RAY INTERPOLANTS FOR FAST RAY-TRACING REFLECTIONS AND REFRACTIONS[1]

Fatma Betul Atalay **David M. Mount**

Department of Computer Science
University of Maryland, College Park
{betul,mount}@cs.umd.edu

ABSTRACT

To render an object by ray tracing, one or more rays are shot from the viewpoint through every pixel of the image plane. For reflective and refractive objects, especially for multiple levels of reflections and/or refractions, this requires many expensive intersection calculations. This paper presents a new method for accelerating ray-tracing of 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. 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. In most cases, object boundaries and other discontinuities are handled by applying various heuristics. 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.

Keywords: ray tracing, rendering reflections and refractions, interpolation

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 [Whitt80]. However, ray tracing remains a computationally expensive technique. The primary expense in ray tracing lies in intersection calculations, especially for scenes that contain complex objects, such as Bezier or NURBS surfaces, and in case of multiple levels of reflections and/or refractions.

In this paper, we present a method to accelerate ray tracing of reflective and refractive objects by eliminating intersection calculations. Our algorithm facilitates fast, approximate rendering of the object from any viewpoint, and would be most useful when the same object is rendered from multiple viewpoints in a sequence of frames. The key insight to our method is that a ray intersecting a reflective or refractive object goes through a set of reflections and/or refractions, and finally exits the object as an output ray. Therefore, we can model the object as a function f that maps input rays to output rays. 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 nearby samples during rendering.

An important contribution of our work is handling discontinuity regions in which reflection/refraction patterns of nearby rays might differ substantially. We present a set of heuristics to permit interpolation when the parts of an interpolant lie on different sides of a discontinuity. We find a model of the discontinuity and while avoiding interpolation across the discontinuity boundary, we still interpolate on either side. In cases where we cannot find sufficient evidence to interpolate, we perform ray tracing as a last resort.

The rest of the paper is organized as follows. The next section summarizes previous related research. In Sec-

[1]The support of the National Science Foundation under grant CCR-0098151 is gratefully acknowledged.

tion 3, we explain the construction of our data structure. Section 4 outlines the rendering phase and the heuristics used for handling discontinuities. In Section 5, we describe computing local illumination. The experiments are presented in Section 6. Finally, we conclude with Section 7.

2 PREVIOUS WORK

Early research concentrated on accelerating ray tracing by reducing the cost of intersection computations using bounding volume hierarchies [Rubin80], space partitioning structures [Glass84, Kapla85], and methods exploiting ray coherence [Arvo87, Heckb84].

Recent research has focused on fast generation of ray-traced images from multiple viewpoints. These systems exploit frame-to-frame coherence and reuse pixels from the previous frame by reprojection and only recompute or possibly refine the potentially incorrect pixels [Adels95, Walte99].

The Interpolant Ray Tracer system described by Bala, Dorsey and Teller introduced the *radiance interpolant* to accelerate shading by quadrilinearly interpolating radiance samples cached in an adaptive 4D data structure while conservatively bounding the error [Bala99]. We differ in that we are primarily interested in fast rendering of reflective and refractive objects. Our data structure maps rays to rays rather than rays to radiance, and we interpolate among rays. By this method, we decouple local geometry of the object from the environment, and much less sampling of rays is sufficient than sampling of radiance to render reflective/refractive objects. To render reflected textures, the Interpolant Ray Tracer system shoots additional reflection rays, which is expensive, especially for multiple reflections. Their interpolation requires that the ray trees of all 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.

Image-Based Rendering methods constitute another line of research to support fast rendering of scenes. Among them, the most relevant to our work is the Lumigraph [Gortl96] and Light Field Rendering [Levoy96] techniques. Both are based on dense sampling of the *plenoptic function* [Adels91]. These systems have a preprocessing phase where the 4D plenoptic function is sampled by uniformly subdividing in all four dimensions. 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. Schirmacher, *et al.* [Schir99] and Sloan, *et al.* [Sloan97] proposed extensions to the Lumigraph.

There exist approaches other than ray tracing to render fast approximations of reflective/refractive objects. The oldest such method is environment mapping [Blinn76]. It assumes that the environment is sufficiently far away from the reflective object. Another method explained in [Ofek98] is based on mirroring the scene objects with respect to a reflector. It works for curved reflectors relying on high resolution tessellation of both the reflector and the reflected objects and focuses on a single level of reflection. Heidrich, *et al.* proposed a light field method for rendering refractive objects [Heidr99]. Their method is similar to ours in that they interpolate rays rather than radiance. However, since their system is built on a light field structure, it relies on dense sampling of rays for capturing clear object boundaries and handling discontinuities. Thus, their storage requirements are high. Our method, on the other hand, samples rays adaptively and applies various heuristics to achieve high quality discontinuity rendering at lower sampling rates.

3 SAMPLING PHASE

3.1 Representation of Rays as 5D Points

In ray tracing implementations, a common way to represent a ray R is by its origin P and direction vector $\vec{d}$. However, since we need a representation that will allow us to subdivide the direction space easily, we employ the *direction cube* representation as described by [Arvo87] mapping the 3D direction vector of any given ray to 2D coordinates. 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*. 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. 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 5D point.

3.2 The RI-Tree

In this section, we introduce our two-level data structure, the *RI-Tree* which stands for Ray Interpolant Tree. 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.

The first level of the RI-Tree corresponds to the viewpoint space. It consists of six separate quadtrees corresponding to the faces of the bounding box. They recursively decompose the space of viewpoints. We refer to these six quadtrees as the *viewpoint tree*. This level is uniformly subdivided and the four corners of each leaf cell constitute the viewpoint samples as depicted in Fig.1(a).

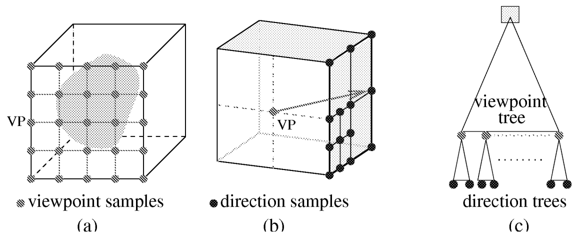


Figure 1: (a)Viewpoint tree (one face) (b)Direction hemicube from *VP* (c)RI-Tree

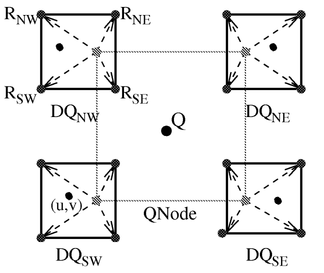


Figure 2: The leaf cells containing Q and (u,v)

The second level corresponds to the directional space. From each viewpoint sample in the viewpoint tree, rays are sampled in various directions. The space of all possible directions from any viewpoint towards the object 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. We impose five quadtrees on the five faces of the hemicube. These five trees are referred to as the *direction tree*. Each viewpoint sample *VP* has a 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 as shown in Fig.1(b). For each direction sample, the corresponding ray—which is referred to as the *input ray*—is traced through the object and the final ray that comes out of the object—which is referred to as the *output ray*— is stored in the leaf cell associated with that direction sample.

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. For this reason, the subdivision at the directional level is carried out adaptively, based on output ray distance. However, we impose an upper limit on its depth to prevent the tree from growing excessively.

4 RENDERING PHASE

4.1 Simple Two-level Interpolation

Our goal is to compute the output ray for any input ray without tracing the input ray through the object. Instead, we try to utilize the coherence of rays, and compute an approximate output ray by interpolation using sampled rays. Since the RI-Tree stores the origins and the direction coordinates of the 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 R onto the bounding box enclosing our object in order to set its origin to a point in our viewpoint space. Assume for the sake of concreteness that, R intersects the box at point Q. Now, our query ray is represented as $R = (Q, \vec{d})$. Next, we locate the leaf cell of the viewpoint tree in which Q lies. Let *QNode* denote this leaf cell. Then, we convert $\vec{d}$ to the 2D direction coordinates, (u, v). For each viewpoint *VP* corresponding to the four corners of *QNode*, we traverse its direction tree and locate the leaf cell in which (u, v) lies. Let DQ_{VP} denote the leaf cell in direction tree of *VP*. As shown in Fig.2, at this point, we have four directional leaf cells associated with the four *VP*s of *QNode*. These four cells provide us sixteen *candidate* rays to be used in interpolation: output rays for the sixteen sampled rays originating from the four viewpoints surrounding the origin of the query ray R, in four directions surrounding the direction of R.

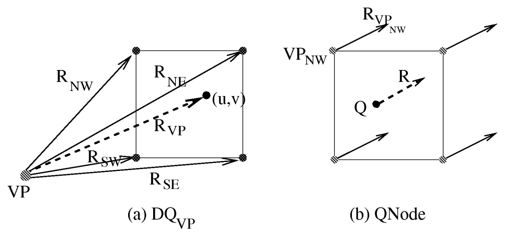


Figure 3: Two level interpolation

The first set of interpolations are done at the direction tree level. For each DQ_{VP}, we compute an approximate output ray for the ray labeled as R_{VP} in Fig.3(a). R_{VP} is the ray originating from *VP* and has the direction coordinates, (u, v), which are the direction coordinates of the query ray R. It serves as an intermediate query ray. The approximate output ray for R_{VP}, denoted $f^*(R_{VP})$, is computed by bilinear interpolation of $f(R_{NW})$, $f(R_{NE})$, $f(R_{SW})$ and $f(R_{SE})$.

After we compute an interpolated output ray, $f^*(R_{VP})$ for each DQ_{VP}, we propagate these intermediate output rays to the viewpoint tree level. As shown in Fig.3(b), R_{VP}s are parallel rays in the direction of our original query ray, R, and originating from the four viewpoints surrounding the origin of R. We compute the output ray for R, $f^*(R)$ by bilinear interpolation of $f^*(R_{VP})$s. Consequently, an approximate output ray for R is computed by the interpolation of sixteen output ray samples.

4.2 Advanced Interpolation for Discontinuities

The simple interpolation method makes no assumptions about the structure of the object, 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 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.

Another solution might be to follow a conservative approach as Bala, *et al.*[Bala99]. If a discontinuity is detected, they do not interpolate but ray-trace. This method reduces the cases one can benefit from interpolation. Our approach, however, is to apply interpolation much more aggressively. We assume that at lower levels of the tree, discontinuities crossing a cell will be of a simple nature and can be treated as a line segment. So, we find a model of the discontinuity and while avoiding interpolation across the discontinuity boundary, we still interpolate on either side.

Patches and Equivalence Classes: In order to explain how the discontinuities are handled, we will describe the structure of our objects. Our algorithm is designed to handle the objects that are specified as a collection of smooth surfaces, referred to as "patches". The patches that share a common edge may or may not be joined with sufficiently high continuity to permit interpolation across the boundary. For example, in Fig.4(a), we can interpolate between patches **A** and **B**, but not between patches **C** and **D**. 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 to be connected continuously. Each patch is assigned a *patch-identifier* and each patch-identifier is associated with a *class-identifier* denoting its equivalence class. Associated with each sampled ray, we store the patch-identifier of the first patch it hits.

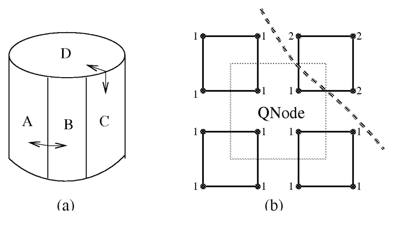


Figure 4: Two-patch case

Two-patch Condition: In the advanced interpolation method, all the sixteen rays might not be used for interpolation. Moreover, the interpolation method might not be applied at all. To determine whether to compute the output ray by interpolation or by tracing the ray, 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. If p_1, p_2,..., p_{16} are grouped in at most two equivalence classes, the *two-patch condition* is satisfied, implying that there is either none or a single discontinuity crossing the region surrounded by the sixteen ray hits. In this case, we can model the discontinuity and interpolate on either side of it. This case is shown in Fig.4(b). In the figure, each corner is labeled with the class-identifier of the patch hit by the ray corresponding to that corner. If *two-patch condition* is not satisfied, we assume that multiple discontinuity boundaries exist in the region, and so we ray-trace rather than interpolate.

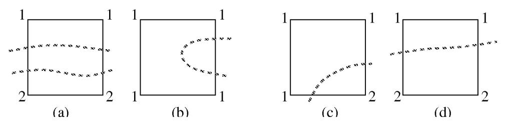


Figure 5: (a)-(b) Bad cases (c)-(d) Good cases

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 Fig.5(a) and (b). For example, in Fig.5(a), just by looking at the four corners, we would decide that the cell is overlapped by two equivalence classes and overlook the third one in between. Similarly for part (b). We do not detect these cases, and assume that they arise very rarely. From this point on, we will assume that if the patches are grouped in two equivalence classes, only the "good" cases depicted in Fig.5(c) and (d) could arise.

Let p_r denote the patch the query ray R hits first. In a two-patch case, among the sixteen candidate rays, we use only the ones that hit a patch in the same equivalence class as p_r. Such a ray is referred to as a *usable ray*. We avoid using the rays hitting the other side of the discontinuity boundary since those would possibly have very different reflection/refraction directions. However, since the query ray R is not actually traced, we do not know p_r. But, we assume that R should hit one of the patches in $PS = \{p_i | 1 \leq i \leq 16\}$, and we compute the exact first intersection of R with the object checking intersections only with the patches in PS. This is not as expensive as a general ray-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. Besides, this computation is required only in the two-patch cases.

Note that, in order to catch discontinuities, we take into account first hits only, however it is possible to have discontinuities within the rest of the raytree. Currently, these discontinuities are handled by not interpolating output rays which are "distant" from each other, as explained in Section 4.4.

Since at least three interpolants are required to perform an interpolation, some cells cannot be used for interpolation due to unusable candidate rays. The algorithm given in Fig.6 summarizes the advanced interpolation method.

```
determine the patch hit by the query ray R;
NumberOfUsableDQs = 0;
for each of the four DQs do
        if (number of usable rays ≥ 3) then
                NumberOfUsableDQs++;
                compute intermediate output ray
                        by interpolating usable rays;
if ( NumberOfUsableDQs ≥ 3) then
        compute $f^*(R)$ using the intermediate
                output rays from successful DQs;
        return $f^*(R)$;
else
        return failure; /* trace the ray */
```

Figure 6: Advanced interpolation algorithm

For the three-interpolant cases, if the point for which we attempt to estimate the output ray lies outside the triangular region formed by the points corresponding to the usable candidate rays, this is not an interpolation anymore—it becomes an extrapolation. Since extrapolated values are less reliable, the user is granted the option of tuning the extrapolation. If the point to be extrapolated is farther from the triangular region—in terms of its barycentric coordinates—than a given threshold, extrapolation is disabled and the cell cannot be used for interpolation.

4.3 Extra Sampling

We can reduce the number of rays ray-traced if we can use the *DQ*s with less than three interpolants instead of eliminating them. These cells are made usable by finding a model of the discontinuity and sampling extra rays at one or both sides of the discontinuity to satisfy the three-interpolant requirement.

During preprocessing, if a leaf cell will not be split any further, but the rays corresponding to the four corners hit patches that are in two different equivalence classes, two "good" 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 case is when two of them are in one equivalence class while the other two are in another class.

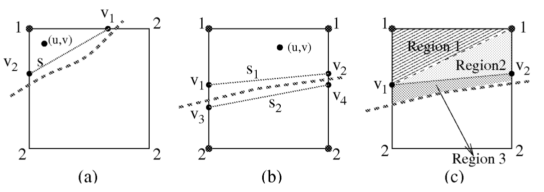


Figure 7: One- and two-interpolant cases

We call the first case the one-interpolant case. Consider the example in Fig.7(a). The corners are labeled with the class-identifier of the patch hit by the ray corresponding to that corner. Let $c(p)$ denote the equivalence class of patch p. In case this node is needed for rendering, and if $c(p_r) = 1$, then two more sampled rays are required on the same side of the discontinuity as the NW corner to perform the interpolation. The dashed curve in the figure depicts the actual discontinuity, and the line segment labeled as s is the approximation we compute to model the discontinuity. In order to define s, we find the two points labeled as v_1 and v_2. By binary search on the upper edge of the cell, we locate the point farthest from the NW corner and the corresponding ray of which hits a patch p, where $c(p) = 1$. This is point v_1. We compute v_2 similarly. Then, we sample the rays corresponding to v_1 and v_2.

The second case is called the two-interpolant case. An example is shown in Fig.7(b). 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 s_1 on one side of the curve and another line segment s_2 on the other side. To define s_1 and s_2, we locate points v_1, v_2, v_3 and v_4 by binary search similar to the one-interpolant case. Then, we sample extra rays corresponding to these points. To illustrate the interpolation in a two-interpolant case, consider Fig.7(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, rays corresponding to NW and NE corners and v_1 are used to interpolate the output ray. If (u, v) is in Region 2, NE corner, v_1 and v_2 are used. And, if (u, v) is in Region 3, then we have the option of using extrapolation using barycentric coordinates computed with respect to the NE corner, v_1 and v_2.

4.4 Handling High Curvature

Normally, we assume that the usable rays associated with the same cell 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 distant from each other. This case arises when the input rays hit at 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. As a measure to determine the distance between two output rays, we use the angular distance between the direction vectors of the rays. If the distance is greater than a given threshold, those rays are not usable as interpolants.

5 LOCAL ILLUMINATION

The final color is determined by blending the local illumination with the reflected/refracted color. We compute the reflected/refracted color by shooting the output ray through the rest of the environment. Since we need intersection points and normals to compute local illumination, we store the first intersection point and the normal along with each ray sample. These are then used to interpolate the intersection point and the normal for the query ray applying the same interpolation method used for interpolating output rays. Only the intersection points and normals that are associated with usable rays are used as interpolants.

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 ray tracer which is utilized when rays are sampled during preprocessing, and when interpolation cannot be done during rendering. Our models are constructed from bicubic Bezier patches.

A *ray-traced* image is generated by tracing each input ray through the object to compute the output ray. In our example images, an "interesting" object is placed in a relatively simple environment. An *interpolated* image is generated by computing the output ray by our interpolation method. 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 tables given below. We also provide the total number of FLOPs and CPU time for the entire image. Since the output ray is traced through the environment, the total FLOPs and CPU time are not improved as much as those for the object-only ones.If we had used the output ray to index an environment map instead, the improvement ratio for the entire image would be close to the object-only ratios.

For our example images, the viewpoint tree has a 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. For each image, we provide the ray-traced image, the image generated by our interpolation method, 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.

Fig.9 shows a reflective bowl on a procedurally textured table placed in a room. The walls are shaded in different gradient colors, or textures. Part (a) shows the interpolated image during whose generation no distance thresholds are imposed among interpolants (see section 4.4). The image in part (b) is generated by imposing an angle threshold, resulting in more ray-traced pixels, but a better quality image. The artifacts around the knob of the bowl which are visible in (a) are remedied in (b) by ray-tracing correct regions. Thus, the user can adjust the distance thresholds according to the desired accuracy. Table 1 gives the performance results for the images in Fig.9. 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 ray tracer does more work for each ray, whereas our algorithm performs the same set of interpolations. Fig.10 shows a refractive vase placed in a similar environment. Table 2 gives the corresponding performance results. Our algorithm is at least three times faster in terms of number of FLOPs. If the angular threshold is lower, the artifacts around the curved interior of the vase are remedied trading off performance.

For the reflective bowl, the preprocessing takes 1 hour, and the size of the data structure (for a single face of the bounding box) is 189 MB. For the refractive vase, it takes 2.5 hours to build a data structure of 191MB. In our current implementation, the data structure has not been optimized for space or preprocessing time.

Recall that, the interpolation method proposed by Bala, *et al.*[Bala99] 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 Fig.9(e), 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 sample adaptively with respect to radiance in [Bala99], 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.

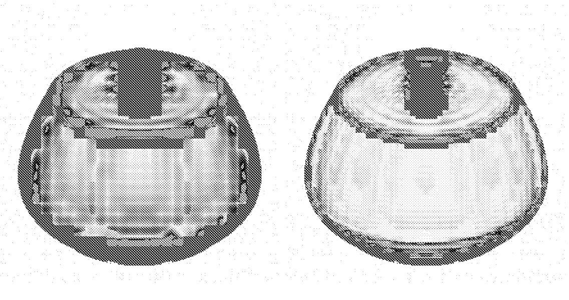


Figure 8: Visualization of output ray distance

Distance between the ray-traced and the interpolated images: The method of Bala, *et al.* [Bala99] 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. Fig.8 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 Fig.9. 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. The angular distance diagram on the left is for a viewpoint tree of depth 5, the right one is for a viewpoint tree of depth 6. As expected, since the right diagram is generated from a more densely sampled tree, the interpolated output rays are closer to the correct output rays. If we had built deeper trees, we would have generated better quality images with a higher performance.

The example images given are generated with the extrapolation off. Enabling extrapolation increases the number of cases we can interpolate. Thus, the performance gain is higher, but at the expense of quality.

7 CONCLUSION

In this paper, we described a ray interpolation method that accelerates ray tracing of reflective or refractive objects. We have introduced the *RI-Tree* data structure storing adaptively-sampled ray interpolants that are 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 under changing illumination and/or geometry of the environment.

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. Our interpolations are open to further optimizations such as applying incremental calculations taking advantage of spatial coherence. The performance gain is achieved at the potential expense of quality. However, slight artifacts in reflected/refracted images are often tolerable. Besides, our system detects and deals with the object boundaries and other strong discontinuities where the artifacts are more likely to be noticed. Moreover, the user is granted the option to improve/reduce quality by tuning a few parameters.

Since the entire data structure is built to facilitate rendering from any arbitrary viewpoint, the preprocessing times are high. For the current version, we assume that the preprocessing cost will be amortized over many renderings in an animation. However, for the future, we plan to address reducing both the preprocessing time and space requirement, by filling the data structure on demand—we will generate samples only when needed as interpolants. We want to incorporate a caching mechanism in order to use previously generated samples for the new viewing position. We also plan to extend this method by supporting objects that are both reflective and refractive.

REFERENCES

[Adels91] E. Adelson and J. Bergen. The plenoptic function and the elements of early vision. *Computational Models of Visual Processing*, pages 1–20, 1991.

[Adels95] S. J. Adelson and L. F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Comp. Graph. and Appl.*, 15(3):43–52, May 1995.

[Arvo87] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *Proc. of SIGGRAPH 87*, 21(4):196–205, 1987.

[Bala99] K. Bala, J. Dorsey, and S. Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. on Graph.*, 18(3), August 1999.

[Blinn76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Commun. of ACM*, 19:542–546, 1976.

[Glass84] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Comp. Graph. and Appl.*, 4(10):15–22, October 1984.

[Gortl96] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Proc. of SIGGRAPH 96*, pages 43–54, August 1996.

[Heckb84] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Proc. of SIGGRAPH 84*, 18(3):119–127, July 1984.

[Heidr99] W. Heidrich, H. Lensch, M. Cohen, and H. Seidel. Light field techniques for reflections and refractions. In *10th Eurographics Rendering Workshop*, June 1999.

[Kapla85] M. R. Kaplan. Space tracing a constant time ray tracer. *State of the Art in Image Synthesis (SIGGRAPH 85 Course Notes)*, 11, July 1985.

[Levoy96] M. Levoy and P. Hanrahan. Light field rendering. *Proc. of SIGGRAPH 96*, pages 31–42, 1996.

[Ofek98] E. Ofek and A. Rappoport. Interactive reflections on curved objects. *Proc. of SIGGRAPH 98*, 14(3):333–342, July 1998.

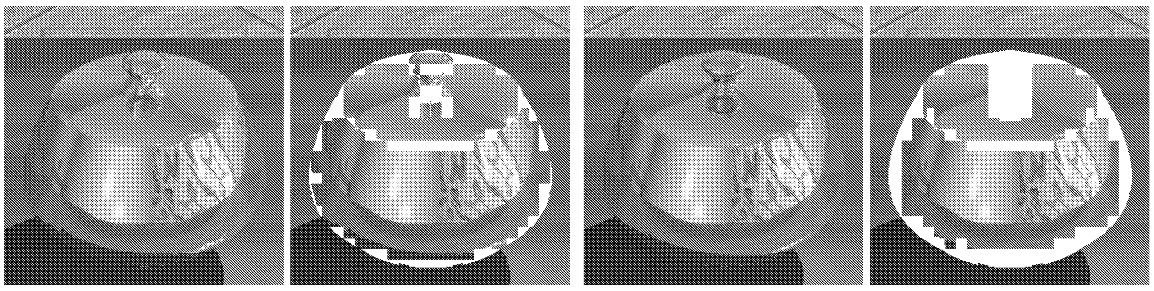
[Rubin80] S. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Proc. of SIGGRAPH 80*, 14(3):110–116, July 1980.

[Schir99] H. Schirmacher, W. Heidrich, and H. P. Seidel. Adaptive acquisition of lumigraphs from synthetic scenes. *Computer Graphics Forum (Eurographics '99)*, 18(3):151–160, September 1999.

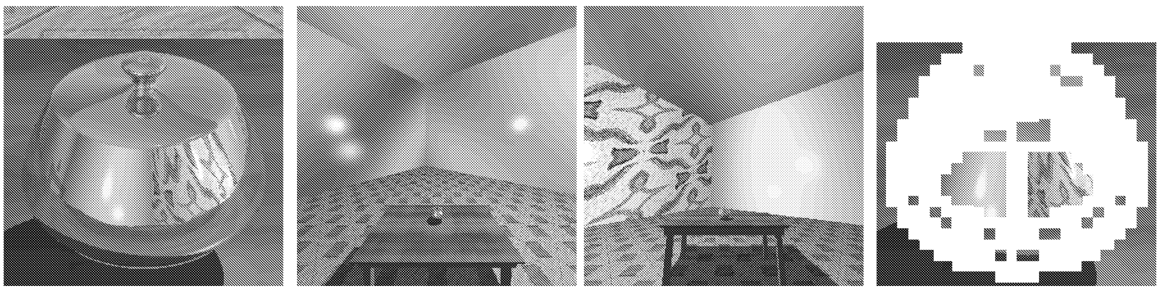
[Sloan97] P. P. Sloan, M. F. Cohen, and S. J. Gortler. Time critical lumigraph rendering. In *Proc. of 1997 Symp. on Interactive 3D Graphics*, pages 17–24, 1997.

[Walte99] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. In *10th Eurographics Workshop on Rendering*, June 1999.

[Whitt80] T. Whitted. An improved illumination model for shaded display. *Commun. of ACM*, 23(6):343–349, June 1980.



(a) (b)

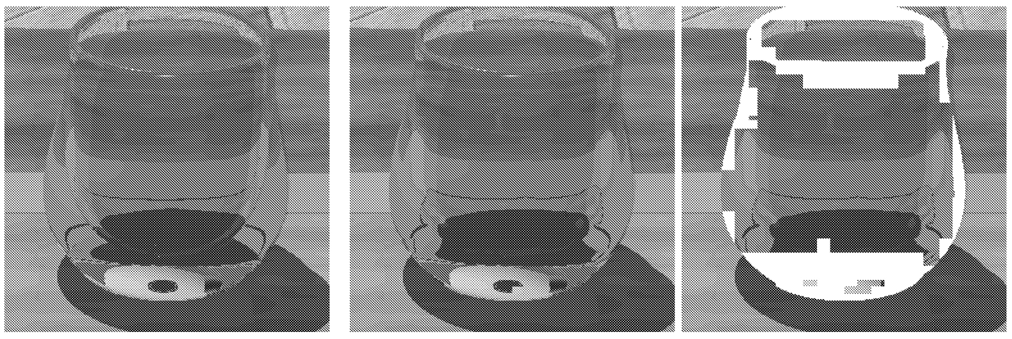


(c) (d) (e)

Figure 9: (a) Interpolated image (no angle threshold) & corresponding color-coded image, white areas show ray-traced rays (b) Interpolated image (angle threshold imposed) & corresponding color-coded image (c) Ray-traced image (d) The room as viewed when looking into the front of the bowl (same direction as in (a) (b) and (c)) and from behind the bowl (e) Interpolation by our implementation of the raytree approach [Bala99].

	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 [Bala99]	57781	613×10^6	36.211	1036×10^6	71.281

Table 1: Performance results for the reflective bowl



(a) (b)

Figure 10: (a) Ray-traced image (b) Interpolated image & corresponding color-coded image.

	RAYS TRACED	FLOPS (object-only)	CPU TIME (sec) (object-only)	FLOPS (total)	CPU TIME (sec) (total)
Ray-traced	90000	2206×10^6	118.047	3343×10^6	191.994
Interpolated	14237	685×10^6	39.649	1822×10^6	97.309

Table 2: Performance results for the refractive vase