 4 bytes per coordinate. Thus, the storage required is $36N$ bytes.

Now consider the amount of storage required for our approach using an octree of depth $O(\log N)$ and a single cube per cell. Assume that each leaf cell in the hierarchy contains k triangles. In this case, the number of leaf cells required would be (n/k), and the total number of cells required for the tree would be $8/7(n/k)$. The amount of storage required for each node in the tree is 6 faces $\times$ 4 bytes per face to store RGBA values. In addition to this, we require 8 pointers for each child $\times$ 4 bytes per pointer. Thus the total storage requirements for each node are 56 bytes. In total, the amount of storage necesary for the hierarchy is $64(n/k)$ bytes. In addition to this, we will need to store indices for primitives with the leaf cell which contains it. This figure will be roughly equal to the number of primitives N times the number of indices required to describe the scene and therefore we will ignore this cost for now, absorbing it into the requirements to store the scene itself. Note that our storage requirements are within a factor of 2 for $k = 1$, and in order for the hierarchical scheme to most effective, k will often be greater than 1. Thus we have argued that our memory requirements are roughly proportional to those of the original scene.

B Achieving Sub-Linear Speeds

It should be possible to simulate the sub-linear speeds which we hope to demonstrate with our algorithm by virtually constructing increasingly large fields of topiaries as discussed in Section 6.3. In order to do this, we can use the timings obtained in drawing a topiary at multiple levels of detail as shown in Plate 4. These images roughly correspond to the topiary models that would be rendered for topiaries in the distance. The times required to render these models are .830 seconds, .209 seconds, .117 seconds, and .049 seconds respectively. Varying the number of topiaries between 1 and 64, and calculating the number of topiaries that would need to be drawn at each level allows us to estimate the time required to draw fields of topiaries. These estimations are shown in the following plot.

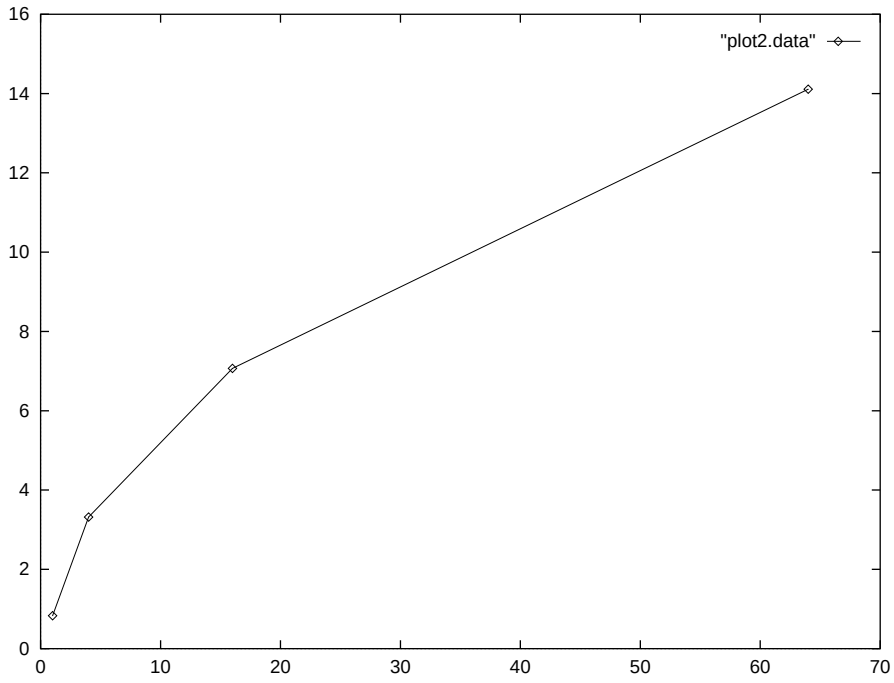

Figure 8: Prediction of performance for increasingly large topiary scenes. The x-axis is the number of topiaries in the virtual scene and the y-axis is the estimation of the amount of required time.