Interpolation over Light Fields
with Applications in Computer Graphics[*]

F. Betul Atalay[†]   David M. Mount[‡]

Abstract

We present a data structure, called a *ray interpolant tree*, or *RI-tree*, which stores a discrete set of directed lines in 3-space, each represented as a point in 4-space. Each directed line is associated with some small number of continuous geometric attributes. We show how this data structure can be used for answering *interpolation queries*, in which we are given an arbitrary ray in 3-space and wish to interpolate the attributes of neighboring rays in the data structure. We illustrate the practical value of the RI-tree in two applications from computer graphics: ray tracing and volume visualization. In particular, given objects defined by smooth curved surfaces, the RI-tree can produce high-quality renderings significantly faster than standard methods. We also investigate a number of tradeoffs between the space and time used by the data structure and the accuracy of the interpolation results.

1 Introduction

There is a growing interest in algorithms and data structures that combine elements of discrete algorithm design with continuous mathematics. This is particularly true in computer graphics. Consider for example the process of generating a photo-realistic image. The most popular method for doing this is *ray-tracing* [18]. Ray-tracing models the light emitted from light sources as traveling along rays in 3-space. The color of a pixel in the image is a reconstruction of the intensity of light traveling along various rays that are emitted from a light source, transmitted and reflected among the objects in the scene, and eventually entering the viewer's eye.

There are many different methods for mapping this approach into an algorithm. At an abstract level, all ray-tracers involve forming an image by combining various continuous quantities, or *attributes*, that have been generated from a discrete set of sampled rays. These continuous attributes include color, radiance, surface normals, and reflection and refraction vectors. These attributes vary continuously either as a function of the location on the surface of an object or as a function of the location of the viewer and the locations of the various light sources in 3-space. The reconstruction process involves combining various discretely sampled attributes in the context of some illumination model.

Producing images by ray-tracing is a computationally intensive process. The degree of realism in the final image depends on a number of factors, including the density and number of samples that are used to

---

[†]Department of Computer Science, University of Maryland, College Park, Maryland. Email: betul@cs.umd.edu.

[‡]Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. Email: mount@cs.umd.edu.

compute a pixel's intensity and the fidelity of the illumination model to the physics of illumination. Scenes can involve hundreds of light sources and from thousands to millions of objects, often represented as smooth surfaces, including implicit surfaces [9], subdivision surfaces [30], and Bezier surfaces and NURBS [14]. Reflective and transparent objects cause rays to be reflected and refracted, further increasing the numbers of rays that need to be traced. In traditional ray-tracing solutions, each ray is traced through the scene as needed to compute the intensity of a pixel in the image [18]. To achieve smoothness avoid problems with aliasing, many rays may be shot for each pixel. A high resolution rendering can easily involve shooting on the order of hundreds of millions of rays. Much of the computational effort involves determining the first object that is intersected by each ray and the location that the ray hits.

In this paper, we propose an approach to help to accelerate this process by reducing the number of intersection calculations. Our algorithm facilitates fast, approximate rendering of a scene from any viewpoint, and is also useful when the scene is rendered from multiple viewpoints, as arises, for example, in computing animations. Rather than tracing each input ray to compute the required attributes, we collect and store a relatively sparse set of *sampled rays* and associate a number of continuous geometric attributes with each sample in a fast data structure. We can then use inexpensive *interpolation* methods to approximate the value of these sampled quantities for other input rays. Using an adaptive strategy, it is possible to avoid oversampling in smooth areas while providing sufficiently dense sampling in regions of high variation. We dynamically maintain a *cache* of the most recently generated samples, in order to reduce the space requirements of the data structure.

The information associated with a given ray is indexed according to the directed line that supports the ray, which in turn is modeled as a point in a 4-dimensional *line space*. Given a ray to be traced, we access the data structure to locate a number of neighboring sampled rays in line space. We then interpolate the continuous attributes from the neighboring rays. The resulting data structure is called an *ray interpolant tree*, or *RI-tree* for short.

1.1   Related Work

The idea of associating radiance information with points in line space has a considerable history, dating back to work in the 1930's by Gershun on vector irradiance fields [16] and Moon and Spencer's concept of photic fields [25], and more recent papers by Levoy and Hanrahan [23] and Gortler, et al. [19]. The term "light field" in our title was coined by Levoy and Hanrahan, but our notion is more general than theirs because we consider interpolation of any continuous attributes, not just radiance. Most methods for storing light field information in computer graphics are based on discretizing the space into uniform grids. In contrast, we sample rays adaptively, concentrating more samples in regions where the variations in attribute values are higher. The most closely related work to ours is the Interpolant Ray-tracer system introduced by Bala, Dorsey, and Teller [7], which combines adaptive sampling of radiance information, caching, and interpolation for rendering convex objects. Our method generalizes theirs by storing and interpolating not only radiance information but other sorts of continuous information, which may be relevant to the rendering process. In Section 3.9 we demonstrate the value of this approach. We also allow nonconvex objects. Unlike their method, however, we do not provide guarantees on the worst-case approximation error.

In addition, there has been earlier research on accelerating ray tracing by reducing the cost of intersection computations using bounding volume hierarchies [22, 27], space partitioning structures [15, 17, 21], and methods exploiting ray coherence [1, 5, 20, 26]. Many of these methods can be applied in combination with ours.

The design of efficient data structures for ray-tracing has been of interest in the field of computational geometry. This includes, for example, the work of Mitchell, Mount and Suri on simple cover complexity [24], the work of Aronov and Fortune on low weight triangulations [4], cost prediction for ray shooting by Aronov, Brönnimann, Chang and Chiang [2, 3] and hierarchical uniform grids (HUG) space partitioning data structure introduced by Cazals, Drettakis and Puech [10]. In a later study, Cazals and Puech compared uniform grids, recursive grids, and HUG for ray-tracing, and demonstrated statistically that recursive grids and HUG outperform uniform grids for non-uniform distribution of scene objects[11].

A simpler variant of this data structure was introduced by the authors in [6]. The current data structure has two main improvements over the previous one. Firstly, the previous data structure was not dynamic. In the preprocessing phase, the entire data structure was built sampling millions of rays originating from the entire space of viewpoints in various directions. This resulted in high preprocessing times, and high space requirements. The current data structure, on the other hand, is built dynamically. Samples are generated on demand—only when they are actually needed, and maintained by a caching mechanism. Secondly, the current system generalizes the framework and can handle scenes containing multiple objects with different surface reflectance properties, whereas the previous system focused on single reflective or transparent object. In addition, a systematic analysis of the performance of the data structure is presented in this paper.

1.2   Design Issues

The approach of computing a sparse set of sample rays and interpolating the results of ray shooting is most useful for rendering smooth objects that are reflective or transparent, for rendering animations when the viewpoint varies smoothly, and for generating high-resolution images and/or antialiased images generated by supersampling [18] in which multiple rays are shot for each pixel of the image.

Although we have motivated the RI-tree data structure from the perspective of ray-tracing, there are a number of applications having to do with lines in 3-space that can benefit from this general approach. To illustrate this, we have studied two different applications, one involving image generation through ray-tracing and the other involving volume visualization with applications in medical imaging for radiation therapy.

There are a number of issues that arise in engineering a practical data structure for interpolation in line space. These include the following.

How and where to sample rays? Regions of space where continuous information varies more rapidly need to be sampled with greater density than regions that vary smoothly.

Whether to interpolate? In the neighborhood of a discontinuity, the number of rays that may need to be sampled to produce reasonable results may be unacceptably high. Because the human eye is very sensitive to discontinuities near edges and silhouettes, it is often wise to avoid interpolating across discontinuities. This raises the question of how to detect discontinuities. When they are detected, is it still possible to interpolate or should we avoid interpolation and use standard ray-tracing instead?

How many samples to maintain? Even for reasonably smooth scenes, the number of sampled rays that would need to be stored for an accurate reconstruction runs well into millions. For this reason, we *cache* the results of only the most relevant rays. What are the space-time tradeoffs involved with this approach?

We investigate these and other questions in the context of a number of experiments based on the applications mentioned above. The paper is organized as follows. In the rest of this section, we give a brief overview of our algorithm in the context of ray-tracing. In Section 2, we explain the construction of our data structure, and its use for answering *interpolation queries*. In Section 3, we report experimental results.

1.3 Algorithm Overview

In order to make things concrete, we will describe our data structure in terms of a ray-tracing application. As mentioned in the introduction, ray-tracing works by simulating the light propagation in a scene. A traditional ray-tracer shoots one or more rays from the viewpoint through each pixel of the image plane. Each ray is traced through the scene and the light intensity that is gathered determines the color of the corresponding pixel. To reduce problems of aliasing, multiple rays may also be traced for each pixel and these results are averaged.

We can distinguish two major tasks in a ray-tracer. The *geometric component* is responsible for calculating the closest visible object point along a specific ray, as the *shading component* computes the color of that point. If the object is reflective or transparent, the reflection and transmission rays are traced recursively to gather their contribution to the intensity. The primary expense in ray-tracing lies in the geometric component, especially for scenes that contain complex objects such as Bezier or NURBS surfaces, and/or reflective and refractive objects.

Each object can be modeled abstractly as a function f mapping input rays to a set of geometric attributes, which are used in color and shading computation. The attributes depend on the object's surface reflectance properties. For objects whose surfaces are neither reflective nor transparent, denoted *simple surfaces*, the function returns the point of intersection and the surface normal at this point. For objects whose surfaces are either reflective or transparent the function additionally returns the *exit ray*, that is, the reflected or refracted ray, respectively, that leaves the object's surface. The exit ray is represented by its origin, the *exit point*, and directional *exit vector*. In general, objects that are both reflective and refractive could be handled by associating multiple exit rays with an input ray, but our implementation currently does not support this. These quantities are depicted in Figure 1 and the function is described schematically below. We will refer to the combination of the underlined attributes below as the *output ray*.

For simple surfaces: $f : \text{Ray} \rightarrow \{\underline{\text{Normal}}, \underline{\text{IntersectionPoint}}\}$
Otherwise: $f : \text{Ray} \rightarrow \{\text{Normal}, \text{IntersectionPoint}, \underline{\text{ExitPoint}}, \underline{\text{ExitVector}}\}$
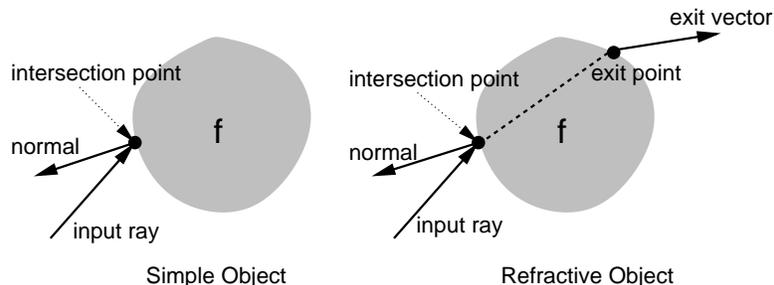
Figure 1: Geometric attributes.

4

For many real world objects, which have large smooth surfaces, f is expected to vary smoothly. In the context of ray-tracing, this is referred to as *ray coherence*. Nearby rays follow similar paths, hit nearby points having similar normal vectors and hence are subject to similar reflections and/or refractions. In the neighborhood of discontinuities, however, nearby input rays may follow quite different paths. In cases where we cannot find sufficient evidence to interpolate, we perform ray-tracing instead.

In a traditional ray-tracer, each object is associated with a procedure that computes intersections between rays and this object. For objects whose boundaries are sufficiently smooth, we replace this intersection procedure with a data structure, which will be introduced in the next section. This data structure approximates the function f through interpolation.

2 The Ray Interpolant Tree

In this section we introduce the main data structure used in our algorithm, the *RI-tree* or *ray interpolant tree*. Each RI-tree is associated with a single *object* of the scene, which is loosely defined to be a collection of logically related surfaces. The object is enclosed by an axis-aligned bounding box. The data structure stores the geometric attributes associated with some set of sampled rays, which may originate from any point in space and intersect the object's bounding box.

2.1 Parameterizing Rays as Points

We will model each ray by the directed line that contains the ray. Directed lines can be represented as a point lying on a 4-dimensional manifold in 5-dimensional projective space using Plücker coordinates [29], but we will adopt a simpler representation, called the *two-plane parameterization* [19, 23, 7]. A directed line is first classified into one of 6 different classes (corresponding to 6 *plane pairs*) according to the line's *dominant direction*, defined to be the axis corresponding to the largest coordinate of the line's directional vector and its sign. (Ties may be broken arbitrarily.) These classes are denoted $+X, -X, +Y, -Y, +Z, -Z$. The
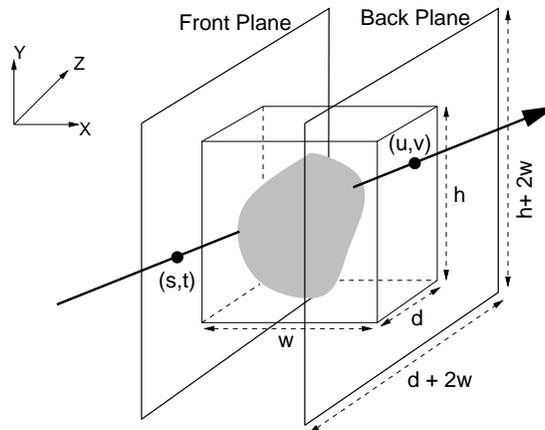
Figure 2: The two-plane parameterization of directed lines and rays. The +X plane pair is shown.

directed line is then represented by its two intercepts (s, t) and (u, v) with the *front plane* and *back plane*, respectively, that are orthogonal to the dominant direction and coinciding with the object's bounding box.
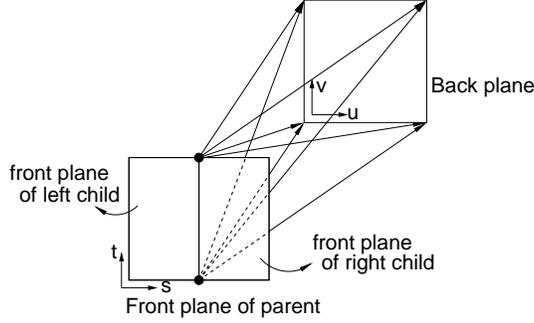
Figure 3: Subdivision along s-axis.

Now, For example, as shown in Figure 2, ray R with dominant direction $+X$ first intersects the front plane of the $+X$ plane pair at (s, t), and then the back plane at (u, v), and hence is parameterized as (s, t, u, v). Note that, the $+X$ and $-X$ involve the same plane pair but differ in the distinction between front and back plane. It is easy to see that if the bounding box has width w (along x), height h (along y) and depth d (along z), then a ray that intersects the bounding box and whose dominant direction is $\pm X$ intersects the front and back planes through two parallel rectangles of height $h + 2w$ and depth $d + 2w$. (See Figure 2.)

2.2 The RI-tree

The RI-tree is a binary tree based on a recursive subdivision of the 4-dimensional space of directed lines. It consists of six separate 4-dimensional kd-trees [8, 28] one for each of the six dominant directions. The root of each kd-tree is a 4-dimensional hypercube in line space containing all rays that are associated with the corresponding plane pair. The 16 corner points of the hypercube represent the 16 rays from each of the four corners of the front plane to the each of the four corners of the back plane. Each node in this data structure is associated with a 4-dimensional hyperrectangle, called a *cell*. The 16 corner points of a leaf cell constitute the ray samples, which form the basis of our interpolation. When the leaf cell is constructed, these 16 rays are traced and the associated geometric attributes are stored in the leaf.

2.3 Adaptive Subdivision and Cache Structure

The RI-tree grows and shrinks dynamically based on demand. Initially only the root cell is built by sampling its 16 corner rays. A leaf cell is is subdivided by placing a cut-plane at the midpoint orthogonal to the coordinate axis with the longest length. In terms of the plane pair, this corresponds to dividing the corresponding front or back plane through the midpoint of the longer side. We partition the existing 16 corner samples between the two children, and sample eight new corner rays that are shared between the two child cells. (These new rays are illustrated in Figure 3 in the case that the s-axis is split.)

Rays need to be sampled more densely in some regions than others, for example, in regions where geometric attributes have greater variation. For this reason, the subdivision is carried out adaptively based on the distance between output attributes. The distance between two sets of output attributes are defined as the distance between their associated output rays. We define the *distance* between two rays to be the L_2 distance between their 4-dimensional representations. To determine whether a cell should be subdivided, we first compute the correct output ray associated with the midpoint of the cell, and then we compute an

approximate output ray by interpolation of the 16 corner rays for the same point. If the distance between these two output rays exceeds a given user-defined *distance threshold* and the depth of the cell in the tree is less than a user-defined *depth constraint*, the cell is subdivided. Otherwise the leaf is said to be *final*.

If we were to expand all nodes in the tree until they are final, the resulting data structure could be very large, depending on the distance threshold and the depth constraint. For this reason we only expand a node to a final leaf if this leaf node is needed for some interpolation. Once a final leaf node is used, it is marked with a time stamp. If the size of the data structure exceeds a user-defined *cache size*, then the the tree is pruned to a constant fraction of this size by removing all but the most recently used nodes. In this way, the RI-tree behaves much like an LRU-cache.

Comment on the depth constraint: An absolute bound on the maximum tree depth is a rather unnatural parameter. It is, however, possible to infer this value based on some more geometrically natural parameters. For example, consider instead a *angular similarity constraint*, θ, which bounds the maximum angle between any two input rays that fall within the same leaf cell. In other words, if the angle between any two input rays that lie within the same leaf cell of the subdivision is at most θ, then these samples are sufficiently close to one another that further subdivision is not required.

Let us illustrate how to compute the tree depth constraint from θ. We assume for simplicity that the object has been enclosed within a bounding cube. Since angles are not affected by uniform scaling, we may assume that this is a unit cube. It follows from our parameteri-

Figure 4: Maximum angle is achieved by the cross diagonals of orthogonally opposing faces.

zation, that for each of the dominant directions, it suffices to consider rays that intersect a pair of parallel squares lying on the front and back planes for this direction, whose side lengths are 3 units. (Consider Figure 2 in the case $w = h = d = 1$.) Now, consider any leaf cell of the associated tree for this direction. Such a cell corresponds to the set of rays passing through two rectangular faces, one on each of these two parallel squares. Since the sides of each rectangle are alternatingly split along the side of maximum length, the worst case arises when both sides of these faces are of equal length, say r. (See Figure 4.) Among the 16 corner rays sampled for any face pair, it can be shown that the maximum angle occurs between the cross diagonals of two faces that are aligned orthogonally opposite one another, that is, so that the line connecting the centers of these two faces is orthogonal to both faces. In this case, the diagonals are of equal length and intersect at their midpoints, from which we have

$$\tan\frac{\theta}{2} = \frac{r\sqrt{2}/2}{1/2} = r\sqrt{2},$$

And so, $r = (\tan(\theta/2))/\sqrt{2}$. Since four splits are required to halve the side length of a cell, and the initial (root) face is of side length 3, it follows that the maximum depth constraint, as a function of the angular similarity constraint, is

$$depth(\theta) = 4\lg\frac{3}{r} = 4\lg\frac{3\sqrt{2}}{\tan\frac{\theta}{2}} = 4\lg\left(3\sqrt{2}\cot\frac{\theta}{2}\right),$$

where $\lg$ denotes logarithm base 2.

We set a fixed depth constraints in our experiments, but this formulation in terms of an angular similarity constraint would be a more appropriate parameter for software design purposes.

7

2.4 Rendering and Interpolation Queries

Recall that our goal is to use interpolation between sampled output rays whenever things are sufficiently smooth. RI-tree can be used to perform a number of functions in rendering, including determining the first object that a ray hits, computing the reflection or refraction (exit) ray for nonsimple objects, and answering visibility queries, which are used for example to determine whether a point is visible to a light source or in a shadow.

Let us consider the interpolation of a given input ray R. We first map R to the associated point in the 4-dimensional directed line space and, depending on the dominant direction of this line, we find the leaf cell of the appropriate kd-tree through a standard descent. Since the nodes of the tree are constructed only as needed, it is possible that R will reside in a leaf that is not marked as *final*. This means that this particular leaf has not completed its recursive subdivision. In this case, the leaf is subdivided recursively, along the path R would follow, until the termination condition is satisfied, and the final leaf containing R is now marked as *final*. (Other leaves generated by this process are not so marked.)
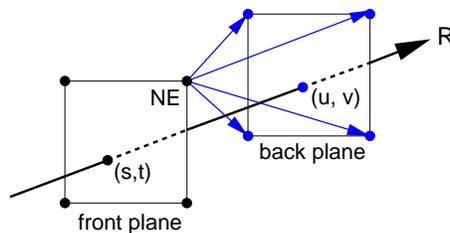
Figure 5: Sampled rays within a directional group.

Given the final leaf cell containing R, the output attributes for R can now be interpolated. Interpolation proceeds in two steps. First we group the rays in groups of four, which we call the *directional groups*. Rays in the same group originate from the same corner point on the front plane, and pass through each of the four corners of the back plane (For example, Figure 5 shows the rays that originate from the north-east corner of the front plane). Within each directional group, bilinear interpolation with respect to the (u, v) coordinates is performed to compute intermediate output attributes. The outputs of these interpolations are then bilinearly interpolated with respect to the (s, t) coordinates to get the approximate output attributes for R. Thus, this is essentially a quadrilinear interpolation.

2.5 Handling Discontinuities and Regions of High Curvature

Through the use of interpolation, we can greatly reduce the number of ray samples that would otherwise be needed to render a smooth surface. However, if the ray-output function f contains discontinuities, as may occur at the edges and the outer silhouettes of the object, then we will observe bleeding of colors across these edges. This could be remedied by building a deeper tree, which might involve sampling of rays up to pixel resolution in the discontinuity regions. This could result in unacceptably high memory requirements. Instead our approach will be to detect and classify discontinuity regions. In some cases we apply a more sophisticated interpolation. Otherwise we do not interpolate and instead simply revert to ray-tracing. We will present a brief overview of how discontinuities are handled here. Further details are presented in [6].

Our objects are specified as a collection of smooth surfaces, referred to as *patches*. Each patch is assigned a *patch-identifier*. Associated with each sample ray, we store the patch-identifier of the first patch

it hits. Since each ray sample knows which surface element it hits, it is possible to disallow any interpolation between different surfaces. It is often the case, however, that large smooth surfaces are composed of many smaller patches, which are joined together along edges so that first and second partial derivatives vary continuously across the edge. In such cases interpolation is allowed. We assume that the surfaces of the scene have been provided with this information, by partitioning patches into surface equivalence classes. If the patch-identifiers associated with the 16 corner ray samples of a final leaf are in the same equivalence class, we conclude that there is no discontinuities crossing the region surrounded by the 16 ray hits, and we apply the interpolation process described above.

Requiring that all 16 patches arise from the same equivalence class can significantly limit the number of instances in which interpolation can be applied. After all, linear interpolation in 4-space can be performed with as few as 5 sample points. If the patch-identifiers for the 16 corner samples of the leaf arise from more than two equivalence classes, then we revert to ray tracing. On the other hand, if exactly two equivalence classes are present, implying that there is a single discontinuity boundary, then we perform an intersection test to determine which patch the query ray hits. Let p_r denote this patch. This intersection test is not as expensive as a general tracing of the ray, since typically only a few patches are involved, and only the first level intersections of a ray-tracing procedure is computed (that is, no reflection rays or light rays need be traced). Among the 16 corner ray samples, only the ones that hit a patch in the same equivalence class as p_r are *usable* as interpolants. These are the ray samples hitting the same side of a discontinuity boundary as the query ray. If we determine that there is a sufficient number of *usable* ray samples, we then interpolate the ray. Otherwise, we use ray-tracing. See [6] for further details.

Even if interpolation is allowed by the above criterion, it is still possible that interpolation may be inappropriate because the surface has high curvature, resulting in very different output rays for nearby input rays. High variations in the output ray (i.e. normal or the exit ray), signal a discontinuous region. As a measure to determine the distance between two output rays, we use the angular distance between their directional vectors. If any pairwise distance between the output rays corresponding to the usable interpolants is greater than a given *angular threshold*, then interpolation is not performed.

3 Experimental Results

The data structure described in the previous section is based on a number of parameters, which directly influence the algorithm's accuracy and the size and depth of the tree, and indirectly influences the running time. We have implemented the data structure and have run a number of experiments to test its performance as a function of a number of these parameters. We have performed our comparisons in the context of two applications.

Ray-tracing: This has been described in the previous section. We are given a scene consisting of objects that are either simple, reflective or transparent and a number of light sources. The output is a rendering of the scene from one or more viewpoints.

Volume Visualization: This application is motivated from the medical application of modeling the amount of radiation absorbed in human tissue [12]. We wish to visualize the absorption of radiation through a set of nonintersecting objects in 3-space. In the medical application these objects may be models of human organs, bones, and tumors. For visualization purposes, we treat these shapes as though they are transparent (but do not refract light rays). If we imagine illuminating such a scene by x-rays, then

the intensity of a pixel in the image is inversely proportional to the length of its intersection with the various objects of the scene. For each object stored as an RI-tree, the geometric attribute associated with each ray is this intersection length.

We know of no comparable algorithms or data structures with which to compare our data structure. Existing image-based data structures [19, 23] assume a dense sampling of the light field, which would easily exceed our memory resources at the resolutions we would like to consider. The Interpolant Ray-tracer system by Bala, et al. [7] only deals with convex objects, and only interpolates radiance information. Our data structure can handle nonconvex objects and may interpolate any type of continuous attributes. In the ray-tracing application, for example, we prefer interpolating normals (for a simple object), rather than interpolating radiance. In Section 3.9, we provide a comparison of radiance versus normal interpolation in such a case.

3.1 Test Inputs

We have generated a number of input scenes including different types of objects. As mentioned earlier, for each object in a scene we may choose to represent it in the traditional method or to use our data structure. Our choice of input sets has been influenced by the fact that the RI-tree is most beneficial for high-resolution renderings of smooth objects, especially those that are reflective or transparent. We know of no appropriate benchmark data sets satisfying these requirements, and so we have generated our own data sets.

Bezier Surface: This surface is used to demonstrate the results of interpolation algorithm for smooth reflective objects. It is a reflective surface consisting of 100 Bezier patches, joined with C^2 continuity at the edges. The surface is placed within a large sphere, which has been given a pseudo-random procedural texture [13]. Experiments run with the Bezier surface have been averaged over renderings of the surface from 3 different viewpoints. Figure 13(a) shows the Bezier surface from one viewpoint. We rendered images of size 600×600 without anitaliasing. (That is, only one ray is shot per pixel.)

Random volumes: We ran another set of experiments on randomly generated refractive, nonintersecting, convex Bezier objects. In order to generate nonintersecting objects, a given region is recursively subdivided into a given number of non-intersecting cells by randomly generated axis-aligned hyperplanes, and a convex object is generated within each such cell. Each object is generated by first generating a random convex planar polyline that defines the silhouette of right half of the object. The vertices of the polyline constitute the control points for a random number (n) of Bezier curves, ranging from 5 to 16. Then a surface of revolution is generated, giving rise to $4n$ Bezier surface patches. The volumes are used both for the ray-tracing and the volume visualization experiments. For ray-tracing we rendered anti-aliased images of size 300×300 (with 9 rays shot per pixel). For volume visualization we rendered 600×600 images without antialiasing. Results are averaged over three different random scenes containing 8, 6, and 5 volumes respectively. Figure 14 shows a scene of refractive volumes.

Tomatoes: This is a realistic scene used to demonstrate the performance and quality of our algorithm for real scenes. The scene consists of a number of tomatoes, modeled as spheres, placed within a reflective bowl, modeled using Bezier surfaces. This is covered by a reflective and transparent but non-refractive plastic wrap (the same Bezier surface described above). There is a Bezier surface tomato next to the bowl, and they are both placed on a reflective table within a large sphere. The wrap reflects the procedurally textured sphere. The scene is shown in Figure 12.
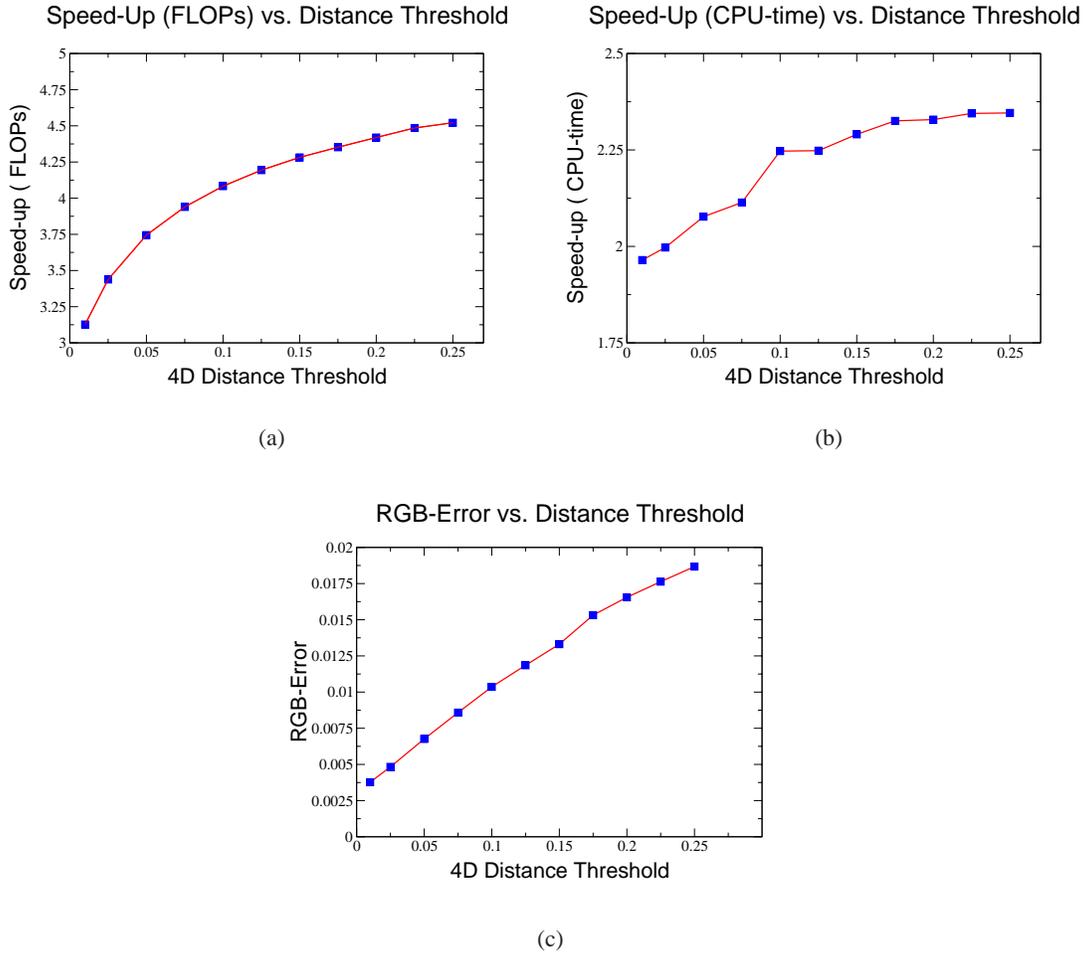
Figure 6: Varying the distance threshold. (Angular threshold = $30°$, maximum tree depth = 28, for a 600×600 image, without antialiased). Note that the y-axis does not always start at 0.

3.2 Metrics

We measured the *speedup* and *actual error* committed as a function of four different parameters. Speedup is defined both in terms of number of floating point operations, or *FLOPs*, and CPU-time. FLOP speedup is the ratio of the number of FLOPs performed by traditional ray-tracing to the number of FLOPs used by our algorithm to render the same scene. Similarly, CPU speedup is the ratio of CPU-times. Note that FLOPs and CPU-times for our algorithm include both the sampling and interpolation time. FLOP counts are machine independent, but they tend to underestimate the time spent in data structure access. However, our experience has shown that this access time is not a dominant component of the overall running time.

The actual error committed in a ray-tracing application is measured as the average L_2 distance between the RGB values of corresponding pixels in a ray-traced image and the interpolated image. RGB value is a 3-dimensional vector with values normalized to the range $[0, 1]$. Thus the maximum possible error is $\sqrt{3}$. The error in a volume visualization application is measured as the average distance between the actual length

attribute and the corresponding interpolated length attribute.

3.3   Varying the Distance Threshold

Recall that the distance threshold, described in Section 2.3, is used to determine whether an approximate output ray and the corresponding actual output ray are close enough (in terms of L_2 distance) to terminate a subdivision process. We varied the distance threshold from 0.01 to 0.25 while the other parameters are fixed. The results for the Bezier surface scenes are shown in Figure 6.

As expected, the actual error decreases as the threshold is lowered, due to denser sampling. But, the overhead of more sample computations reduces the speedup. However, even for low thresholds where the image quality is high, the CPU-speedup is greater than 2 and the FLOP-speedup is greater than 3. These speedups can be quite significant for ray-tracing, where a single frame can take a long time to render.

Figure 13 (b) and (c) demonstrate how the variation in error reflects the changes in the quality of the rendered image. Notice the blockiness in part (c) when the data structure is not subdivided as densely as in part (b).

(a)   (b)
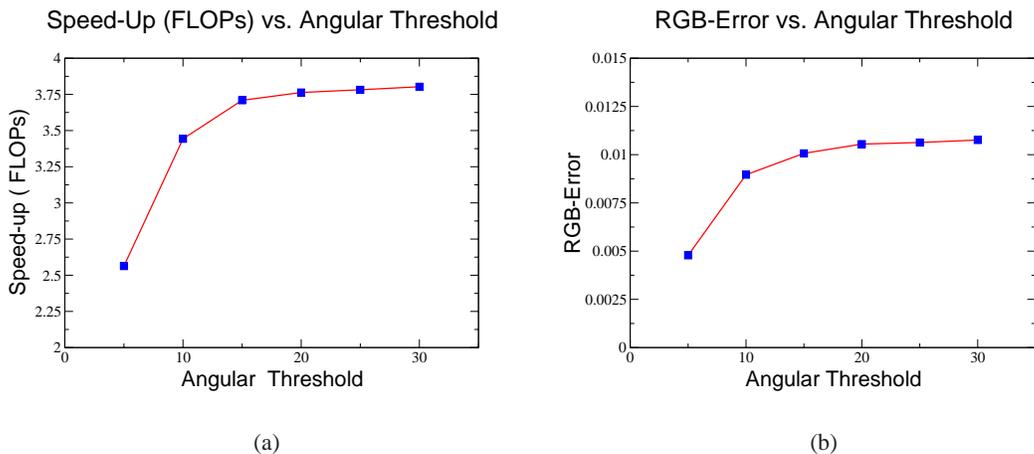
Figure 7: Varying angular threshold (distance threshold=0.25, maximum depth=28, 300×300, antialiased).

3.4   Varying the Angular Threshold

The angular threshold, described in Section 2.5, is applied to each query to determine whether the surface curvature variation is too high to apply interpolation. We investigated the speedup and error as a function of the angular threshold over the renderings of three different random volume scenes. The angular threshold is varied from $5°$ to $30°$. The results are shown in Figure 7.

For lower thresholds, fewer rays could be interpolated due to distant interpolants, and those rays are traced instead. In this case, the actual error committed is smaller but at the expense of lower speedups. However, the speedups are still acceptable even for low thresholds.

12
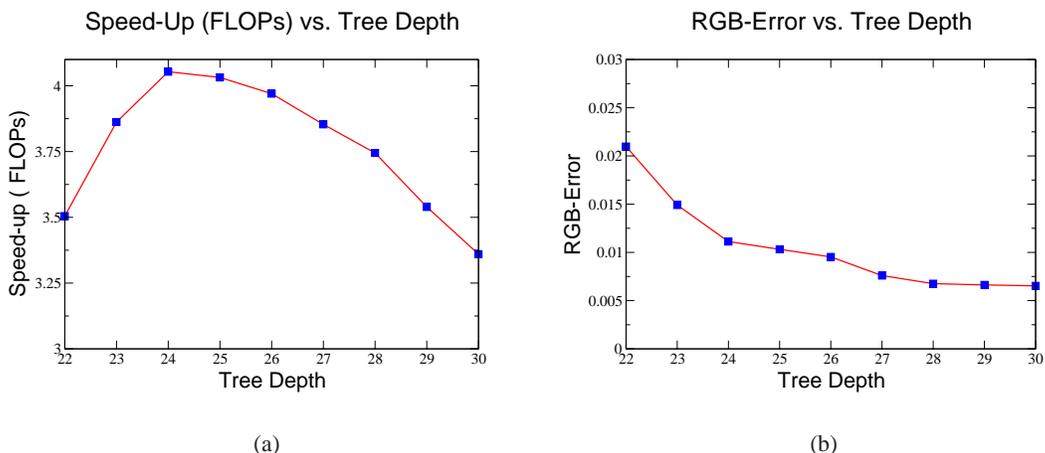
Figure 8: Varying tree depth (distance threshold=0.05, angular threshold=30, 600×600, non-antialiased).

3.5 Varying the Maximum Tree Depth

Recall that the maximum tree depth, described in Section 2.3, is imposed to avoid excessive tree depth near discontinuity boundaries. We considered maximum depths ranging from 22 to 30—corresponding to angular similarity ranges of $11°$ down to $2.7°$ (Because this is a kd-tree in 4-space, four levels of descent are generally required to halve the the diameter of a cell). The results for the Bezier surface scenes are shown in Figure 8. The angular threshold is fixed at $30°$, and the distance threshold is fixed at 0.05.

As the tree is allowed to grow up to a higher depth, rays are sampled with increasing density in the regions where the geometric attributes have greater variation, and thus, error committed by the interpolation algorithm decreases with higher depths. The speedup graph shows a more interesting behavior. Up to a certain depth, the speedup increases with depth. Speedups are poor for very low-depth trees, since many of the interpolants cannot pass the angular threshold test, and so many rays need to be traced rather than interpolated. However, the speedup decreases with very large depth values, since the overhead caused by denser sampling starts to dominate.

It seems that a wise choice of depth would be a value that results in both a lower error, and reasonable speedup. For example for the given graph, depth around 28 would be a good choice for this image. (This corresponds to angular similarity constraint of roughly $3.8°$.) However, peak performance depends on a number of parameters that are particular to the ray tracing application, such as the expected cost of a single ray shoot. In addition, Tables 4–6 shows the required memory when depth is varied. When the tree is unnecessarily deep, not only does the speedup decrease, but space requirements increase as well.

3.6 Varying the Cache Size

As mentioned earlier, the RI-tree functions as an LRU cache. If an upper limit for the available memory—the cache size—is specified, the least recently used paths are pruned based on time stamps set whenever a path is accessed. Excessively small cache sizes can result in frequent regeneration of the same cells. For the Bezier surface scene, we have varied the cache size from 0.128 to 2.048 megabytes (MB). The resulting speedup graph is shown in Figure 9.

13
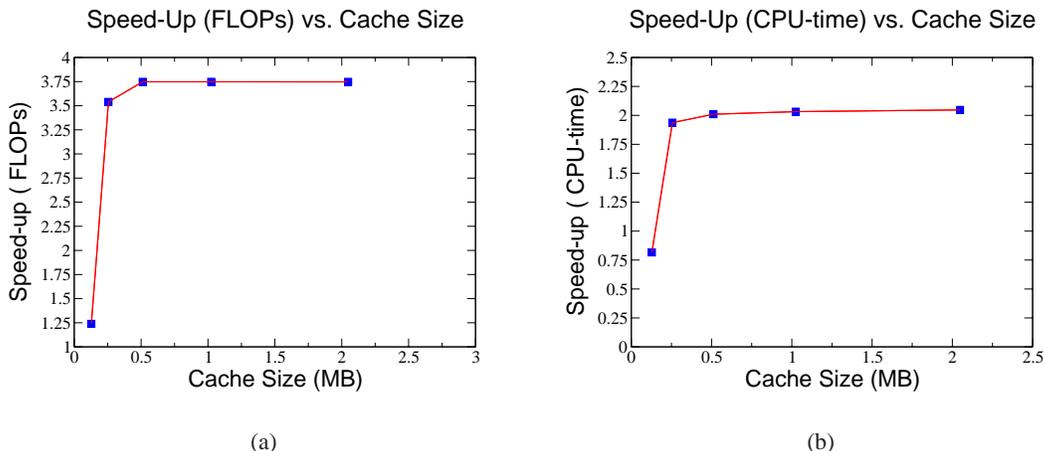
Figure 9: Varying cache size (distance threshold = 0.05, angular threshold = 30, maximum tree depth = 28, 600×600 image, non-antialiased).

Notice that we used small cache sizes to demonstrate the sudden increase in speedup as the cache size approaches a reasonable value. Normally, we set the cache size to 100MB which is high enough to handle bigger scenes with many data structures. There are additional parameters involved in garbage collection, such as what percentage of the cache should be pruned. In these experiments, each garbage collection prunes 70% of the cache.

3.7 Volume Visualization Experiments

We have tested the algorithm for the volume visualization application using the same random volumes we used for refractive objects. Images are 600×600 and not antialiased. Results of our sample runs are shown in Table 4-6. The FLOP speedup varies from 2.817 to 3.549, and CPU speedup varies from 2.388 to 2.814. For higher resolutions, or anti-aliased images the speedups could be higher. The error could be as low as 0.008 for low distance thresholds, and is still at a reasonable value for higher thresholds. Figure 15 shows the actual image, and the interpolated image visualizing one of the random volume scenes. All objects have 0.5 opacity, and all have solid gray colors.

Dist Thresh	Ang Thresh	Tree Depth	Speedup (FLOP)	Speedup (CPU-time)	Error	Memory (MB)
0.25	30	28	2.65	1.89	0.00482	34
0.05	10	28	2.40	1.75	0.00190	47

Table 1: Sample results for tomatoes scene (1200×900 non-antialiased).

3.8 Performance and Error for Tomatoes Scene

Finally, we have tested the algorithm on the tomatoes scene generating an image of size 1200×900, non-antialiased. Table 1 shows sample results for the tomato scene and Figure 12 shows the corresponding

images. Figure 12(a) shows the ray-traced image. Part (b) shows the interpolated image, and a corresponding color-coded image in which the white regions denote the pixels that were traced rather than interpolated. Part (c) shows the interpolated image generated with lower thresholds and the corresponding color-coded image. Notice that the artifacts in part (b) are corrected in part (c).

Note that the closest objects along the eye rays are correctly determined by interpolation, as are the reflection rays from the wrap and the bowl, and the shadows. The sky is reflected on the wrap. As expected, for lower threshold values we can get a very high quality image and still achieve speedups of 2 or higher. If quality is not the main objective, we can get approximate images at higher speedups. The error given is the average RGB-error as explained above.

3.9 Radiance versus Ray Interpolation

As we have mentioned before, in a ray-tracing application, we prefer interpolating normals (for a simple object), rather than interpolating radiance. Consider the Bezier surface scene, where the surface is simple (neither reflective nor refractive), but the specularity is high. Figure 10 shows the ray-traced image[1] of a part of the surface illuminated by 2 light sources.

(a)   (b)
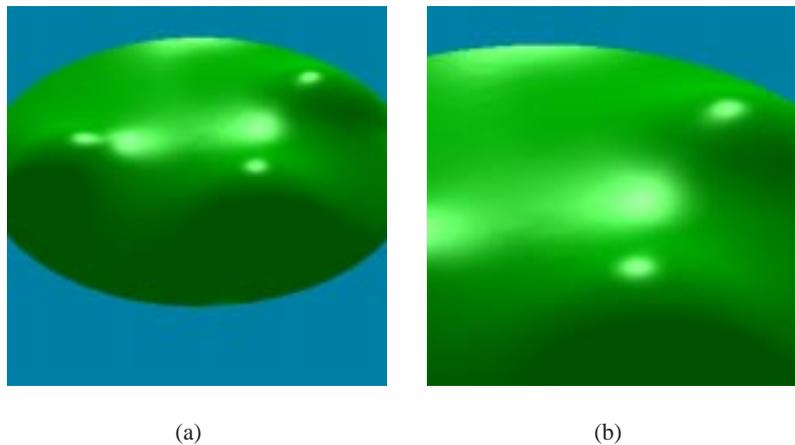
Figure 10: (a) Ray-traced simple Bezier surface (b) Upper right part zoomed.

The comparison of radiance versus normal interpolation is somewhat analogous to the difference in the interpolation methods used in Gouraud and Phong shading. If we choose to interpolate radiance, similar to Gauraud shading, we may have to collect samples much more densely, in the regions near specular highlights in order to achieve greater fidelity, since in those regions radiance varies rapidly. However, when we choose to interpolate normals and then compute shading with respect to the interpolated normal, dense sampling is only required in areas where the surface is not smooth enough, regardless of the radiance.

Consider Figure 11. In part (a), the leftmost image shows the surface rendered by using RI-tree, where normals are interpolated. The maximum depth allowed is 28, and this is enough to obtain a high quality approximation to the ray-traced image in Figure 10. The middle image is a visualization of the RI-tree cells. The rightmost image color codes the depth of the leaf cell used to interpolate the associated pixel. The

---

[1]For full resolution images visit http://www.cs.umd.edu/~mount/Projects/InterpOLF.

15

(a)

(b)

(c)

Leaf Depth   0   8   16   24   32
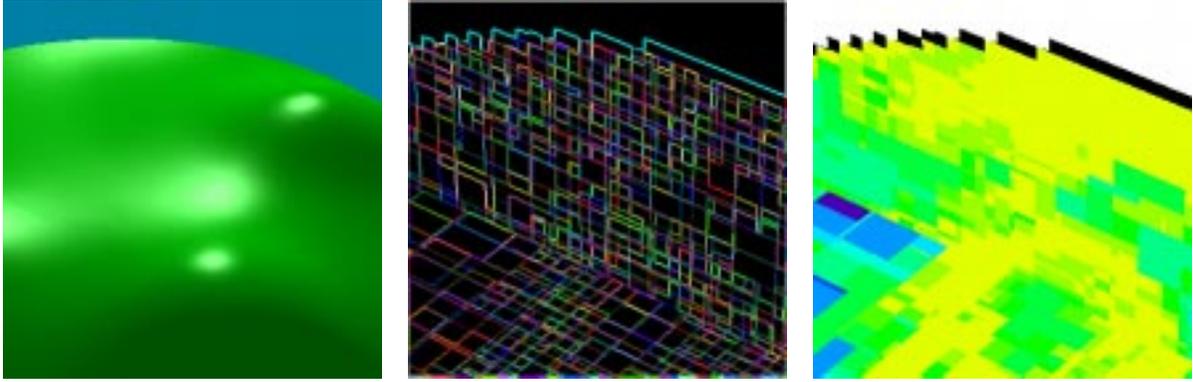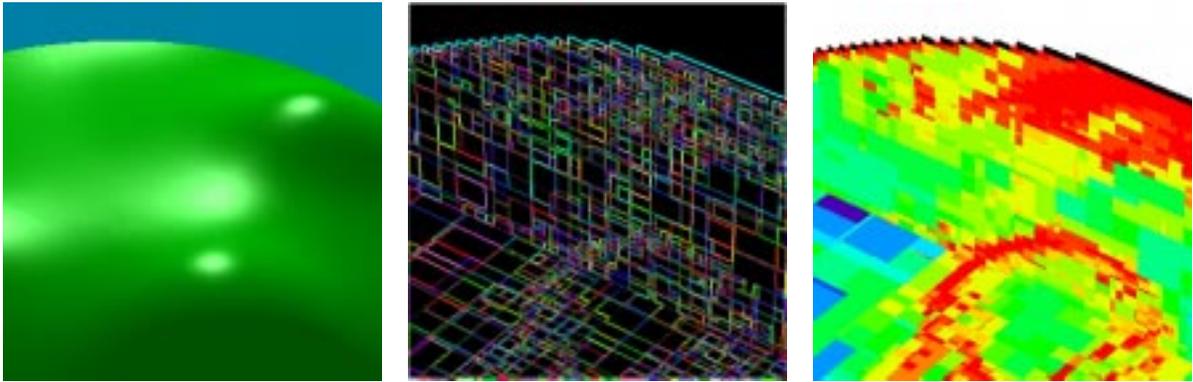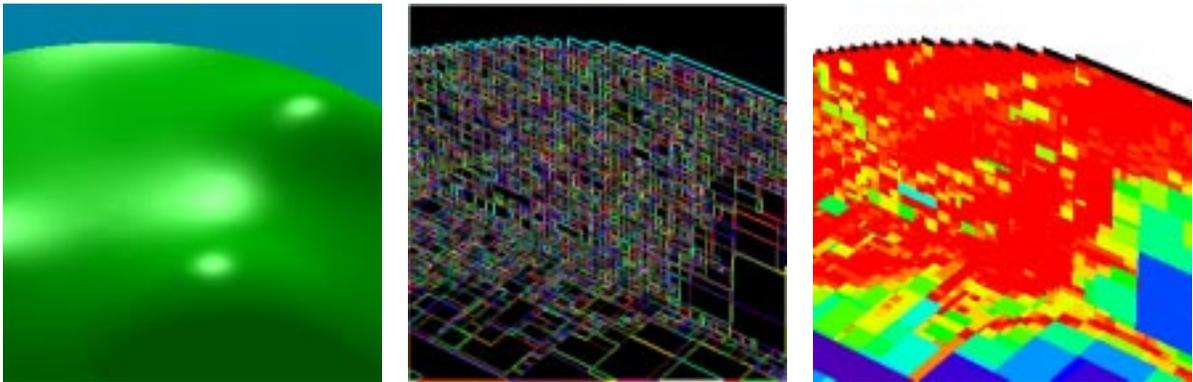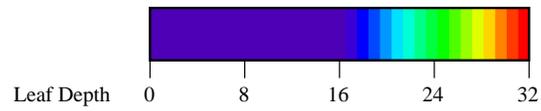
(d)

Figure 11: (a) Normal-interpolation, max. depth = 28, no. of nodes = 7.4K. (b) Normal-interpolation, max. depth =32, no. of nodes = 13K. (c) Radiance interpolation, max. depth = 32, no. of nodes = 25K. (d) Depth-color scale.

16

depth-color scale is given in part (d). Part (b) gives the corresponding images when the depth is allowed to grow up to 32, some cells are refined more around regions of high curvature.

Part (c) shows the images when the image is rendered by radiance interpolation such that the tree is refined according to the variance in radiance. To generate an image of comparable quality to (a) and (b), the maximum depth should be set to at least 32. (For values lower than 32, the highlights are noticeably distorted.) Number of nodes are 7.4K, 13K, and 25K for parts (a), (b) and (c) respectively. Note that Part (a) is a high quality approximation to the ray-traced version, and to achieve the same quality by radiance interpolation, 3 times more nodes should be generated.

3.10 Animations

In an animation sequence, since many samples will be reused by subsequent frames, we expect that the performance gain after the first frame would be higher. We have tested the algorithm on the following two animation sequences. In both cases, the degree of speedup for the first frame was considerably lower than subsequent frames, since the data structure is built from scratch for the first frame, and subsequent frames can reuse some or all of the existing structure.

Light animation: In the first sequence, we use our tomatoes scene, illuminated by only one spotlight. During the animation, the viewpoint is fixed, but the spotlight is swinging, thus the illumination in the scene is different in each frame. Since, the viewpoint is fixed, no more nodes/samples are generated after the first frame, and so there is no sampling cost. The performance for the first three frames are given in Table 2. The speed-ups are roughly constant after the first frame.

Viewpoint animation: In the second sequence, we use the original tomatoes scene illuminated by 9 light sources. In each frame, the viewpoint is rotated around the cup by $1°$. And so, even though many samples are reused, some new samples are also generated in each frame. The performance for the first three frames are given in Table 3. The speed-ups are roughly constant after the first frame.

Dist Thresh	Ang Thresh	Tree Depth	Speedup (FLOP)			Speedup (CPU-time)		
			Frame1	Frame2	Frame3	Frame1	Frame2	Frame3
0.25	30	28	2.35	2.98	2.98	1.83	2.23	2.21
0.05	10	28	2.02	2.67	2.67	1.67	2.04	2.09

Table 2: Sample results for light animation (1200×900 non-antialiased).

Dist Thresh	Ang Thresh	Tree Depth	Speedup (FLOP)			Speedup (CPU-time)		
			Frame1	Frame2	Frame3	Frame1	Frame2	Frame3
0.25	30	28	2.65	2.97	2.98	1.89	2.04	2.02
0.05	10	28	2.40	2.83	2.83	1.75	1.99	1.98

Table 3: Sample results for viewpoint animation (1200×900 non-antialiased).

4 Conclusions

In this paper, we have introduced the RI-tree data structure and illustrated its use in the context of a ray-tracing application. We have shown that the RI-tree can produce high-quality renderings faster than ray-tracing by storing an adaptively sampled set of rays and associated with them a number of geometric attributes, and then using inexpensive interpolation methods to approximate the values of these attributes for new input rays. The data structure is built dynamically, generating samples when they are required, and maintained as an LRU cache to bound the memory used.

Our approach is most useful for rendering smooth objects that are reflective or transparent, for rendering animations when the viewpoint varies smoothly or when the illumination varies from frame to frame, and for generating high resolution images and/or antialiased images generated by supersampling in which multiple rays are shot for each pixel of the image.

The performance gain is achieved at the potential expense of quality. However, our system detects and deals with the object boundaries and other strong discontinuities where the artifacts are more likely to be noticed. We have demonstrated the performance-quality tradeoff by experimenting with the few parameters that control the quality of approximation.

Although, we have explained the data structure in the context of a ray-tracing application, storing and interpolating certain geometric attributes, the data structure is flexible enough to interpolate any information, and it could be used for any other application having to do with lines in 3-space, given a sampling routine, and a measure of discrepancy between samples.

Note that the kd-tree decomposition is not a cell complex, and hence the interpolation is not continuous. Although there are methods which will further subdivide the kd-tree to produce a cell complex, these approaches do not scale well with dimension due to the exponential increase in the number of vertices and explosion of cases that need to be considered. This raises the question of whether we can implement a hierarchical cell complex in 4-dimensional space, which would allow us to perform interpolation queries efficiently. We are currently working on developing such a structure.

References

[1] J. Amanatides. Ray tracing with cones. *Computer Graphics (Proc. of SIGGRAPH 84)*, 18(3):129–135, 1984.

[2] B. Aronov, H. Bronnimann, A. Y. Chang, and Y. Chiang. Cost prediction for ray shooting. In *Proc. 18th ACM Symp. on Comput. Geom.(SoCG'02)*, pages 293–302, 2002.

[3] B. Aronov, H. Bronnimann, A. Y. Chang, and Y. Chiang. Cost-driven octree consruction schemes: An experimental study. In *Proc. 19th ACM Symp. Comput. Geom. (SoCG'03)*, pages 227–236, 2003.

[4] B. Aronov and S. Fortune. Approximating minimum weight triangulations in three dimensions. *Discrete Comput. Geom.*, 21(4):527–549, 1999.

[5] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *Computer Graphics (Proc. of SIGGRAPH 87)*, 21(4):196–205, 1987.

[6] F.B. Atalay and D.M. Mount. Ray interpolants for fast ray-tracing reflections and refractions. *Journal of WSCG (Proc. International Conf. in Central Europe on Comp. Graph., Visualization and Comp. Vision)*, 10(3):1–8, 2002.

[7] K. Bala, J. Dorsey, and S. Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. on Graph.*, 18(3), August 1999.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. of ACM*, 18(9):509–517, 1975.

[9] J. Bloomenthal. *An Introduction to Implicit Surfaces*. Morgan-Kaufmann, San Francisco, 1997.

[10] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: A new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):371–382, 1995.

[11] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray tracing. In *Proc. 13th ACM Symp. Comput. Geom. (SoCG'97)*, pages 11–20, 1997.

[12] J. B. Van de Kamer and J. J. W. Lagendijk. Computation of high-resolution SAR distributions in a head due to a radiating dipole antenna representing a hand-held mobile phone. *Physics in Medicine and Biology*, 47:1827–1835, 2002.

[13] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modelling*. Academic Press Professional, San Diego, 1998.

[14] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.

[15] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray tracing system. *IEEE Comp. Graph. and Appl.*, 6(4):16–26, April 1986.

[16] A. Gershun. The light field. *Journal of Mathematics and Physics*, XVIII:51–151, 1939. Moscow, 1936, Translated by P. Moon and G. Timoshenko.

[17] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Comp. Graph. and Appl.*, 4(10):15–22, October 1984.

[18] A. S. Glassner(editor). *An Introduction to Ray Tracing*. Academic Press, San Diego, 1989.

[19] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 43–54, August 1996.

[20] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Computer Graphics (Proc. of SIGGRAPH 84)*, 18(3):119–127, July 1984.

[21] M. R. Kaplan. Space tracing a constant time ray tracer. *State of the Art in Image Synthesis (SIGGRAPH 85 Course Notes)*, 11, July 1985.

[22] T. L. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proc. of SIGGRAPH 86)*, 20(4):269–278, August 1986.

[23] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 31–42, August 1996.

[24] J.S.B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. *International Journal of Computational Geometry*, 7(4):317–347, 1997.

[25] P. Moon and D. E. Spencer. *The Photic Field*. MIT Press, Cambridge, 1981.

[26] M. Ohta and M. Maekawa. Ray coherence theorem and constant time ray tracing algorithm. *Computer Graphics 1987 (Proc. of CG International '87)*, pages 303–314, 1987.

[27] S. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics (Proc. of SIGGRAPH 80)*, 14(3):110–116, July 1980.

[28] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[29] D. M. Y. Sommerville. *Analytical Geometry in Three Dimensions*. Cambridge University Press, Cambridge, 1934.

[30] D. Zorin, P. Schröder, and W. Sweldens. Interpolating subdivision for meshes with arbitrary topology. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 189–192, 1996.

Test Input	Dist Thresh	Speedup (FLOP)	Speedup (CPU-time)	Error	Memory (MB)
Bezier Surface	0.010	3.12704	1.96466	0.00377	2.925
	0.025	3.43796	1.99712	0.00483	2.371
	0.050	3.74473	2.07705	0.00676	1.931
	0.075	3.93950	2.11372	0.00858	1.699
	0.100	4.08325	2.24707	0.0103	1. 549
	0.125	4.19358	2.24816	0.01185	1.442
	0.150	4.28194	2.29041	0.01331	1.361
	0.175	4.35214	2.32532	0.01532	1.301
	0.200	4.41940	2.32863	0.01655	1.253
	0.225	4.48503	2.34465	0.01763	1.212
	0.250	4.52146	2.34591	0.01867	1.185
Random Volumes	0.010	3.12173	2.63532	0.00627	19.252
(ray-tracing)	0.025	3.26194	2.64317	0.00645	17.518
	0.050	3.40527	2.71941	0.00679	15.799
	0.075	3.49906	2.76194	0.00722	14.765
	0.100	3.56870	2.79244	0.00780	14.088
	0.125	3.62689	2.84409	0.00853	13.603
	0.150	3.67422	2.88046	0.00890	13.183
	0.175	3.70776	2.89190	0.00945	12.875
	0.200	3.74041	2.92770	0.00989	12.583
	0.225	3.77341	2.94416	0.01048	12.331
	0.250	3.80292	2.91917	0.01076	12.094
Random Volumes	0.050	2.95084	2.42804	0.00850	11.773
(volume	0.150	3.31043	2.67274	0.01179	9.503
visualization)	0.250	3.54958	2.81416	0.01488	8.344

Table 4: Varying the distance threshold: Speedup and actual error on Bezier Surface and Random Volumes (ray-tracing and volume visualization).

Test Input	Tree Depth	Speedup (FLOP)	Speedup (CPU-time)	Error	Memory (MB)
Bezier Surface	22	3.50486	2.05642	0.02098	0.565
	23	3.86223	2.14654	0.01491	0.706
	24	4.05344	2.21946	0.01112	0.881
	25	4.03178	2.17521	0.01032	1.084
	26	3.97010	2.15906	0.00953	1.318
	27	3.85335	2.05680	0.00760	1.603
	28	3.74473	2.07705	0.00676	1.931
	29	3.53944	2.04811	0.00663	2.265
	30	3.36016	1.97434	0.00653	2.629
Random Volumes	22	3.10450	2.56453	0.01859	3.729
(ray-tracing)	23	3.41967	2.63197	0.01708	4.431
	24	3.70909	2.74675	0.01526	5.441
	25	3.85445	2.90357	0.01449	6.560
	26	3.85108	2.93271	0.01305	7.989
	27	3.84435	2.87188	0.01187	9.660
	28	3.80292	2.91917	0.01076	12.094
	29	3.56893	2.79997	0.01026	14.361
	30	3.34045	2.73197	0.00987	17.413

Table 5: Varying the tree depth: Speedup and actual error on Bezier Surface and Random Volumes (ray-tracing and volume visualization).

Input Scene	Ang Thresh	Speedup (FLOP)	Speedup (CPU-time)	Error	
Bezier Surface	5	2.68103	1.68226	0.00424	
	10	3.51840	2.01129	0.00591	
	15	3.68553	2.11734	0.00663	
	20	3.72731	2.12195	0.00673	
	25	3.74471	2.11754	0.00676	
	30	3.74473	2.07705	0.00676	
Random Volumes	5	2.56317	2.15410	0.00478	
(ray-tracing)	10	3.44274	2.67800	0.00896	
	15	3.70928	2.83973	0.01007	
	20	3.76320	2.88208	0.01055	
	25	3.78206	2.89311	0.01063	
	30	3.80292	2.91917	0.01076	
Random Volumes	10	2.81703	2.38833	0.01047	
(volume	15	3.21517	2.62693	0.01340	
visualization)	20	3.40653	2.73348	0.01411	
	30	3.54958	2.81416	0.01488	

Table 6: Varying the angular threshold: Speedup and actual error on Bezier Surface and Random Volumes (ray-tracing and volume visualization).
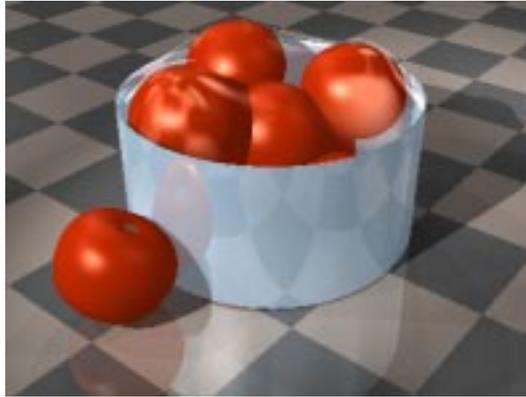
(a)
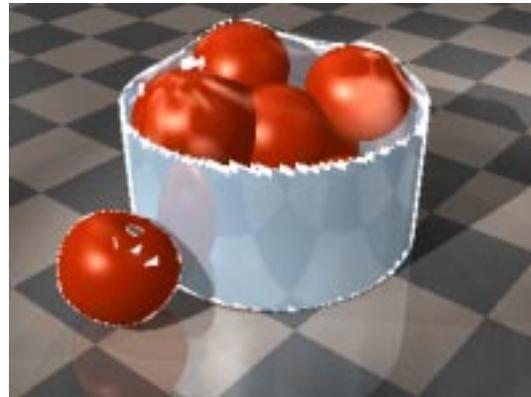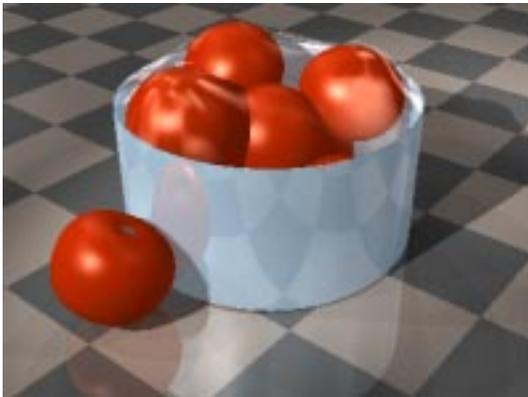
(b)
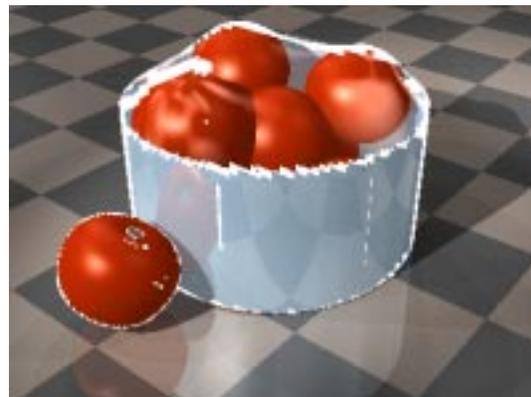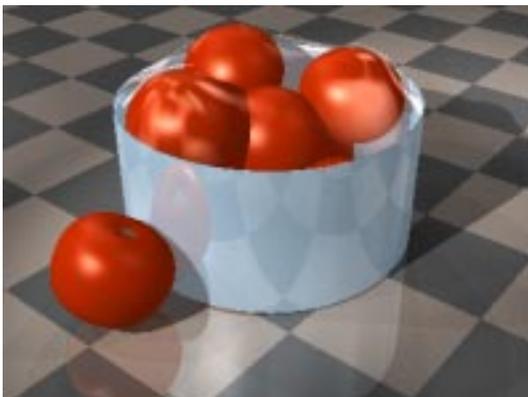
(c)

Figure 12: (a) Ray-traced image, (b) Interpolated image (distance threshold=0.25, angular threshold=30) and corresponding color-coded image, white areas show the ray-traced regions, (c) Interpolated image (distance threshold=0.05, angular threshold=10) and corresponding color-coded image, showing ray-traced pixels.
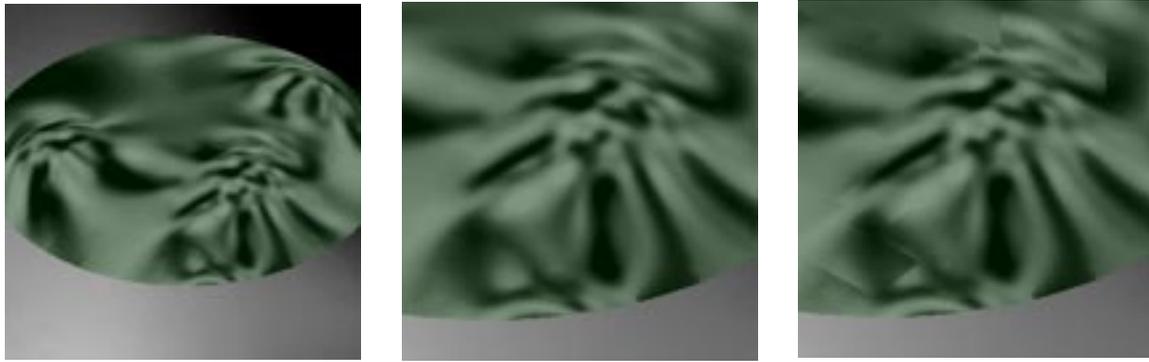
(a)   (b)   (c)

Figure 13: (a) Ray-traced image, (b) Lower right part of interpolated image (distance threshold=0.01), error = 0.00377, (c) Lower right part of interpolated image (distance threshold=0.15), error = 0.01331.
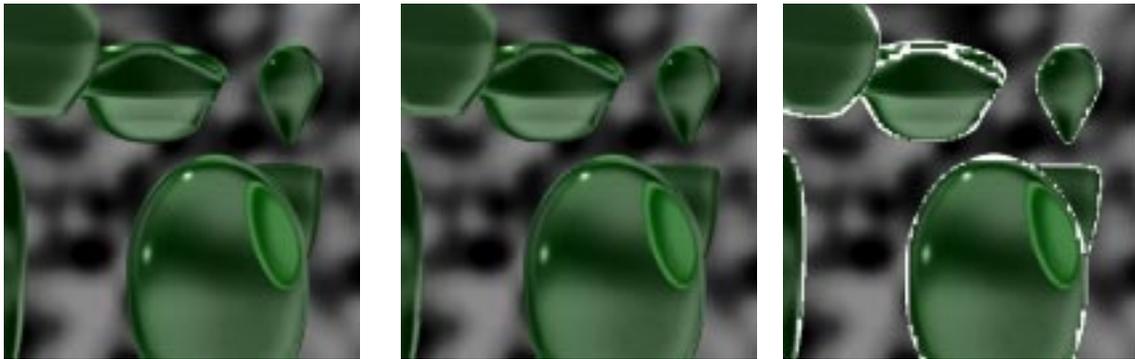
(a)   (b)

Figure 14: (a) Ray-traced image, (b) Interpolated image (distance threshold=0.05) and the corresponding color-coded image where white regions indicate pixels that were ray-traced.
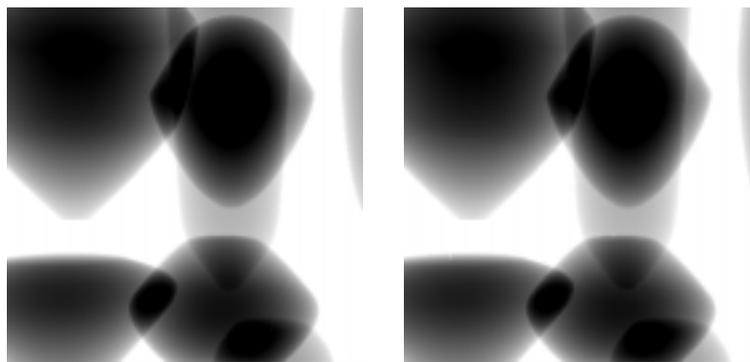
(a)   (b)

Figure 15: (a) Ray-traced image, (b) Interpolated image (distance threshold=0.25).