Ray Interpolants for Fast Ray Tracing of Reflective and Refractive Objects

F. Betul Atalay [*] David M. Mount[†]

Abstract

To render an object from multiple viewpoints by ray tracing, each frame is computed by tracing one or more rays from the viewpoint through every pixel of the image plane. For reflective and refractive objects, especially if multiple reflections and/or refractions occur, this requires many expensive intersection calculations. For complex objects, such as Bezier or NURBS surfaces, the intersection computations are even more expensive. This paper presents a new method for accelerating ray tracing of complex reflective and refractive objects by substituting accurate-but-slow intersection calculations with approximate-but-fast interpolation computations. Our approach is based on modeling the reflective/refractive object as a function that maps input rays entering the object to output rays exiting the object. As preprocessing, a quadtree-based two level data structure is built, storing discrete samples of this function. Sampling is done adaptively from multiple viewpoints in various directions. During rendering, rather than tracing each input ray through the object, we interpolate the collection of nearby ray samples to compute an approximate output ray. In most cases, object boundaries and other discontinuities are handled by applying various heuristics. We also rely on dense sampling in discontinuity regions, due to our adaptive sampling mechanism. In cases where we cannot find sufficient evidence to interpolate, we perform ray tracing as a last resort. We provide performance studies to demonstrate the efficiency of this method.

1 Introduction

High quality, physically accurate rendering of complex illumination effects such as reflection, refraction, and specular highlights is highly desirable in computer-generated imagery. The most popular technique for generating these effects is ray tracing [25]. However, ray tracing remains a computationally expensive technique. The primary expense in ray tracing lies in the intersection calculations, particularly for the scenes that contain complex objects, and in case of multiple reflections and/or refractions. Early research concentrated on accelerating ray tracing by reducing the cost for intersection computations using bounding volume hierarchies [13, 21], space partitioning structures [8, 12], and methods exploiting ray coherence [3, 4, 10, 19].

In this paper, we present a method to accelerate ray tracing of reflective and refractive objects. Our method would be most useful when the same object is rendered from multiple viewpoints in a sequence of frames. Our approach is based on modeling the reflective/refractive object as a function f that maps input rays entering the object to output rays exiting the object. We are interested in computing the output ray without actually tracing the input ray through the object. This is achieved by adaptively sampling rays from multiple viewpoints in various directions, as a preprocessing phase, and then interpolating the collection of nearby samples to compute an approximate output ray for any input ray. By this method, the object can be rendered from any viewpoint.

Recent research has focused on interactive ray tracing [20] and accelerating animation sequences. This requires fast generation of ray traced images from multiple viewpoints. These systems exploit frame-to-frame coherence. Other systems reuse pixels from the previous frame by reprojection and only recompute or possibly refine the potentially incorrect pixels [2, 22]. Walter, *et al.* cache the results while rendering a frame and reproject previously cached samples to approximate the current frame [24]. Similarly, in Larson's Holodeck system, rays are computed, cached and reused for subsequent frames by utilizing a 4D data structure [14].

Besides using reprojection to accelerate visibility, the Interpolant Ray Tracer system described by Bala, Dorsey and Teller in [5] introduced the concept of *radiance interpolant*—shading is accelerated by quadrilinearly interpolating radiance samples cached in an adaptive 4D data structure while conservatively bounding the error. Similar to our system, they rely on the fact that radiance is a smoothly varying function over the ray space most of the time, and a sparse set of samples can be interpolated to approximate radiance. However, we differ in that we are primarily interested in fast rendering of reflective and refractive objects from multiple viewpoints. Our data structure is designed to map rays to rays rather than mapping rays to radiance, and we interpolate among rays. With ray interpolation, handling textures does not require any extra effort. To render reflected textures, the Interpolant Ray Tracer system

[*]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. Research supported in part by the National Science Foundation under grant CCR–9712379.

shoots additional reflection rays, which can be expensive especially for multiple reflections/refractions. Their quadrilinear interpolation requires that the ray trees of all sixteen samples used for interpolation be identical to constitute a valid interpolant. For reflective/refractive objects this strong requirement significantly reduces the cases where interpolation could be substituted for ray tracing. Instead, we apply heuristics that would allow us to use interpolations in more cases while trading off quality to some extent. Also, in addition to simple objects, our method supports rendering bicubic patches, and this creates a variety of other issues which will be described in detail later.

Image-Based Rendering (IBR) methods constitute another line of research to support interactive rendering of scenes. A good survey can be found in [17]. These systems capture and store a database of reference images of a scene from a set of viewpoints, and use them to render the scene from new positions. Among the IBR methods, the most relevant to our work is the Lumigraph [9] and Light Field Rendering [15] techniques. Both systems are based on dense sampling of the *plenoptic function* [1]. The plenoptic function is a 5D quantity describing the flow of light at every position (x, y, z) for every direction (θ, ϕ). By considering only the light leaving a bounded object (or scene), the domain of the plenoptic function can be reduced to 4D, since the radiance along a ray is constant. Each ray is represented by its intersection points (s, t) and (u, v) on two parallel planes, and hence as a 4D quantity. These systems have a preprocessing phase where the 4D function is sampled by uniformly subdividing in all four dimensions, resulting in a regular grid structure on both planes. The radiance along any ray from any viewpoint can then be approximated by quadrilinearly interpolating the radiance values for the nearest sixteen ray samples. To have reasonable quality of complex effects such as reflection, refraction and specular highlights, these methods should sample very densely.

Camahort, *et al.* proposed alternative ray parameterizations to acquire and reconstruct light fields in a nearly uniform fashion [7]. Sloan, *et al.* considered a taxonomy of methods to improve the performance of lumigraph rendering trading off quality for time [23].

Heidrich, *et al.* proposed a light field method focusing on rendering refractive objects [11]. Their method has similarities to ours in that they use a mapping from rays to rays, and interpolate among rays. However, since their system is built on a lumigraph/light field structure, it relies on dense sampling of the rays for capturing clear object boundaries and handling discontinuities. This results in the main problem with the light field methods: large storage requirements. Our method, on the other hand, samples rays adaptively and applies a variety of heuristics to achieve high quality discontinuity rendering at lower sampling rates. Moreover, we store normal vectors and intersection points for sampled rays and approximate normals and intersection points for the query rays by using similar interpolation mechanisms in order to compute the local illumination for the object.

There have been approaches other than ray tracing to render fast approximations of reflective/refractive objects. The oldest such method is environment mapping [6]. It is based on the assumption that the environment is sufficiently far away from the reflective object. We do not impose such restrictions in our system. Another novel method explained in [18] is based on mirroring the scene objects with respect to a reflector. These virtual objects are rendered as ordinary objects to generate a reflection image which is later blended with the primary image. Their method works for curved reflectors relying on high resolution tesselation of both the reflector and the reflected objects and focuses on a single level of reflection. See also [16] on mirroring the scene objects with respect to planar reflectors.

The rest of the paper is organized as follows. In the next section we give a brief overview of our algorithm. In Section 3, the sampling phase is presented explaining the construction of our data structure. Section 4 outlines the rendering phase describing our ray interpolation mechanism and heuristics used for handling discontinuities. In Section 5, the other details of our algorithm such as extrapolation and local illumination interpolation are described. The results of the experiments are presented in Section 6. Finally, we conclude with Section 7.

2 Algorithm Overview

To render an object from multiple viewpoints, each frame is computed by tracing rays from the viewpoint through every pixel of the image. For reflective and refractive objects, especially if multiple reflections and/or refractions occur, this requires many expensive intersection calculations. For complex objects, such as Bezier or NURBS surfaces, the intersection computations are even more expensive. Our algorithm aims to accelerate ray tracing of complex reflective and refractive objects by eliminating intersection calculations, and facilitates fast, approximate rendering of the object from any viewpoint.

Ray-to-ray mapping and ray coherence: The key insight to our method is that a ray intersecting a reflective or refractive object goes through a set of reflections or refractions, and finally exits the object as an output ray. Therefore, we can think of the object as a function f that maps input rays to output rays. (Currently, our method assumes that there is a single output ray for each input ray, so we cannot handle objects that are both reflective and refractive. We leave this as a future enhancement.) For many real world objects which have large smooth surfaces, f is expected to vary smoothly. This is due to ray coherence, that is, nearby rays follow similar reflection/refraction patterns in smooth regions, and so output rays corresponding to nearby input rays are also close to each other. This leads to the idea that, rather than computing each and every output ray by tracing the input ray through the object, we can precompute and store sparse samples of rays in a data structure, and interpolate these samples to get an approximate output ray for any given input ray. Basically, f is discretized by means of a data structure, and an approximation f^* to the actual function f is reconstructed by interpolating the nearby samples during rendering.

For discontinuous regions, however, nearby rays are reflected/refracted in very different directions. In these cases where we cannot rely on the smoothness of f, we sample more densely in the preprocessing phase. We also apply various heuristics while interpolating in the rendering phase. In cases where we cannot find sufficient evidence to interpolate, we perform ray tracing as a last resort.

Data structure: We construct a two-level hierarchical data structure, called *RI-Tree*, which stores adaptively sampled rays from all possible viewpoints in all possible directions. The first level corresponds to the viewpoints from which rays are sampled and it consists of six quadtrees imposed onto the faces of an axis-aligned bounding box enclosing the object. This follows from the fact that sampling the set of viewpoints on the bounding box is sufficient to sample the space of all viewpoints seeing the object. Each viewpoint sample contains a second level of quadtrees corresponding to the hemisphere of possible directions through which rays originating from that particular viewpoint are sampled. These directional quadtrees are adaptively subdivided to provide dense sampling at the regions of discontinuity.

Rendering: Consider the case of rendering an image from a particular viewpoint. We compute an input ray R through each pixel of the image. First, we determine which face of the bounding box is intersected by R. We search for the four closest viewpoints in the quadtree corresponding to the intersected face. Then, in the directional quadtrees of each of the four viewpoints, we locate the ray samples corresponding to the four directions closest to the direction of R. This results in sixteen samples to be used in the interpolation.

Our interpolation method works bottom-up, first interpolating on the directional level, and then on the viewpoint level using the propagated results from the directional level. In the simplest case, all the sixteen samples are usable in the interpolation and our interpolation method is similar to the *quadrilinear interpolation* of the sixteen output rays—four bilinear interpolations on the directional level and one bilinear interpolation on the viewpoint level. However, for the discontinuity regions, some of these sixteen samples are invalidated by various heuristics to avoid interpolation among the rays that are not close to each other. In many cases, we can still interpolate using fewer than sixteen rays. If there are not enough usable ray samples remaining, we simply use the ray tracer to compute the output ray. The output ray is then shot through the rest of the environment to get the reflected/refracted color. Ray interpolation allows us to capture the reflections/refractions of textured environments correctly.

We also store the surface normal vectors and the intersection points along with each sample. These are used to interpolate normal vectors and intersection points for any ray R, so that we can compute the local illumination for the object. The reflected/refracted color and the local illumination are blended using the reflection/refraction coefficients of the object.

The performance gain is achieved at the potential expense of quality. However, slight degradation in quality for reflected/refracted environments is tolerable in many cases. Besides, our system detects and deals with the object boundaries and other strong discontinuities where the artifacts are more likely to be noticed.

3 Sampling Phase

3.1 Representation of Rays as 5D points

In ray tracing implementations, a common way to represent a ray is by its origin and direction vector. Thus, a ray in 3 dimensional space can be viewed as a 6D point. However, geometrically a ray has only five degrees of freedom since two spherical angles are sufficient to define a unique direction vector. This means that we can represent a ray as a 5D point, consisting of its origin and two spherical angles.

We prefer a low dimensional representation, but also we need a representation that will allow us to subdivide the direction space easily. So, instead of using spherical angles, we employ the *direction cube* concept as described by [4] in order to map the 3D direction vector of any given ray to 2D coordinates.

Let $r = (P, \vec{d})$ denote a ray with its origin P, and direction vector $\vec{d}$. Suppose that r is enclosed by an axis aligned cube of side length 2 centered at P. It will hit one of the six faces of the cube depending on its *dominant axis*. The dominant axis of a ray is the axis of largest absolute value component of $\vec{d}$, and can be one of $+X, -X, +Y, -Y, +Z, -Z$. Each face of the cube is labeled by its corresponding dominant axis.

Note that among the rays hitting a single face of the cube, each distinct direction corresponds to a unique intersection point on the cube face. So, once it is determined which face the ray intersects, $\vec{d}$ can be mapped to a 2D point, $(u, v) \in [-1, 1] \times [-1, 1]$, which is the intersection point on that cube face. The direction coordinates, (u, v) are calculated by the

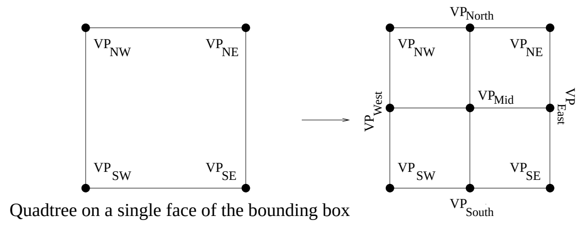


Figure 1. Subdivision in the viewpoint tree

following equations:

$$\vec{w} = \frac{\vec{d}}{\max[|d_x|, |d_y|, |d_z|]},$$

$$(u, v) = \begin{cases} (w_y, w_z) & \text{if} \quad w_x = \pm 1 \\ (w_x, w_z) & \text{if} \quad w_y = \pm 1 \\ (w_x, w_y) & \text{if} \quad w_z = \pm 1. \end{cases}$$

By this method, the ray space is partitioned into six directional groups, and a one-to-one mapping is established between each partition and $[-1, 1] \times [-1, 1]$. Consequently, a ray is represented by its origin, P, its direction group, g, and its direction coordinates (u, v). For a fixed number of direction groups, this can be thought of as a point in 5D space.

3.2 The RI-Tree

In this section, we introduce the main data structure used in our algorithm, the *RI-Tree* which stands for Ray Interpolant Tree. It is a two-level quadtree.

The idea is to enclose our reflective or transparent object within a bounding box, and sample rays originating from viewpoints located on the bounding box in various directions. This is equivalent to sampling rays that originate from any viewpoint in the space and intersect the bounding box. Any such ray can be projected onto the bounding box and the origin of the ray can be reset to the point of intersection with the box. In this way, the space of viewpoints—and so space of rays to be sampled—is reduced.

The viewpoint space consists of all possible viewpoints on the bounding box. The first level of our data structure consists of six independent quadtrees, corresponding to the faces of the bounding box. They recursively decompose the space of viewpoints. We will refer to combination of these six quadtrees as the *viewpoint tree*.

As shown in Figure 1 each leaf cell in the quadtree contains viewpoints corresponding to its four corners. Neighboring cells share viewpoints, but they are not sampled more than once. From each viewpoint, rays will be sampled in various directions. The space of all possible directions from any viewpoint towards the object enclosed by the box constitutes a hemisphere of directions. The hemisphere can be replaced by a *direction hemicube* creating five separate viewing frustums. Independent direction hemicubes for nearby viewpoints are able to better capture the variations in rays that are viewpoint specific, which is important for our approach. Since we already represent the directions as 2D coordinates on a direction cube, we can impose five separate quadtrees on the five faces of the hemicube, thus we have the means to apply subdivision in the direction space.

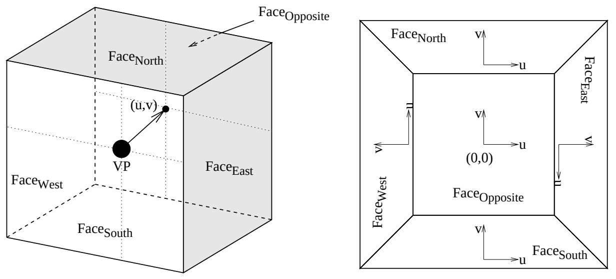
So, each viewpoint in the viewpoint tree contains five directional quadtrees as a second level of trees. We will refer to the combination of these five trees as the *direction tree*. The 2D coordinates of the four corners of each leaf cell in the direction tree represents the directions of the rays we sample. That is, each cell contains four directions corresponding to its four corners. The rays are sampled in the discrete set of directions covered by the direction tree, and the output of sampling, referred to as the *output ray*, corresponding to each direction is stored in the leaf cell containing that particular direction.

3.3 Building the RI-Tree via Adaptive Subdivision

The six quadtrees constituting the viewpoint tree are built independently in the same way. We will describe the construction of one of the quadtrees, let's say the one corresponding to the $+X$ face of the bounding box. In our current model, we apply a uniform subdivision while building the viewpoint tree, so the viewpoint tree is actually a grid structure. However, the quadtree representation will allow us to apply nonuniform subdivision if desired.

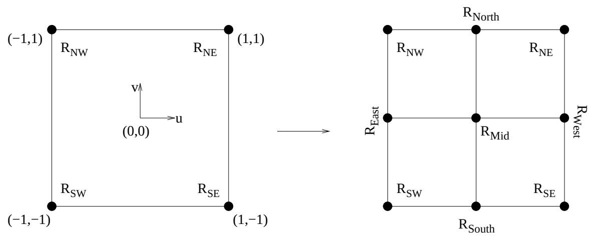
We start with a single cell which covers the whole $+X$ face of the bounding box. Initially we *sample* four viewpoints located at the four corners of this cell. These are referred to as VP_{NW}, VP_{NE}, VP_{SW} and VP_{SE}. We will explain what is meant by *sampling a viewpoint* later. Then, we start recursively subdividing the leaf cells until a termination condition is reached. The subdivision works as follows: Divide the cell into four equal-sized quadrants. Sample new viewpoints at the locations labeled as VP_{Mid}, VP_{North}, VP_{South}, VP_{East} and VP_{West} in Figure 1. Each new leaf cell inherits one viewpoint from its parent, and uses three of the newly sampled viewpoints to assign to its four corners.

What is meant by sampling a viewpoint? As mentioned before, each viewpoint contains five direction quadtrees representing the space of possible directions to sample rays in. Sampling a viewpoint VP refers to constructing the direction tree of that VP. The direction tree construction proceeds in an adaptive manner. To construct a quadtree corresponding to a single face of the hemicube, we start with one cell that covers the whole face. Consider the case of the quadtree for the opposite face of the hemicube. For that face, $(u, v) \in [-1, 1] \times [-1, 1]$. We sample the rays orig-



$\text{Face}_{\text{Opposite}}$
$\text{Face}_{\text{North}}$
(u,v)
VP
$\text{Face}_{\text{West}}$
$\text{Face}_{\text{East}}$
$\text{Face}_{\text{South}}$
$\text{Face}_{\text{North}}$
$\text{Face}_{\text{East}}$
(0,0)
$\text{Face}_{\text{Opposite}}$
$\text{Face}_{\text{West}}$
$\text{Face}_{\text{South}}$
u
v

(a) The Direction Hemicube of VP



(b) Subdivion of the opposite face of the Direction Hemicube

Figure 2. Subdivision in the direction tree

inating from VP in the directions represented by the four corners. To sample a ray—which is referred to as the input ray, we shoot it through our enclosed object and compute an output ray as the final ray that comes out of the object after a possible series of reflections/refractions. We store the output ray in the cell associated with the corner that generated the input ray. In short, our data structure can be thought of as storing associations between the input and output rays. The output rays are stored as 5D points whereas the input rays are deduced from the structure of the quadtree.

After all the four corners have been sampled, we determine whether there is a need for further subdivision. This is where the adaptive nature of subdivisions in the direction tree comes into the picture. The reason for applying adaptive subdivisions is the fact that rays need to be sampled more densely in some regions than others. These are the regions where strong discontinuities exist, causing the reflection/refraction patterns of the nearby input rays differ substantially.

We decide whether to subdivide a cell as follows: We sample the ray with the direction coordinates that coincide with the midpoint of the cell, i.e. (0,0) for the root cell. This gives us the actual output ray computed by ray tracing. Then, we compute an approximated output ray by interpolation of the four already sampled rays which are labeled as R_{NW}, R_{NE}, R_{SW} and R_{SE} in Figure 2 (We describe the interpolation mechanism in the next section). If the output ray and the approximate output ray are close enough, we assume that it is possible to compute a reasonably good output ray for any input ray direction within that cell by interpolation of the output rays corresponding to the cell corners, and we stop subdividing. Otherwise, we divide the cell into four equal-sized quadrants, sample four more rays with the direction coordinates at locations labeled as $R_{North}, R_{South}, R_{East}$, and R_{West} in Figure 2. Then, we assign the appropriate output rays to the the corners of the four new quadrants. We continue this process recursively until no more subdivisions are needed or some termination condition is satisfied.

4 Rendering Phase

4.1 Overview of the Simple Two-level Interpolation

Our objective is to compute the output ray for any input ray without actually tracing the input ray through the object. Instead, we try to utilize the coherence of rays, and use already sampled rays to compute an approximate output ray.

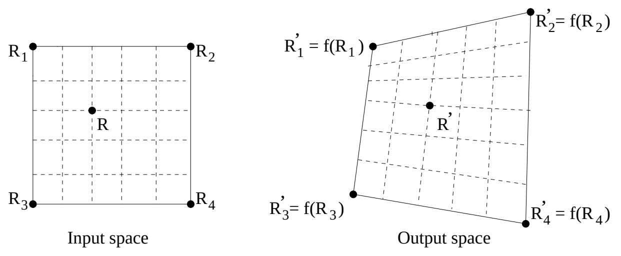


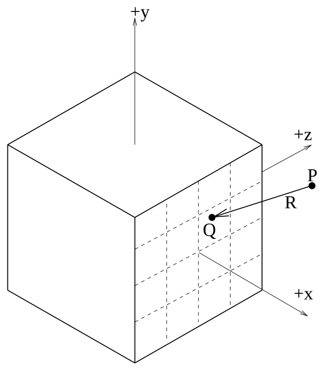
Figure 3.

In Figure 3, let R_1, R_2, R_3 and R_4, represent input rays that were sampled in the preprocessing step. Recall that the rays are modeled as 5D points, but for visualization purposes, we give a 2D illustration of the concept. Our sampling function mapped R_1, R_2, R_3 and R_4 to $R_1{}', R_2{}', R_3{}'$ and $R_4{}'$ respectively. Intuitively, for any input ray, R, within the cell formed by R_1, R_2, R_3 and R_4 in the input space, we can compute an approximate output ray, R', by bilinear interpolation of $R_1{}', R_2{}', R_3{}'$ and $R_4{}'$ in the output space using the bilinear coefficients derived from the input space.

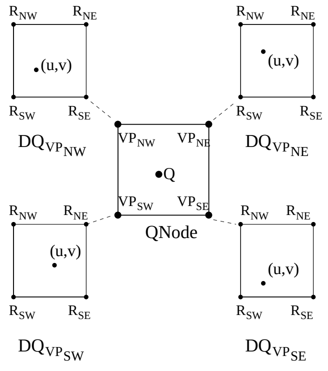
Since our data structure stores the origins and the direction coordinates of the input rays on two separate levels, we apply a two-level interpolation scheme.

Consider the case of computing an approximate output ray for an input ray, $R = (P, \vec{d})$. First, we project this ray onto the bounding box enclosing our object in order to set its origin to a new point on the bounding box face, that is, to a point in our viewpoint space. Assume for the sake of concreteness that, ray R intersects the $+X$ face of the box at point Q. Now, our query ray is represented as $R = (Q, \vec{d})$. Figure 4(a) illustrates this projection.

Next, we locate the quadtree leaf cell in which Q lies. This requires a traversal of the quadtree corresponding to



(a) Projection of ray, R, onto the +X face of the bounding box



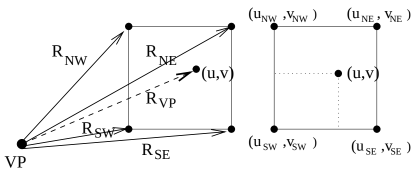
(b) QNode and DQs

Figure 4. Locating the leaf cells containing Q and (u,v)

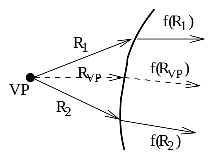
the $+X$ face. Let $QNode$ denote this leaf cell. As explained in the previous section, viewpoints VP_{NW}, VP_{NE}, VP_{SW} and VP_{SE} at the corners of $QNode$ were sampled in the preprocessing phase.

Next, we convert $\vec{d}$ to the direction cube representation, consisting of the direction cube face, g, and the 2D direction coordinates, (u, v). In the direction trees of viewpoints $VP_{NW}, VP_{NE}, VP_{SW}$ and VP_{SE}, we traverse the quadtree corresponding to the face g, and locate the quadtree cell in which (u, v) lies. Let DQ_{VP} denote the leaf cell in direction tree of VP. As shown in 4(b), at this point, we have $DQ_{VP_{NW}}$, $DQ_{VP_{NE}}$, $DQ_{VP_{SW}}$ and $DQ_{VP_{SE}}$ denoting the four leaf cells, one for each direction tree. These four DQs provide us the output rays for a total of sixteen sampled rays originating from the four viewpoints surrounding the origin of the query ray, R, in four directions surrounding the direction of R.

The first phase of interpolations are done at the direction tree level, and the results from that level are propagated up to the viewpoint tree level for one more interpolation to get the final interpolated output ray for R.



(a) DQ_{VP}



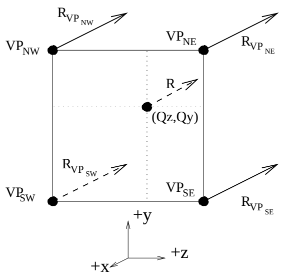
(b) Refraction of similar rays originating from the same viewpoint

Figure 5. Direction tree interpolation

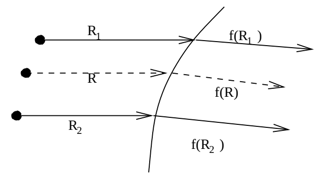
Interpolation at the Direction Tree Level: For each DQ_{VP}, we compute an approximate output ray for the ray labeled as R_{VP} in Figure 5(a). This is the ray originating from VP and has the direction coordinates, (u, v), which are the direction coordinates of the original query ray R. It serves as an intermediate query ray. As shown in the figure, the rays corresponding to the four corners of DQ_{VP} surround the query ray, R_{VP}. We expect that, due to the coherence of rays, R_{VP} would follow a similar path to these four rays. Figure 5(b) gives a 2D illustration of how similar rays originating from the same viewpoint are expected to behave on a single refractive surface. R_1, R_2 and R_{VP} are all originating from VP and R_1 and R_2 surround the query ray, R_{VP}. Assume that R_1 and R_2 are sampled in the preprocessing phase and returned the output rays, $f(R_1)$ and $f(R_2)$. We assume that, if R_{VP} follows a path between the paths of its surrounding rays, $f(R_{VP})$ would be surrounded by $f(R_1)$ and $f(R_2)$. Our case is a 3D analog of this. The rays corresponding to the corners of DQ_{VP} form a pyramid, and R_{VP} lies within that pyramid.

To perform the interpolation, the bilinear coefficients of (u, v) with respect to (u_{NW}, v_{NW}), (u_{NE}, v_{NE}), (u_{SW}, v_{SW}) and (u_{SE}, v_{SE}) are determined. Then, using these coefficients, the approximate output ray for R_{VP}, denoted $f^*(R_{VP})$, is computed by standard bilinear interpolation of $f(R_{NW}), f(R_{NE}), f(R_{SW})$ and $f(R_{SE})$.

Interpolation at the Viewpoint Tree Level: After we compute an interpolated output ray, $f^*(R_{VP})$ for each DQ_{VP}, we propagate these intermediate output rays to viewpoint tree level for the final interpolation. In Figure 6(a), these rays are labeled as $R_{VP_{NW}}$, $R_{VP_{NE}}$, $R_{VP_{SW}}$, and $R_{VP_{SE}}$. These are parallel rays in the direction of our original query ray, R, and originating from the four viewpoints surrounding the origin of R. This implies that R is surrounded by these four R_{VP}s. Now that we have the $f^*(R_{VP})$s computed in the direction tree level, we can apply same kind of interpolation method on these parallel rays to compute $f^*(R)$.



(a) QNode (on +X face)



(b) Refraction of parallel rays originating from different but close viewpoints.

Figure 6. Viewpoint tree interpolation

A 2D illustration of the idea is given in Figure 6(b), where R_1, R_2 and the query ray R are parallel rays originating from different viewpoints. Since R is surrounded by R_1 and R_2, it is expected that R would follow a path between the paths followed by R_1 and R_2, and finally, $f(R)$

would be surrounded by $f(R_1)$ and $f(R_2)$.

As in the direction tree level interpolation, we compute the bilinear coefficients of Q with respect to the four corners of *QNode*. Then, using these coefficients, we compute the output ray for R, $f^*(R)$ by standard bilinear interpolation of $f^*(R_{VP_{NW}}), f^*(R_{VP_{NE}}), f^*(R_{VP_{SW}})$ and $f^*(R_{VP_{SE}})$. Consequently, an approximate output ray for R is computed by the interpolation of sixteen output ray samples.

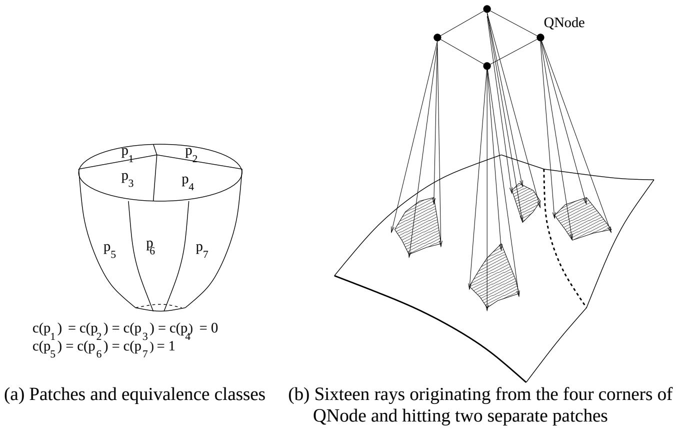
A Note on the Depth of the RI-Tree: As we have explained in the previous section, the viewpoint tree is a fixed depth tree where the appropriate depth is determined empirically. However, the direction tree grows adaptively, and its depth varies at different places. For space efficiency, we impose an upper limit on its depth to prevent the tree from growing excessively. The deeper the tree, the more densely the rays are sampled, so, the interpolation method would produce higher quality images.

4.2 Advanced Interpolation for Discontinuities

The simple interpolation method makes no assumptions about the structure of the object in the scene, and applies the same interpolation procedure everywhere. When there are no strong discontinuities in the scene, the simple interpolation method performs well even when the RI-Tree is not deep. On the other hand, if the ray input-output function contains discontinuities, as may occur at the edges and the outer boundary of the object, then we will observe bleeding of the colors across the edges. This could be remedied by building a deeper tree, which would involve sampling of rays at pixel resolution in the discontinuity regions, but this would result in unacceptably high memory requirements.

To keep the memory requirements reasonably low, instead of building a deeper tree, we devised a series of interpolation heuristics to render the discontinuity regions. We apply the interpolation methods at the smooth regions, and we ray trace the rays at the discontinuity regions. In order to minimize the number of the rays actually traced, we modified the simple interpolation algorithm to be able to use the interpolation methods as much as possible.

Bezier Patches and Equivalence Classes: In order to explain how the discontinuities are detected and handled, we will describe the structure of our objects. Many real-world objects are inherently smooth, so, our algorithm is designed to handle the objects that are specified as a collection of smooth surfaces, referred to as "patches". A patch could be a simple polygonal surface, or a more complex one such as a Bezier or NURBS surface. The patches that share a common edge may or may not be joined with sufficiently high continuity to permit interpolation across the boundary. To provide this information, we use a simple method. We group the patches into equivalence classes. Two adjacent patches in the same equivalence class are assumed



p1
p2
p3
p4
p5
p6
p7
c(p1) = c(p2) = c(p3) = c(p4) = 0
c(p5) = c(p6) = c(p7) = 1
QNode
(a) Patches and equivalence classes
(b) Sixteen rays originating from the four corners of QNode and hitting two separate patches

Figure 7.

to be connected continuously. If two patches are in different classes, that means there is a noticeable edge between them across which we do not want to interpolate. In our object representation, we assign a *patch-identifier* to each patch and we associate each patch-identifier with a *class-identifier* denoting its equivalence class. Figure 7(a) shows an example object made from patches that are in two different equivalence classes.

Storing the Patch-identifiers in the RI-Tree: In order to make use of patch grouping in our interpolation method, for each ray sampled in the preprocessing phase we store the patch-identifier of the first patch that the ray hits. If it does not hit any patches, we store a special patch-identifier, -1, for the outside environment. It is in an equivalence class by itself.

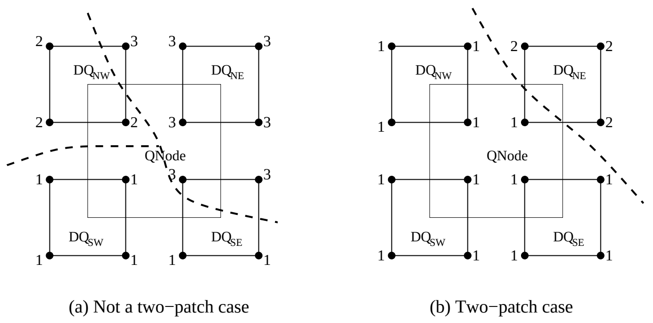


Figure 8. Two-patch case

Two-patch Condition: To compute the output ray for any ray R, as in the case of the simple interpolation, we first locate the four directional quadtree cells containing the sixteen rays to be used as interpolants. Recall Figure 4. But, all the sixteen rays might not be used for interpolation. Moreover, the interpolation method might not be applied at all. The first step is to determine whether to compute the output ray by interpolation or by ray tracing the ray. At this point, we introduce the concept of a *two-patch condition*. Let p_1, p_2,..., p_{16} denote the patch-identifiers associated with the sixteen candidate rays, and $c(p_i)$ denote the class-identifier of patch p_i. If the cardinality of the set $\{c(p_i)|1 \leq i \leq 16\}$ is greater than 2, then we conclude that there is more than one discontinuity boundary in the region surrounded by the intersection points of the sixteen rays with the object. This case is shown in Figure 8(a). In this case, we ray trace the ray, instead of interpolating the result. Otherwise, the *two-patch condition* is satisfied, implying that either all the patches hit by the sixteen rays are in the same equivalence class, or they are divided into two equivalence classes, say c_1 and c_2. In the figure, each corner is labeled with the class-identifier of the patch hit by the ray corresponding to that corner. Note that, for simplicity the discontinuity boundary is depicted on the input ray space. Also, we will use the phrase "discontinuity crossing/overlapping the cell" to mean the discontinuity crossing/overlapping the region surrounded by the points hit by the rays corresponding to that cell.

If all the sixteen patches are in the same equivalence class, then it is reasonable to assume that the query ray R would hit a patch in the same equivalence class as well, so all of the sixteen candidate rays could be used as interpolants. If the patches are partitioned into classes c_1 and c_2, then we assume that there is a single discontinuity boundary dividing the region surrounded by the sixteen ray hits, into two regions, c_1 and c_2. This case is shown in 8(b) and is referred to as the *two-patch case*.

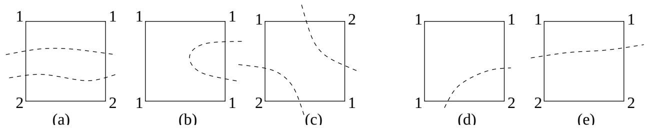


Figure 9. (a)-(c) Problematic cases (d)-(e) Good cases

Before we continue, let us mention a few problematic cases that could arise. Since the knowledge of the "number of different equivalence classes overlapped" is based on the information obtained from the vertices, we might be mistaken. Consider the cases depicted in Figure 9(a), (b) and (c). For simplicity, we illustrate the problematic cases on the region surrounded by the four ray hits associated with a single *DQ*, but they can be generalized to the region surrounded by the sixteen ray hits associated with a *QNode*. In the case shown in Figure 9(a), just by looking at the four corners of the *DQ*, we would decide that the cell is overlapped by two equivalence classes and overlook the third one in between. Figure 9(b) depicts a case where we would conclude that it is continuous region whereas there is a discontinuity boundary overlapping the cell. In the case shown in Figure 9(c), the rays hit patches in two different equivalence classes only, but there are two discontinuity boundaries crossing the cell. We do not detect these cases, and assume that they arise very rarely. So, from this point on, we will assume that if the patches hit by the rays associated with a *DQ* (or a *QNode*) are grouped in two equivalence classes, there is a single discontinuity boundary between them. Then, we assume that we can approximate the discontinuity with a line segment as explained later in this section. The good cases are depicted in Figure 9(d) and (e).

In the two-patch case, suppose that we know which patch the query ray R hits first. Let p_r be the patch-identifier of this patch. If p_r is in the c_1 equivalence class, among the sixteen candidate rays, we want to use only the ones whose first hits are in the c_1 region. We avoid using the ones in the other region since those rays are known to hit a completely different surface and would possibly have very different reflection/refraction directions, even if the input rays are very close to each other. When a candidate ray hits a patch that is in the same equivalence class as p_r, we call it a *usable ray* implying that it could be used for interpolation of the output ray of the query ray R.

However, since the query ray R is not ray traced, we do not know which patch it hits first, that is, we do not know which side of the discontinuity boundary it falls. But, we assume that it should hit one of the patches in $PS = \{p_i|1 \leq i \leq 16\}$. At this point, we compute the exact first intersection point of R with the object, but we check for the intersections only with the patches in PS. Computing an exact intersection point is an expensive operation, but, typically only a few patches are involved (certainly never more than sixteen), and we compute only the first level intersections of a ray tracing procedure. So, this is not as expensive as a general ray tracing of the ray. Besides, this computation is required only in the two-patch cases.

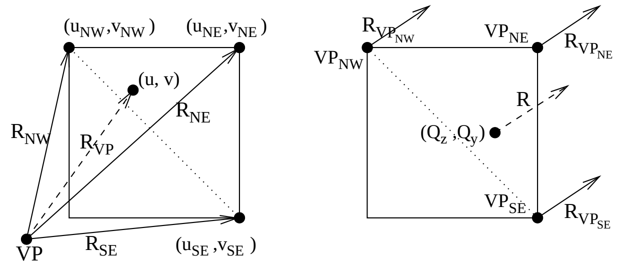
Since we cannot use all of the sixteen rays, and at least three interpolants are required to perform an interpolation, it is possible that some nodes cannot be used for interpolation at all.

The algorithm given in Figure 10 summarizes the two-patch case interpolation. If it is a two-patch case, we proceed with the direction tree interpolation as follows. For each *DQ*, if at least three of the four rays corresponding to the corners are associated with a patch-identifier that is in the same equivalence class as p_r, we can use that *DQ* for interpolation. Otherwise, we eliminate it from the interpolation process.

There are two cases. When all four rays could be used as interpolants, we apply the same direction tree interpolation as in the simple interpolation method. In the case where only three of the rays could be used as interpolants, instead of bilinear interpolation, barycentric interpolation is used. Consider the case in Figure 11(a), where only NW, NE and

```
if (QNode is a two-patch case) then
    determine the patch R hits;
    NumberOfSuccessfulDQs = 0;
    for each of the four DQs do
        if (number of usable rays $\geq$ 3) then
            $/*$ interpolate the DQ $*/$
            NumberOfSuccessfulDQs++;
            compute intermediate output ray
                by interpolating usable rays;
    if ( NumberOfSuccessfulDQs $\geq$ 3) then
        $/*$ interpolate QNode $*/$
        compute $f^*(R)$ using the intermediate
            output rays from successful DQs;
        return $f^*(R)$;
    else
        return failure; $/*$ ray trace the ray $*/$
```

Figure 10. Advanced interpolation algorithm



(a) Three−interpolant direction tree interpolation

(b)Three−interpolant viewpoint tree interpolation

Figure 11. Three-interpolant interpolations

SE corners could be used to interpolate the output ray for R_{VP}. Barycentric coefficients of (u, v) are computed with respect to the coordinates of the NW, NE and SE corners, and the intermediate output ray is calculated by barycentric interpolation of $f(R_{NW}), f(R_{NE})$ and $f(R_{SE})$ using these coefficients. However, there is an issue we have to point at. If (u, v) lies outside the triangular region formed by the NW, NE, and SE corners, this is not an interpolation anymore—it becomes an extrapolation. In this case, especially when (u, v) is very close to the SW corner, the output ray approximation might not be very good. We will explain our strategy on the extrapolations issue later.

After all the *DQ*s are processed, each *DQ* is either eliminated from the interpolation, or $f^*(R_{VP})$ is computed for it. If at least three of the *DQ*s are successfully interpolated, the intermediate output rays are propagated to the viewpoint tree level and the final interpolation is done using the $f^*(R_{VP})$s. Similar to the case of the direction tree interpolation, if there are only three interpolants, barycentric coordinates of Q are computed and used in the interpolation to get $f^*(R)$.

Fig 11(b) illustrates the three-interpolant viewpoint tree interpolation. Four-interpolant case is same as the simple interpolation at the viewpoint tree level. If there are less than three interpolants, we give up interpolation, and ray trace R.

4.3 One- and Two-interpolant Cases and Extra Sampling

We can reduce the number of rays raytraced if we can make use of the *DQ*s with less than three interpolants instead of eliminating them. Some of the *DQ*s could be made usable by sampling extra rays at certain points in the preprocessing phase in order to get three usable rays.

In the preprocessing phase, if a leaf cell is not going to be split any further, but the rays corresponding to the four corners hit patches that are in two different equivalence classes, two cases might arise. The first one is when one of the corners is in one equivalence class by itself where as the other three are in another class. The second one is when two of them are in one equivalence class while the other two are in another equivalence class. In this case, if the two corners which are in the same class are not neighbors, the *DQ* remains unusable, so we eliminate it from consideration. Cases like the one shown in Figure 9(c) are eliminated at this stage.

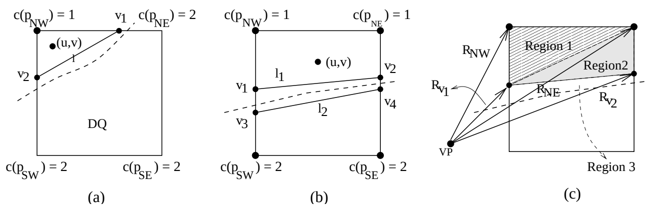


Figure 12. One- and two-interpolant cases

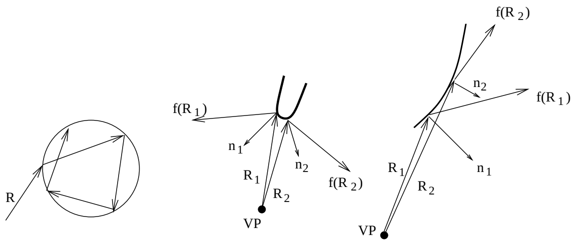
We call the first case the one-interpolant case and the *DQ* is made usable as follows. Consider the example in Figure 12(a). Let p_{NW}, p_{NE}, p_{SW} and p_{SE} denote the patches hit by the rays corresponding to the four corners. For concreteness, assume that $c(p_{NW}) = 1$, and $c(p_{NE}) = c(p_{SW}) = c(p_{SE}) = 2$. In case this node is needed for rendering, and if $c(p_r) = 2$ (that is when the query ray hits a patch in equivalence class 2) then there is no problem, and we can proceed with the interpolation. However, if $c(p_r) = 1$, then two more sampled rays are required to use with the NW corner to perform the interpolation. The dashed curve in the figure depicts the actual discontinuity, and the line segment labeled as l is the approximation we compute to represent the discontinuity. In the preprocessing phase, in order to define l, we find the two points labeled as v_1 and v_2. We use binary search to locate the farthest point from the NW corner on the upper edge of the cell of which the corresponding ray hits a patch p, where $c(p) = 1$. That is v_1. We compute v_2 similarly. Then, we sample the rays with the direction coordinates v_1 and v_2. During rendering, we use these extra sampled rays. To perform the interpolation, we simply use the barycentric coordinates of (u, v) with respect to the single corner that was already usable (as the NW corner in Figure 12(a)), v_1 and v_2. Then, we compute the output ray as in the three-interpolant case.

The second case is called the two-interpolant case. An example is shown in Figure 12(b) where p_{NW} and p_{NE} are in the same equivalence class, while p_{SW} and p_{SE} are in another equivalence class. During the rendering phase, if we want to use this node for interpolation, in either the case of $c(p_r) = 1$ or $c(p_r) = 2$, at least one more point is required to do the interpolation. In the figure, the dashed curve represents the actual discontinuity. We approximate it by a line segment l_1 on one side of the curve and another line segment l_2 on the other side. The points defining l_1 are labeled as v_1 and v_2 and the rays with the direction coordinates at these points have to hit a patch p, where $c(p) = 1$. The points v_3 and v_4 define the line segment l_2 and the rays with the direction coordinates v_3 and v_4 should hit a patch p, where $c(p) = 2$. In the preprocessing phase, we locate these four points by binary search to get the best approximation to the actual discontinuity. And then, we sample extra rays corresponding to them. So, in the rendering phase, we will be able to use the rays corresponding to v_1 and v_2 along with the rays corresponding to the NW and NE corners for interpolation when $c(p_r) = 1$. If $c(p_r) = 2$ then v_3 and v_4 can be used with the SW and SE corners. To illustrate the interpolation in a two-interpolant case, consider Figure 12(c). The cell is subdivided into three regions (denoted Region 1, 2 and 3). Suppose that $c(p_r) = 1$. If (u, v) is in Region 1, barycentric interpolation of R_{NW}, R_{NE} and R_{v_1} is used to interpolate the output ray. If (u, v) is in Region 2, barycentric interpolation of R_{NE}, R_{v_1} and R_{v_2} are used. And, if (u, v) is in Region 3, and we have the option of using extrapolation with barycentric coordinates computed with respect to the NE corner, v_1 and v_2.

5 Other Issues

5.1 Extrapolation

Extrapolation arises when we attempt to estimate output rays for the points outside the convex hull of the sampled points. In our algorithm, extrapolation might be required in the one-, two- or three-interpolant cases. When the extra sampled points are used, the one- and two-interpolant cases boil down to a three-interpolant case where the interpolation is performed using the barycentric coordinates with respect to the three vertices of a triangle. When the point for which we want to compute an interpolated value is outside the triangular region, the value computed is an extrapolated value. Since extrapolated values are less reliable, the user is granted the option of tuning the extrapolation. Extrapolation can be disabled in which case, if an extrapolation is required in a particular cell during the rendering phase, the cell is labeled as unusable. Alternatively, a threshold value can be specified to control at what level the extrapolation will be done. Basically, if the point to be extrapolated is farther from the triangular region—in terms of its barycentric coordinates—than a given threshold value, extrapolation is not used and the cell cannot be used to compute an output ray.



(a) f(R) = null ray (b) R_1 and R_2 are close but $f(R_1)$ and $f(R_2)$ are not.

Figure 13.

5.2 Handling Repeated Reflections and Refractions

Previously, we defined a *usable ray* as a ray that hits a patch in the same equivalence class as the patch hit by the query ray. This was the only constraint for a ray to be used as an interpolant. Here, we introduce a new constraint. Till now, we assumed that our function f returns an output ray for any input ray, however, it is possible that an input ray gets trapped in the object, for example, consider a refractive object in which a ray is repeatedly reflected due to total internal reflection and does not escape the object as depicted in Figure 13(a). We handle this case by defining a special ray called the *null ray*. If an input ray does not escape the object after a fixed number of reflections, the function f returns the *null ray* instead of the output ray. During the rendering phase we do not use such rays as interpolants.

So, we redefine a *usable ray* as a ray that hits a patch in the same equivalence class as the patch hit by the query ray, and its associated output ray is not the *null ray*.

5.3 Handling High Curvature

Normally, we assume that the rays associated with the same cell and that are usable for interpolation are localized to a small area and their reflection/refraction patterns would be similar. However, there might be cases where we have two input rays that are close to each other, but the corresponding output rays are very distant from each other. This case arises especially when the input rays hit at those areas of high curvature—or more generally under any circumstance in which the surface normals vary rapidly for nearby input rays causing them to be reflected/refracted in very different directions. In that case, we do not interpolate among those rays. We use two measures to determine the distance between two output rays. The first one is the Euclidean distance between the origins of the rays. If it is greater than a given threshold, those rays are not usable as interpolants. The second measure is the angle between the direction vectors of the rays. Similarly, if this angle is greater than a given threshold, this signifies that nearby input rays are reflected/refracted in very different directions. Examples of this case are shown in Figure 13(b).

After usable rays are determined by the previous constraints, we apply the distance measures pairwise among the interpolants, and if any pair fails then the interpolation cannot be done.

5.4 Intersection Point and Normal Interpolation for Local Illumination Computations

In the framework described up to now, to render a pixel through which an input rays passes, we have only the means to determine the reflected/refracted color component. The output ray is used to access the environment, and the color of the point it hits constitutes the reflected/refracted color. However, to render realistic images, the object itself has to be shaded. For local illumination computations though, we need the intersection point and the surface normal at that point. Since, we do not actually compute intersections during the rendering phase, we calculate the interpolated values for the intersection point and the normal as well. For this reason, for each ray sampled in the preprocessing phase, we store the first intersection point P of the ray with the object and the normal vector n at that point. Then, during the rendering phase, we apply the same interpolation method that is used for interpolating the output ray. Only the intersection points and normals that are associated with the usable rays are used as interpolants. By using the interpolated values of the intersection point and the normal we can compute an approximate color of the object and the final color is the weighted sum of the object color and the reflected/refracted color.

6 Results

In this section, we present preliminary results of our algorithm. The images are generated on a 400 MHz sparc processor.

Our system is built on a basic ray tracer which is utilized when rays are being sampled during preprocessing, and when interpolation cannot be performed during rendering. Our models are constructed from bicubic Bezier patches. A Bezier patch is subdivided into smaller patches until the patches are flat enough. Each patch is stored as a bounding sphere hierarchy. This simulates a bounding volume hierarchy acceleration method for the ray tracer so that we are fair to the basic ray tracer in our comparisons.

A *ray traced* image is generated by ray tracing each input ray through the reflective/refractive object to compute the output ray. The output ray is then shot through the rest of the environment. In our example images, an “interesting” object is placed in a relatively simple environment. An *interpolated* image is generated by computing the output ray using our interpolation algorithm. We compare the number of floating point operations(FLOPs) and the CPU time for the generation of the output rays and the computation of the local illumination of the object, since that is the part accelerated by our algorithm. These are labeled as “object-only” in the performance tables given below. However, we also provide the total number of FLOPs and CPU time for the entire image. Since the output ray is ray traced through the environment, the total FLOPs and CPU time are not improved as much as those for the object-only case. If we had used the output ray to access an environment map instead, the improvement ratio for the total FLOPs and CPU time would be similar to the object-only ratios. Since CPU time is system dependent, we emphasize the number of FLOPs in our comparisons.

For the images we present, the viewpoint tree has a fixed depth of 5. The direction trees are adaptively divided, and their depths vary between 1 and 6. All images are of 300×300 resolution, and a single input ray is shot through each pixel resulting in a total of 90K rays. For the above depths, the preprocessing phase takes an hour to a few hours depending on the complexity of the object and whether the object is reflective or refractive. The size of the data structure (for a single face of the viewpoint cube) is 189 MB for the reflective bowl, 175 MB for the reflective vase, and 191 MB for the refractive vase examples. In our current implementation, the data structure has not been optimized for space or preprocessing time.

For each image, we provide the correct ray traced image, the image generated by our interpolation algorithm, and a corresponding image showing which parts are successfully interpolated and which parts were ray traced due to unusable interpolants. The white pixels correspond to the ray traced regions.

Figure 15 shows a reflective closed bowl on a procedurally textured table placed in a room. The walls of the room are shaded in different gradient colors, or textures. Part (b) shows the interpolated image during whose generation no distance thresholds are imposed among interpolants (see section 5.3). The image in part (c) is generated by imposing an angle threshold between interpolants, resulting in more ray traced pixels, but a better quality image. The artifacts around the knob of the bowl which are visible in (b) are remedied in (c) by ray tracing correct regions. Thus, we allow the user to adjust the distance thresholds according to the desired accuracy. Table 1 gives the performance results for the images in Figure 15. Our algorithm is twice as fast in terms of the number of FLOPs. Since this is a closed, reflective object, the actual ray tracer does not perform multiple reflections/refractions for a single ray. The performance gain is higher in the case of refractive objects, because the basic ray tracer does more work for each ray, whereas our algorithm performs the same set of interpolation calculations.

Recall that, the interpolation method proposed by Bala, Dorsey and Teller [5] requires that all sixteen interpolants have identical raytrees, whereas our method applies interpolation much more aggressively. To demonstrate the advantage of our method, we have simulated the case when interpolation is not allowed unless the raytrees of all sixteen interpolants are not identical. In Figure 15(d), white pixels show the areas where interpolation cannot be done. For reflective/refractive objects, very few pixels could be interpolated with this method. Obviously, since they use radiance interpolants and adaptively sample with respect to radiance in [5], they would have sampled more densely around radiance discontinuities, therefore the white region would be thinner. But, the white region would still exist around radiance discontinuities, whereas in our algorithm these regions are handled by ray interpolation and not considered as discontinuities.

Figure 16 shows a reflective vase placed in a similar environment. The images are given in increasing detail, part (a) is from far, (b) is a closer look, and in (c), a small section of the vase is zoomed. Table 2 gives the corresponding performance results for each set of images. Figure 17 shows a refractive vase. Table 3 gives the corresponding performance results. Our algorithm is at least three times faster in terms of the number of FLOPs. In part (b), the angular distance threshold is higher than part (c), so, the image in (c) is of better quality providing a trade off in performance. Note that in (c), the additional region ray traced is exactly where the artifacts occur in (b).

Distance between the correct image and the interpolated image: Recall that, the method of Bala, *et al.* [5] provides guarantees on maximum errors in radiance, but ours does not. Since our method is based on estimating output rays, the appropriate quality measure would be the distance between the actual output ray and the interpolated output ray. Figure 14 is presented to visualize the distance between the correct output ray and the interpolated output ray corresponding to each pixel of the image shown in Figure 15. We provide diagrams only for the angular distance. The red pixels correspond to the ray traced areas. The angle between the correct and the interpolated output rays corresponding to the green pixels is greater than $2°$. For the rest of the pixels, the angular distance between the correct and the interpolated output rays varies between $0°$ to $2°$. These are depicted in gray scale: lighter pixels correspond to smaller angular distances. In part (a) the angular distance diagram is given for a viewpoint tree of depth 5, in (b) for a viewpoint tree of depth 6. Both have direction trees of maximum depth 6. As seen in the figures, since (b) is generated from a more densely sampled tree, the interpolated output rays are closer to the correct output rays–the figure in (b) is much lighter than the one in (a). Also, (b) has much less green areas than (a).

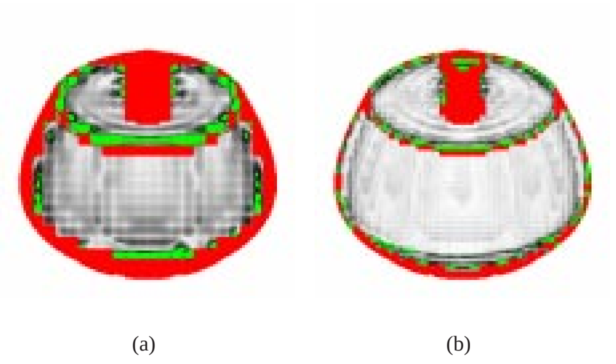

(a) (b)

Figure 14.

Full sampling versus adaptive sampling: We have sampled the direction trees adaptively to save space. For example, the data structure used for the reflective bowl requires 189 MB when sampled adaptively at a maximum depth of 6. When the direction trees are sampled fully to depth 6, the data structure requires 1,370 MB space, which is more than six times as much as the adaptive tree. And, the image generated from both trees do not differ in terms of quality and performance, implying that adaptive sampling is sufficient. Obviously, if we had built deeper viewpoint and/or direction trees, we would have generated better quality images with a higher performance, since the ray traced region would have been much thinner.

All images presented here are generated with the extrapolation option off. As explained in Section 5.1, allowing extrapolation increases the number of cases where the output ray is interpolated rather than ray traced, thus, the gain in performance is much more. However, there would be more artifacts around discontinuities.

7 Conclusion

In this paper, we have explored a ray interpolation method that accelerates ray tracing of complex objects that are reflective or refractive. We have introduced the *RI-Tree* data structure storing adaptively sampled ray interpolants used to interpolate an approximate output ray for any input ray that hits the object, instead of tracing the input ray through the object. The RI-Tree allows the object to be rendered from any viewpoint in any direction. Moreover, the same RI-Tree could be used to render the object in different environments.

According to the preliminary results, our algorithm speeds up ray tracing for relatively complex objects. The performance gain is significant, especially if an input ray goes through multiple levels of reflections/refractions before escaping the object, since our algorithm performs a fixed set of interpolations independent of the number of reflections/refractions. The gain in performance is achieved at the potential expense of quality. However, slight artifacts in reflected/refracted images are often tolerable. Moreover, the user is granted the option to improve/reduce quality by tuning a few parameters. Also, if a deeper RI-Tree is built, both quality and performance would improve trading off space.

Currently, the space required by our data structure is not optimized. However, adaptive sampling reduced the space requirements substantially, even though we exercised adaptivity only at the directional level. We will explore adaptive sampling at the viewpoint level as well. Our interpolation calculations are also open to further optimizations. We do not impose any restrictions on the geometry of the object, but, we do not support objects that are both reflective and refractive. We leave this as a further extention.

References

[1] E. Adelson and J. Bergen. The plenoptic function and the elements of early vision. *Computational Models of Visual Processing*, pages 1–20, 1991.

[2] S. Adelson and L. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3):43–52, May 1995.

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

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

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

[6] J. Blinn and M. Newell. Texture and reflection in computer generated images. *CACM*, 19:542–546, 1976.

[7] E. Camahort, A. Lerios, and D. Fussell. Uniformly sampled light fields. *Rendering Techniques '98 (9th Eurographics Workshop on Rendering)*, pages 117–130, 1998.

[8] A. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

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

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

[11] W. Heidrich, H. Lensch, M. Cohen, and H. Seidel. Light field techniques for reflections and refractions. In *10th Eurographics Rendering Workshop*, June 1999.

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

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

[14] G. Larson. The holodeck: A parallel ray-caching rendering system. In *2nd Eurographics Workshop on Parallel Graphics and Visualisation*, September 1998.

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

[16] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings SIGGRAPH Symposium on Interactive 3D Graphics*, pages 105–106, 1995.

[17] L. McMillan and S. Gortler. Image based rendering: A new interface between computer vision and computer graphics. *Computer Graphics*, 33(4):61–64, November 1999.

[18] E. Ofek and A. Rappoport. Interactive reflections on curved objects. *Computer Graphics (Proceedings of SIGGRAPH 98)*, 14(3):333–342, July 1998.

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

[20] S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.

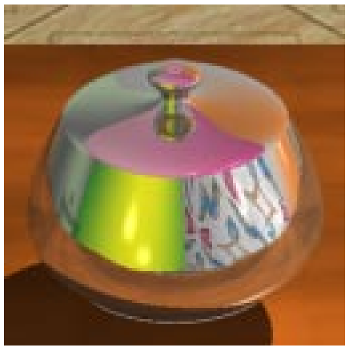
[21] S. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics (Proceedings of SIGGRAPH 80)*, 14(3):110–116, July 1980.

[22] J. S. Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, September 1988.

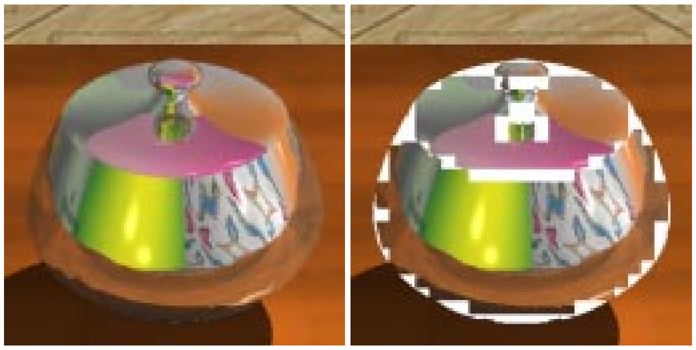
[23] P. Sloan, M. Cohen, and S. Gortler. Time critical lumigraph rendering. In *Proceedings of 1997 Symposium on Interactive 3D Graphics*, pages 17–24, 1997.

[24] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. In *10th Eurographics Workshop on Rendering*, June 1999.

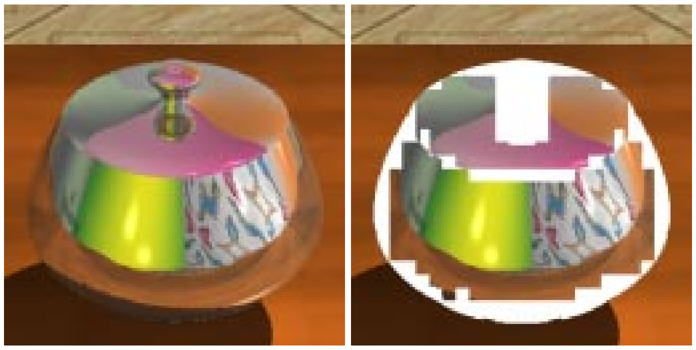
[25] T. Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, June 1980.



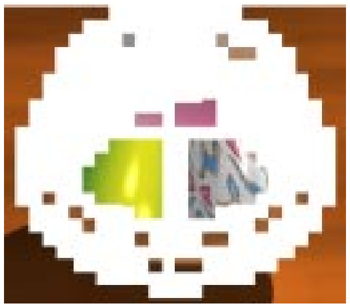
(a)



(b)



(c)

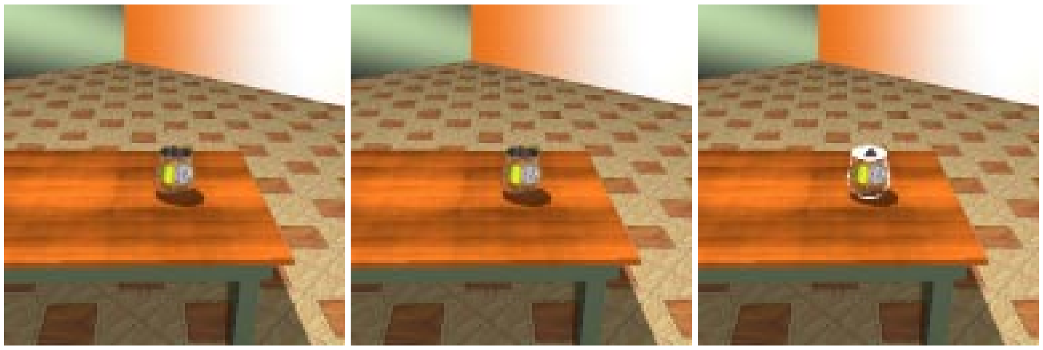


(d)

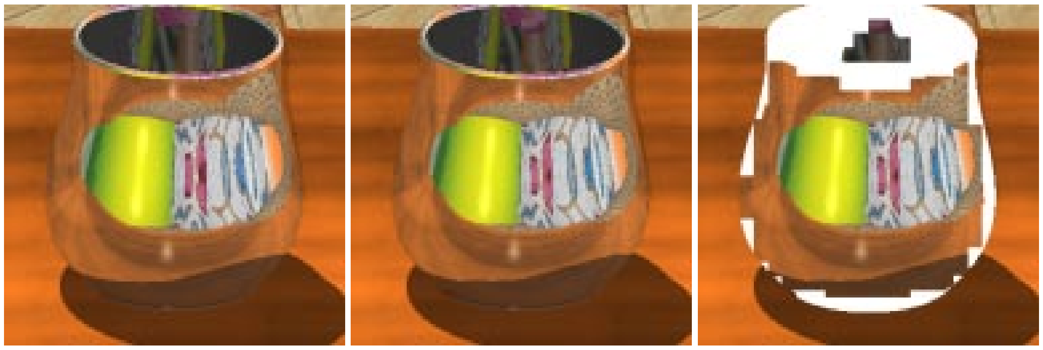
Figure 15. (a) Ray traced image (b) Interpolated image (no angle threshold) & corresponding image with white areas showing ray traced rays (c) Interpolated image (angle threshold imposed) & corresponding image with white areas showing ray traced rays (d) Interpolation by our implementation of the raytree approach [5].

	RAYS TRACED	FLOPS (object-only)	CPU TIME (sec) (object-only)	FLOPS (total)	CPU TIME (sec) (total)
Ray traced	90000	691×10^6	37.557	1115×10^6	73.898
Interpolated (no angle threshold)	6844	298×10^6	19.485	724×10^6	52.206
Interpolated (angle threshold imposed)	12716	405×10^6	26.707	830×10^6	59.452
Raytree Approach [5]	57781	613×10^6	36.211	1036×10^6	71.281

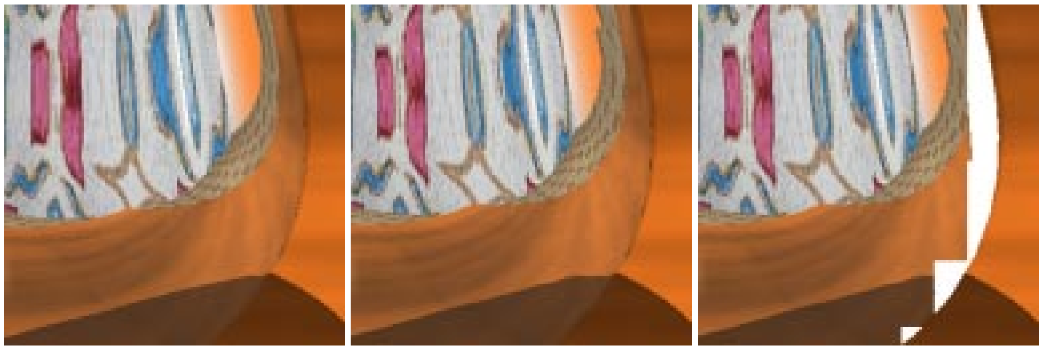
Table 1.



(a)



(b)

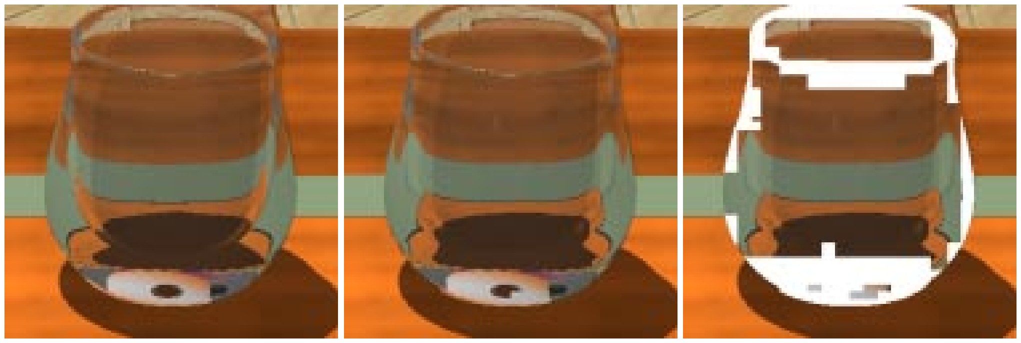


(c)

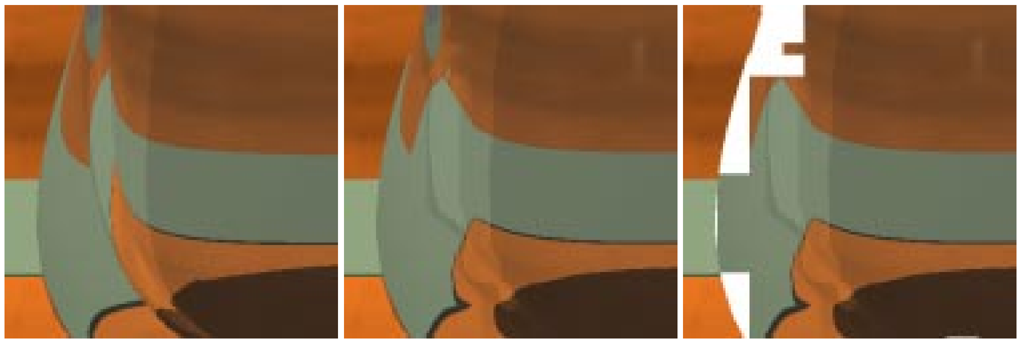
Figure 16. In each row left to right :ray traced image, interpolated image, and the corresponding image where the white pixels depict the ray traced region. (a) from far (b) closer look (c) zoomed section.

	RAYS TRACED	FLOPS (object-only)	CPU TIME (sec) (object-only)	FLOPS (total)	CPU TIME (sec) (total)
Ray traced (far)	90000	63×10^6	5.257	378×10^6	38.053
Interpolated (far)	378	27×10^6	2.007	342×10^6	30.656
Ray traced (close)	90000	1271×10^6	67.760	1851×10^6	111.702
Interpolated (close)	13423	612×10^6	35.290	1197×10^6	75.143
Ray traced (zoom)	90000	1407×10^6	75.956	1966×10^6	121.931
Interpolated (zoom)	5502	246×10^6	21.627	813×10^6	67.518

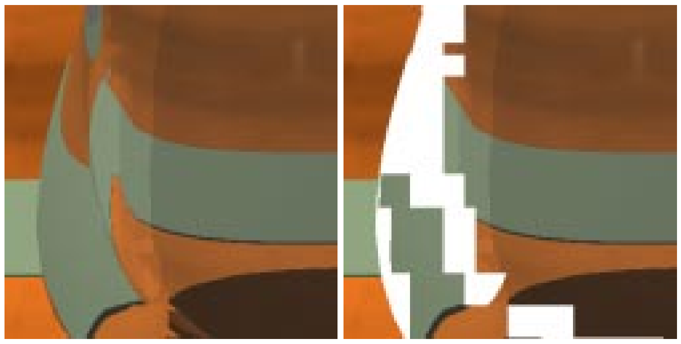
Table 2.



(a)



(b)



(c)

Figure 17. (a) and (b) in each row left to right:ray traced image, interpolated image, and the corresponding image where the white pixels depict the ray traced region. (c) for a smaller angle threshold: interpolated image, and the corresponding image where the white pixels depict the ray traced region for the same zoomed section in (b).

	RAYS TRACED	FLOPS (object-only)	CPU TIME (sec) (object-only)	FLOPS (total)	CPU TIME (sec) (total)
Ray traced (close)	90000	2206×10^6	118.047	3343×10^6	191.994
Interpolated (close)	14237	685×10^6	39.649	1822×10^6	97.309
Ray traced (zoom)	90000	3602×10^6	189.725	4828×10^6	264.571
Interpolated (zoom)	5510	365×10^6	26.712	1576×10^6	100.273
Interpolated (zoom) (smaller angle threshold)	14725	715×10^6	42.717	1926×10^6	112.156

Table 3.