Appears in *The Tenth Eurographics Workshop on Rendering*, Spain, June 1999.

Interactive Ray-Traced Scene Editing Using Ray Segment Trees

Kavita Bala Julie Dorsey Seth Teller
Laboratory for Computer Science
Massachusetts Institute of Technology
{kaybee,dorsey,seth}@graphics.lcs.mit.edu

Abstract. This paper presents a ray tracer that facilitates near-interactive scene editing with incremental rendering; the user can edit the scene both by manipulating objects and by changing the viewpoint. Our system uses object-space radiance interpolants to accelerate ray tracing by approximating radiance, while bounding error. We introduce a new hierarchical data structure, the *ray segment tree* (RST), which tracks the dependencies of radiance interpolants on regions of world space. When the scene is edited, affected interpolants are rapidly identified— typically in 0.1 seconds—by traversing these ray segment trees. The affected interpolants are updated and used to re-render the scene with a 3 to 4× speedup over the base ray tracer, even when the viewpoint is changed. Although the system does no pre-processing, performance is better than for the base ray tracer even on the first rendered frame.

Keywords: Ray tracing, radiance interpolation, scene manipulation, incremental rendering

1 Introduction

Ray tracing is a popular technique for producing high-quality imagery. Ray tracers typically support specular and diffuse reflectance functions, and general geometric primitives, producing high quality view-dependent images. However, this quality is achieved by compromising interactivity; ray tracing is not commonly used in interactive applications such as editing and viewing because of the high cost of computing each frame.

In recent years, strides have been made in facilitating interactive scene manipulation with ray tracing. Several researchers have developed ray tracers supporting scene editing that incrementally render parts of the scene that might be affected by a change. Cook's shade trees [4] maintain a symbolic evaluation of the local illumination at each pixel of a frame. When an object's material properties are changed, if the shade trees remain the same, they are re-evaluated with the new material properties. Séquin and Smyrl [9] extend these shade trees to include reflections and refractions. Their trees represent the entire radiance contribution by the scene at each pixel. In their system, changes to an object's material properties (e.g., color, specular coefficient) are the only user-specified edits permitted. Murakami and Hirota [8] and Jevans [7] extend these techniques to support geometry changes (e.g., object is moved) by associating rays with the voxels they traverse. A scene change affects voxels and their associated rays.

More recently, Brière and Poulin [3] introduced a system that maintains *color trees* and *ray trees* to separately accelerate updates to object attributes and geometry. Attribute changes involve adjustments to an object's color, reflection coefficient etc., while geometry changes include changes such as moving an object. Their system reflects attribute changes in about 1-2 seconds, and geometry changes in 10-110 seconds.

All of the above techniques are *pixel-based*; that is, additional information is maintained for each pixel and used to recompute radiance as the user edits the scene. The chief drawback of these systems is that they are completely view-dependent; while a user can edit the scene, he cannot adjust the viewpoint. Also, for high resolution images, the memory requirements can be large.

Another related area of research is the use of line or ray space to accelerate editing and rendering in global illumination algorithms. Arvo and Kirk [1] represent bundles of rays as 5D bounding volumes that are used to accelerate ray-object intersections, but do not accelerate shading or editing. In the context of editing for radiosity applications, Dretakkis and Sillion [5] augment the link structure of hierarchical radiosity with additional line-space information to track links affected by the addition or deletion of objects. The hierarchical link structure, and hence the implicit line space, makes it possible to identify affected regions rapidly when an object is edited. Their system is not pixel-based; therefore, a user can change the viewpoint after an update. However, their algorithms apply only to radiosity systems for scenes with diffuse materials.

In previous work [2, 11] we presented a system to accelerate ray tracing using per-object 4D *radiance interpolants* used to approximate radiance, while guaranteeing bounds on error. A radiance interpolant records view-dependent radiance for a set of rays that intersect an object. The system uses an error predicate to guarantee that the interpolant approximates radiance for every ray covered by that set of rays to within a user-specified error bound. For each pixel, the system finds the interpolant that covers that eye ray, and if it exists, uses it to interpolate radiance. Interpolants are built lazily and adaptively as needed and stored in 4D trees called *linetrees*. When the viewpoint changes, some interpolants from the previous frame are reused. Thus, our system accelerates ray tracing while allowing the viewpoint to move; however, objects in the scene cannot be modified.

In this paper, we present a system that supports interactive scene editing *while permitting changes in the viewpoint*. Our work draws on the work of Brière and Poulin, Dretakkis and Sillion, and our previous work on radiance interpolants, while providing additional functionality. Our system builds radiance interpolants to accelerate rendering by approximating radiance. When an object is updated, only a subset of global line space is affected. We introduce space-efficient hierarchical *5D ray segment trees* to track the regions of ray space that affect an interpolant. When the scene is edited, trees are traversed to rapidly identify and invalidate the interpolants that are affected by the edit. When a new frame is rendered from the same or a different viewpoint, interpolants that are still valid are reused to accelerate rendering.

First, we review 4D per-object interpolants in Section 2. In Section 3, we show how to augment interpolants to support interactive scene editing, and describe how these augmented interpolants can be used with global linetrees to track the regions of line space affected by an interpolant. In Sections 4 and 5, we address the limitations of the 4D global linetrees by using 5D ray segment trees, and explain how these trees can be used to find all interpolants that might be affected by a scene edit. Finally, we present results in Section 6, and conclude with a discussion of future work in Section 7.

2 Radiance Interpolants

In this section, we review how 4D radiance interpolants are used to accelerate ray tracing by exploiting spatial coherence, both object and screen-space, and also temporal coherence. The key idea of this system is to accelerate ray tracing by approximating radiance while guaranteeing error bounds. Every ray intersecting an object has an associated radiance. Assuming that a ray intersects an object, and the object is surrounded by a transparent medium, the ray can be parameterized using four coordinates. The space of all such rays is the four-dimensional space of directed lines, called *line space*. A radiance interpolant represents the radiance of an object over a region of line space; the interpolant is said to *cover* that region of line space. The region of line space covered by an interpolant is a four-dimensional hypercube, and every ray covered by the interpolant lies inside the hypercube. The interpolant records a radiance sample for each of the sixteen vertices of the hypercube. The radiance associated with any ray covered by the interpolant is then approximated by quadrilinearly interpolating these radiance samples. When radiance is coherent, radiance interpolants are an efficient way to compute and represent the radiance of a scene.

Radiance interpolants have two important properties for interactive scene editing:

- Interpolants do not depend on the viewpoint; therefore, the viewpoint can be changed freely without invalidating them. As long as rays from the current viewpoint are covered by an interpolant, the interpolant can be reused.
- An update to the radiance samples stored in an interpolant effectively updates radiance for *all* rays covered by that interpolant. Thus, interpolants are an efficient way to structure the update of radiance information when a scene is edited.

2.1 Ray parameterization

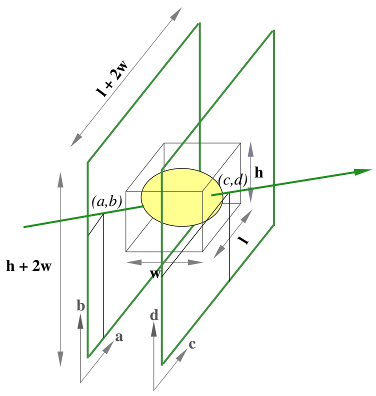


Fig. 1. Ray parameterization in 3D. The ray is parameterized by (a, b, c, d), its intercepts with the front and back faces of the expanded face pair.

Every ray intersecting an object is parameterized by four coordinates (a, b, c, d), which are the intercepts that it makes with two parallel faces surrounding that object (see Figure 1). To completely cover the space of rays that intersect the object, six pairs of parallel faces are considered. Each face pair is defined by two parallel faces and a principal direction that is perpendicular to the faces. The principal directions of the six face pairs are $\pm\hat{x}$, $\pm\hat{y}$ and $\pm\hat{z}$. Every ray intersecting o is uniquely associated with the face pair whose principal direction is closest to the ray's direction; that is, the principal direction onto which the ray has the maximum positive projection. To ensure that every ray associated with a face pair intersects both parallel faces, the faces are sized as shown in Figure 1. Once the face pair associated with a ray is identified, the ray is intersected with its front and back faces to compute its (a, b) and (c, d) coordinates respectively.

2.2 4D Radiance Interpolants

When rendering a pixel, the system must be able to find an interpolant (if any) that covers the eye ray corresponding to that pixel. To accelerate interpolant lookup, interpolants are stored in *linetrees* that are the 4D analogues of octrees. There is a linetree associated with each face pair of an object. For every eye ray, once the object it intersects is known, an interpolant is found by walking recursively down from the root of the linetree of the appropriate face pair. The radiance for the eye ray is then quadrilinearly interpolated using the radiance samples in the interpolant.

Interpolation error arises from discontinuities and non-linearities in the radiance function. An *error predicate* is tested to automatically detect both these conditions. An interpolant is not constructed if the error predicate indicates conservatively that its interpolation error would exceed a user-specified bound. The error predicate uses information about ray trees [3] to identify the regions of line space that have smoothly varying radiance that is approximated well by quadrilinear interpolation. The error predicate ensures that linetrees are subdivided adaptively; interpolants cover large regions of line space where radiance varies smoothly, and conversely, where radiance changes rapidly, interpolants cover small regions of line space. Thus, adaptive subdivision of linetrees prevents erroneous interpolation while allowing reuse when possible. Currently, the error predicate only supports convex objects, as discussed in Section 7.

Visibility determination at pixels is another important function of the ray tracer and is needed in order to find the correct interpolant for an eye ray. We use a conservative algorithm for *reprojection* of linetree cells to accelerate visibility determination. This algorithm exploits the temporal frame-to-frame coherence in the user's viewpoint, while guaranteeing that the correct visible surface is detected for each pixel. A fast scan-line algorithm accelerates rendering using the reprojected linetrees. See [2] for details.

This algorithm has the important property that it is entirely on-line; no pre-processing is necessary to construct radiance interpolants. Radiance interpolants are generated lazily and adaptively as the scene is rendered from various viewpoints. This on-line property is useful for interactive applications.

3 Interpolants and Scene Editing

In this section, we describe how to identify interpolants that are affected by an object edit. First, we present a *global* four-dimensional parameterization of rays and show that the region of line space affected by an object edit is a subset of global line space. We then describe how a hierarchical global linetree can be used to rapidly identify and invalidate the interpolants affected by an object edit.

3.1 Global Line Space Parameterization

Global line space is the space of all directed lines that intersect the scene. In the previous section, we presented a four dimensional parameterization of rays intersecting an object. We use a similar parameterization for rays intersecting the scene, except that the face pairs (as shown in Figure 1) surround the scene, and $w \times l \times h$ is the size of the bounding box of the entire scene. As explained in Section 2, every ray intersecting the scene is associated with a face pair, and the ray is parameterized by the four intercepts it makes with the two parallel faces of the face pair. Note that per-object line space coordinates can be easily transformed into global line space coordinates.

For simplicity, we explain ideas in 2D in this paper; the extension of these ideas to

3D is straightforward. Each 2D ray is represented by two intercepts (a, c) that it makes with a pair of parallel 2D line segments; this representation is a 2D analogue of the ray parameterization in Section 2.1. For example, in Figure 2-(a), the horizontal lines surrounding the circle represent a 2D face pair for *global line space*. Four such face pairs are needed to represent all the rays that intersect the scene.

3.2 Line Space affected by an Object Edit

A basic intuition is that interpolants can be updated efficiently to reflect an object edit because an object edit affects a contiguous subset of line space, as shown in Figures 2-(a) and (b). On the left in Figure 2-(a), a circle C in world space, a ray that intersects C, and its associated face pair are shown. On the right is a Cartesian representation of 2D line space. Every directed line in world space is a point in line space. The set of rays that intersects the circle in world space corresponds to the interior of a hyperbola in line space as shown on the right in Figure 2-(a) (See Appendix A for details). When the circle is edited, the radiance of every ray in the shaded region of line space could be affected. Therefore, every interpolant that includes a ray in the shaded region should be updated. If all the rays covered by an interpolant lie outside this hyperbola the interpolant is not affected and can be reused. In the next section, we explain how to find the rays covered by an interpolant.

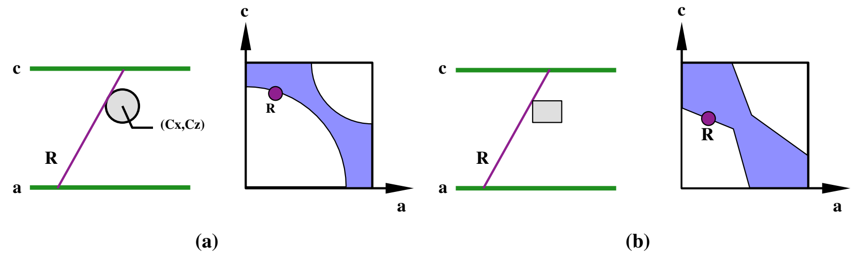


Fig. 2. When the circle or rectangle are edited, the shaded region of line space (on the right of the corresponding figures) is affected.

If instead of a circle, a rectangle is edited (shown in Figure 2-(b)), an hourglass-shaped region (similar to a hyperbola) of line space is affected by the edit. The important point is that the region of line space affected by an edit is a well-defined subset of line space that can be identified efficiently using a hierarchical tree to represent line space. This is true for object edits in 3D world space (4D line space) as well.

3.3 Interpolant Dependencies

Interpolants are an efficient way to structure the update of radiance when a scene is changed, since an update to the radiance samples of the interpolant effectively updates the radiance for all rays covered by the interpolant. In this section, we explain how interpolants are affected by scene editing.

First we review the concept of ray trees, which are important for understanding interpolant dependencies. In a Whitted ray tracer [13], when the radiance for a ray is computed, an associated ray tree can be built that records all the sources of radiance that contributed to the total radiance of the ray [2, 3, 9]. Each internal node in the ray tree corresponds to an intersection of a ray with a surface, and the leaf nodes are lights. An arc of a ray tree represents a *ray segment* from a surface to either another surface or a light. A ray tree node has position-independent and position-dependent information [3]. The position-independent information includes the object intersected by the ray, a list of every light that contributes to the radiance at that point, and a list of every occluder that blocks some light. The position-dependent component includes the point of intersection of the ray, the normal at that point, texture coordinates (if the object is textured), and pointers to the reflected and refracted ray trees (if they exist). These reflected and refracted ray trees are computed recursively. To conserve memory, the position-independent information in ray trees is shared when possible [2, 3].

In our earlier work [2], the error predicate used the sixteen radiance samples and their associated ray trees to determine if the interpolant approximates radiance to within a user-specified error bound over the region of line space that is *covered* by the interpolant. To detect radiance discontinuities, the error predicate requires that the position-independent components of the sixteen extremal ray trees be the same. Since the object associated with the interpolant is convex, every ray covered by the interpolant is bounded by the extremal rays, even after one or more reflections. An additional shaft-cull [6] then guarantees that *every* ray covered by the interpolant also shares the same position-independent ray tree component.

The sixteen extremal ray trees differ only in their position-dependent information. Consider one set of sixteen corresponding arcs from the extremal ray trees; each arc is a ray segment. The corresponding ray segment of every interior ray lies in the 3D volume bounded by these sixteen ray segments. Therefore, when a scene edit affects that 3D volume of space, the interpolant should be invalidated to guarantee correctness. We represent this 3D volume conservatively as a shaft [6]. The set of all shafts represented by an interpolant's ray trees is similar to the *tunnels* used by Brière and Poulin [3], although in that work ray dependencies are captured only for a fixed viewpoint. In Plate A, some tunnels associated with interpolants for the three-sphere scene are shown.

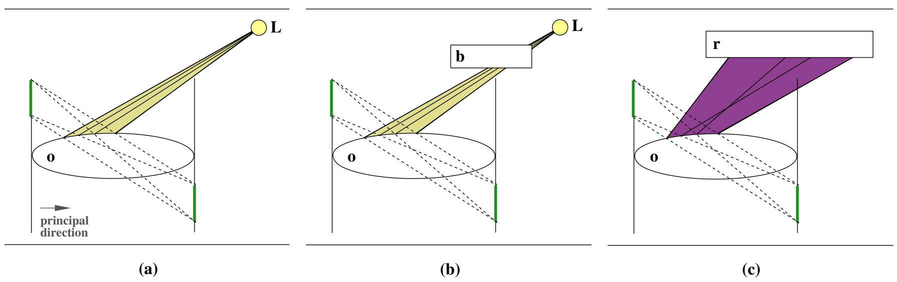


Fig. 3. Rays that affect an interpolant: (a) Light rays, (b) Occluder rays, (c) Reflected rays.

Now we consider the different types of ray-tree arcs and the 3D volumes they cover. Figure 3 depicts three such arcs, corresponding to unoccluded light rays, occluded light rays, and reflected rays. In the figure, the ellipse o is the object for which an interpolant I is built. The interpolant is associated with the face pair shown as two vertical line segments surrounding o. The dotted lines show the four extremal rays (in 2D) that are used to build I. The two horizontal lines at the top and bottom of the scene show one of the face pairs of global line space. The volume that affects each arc (and therefore affects the interpolant) is shaded in each figure.

In Figure 3-(a), the four extremal rays intersect o, and their radiance is evaluated by shooting rays to the light L which is visible to every ray covered by *I*. Therefore, *I* depends on the shaded region shown in the figure. In Figure 3-(b), the light rays for the interpolant are all occluded by the same occluder b. If b is opaque, *I* depends only on the occluder b. If b is transparent, *I* depends on the shaded region shown in the figure. In Figure 3-(c), the volume of space that affects the arcs corresponding to reflections in the interpolant is shaded. Thus, the regions of world space that affect an interpolant can be determined using the ray trees associated with the extremal rays of the interpolant. An interpolant can become invalid only if the scene edit affects one of these regions.

3.4 Finding Affected Interpolants using Global Linetrees

Given a scene edit, we want to efficiently identify the interpolants that could be affected by the edit. We now discuss how global line space can be used to track the regions of 3D space that affect an interpolant. This is similar to the approach taken by Drettakis and Sillion [5] for radiosity systems.

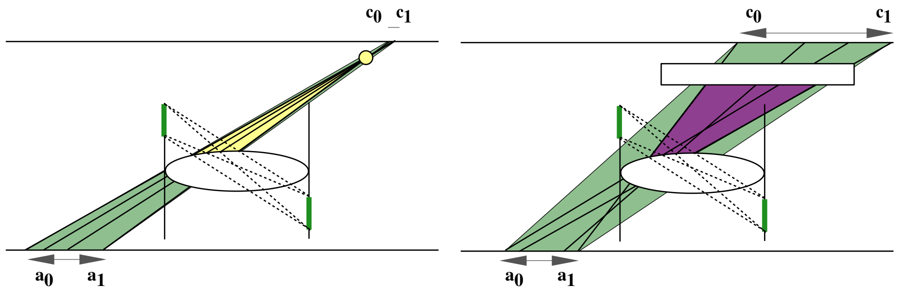


Fig. 4. Global line space for: (a) Light rays, (b) Reflected rays.

When the scene is edited, the edit affects a 3D volume. An interpolant depends on that 3D volume if any tunnel associated with the interpolant intersects that volume. We would like to rapidly identify all such tunnels. Each of the tunnel sections is contained in some region of global line space. This region can be characterized conservatively by extending the sixteen extremal rays that define the tunnel section until they intersect the appropriate global face pair. For example, Figures 4-(a) and (b) show this computation in 2D. In Figure 4-(a), the four extremal rays from the object o to the light L are extended to intersect a global face pair (shown as horizontal lines surrounding the scene). The a and c ranges of these intersections are computed. The corresponding rectangular region in line space, $[a_0, a_1] \times [c_0, c_1]$, is a conservative characterization of the volume that affects the interpolant. In the figure, this region of line space is shown in medium gray. Similarly, in Figure 4-(b), the extremal reflected rays are intersected with the global face pair and the medium gray region shows the corresponding region of line space. This characterization is conservative because it covers a larger 3D volume than its tunnel section. In the next section this characterization is made more precise.

In 4D, each tunnel section of an interpolant is conservatively represented by 8 coordinates (a_0, b_0, c_0, d_0) — (a_1, b_1, c_1, d_1) that define a 4D bounding box in line space. The tunnel sections affected by an object edit can be rapidly identified in the following manner: a linetree is constructed for each face pair of global line space. Each node of the linetree corresponds to a 4D bounding box in line space. A leaf node in the linetree contains pointers to every interpolant that *depends* on the region of line space represented by the node. In other words, for every interpolant included in the linetree node, the 4D bounding box of the linetree node intersects the 4D bounding box that conservatively represents at least one of the interpolant's tunnel sections. This hierarchical linetree can be used to rapidly identify the interpolants affected by an object edit.

4 Interpolants and Ray Segments

The previous section described a data structure that conservatively tracks the regions of line space that affect an interpolant. However, this representation is too conservative. In this section, we introduce a 5D parameterization of rays to address this limitation, and describe *ray segment trees* that improve on the linetrees of the previous section.

4.1 Limitations of Line Space

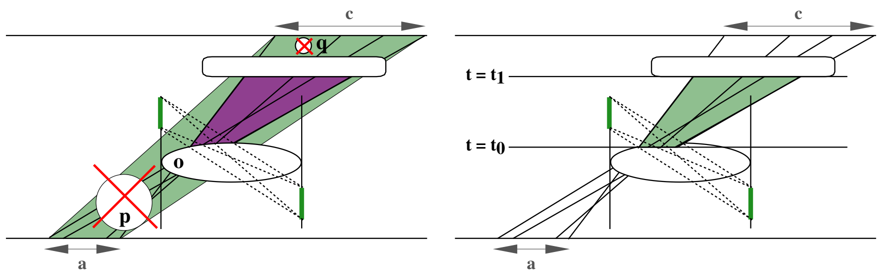


Fig. 5. (a) Line space vs. Ray space, (b) using the extra distance dimension t.

The main disadvantage of the 4D representation of lines is that it is too conservative. This problem is illustrated in Figure 5-(a), which shows an interpolant for o. The tunnel section corresponding to reflected rays from o is shown in dark gray, while the corresponding conservative line space representation is shown in light gray. When the circles p and q are edited, they intersect the 4D bounding box represented by the tunnel section. Therefore, o's interpolant, stored in some leaf of the global linetree, is flagged as a potential candidate for invalidation. However, o's interpolant is only affected by changes in the dark gray region and this invalidation is unnecessary. We address this problem by introducing an extra parameter t for rays. Intuitively, this parameter represents the distance along the 4D lines. In Figure 5-(b), the light gray line space region is bounded by $t = t_0$ and $t = t_1$. Using this extra parameter, o's interpolant is not flagged for invalidation when the circles are updated.

4.2 Global ray segment trees

To efficiently identify the interpolants that are affected by an edit, the system maintains six *global ray segment trees* (RSTs). Each RST node stores ten coordinates $(a_0, b_0, c_0, d_0, t_0)$ to $(a_1, b_1, c_1, d_1, t_1)$ that define a 5D bounding box in ray segment space. The t dimension represents the distance along the principal direction of the face pair. The front face of the face pair is at $t = 0$ and the back face is at $t = 1$. The root node of the tree spans the region from $(0, 0, 0, 0, 0)$ to $(1, 1, 1, 1, 1)$. When an RST node is subdivided, each of its five axes is subdivided simultaneously. Each of the 32 children of the RST node covers the region of 5D ray space that includes all rays from its front face to its back face. While this branching factor may seem high, the tree is sparse, keeping memory requirements modest.

Figure 6 shows RST nodes for 2D rays. The parent node from (a_0, c_0, t_0) to (a_1, c_1, t_1) is shown on the top left, and a-c-t ray segment space (a three dimensional unit cube) is shown on the top right. The parent represents all rays entering its front face and leaving its back face. When the parent is subdivided, the rays represented by its eight children are as shown. Children 0 through 3 correspond to the ray segments that start at the front face at $t = t_0$ and end at the middle face at $t = \frac{t_0+t_1}{2}$. Similarly, children 4 through 7 start at the middle face and end at $t = t_1$. When the parent is subdivided, truncated segments of the parent's rays (shown in black in the figure) lie in different children.

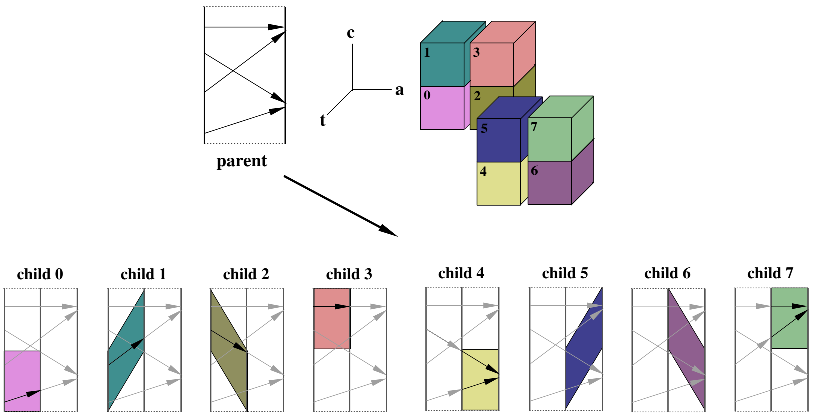


Fig. 6. Subdivision of Ray Segment Trees.

4.3 Inserting interpolants in the RSTs

RSTs are populated with interpolants by a recursive insertion algorithm that starts from the root RST node. For each tunnel section of the interpolant, its sixteen extremal rays are intersected with the current RST node to compute a 5D bounding box that includes the tunnel section. If this bounding box intersects a leaf RST node, a pointer to the interpolant is inserted in the node. For a non-leaf node, the algorithm recursively inserts the interpolant into the children of the RST node that intersect its 5D bounding box. As described in Section 3, the leaf node in an RST stores a list of pointers to every interpolant that *depends* on the region of ray segment space covered by that node and a list of the 5D bounding boxes of the interpolant's corresponding tunnel section.

5 Using Ray Segment Trees

In this section, we describe how interpolants affected by an object edit are rapidly identified and invalidated using RSTs. Brière and Poulin [3] describe two main categories of object edits: attribute changes (including changes to an object's color, specular or diffuse coefficient), and geometry changes (including insertion or deletion of an object). In their work, attribute and geometry changes are handled using different mechanisms, since attribute changes can be dealt with rapidly, while geometry changes require more time. In our work, RSTs permit rapid identification of affected interpolants; therefore, we use the same mechanism to identify affected interpolants for both types of changes.

5.1 Identifying Affected Interpolants

When an object is edited, we use 3D shafts [6] to identify every region of ray segment space, and therefore every associated interpolant, that is affected by the edit. The identification algorithm is recursive and starts at each of the six root RST nodes with a world-space region v (the object's bounding box) that is affected by an object edit. For each RST node visited recursively, a shaft is built enclosing the 3D volume between the front and back face of the node. The shaft consists of six planes: four planes from each edge of the node's front face to the corresponding edge of its back face, and two planes that correspond to its front and back faces. If the shaft intersects v, the children of the RST node are recursively tested for intersection with v. When the shaft of a RST node does not intersect v, the descendants of that node are not visited. If the RST node is a leaf, it has a list of pointers to interpolants that depend on the 3D volume represented by the node's shaft, and the 5D bounding boxes of their corresponding tunnel sections. A 3D shaft is constructed for each such tunnel section. If that shaft intersects v, the interpolant is flagged as a candidate for update. Our approach is similar to the shafts presented by Drettakis and Sillion [5], though they implicitly use the radiosity link structure to construct shafts. In [3], shafts are constructed for pixel-based rays. Plate A shows the interpolants that depend on the reflective mirror for the museum scene shown in Plate B.

One class of affected interpolants (depicted in Figure 3-(c)), can be identified rapidly using a different mechanism. While building an interpolant for o, if a light is occluded by an opaque object b, that tunnel section of the interpolant can only be affected when b moves. Therefore, we maintain a separate list of interpolants for occluders; when b is edited, its list of interpolants is marked for invalidation.

5.2 Interpolant Invalidation

The algorithm to identify affected interpolants is conservative: it might flag interpolants for update even if they are not affected by an object edit, because shaft culling against the edited object's bounding box is conservative. Therefore, we perform an additional check on the position-independent component of the interpolant's ray tree to determine if the interpolant is affected by the edit. For example, when o's color is edited, the edit affects an interpolant I if either I is o's interpolant, or I depends on o indirectly, for example through reflections. For a geometry change, such as the deletion of an object o, an interpolant I should be invalidated if I is o's interpolant, or o appears in the ray tree of I, for example, as an occluder or a reflection. Note that, as in [3], we treat an object movement as a deletion from its old position and an insertion to its new position.

When an interpolant is invalidated, the memory allocated to the corresponding object's linetree node is automatically garbage collected and the node itself is marked for deletion. If recursively, all that linetree node's siblings are also invalid, their space is reclaimed, and therefore, the parent is reclaimed. For example, consider an object o_1 that blocks the light to another object o_2, causing o_2's linetrees to be subdivided around o_1's shadow. When o_1 is deleted, o_2's interpolants are compacted, so that no unnecessary subdivision of o_2's linetrees takes place around the shadow that no longer exists.

To support rapid editing for attribute changes, interpolants could be augmented to include extra information such as the surface normal and point of intersection, for each of the sixteen extremal rays. Using this extra information, the interpolants could be updated by computing the difference in radiance due to the change in o's material properties. However, this extra position-dependent information increases the memory requirements of interpolants. Therefore, we invalidate interpolants for both attribute and geometry changes and lazily recompute them as needed.

6 Performance Results

We have extended our accelerated interpolant ray tracer to maintain and use RSTs for scene editing. The interpolant ray tracer, and the base ray tracer it is compared with, both implement classical Whitted ray tracing [13] with textures and use the Ward isotropic local shading model [12]. The system can handle convex primitives such as cubes, spheres, cylinders, cones, disks and polygons, and CSG union and intersection operations on these primitives. The base ray tracer contains several standard performance optimizations (see [2] for details).

Our timing results were obtained for the museum scene shown in Plate B. The scene has more than 1100 primitives (a tesselation of these primitives requires about 500k polygons). All timing results are reported for frames rendered at 1200×900 resolution on a single 250MHz processor of an SGI Infinite Reality. We report results for three edits (shown in Plate B): the top of the sculpture is deleted (Edit-(a)), the bottom of the sculpture is deleted (Edit-(b)), a green cube is moved in on the right (Edit-(c)). Camera translations and rotations correspond to small adjustments of the viewpoint; a forward translation is by 0.2 feet (the room size is 45×30 sq. ft.), while a rotation is by $2.5°$. When the user changes the viewpoint, new interpolants are built as required.

Plate B shows the impact of edits on interpolants. On the left, rendered images are shown, and on the right are error-coded images showing the regions of interpolation failure and success. Green and yellow pixels are not interpolated due to radiance discontinuities such as shadow edges and object silhouettes. Magenta pixels are not interpolated because of adaptive error-driven subdivision. Pixels that are successfully interpolated are shown in dark blue. The red pixels show the interpolants that are invalidated and rebuilt when the scene is edited. For example, after Edit-(a), the top of the sculpture and the shadow behind it are updated; the new interpolants lazily built to cover those pixels are shown in red. After Edit-(b), the interpolants associated with the bottom of the sculpture and its reflection in the mirror are found and invalidated.

In Table 1, we present results for time and memory usage for scene edits. A change to the viewpoint is considered a scene edit, except that no interpolants are invalidated by the viewpoint change. This is because interpolants do not explicitly depend on the current viewpoint. For each of the edits, traversing the RSTs and invalidating the corresponding linetrees is extremely fast, on the order of *a tenth of a second*. Depending on the type of edit, and its impact on interpolants, updating interpolants lazily while rerendering a frame takes 26 to 28 seconds. Of this time, building new interpolants lazily, shown in red in the plate, takes 1 to 3 seconds. Similar results are obtained when the object's attributes (e.g., color) are changed. As the camera position is changed, frames are rendered in 26 to 31 seconds, depending on the camera movement; the greater the reuse of interpolants from the previous frame, the shorter the rendering time.

The memory requirements of this system are modest: each edit requires an additional 0.6 to 1 MB of memory. Camera movements typically require 0.7 to 2.1 MB of memory, depending on the type and extent of the movement. Additionally, in [2], we

Edit	Base ray tracer	Interpolant ray tracer with RSTs		
		Time (in secs)		Memory (in MB)
		Traverse RSTs and Invalidate linetrees	Update and Re-render	
Edit-(a)	109.0	0.11	25.6	0.7 M
Edit-(b)	108.6	0.10	28.2	1.0 M
Edit-(c)	109.2	0.09	27.4	0.6 M
Pan camera	108.2	—	26.9	0.7 M
Step forward	108.5	—	31.2	2.1 M

Table 1. Time and memory usage for edits and camera movements.

Ray Tracer System	Time (in secs)	Memory (in MB)
Base	109.0	—
Interpolant	79.5	18.0 M
Interpolant with RSTs	80.4	23.5 M

Table 2. Time and memory usage for ray tracers supporting interpolants and RSTs.

have implemented a least-recently used (LRU) memory-management technique to limit memory used for interpolants to a user-specified maximum. This technique imposes a performance penalty of only 1% when memory usage is restricted to 40MBs. This LRU system can be easily extended to limit the memory used for RSTs.

Table 2 shows system performance when rendering the first frame. Unlike in [3], the ray tracer using interpolants is 25% faster than the base ray tracer *even on the first frame*: interpolants exploit the spatial coherence within a frame. Note that our algorithm is an on-line algorithm; no pre-processing is needed to build either linetrees or RSTs. The overhead of creating RSTs is small: less than 1 second. For the first frame, interpolants require 18 MBs of memory, while RSTs require an additional 5.5 MBs. Subsequent frames require much less memory, as shown in Table 1.

7 Conclusions and Future Work

We have presented an incremental ray tracer for scene manipulation that permits the user to edit the scene and the current viewpoint. The system maintains ray segment trees to track the dependencies of interpolants on regions of world space. When the scene is edited, the RSTs are rapidly traversed, in roughly a tenth of a second, to identify and invalidate the interpolants affected by the edit. The interpolants are rebuilt as needed. For full-screen images, the scene is re-rendered with 3 to 4× speedup over the base ray tracer. For small adjustments to the viewpoint, incremental rendering is effective. For large changes in the camera position and even for the first frame, the system is still faster than the base ray tracer.

We believe this is a promising approach to support scene editing in ray tracers. There are several extensions that will improve the system. To support faster re-rendering when the scene is edited without changing the viewpoint, we could add screen-space acceleration structures similar to those described by Brière and Poulin [3]. Since interpolants succeed for a large number of pixels (e.g., 90% for the museum scene), these acceleration structures will only be maintained for the small fraction of the pixels that fail interpolation. Therefore, we expect that they will significantly accelerate rendering when the camera is not moved, while having modest memory requirements. These screen-space structures can be invalidated when the camera moves, and re-built as necessary. With this optimization, we expect that both attribute and geometry changes will require less than 3-5 seconds for fixed viewpoints.

One constraint of the interpolant ray tracer is that it can only guarantee error bounds for convex primitives. Interpolants are not built for non-convex or transparent primitives; therefore, rendering of these primitives is not accelerated. For scene editing, the screen-space structures discussed above could still be used to rapidly update these objects. However, we would like to accelerate the rendering of non-convex and transparent objects when the viewpoint changes as well. In [2], we discuss how to extend the error predicate to support non-convex objects by using linear interval arithmetic. We are currently extending our system to support parametric patches and transparent objects.

8 Acknowledgements

We would like to thank Andrew Myers for many useful discussions. This work was supported by an Alfred P. Sloan Research Fellowship (BR-3659), an NSF CAREER award (CCR-9624172), an NSF CISE Research Infrastructure award (EIA-9802220), a ONR MURI Award (SA-15242582386) and a grant from the Intel Corporation.

References

1. ARVO, J., AND KIRK, D. Fast Ray Tracing by Ray Classification. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pp. 196–205.
2. BALA, K., DORSEY, J., AND TELLER, S. Radiance Interpolants for Accelerated Bounded-Error Ray Tracing. In *ACM Transactions on Graphics (1999, to appear)* (Also available at graphics.lcs.mit.edu/∼kaybee/publications.html).
3. BRIÈRE, N., AND POULIN, P. Hierarchical View-dependent Structures for Interactive Scene Manipulation. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 83–90.
4. COOK, R. L., PORTER, T., AND CARPENTER, L. Distributed Ray Tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), vol. 18, pp. 139–147.
5. DRETTAKIS, G., AND SILLION, F. X. Interactive Update of Global Illumination Using a Line-Space Hierarchy. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 57–64.
6. HAINES, E., AND WALLACE, J. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proc. 2^{nd} Eurographics Workshop on Rendering* (May 1991).
7. JEVANS, D. Object Space Temporal Coherence for Ray Tracing. In *Proceedings of Graphics Interface '92* (Toronto, Ontario, May 1992), pp. 176–183.
8. MURAKAMI, K., AND HIROTA, K. Incremental Ray Tracing. In *Photorealism in Computer Graphics*, K. Bouatouch and C. Bouville, Eds. Springer-Verlag, 1992.
9. SÉQUIN, C. H., AND SMYRL, E. K. Parameterized Ray Tracing. In *Computer Graphics (SIGGRAPH '89 Proceedings)*.
10. SOMMERVILLE, D. M. Y. *An Analytical Geometry of Three Dimensions*. University Press, Cambridge, 1959.
11. TELLER, S., BALA, K., AND DORSEY, J. Conservative Radiance Interpolants for Ray Tracing. In *Seventh Eurographics Workshop on Rendering* (June 15-17 1996), pp. 258–269.
12. WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. A Ray Tracing Solution for Diffuse Interreflection. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, pp. 85–92.
13. WHITTED, T. An Improved Illumination Model for Shaded Display. *CACM 23*, 6 (1980), 343–349.

A Appendix

In this appendix, we characterize the region of line space affected by an object edit. In 2D, the region of line space affected by an object edit is a hyperbola. We then extend this result to an object edit in 3D — the region of 4D line space affected by the edit can be characterized by a fourth-order equation.

A.1 In 2D

Consider a scene and its four global segment pairs. In Figure 2-(a), one of the four segment pairs (the pair of thick horizontal lines), with $+\hat{z}$ as principal direction, is shown on the left. When an object o is edited, every ray that passes through o is affected by the edit. We prove that the region of line space (shown as a square on the right of the figure) affected by updates to o is a hyperbola in 2D.

A ray **R** is specified by its intercepts $[a, c]$ on its associated segment pair. Without loss of generality, $\mathbf{R} = [c-a, 1]$. If **R** is a ray on the boundary of the region of line space affected by the edit, it satisfies two additional constraints: **R** intersects the circle o at some point $\mathbf{P} = [X, Z]$, and **R** is tangential to the circle at **P**. We have three constraints: **P** lies on **R**, **R** is perpendicular to the normal at **P**, **P** lies on the circle.

$$[X, Z] = [a, -\frac{1}{2}] + t[c-a, 1], \mathbf{R} \cdot \mathbf{N} = 0, (X - C_x)^2 + (Z - C_z)^2 = R^2$$

Eliminating t, X and Z:

$$[(c-a)C_z + (\frac{a+c}{2} - C_x)]^2 - R^2[1 + (c-a)^2] = 0 \quad (1)$$

Equation 1 is a second order equation in a and c; the discriminant of the equation satisfies the condition of a hyperbola [10]. Thus, when a circle o is edited, the region of 2D line space affected by the edit is a hyperbola — i.e., the rays in the shaded region on the right in Figure 2-(a) are affected by the edit. The parameters of the hyperbola can be derived from o's location and radius.

A.2 In 4D

A similar derivation identifies the region of 4D line space affected by an edit to a 3D sphere o. Each ray **R** associated with the face pair with principal direction $+\hat{z}$ is specified as $[c-a, d-b, 1]$. The region of 4D space affected by an edit to a 3D sphere is characterized by the following equation:

$$\begin{aligned}
[(c-a)C_z + (\frac{a+c}{2} - C_x)]^2 + [(d-b)C_z + (\frac{b+d}{2} - C_y)]^2 & \\
-R^2[1 + (c-a)^2 + (d-b)^2] & \\
+[(c-a)(\frac{b+d}{2} - C_y) - (d-b)(\frac{a+c}{2} - C_x)]^2 &= 0
\end{aligned}$$

While the first two lines of the equation are exactly the 4D generalization of a 2D hyperbola, the third line introduces fourth-order cross terms. Thus, when a 3D sphere o is edited, the region of 4D line space affected by the edit is not a hyperboloid, but it is specified by a fourth-order equation. Every ray *inside* the surface represented by this equation could potentially be affected by the object edit.

Plate A: Tunnels and Interpolant Dependencies

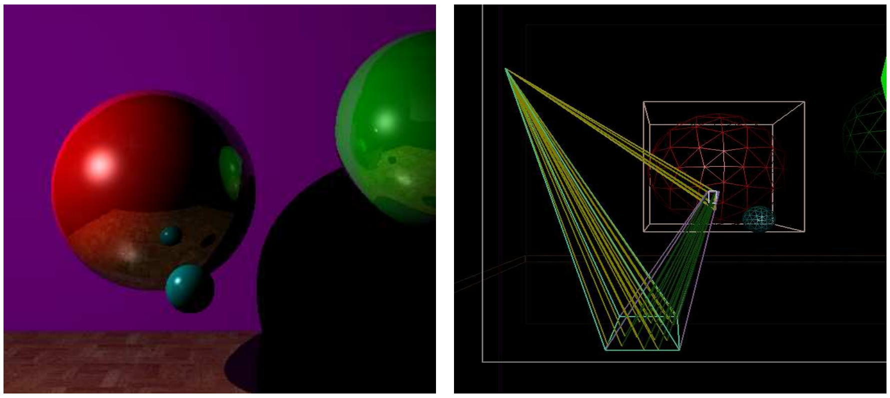


Figure 1: 3 spheres. On the right, one interpolant for the red sphere is shown. The interpolant has: a reflection tunnel to the ground, and a direct tunnel to the light.

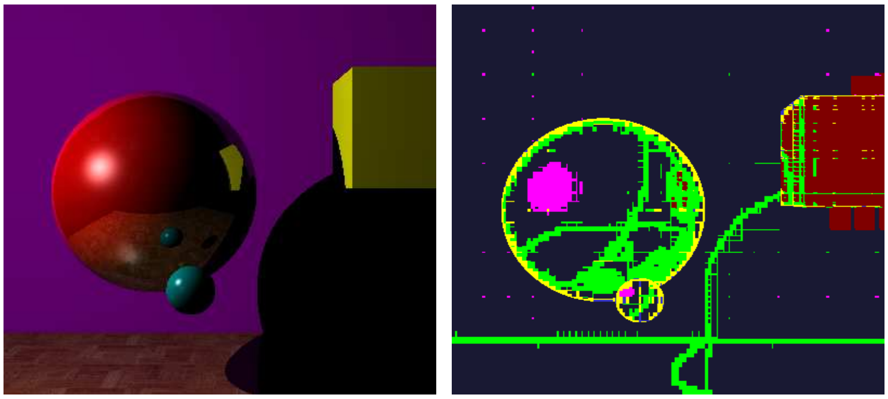


Figure 2: Scene Edit. The green sphere is replaced by the yellow cube. On the right, a color-coded image shows the impact of the edit. Blue-gray pixels are interpolated. Green, yellow and magenta pixels fail due to the error predicate. ***Interpolants are only invalidated and rebuilt for the dark red pixels.***

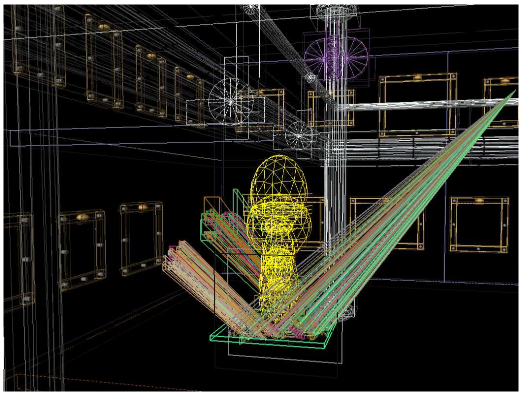
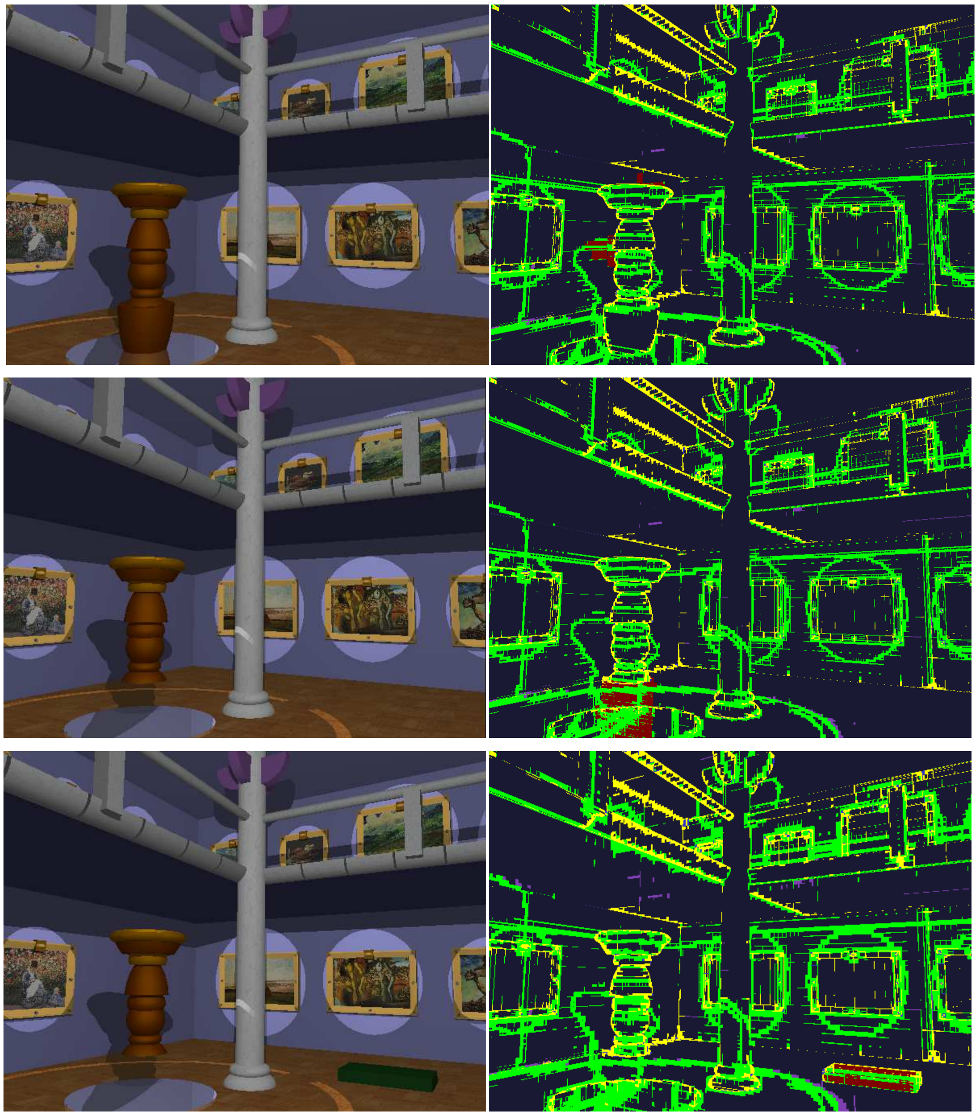


Figure 3: Museum Scene, Interpolant dependencies. Interpolants that depend on the reflective mirror.

Plate B: Scene Edits for Museum Scene



Museum Scene

Top to Bottom:
Edit-(a) - delete top of sculpture
Edit-(b) - delete bottom of sculpture
Edit-(c) - add green bench