In ACM Transactions on Graphics (TOG), Volume 18, issue 3, August 1999.

Radiance Interpolants for Accelerated Bounded-Error Ray Tracing

Kavita Bala, Julie Dorsey and Seth Teller
Laboratory for Computer Science
Massachusetts Institute of Technology

Ray tracers, which sample radiance, are usually regarded as off-line rendering algorithms that are too slow for interactive use. In this paper, we present a system that exploits object-space, ray-space, image-space and temporal coherence to accelerate ray tracing. Our system uses *per-surface interpolants* to approximate radiance, while *conservatively bounding error*. The techniques we introduce in this paper should enhance both interactive and batch ray tracers.

Our approach explicitly decouples the two primary operations of a ray tracer—shading and visibility determination—and accelerates each of them independently. Shading is accelerated by quadrilinearly interpolating lazily acquired radiance samples. Interpolation error does not exceed a user-specified bound, allowing the user to control performance/quality tradeoffs. Error is bounded by adaptive sampling at discontinuities and radiance non-linearities.

Visibility determination at pixels is accelerated by *reprojecting* interpolants as the user's viewpoint changes. A fast scan-line algorithm then achieves high performance without sacrificing image quality. For a smoothly varying viewpoint, the combination of lazy interpolants and reprojection substantially accelerates the ray tracer. Additionally, an efficient cache management algorithm keeps the memory footprint of the system small with negligible overhead.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Ray tracing, color, shading, shadowing, textures*; G.1.1 [**Numerical Analysis**]: Interpolation—*Piecewise polynomial interpolation*; G.1.2 [**Numerical Analysis**]: Approximation—*Linear Approximation*

General Terms: Algorithms, Data Structures, Rendering Systems

Additional Key Words and Phrases: 4D interpolation, error bounds, interactive, interval arithmetic, radiance approximation, rendering, visibility

1. INTRODUCTION

A primary goal of computer graphics is the rapid generation of accurate, high quality imagery. Global illumination algorithms generate realistic images by evaluating radiance, a function over the five-dimensional space of rays. To achieve

Address: {kaybee,dorsey,seth}@graphics.lcs.mit.edu, http://graphics.lcs.mit.edu.
In ACM Transactions on Graphics (TOG), Volume 18, issue 3, August 1999.

reasonable performance, illumination systems trade freedom of viewer motion for scene complexity and accuracy. At one end of the spectrum, ray tracers [Whitted 1980] support generalized geometric primitives and specular and diffuse reflectance functions to produce high-quality view-dependent images; however, this quality is achieved at the expense of interactivity. At the other end of the spectrum, radiosity systems [Goral et al. 1984] support interactive viewing of pre-computed radiosity values, but typically render only diffuse, polygonal environments. Hybrid systems attempt to bridge the gap between these two extremes [Chen et al. 1991; Sillion and Puech 1989; Sillion and Puech 1994; Wallace et al. 1987; Ward et al. 1988]. However, computing the view-dependent component of radiance is expensive; imagery of ray-traced quality is traditionally produced by off-line rendering algorithms that are too slow for interactive use.

In this paper, we present a system that accelerates ray tracing by exploiting spatial and temporal coherence. The goal of our system is to provide an environment in which the user moves around freely as the system renders images of ray-traced quality. To achieve this goal, we decouple and independently accelerate the two primary operations of a ray tracer: the visibility computation that determines the closest visible object along an eye ray, and the shading computation for the visible point so identified.

The key insight in accelerating shading is as follows: ray tracers sample the radiance function at each ray from the viewpoint through every pixel of an image. When the radiance function varies smoothly, a sparse set of samples can be interpolated to approximate radiance. This set of radiance samples is called an *interpolant*. When interpolation is possible, this approximation eliminates the expensive shading computation of the ray tracer. However, interpolation can introduce errors; one important goal of our work is to characterize interpolation error. Radiance is interpolated only when the interpolated radiance is guaranteed to be within a user-specified error bound ϵ of the radiance computed by the base ray tracer.

Visibility computation is accelerated by exploiting frame-to-frame temporal coherence: when the viewpoint changes, objects visible in the previous frame are still typically visible in the current frame. Therefore, interpolants from the previous frame are *reprojected* to the new viewpoint to rapidly determine visibility and shading for pixels in the new frame.

In designing and building this system, we have made several contributions:

—*Radiance interpolation:* Radiance can be approximated rapidly by *quadrilinear* interpolation of a set of samples stored in an interpolant.

—*Linetrees:* A hierarchical data structure called a *linetree* is used to store interpolants. The appropriate interpolant for a particular eye ray is located rapidly by walking down the linetree. Linetrees are subdivided adaptively (and lazily), thereby permitting greater interpolant reuse where radiance varies smoothly, and denser sampling where radiance changes rapidly.

—*Error-driven sampling:* We have developed new techniques for bounding the error introduced by interpolation. When radiance is approximated, the relative error between interpolated and true radiance is less than a user-specified error bound ϵ. The user can vary ϵ to trade performance for quality.

Interpolation error arises both from discontinuities and non-linearities in the ra-

diance function. Our error bounding algorithm automatically and conservatively prevents interpolation in both these cases. Where the error bounding algorithm indicates rapid variations in radiance, radiance is sampled more densely.

—*Visibility by reprojection:* Determination of the visible surface for each pixel is accelerated by a novel *reprojection* algorithm that exploits temporal frame-to-frame coherence in the user's viewpoint, but guarantees correctness. A fast scan-line algorithm uses the reprojected linetrees to further accelerate rendering.

—*Memory management:* Efficient cache management keeps the memory footprint of the system small, while imposing a negligible performance overhead (1%).

The rest of the paper is organized as follows: Section 2 presents related work and discusses how our system differs. Section 3 presents an overview of the interpolant rendering algorithm. Descriptions of the interpolant building mechanism, error bounding algorithm and reprojection follow in Sections 4, 5, and 6. Section 7 discusses the implementation and performance optimizations. Finally, Section 8 presents results and Section 9 concludes with a discussion of future work.

2. RELATED WORK

Many researchers have developed techniques that improve the performance of global illumination algorithms: adaptive 3D spatial hierarchies [Glassner 1984], beam-tracing for polyhedral scenes [Heckbert and Hanrahan 1984], cone-tracing [Amanatides 1984], and ray classification [Arvo and Kirk 1987]. A good summary of these algorithms can be found in [Glassner 1989; Glassner 1995; Sillion and Puech 1994]. In this section, we discuss the related work most relevant to our approach.

2.1 Approximating shading

Systems that accelerate rendering by approximating radiance can be categorized on the basis of the shading models they use, the correctness guarantees (if any) provided for computed radiance, and their use of pre-processing. Some of these systems also approximate visibility by polygonalizing the scene, or by using images instead of geometry.

The RADIANCE system uses ray tracing to produce high quality images while lazily sampling diffuse inter-reflections [Ward 1992; Ward et al. 1988]. RADIANCE uses gradient information to guide the sparse, non-uniform sampling of the slowly varying diffuse component of radiance [Ward and Heckbert 1992]. However, RADIANCE does not bound the error incurred by interpolating diffuse radiance, nor does it interpolate other components of radiance.

Diefenbach's rendering system [Diefenbach and Badler 1997] uses multiple passes of standard graphics hardware to acquire some of the realism of ray tracing. However, the system approximates visibility by discretizing the scene into polygons, and has no correctness guarantees.

Image-based rendering (IBR) systems, such as the Light Field [Levoy and Hanrahan 1996] and the Lumigraph [Gortler et al. 1996], have similarities to our system in that they collect 4D radiance samples that are quadrilinearly interpolated to approximate radiance. However, IBR systems typically have a data acquisition pre-processing phase and are not intended to compute radiance on the fly. Light Fields and Lumigraphs are uniformly subdivided 4D arrays whose fixed size is

determined in the pre-processing phase. This fixed sampling rate does not guarantee that enough samples are collected in regions with high-frequency radiance variations, and may result in over-sampling in regions where radiance is smooth. Additionally, the viewpoint is constrained to lie outside the convex hull of the scene. Recently, layered depth images have been used to reproject both diffuse and specular radiance for new viewpoints [Lischinski and Rappoport 1998]. For small scenes this approach has better memory usage and visual results than the Light Field or Lumigraph. Scenes with greater depth complexity could require excessive memory. Also, though this technique alleviates artifacts for specular surfaces, it still relies on sparse radiance sampling that is not error-driven. IBR techniques have also been used to warp pre-rendered images in animated environments with moving viewpoints [Nimeroff et al. 1995]. However, none of these image-based systems bounds the error introduced by approximating visibility or radiance.

Several researchers exploit image coherence to accelerate ray tracing [Amanatides and Fournier 1984]. Recently, systems for the progressive refinement of ray-traced imagery have been developed [Guo 1998; Pighin et al. 1997]. Guo samples the image sparsely along discontinuities to produce images for previewing. For polyhedral scenes, Pighin et al. compute image-space discontinuities used to construct a constrained Delaunay triangulation of the image plane. This Delaunay triangulation drives a sparse sampling technique to produce previewable images rapidly. The traditional problem with screen-space interpolation techniques is that they may incorrectly interpolate across small geometric details, radiance discontinuities, and radiance non-linearities. While both these systems alleviate the problem of interpolation across discontinuities, neither system bounds error. Also, both systems detect discontinuities in the image plane; therefore, when the viewpoint changes, discontinuities have to be recomputed from scratch.

2.2 Approximating visibility

Algorithms that exploit temporal coherence to approximate visibility at pixels can be categorized by the assumptions they make about the scene and the correctness guarantees they provide. Chapman et al. assume the scene is polygonal and use the known trajectory of the viewpoint through the scene to compute continuous intersection information of rays [Chapman et al. 1990; Chapman et al. 1991]. Several recent systems reuse pixels from the previous frame to render the current frame without any prior knowledge of the viewpoint's trajectory [Badt 1988; Adelson and Hodges 1995; Mark et al. 1997; Chevrier 1997].

Adelson and Hodges apply a 3D warp to pixels from reference images to the current image. This algorithm speeds up visibility for eye rays but does not accelerate shading. Mark et al. also apply a 3D warp to pixels, but treat their reference image as a mesh and warp the mesh triangles to the current image. Their system cannot guarantee correct results for arbitrary movements of the eye. Additionally, both these systems suffer from aliasing effects arising from the fact that pixels are not warped to pixel centers in the current frame; i.e., neither system accurately determines visibility. Chevrier computes a set of key views used to construct a 3D mesh that is interpolated for new viewpoints. If a pixel is not covered by one key view, more key views are used. To handle specularity, one 3D mesh per specular surface is built, and the specular coefficient is linearly interpolated from multiple key im-

ages. While this algorithm decreases some aliasing artifacts, it still may interpolate across shadows or specular highlights; i.e., there are no bounds on error.

2.3 Discussion

In [Bala et al. 1998; Teller et al. 1996], we presented a preliminary algorithm that accelerates shading for a Whitted ray tracer by building 4D interpolants that are reused to satisfy radiance queries. We have added several new contributions to this work: a complete algorithm for bounding error, a reprojection algorithm that exploits temporal coherence, support for texturing, and several performance optimizations. Our work differs from previous rendering systems in several respects:

—We introduce radiance interpolants to approximate radiance while bounding interpolation error conservatively using linear interval arithmetic.

—The user can trade speed for rendering quality using error-driven subdivision.

—Interpolants are built on-line, lazily and adaptively; no pre-processing is required.

—Reprojection accelerates visibility determination by exploiting temporal coherence without introducing *aliasing* artifacts.

3. ALGORITHM OVERVIEW

This section presents an overview of our interactive rendering system.

Base ray tracer. The base ray tracer is a classical Whitted ray tracer [Whitted 1980] extended to implement the Ward isotropic shading model [Ward 1992] and textures. It supports convex primitives (spheres, cubes, polygons, cylinders and cones) and the CSG union and intersection of these primitives [Roth 1982]. To accelerate intersection computations several optimizations have been implemented; these optimizations are discussed in Section 8.1.

Linetrees. Radiance samples are stored in a data structure called the *linetree*, which is the 4D equivalent of an octree. Each object has a set of associated linetrees that store its radiance samples. This hierarchical tree organization permits the efficient lookup of interpolants for each eye ray. See Section 4 for details on linetrees.

Rendering Algorithm. Figure 1 depicts the three rendering paths of the interpolant rendering algorithm: the *fast path*, the *interpolate path*, and the *slow path*. Along the fast path the system exploits temporal coherence by reprojecting linetree cells from the previous frame; this accelerates both visibility and shading for pixels covered by reprojected linetree cells. Shading of other pixels is accelerated by interpolating radiance samples from the appropriate linetree cells (interpolate path). If both reprojection and interpolation fail for a pixel, the base ray tracer renders the pixel (slow path).

The fast path, indicated by the thick green line, corresponds to the case when reprojection succeeds. When a reprojected linetree cell is available for a pixel, the system finds all consecutive pixels in that scanline covered by the same linetree cell, and interpolates radiance for these pixels in screen-space. In our results, this fast path is about 30 times faster than the base ray tracer (see Section 8).

If no reprojected linetree cell is available, the eye ray is intersected with the scene to determine the object visible at that pixel. The system checks for a valid

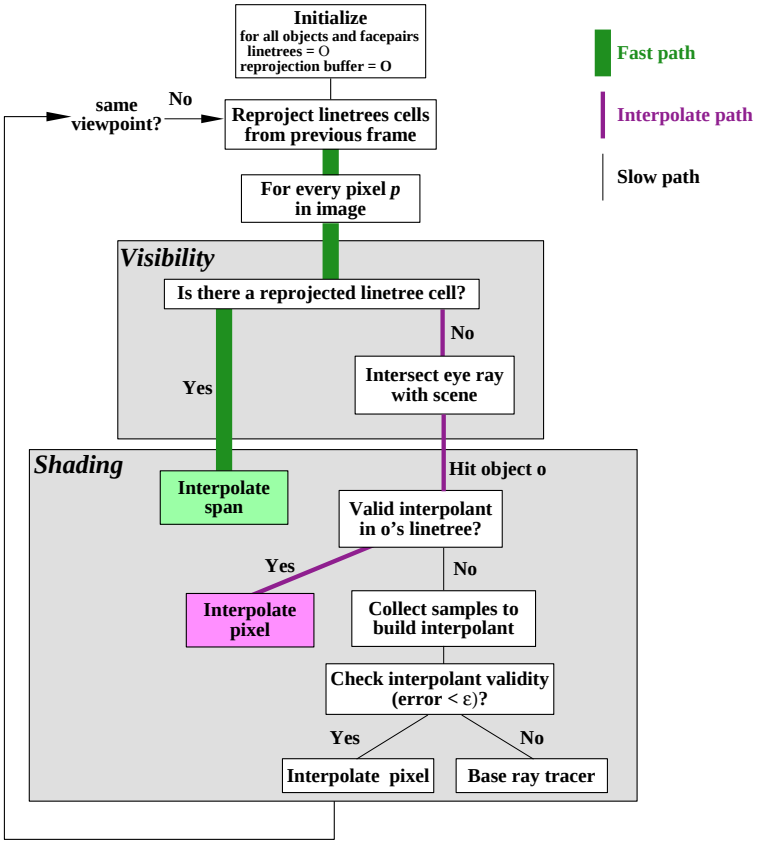

Fig. 1. Algorithm Overview.

interpolant for that ray and object; if it exists, the radiance for that eye ray (and the corresponding pixel) is computed by quadrilinear interpolation. This interpolate path, indicated by the medium-thick maroon line, is about 5 times faster than the base ray tracer.

If an interpolant is not available, the system builds an interpolant by collecting radiance samples for the appropriate new leaf linetree cell. The error bounding algorithm checks the validity of the new interpolant. If the interpolant is valid, the pixel's radiance can be interpolated. If it is not valid, the linetree cell is subdivided, and the system falls back to shading the pixel using the base ray tracer. This is the slow path indicated by the thin black line.

The user's viewpoint is tracked, and when it is stationary, the permitted interpolation error is gradually reduced to produce images of higher quality, until the user-specified error bound is reached. When the user's viewpoint changes, the system renders the scene at the highest speed possible, while preventing interpolation across radiance discontinuities.

4. RADIANCE INTERPOLANTS

Radiance is a function over the space of all rays. Our system lazily samples the radiance function and stores these samples in an auxiliary data structure, called the linetree. For each eye ray, the system finds and interpolates an appropriate set of radiance samples, called an *interpolant*, to approximate radiance.

First, we require a coordinate system describing rays. In this section, we introduce a per-object ray-space coordinate system that describes all rays intersecting an object with four parameters. Then, we present the linetree data structure that stores samples using their ray parameters as keys, and describe how interpolants are built and stored in the linetree.

4.1 Ray parameterization

For simplicity, we first consider 2D rays, and then extend the discussion to 3D rays.

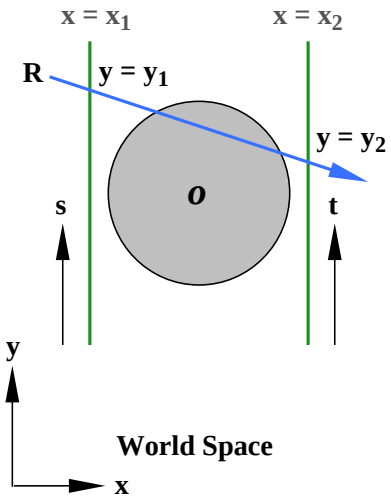

Fig. 2. A segment pair (green) and an associated ray R (blue).

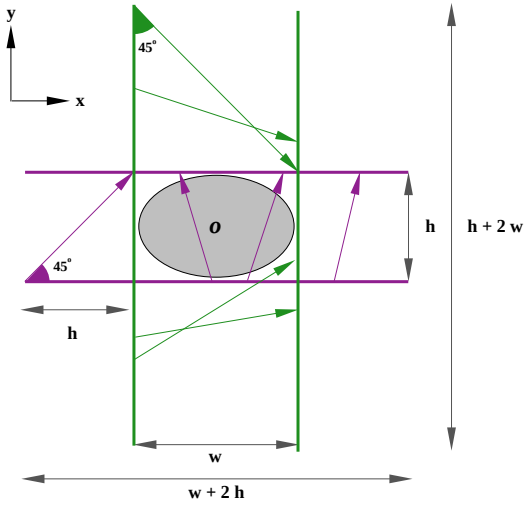

Fig. 3. Two segment pairs (green and maroon) and some associated rays.

4.1.1 *2D ray parameterization.* Every 2D ray can be parameterized by the two intercepts, s and t (see Figure 2), that it makes with two parallel lines (assuming the ray is not parallel to the lines). For example, consider two lines parallel to the y-axis at $x = x_1$ and $x = x_2$, on either side of an object o. Every ray $\mathbf{R}$ intersecting o that is not parallel to the y-axis can be parameterized by the y-intercepts that $\mathbf{R}$ makes with the two parallel lines; i.e., $(s, t) = (y_1, y_2)$. There are three problems with this parameterization: rays parallel to the y-axis cannot be represented; the intercepts of rays nearly parallel to the y-axis are numerically imprecise; and the orientation (right or left) of the ray is not specified by the parameterization.

Our representation avoids these problems by parameterizing each 2D ray with respect to one of four *segment pairs*. A segment pair is defined by two parallel line segments and a principal direction that is perpendicular to the line segments.

The four segment pairs have the principal directions $+\hat{x}$, $-\hat{x}$, $+\hat{y}$, and $-\hat{y}$. In each segment pair, the principal direction vector 'enters' one of the line segments (called the *front* segment) and 'leaves' the other line segment (called the *back* segment). The segment pairs with principal directions $+\hat{x}$ and $-\hat{x}$ have the same parallel line segments and only differ in the designation of front and back line segments. The same is true for the segment pairs with principal directions $+\hat{y}$ and $-\hat{y}$.

Every ray intersecting o is uniquely associated with the segment pair whose principal direction is closest to the ray's direction: the principal direction onto which the ray has the maximum positive projection. Once the segment pair associated with a ray is identified, the ray is intersected with its front and back line segments to compute its s and t coordinates respectively.

To ensure that every ray associated with a segment pair intersects both parallel line segments, the line segments are sized as shown in Figure 3. In the figure, an object o with a bounding rectangle of size $w \times h$ is shown with two of its four segment pairs. The segment pairs with principal directions $\pm\hat{x}$ have line segments of length $(h + 2w)$, while the segment pairs with principal directions $\pm\hat{y}$ have line segments of length $(w + 2h)$. This sizing ensures that the most extreme rays (rays at an angle of 45° to the principal direction) intersect the line segments of the segment pair with which they are associated. The green segment pair with principal direction $\hat{x}$ represents all rays $r = (r_x, r_y)$ with $|r_x| > |r_y|$ and $sign(r_x) > 0$; i.e., all normalized rays with r_x in $[\frac{1}{\sqrt{2}}, 1]$ and r_y in $[-\frac{1}{\sqrt{2}}, +\frac{1}{\sqrt{2}}]$. The maroon rays are associated with the maroon segment pair (principal direction $\hat{y}$).

The four segment pairs represent all rays intersecting the 2D object o. The maximal component of the direction vector of a ray and its sign identify the segment pair with which the ray is associated. The ray is intersected with this segment pair to compute its intercepts (s, t); these (s, t) coordinates parameterize the ray.

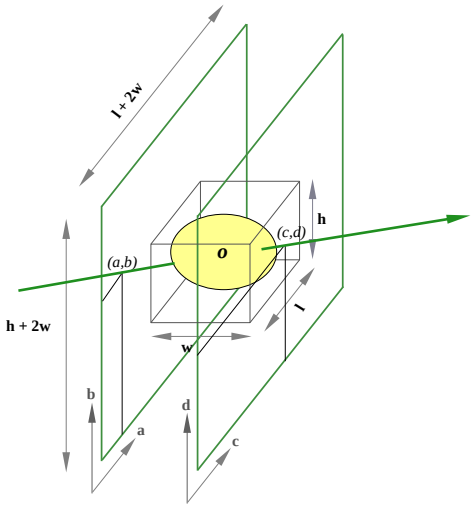

Fig. 4. Ray parameterization in 3D. The face pair is shown in green. The green ray intersects the front face at (a, b), and the back face at (c, d), and is parameterized by these four intercepts.

4.1.2 *3D ray parameterization.* The parameterization of the previous section is easily extended to 3D rays. Every ray intersecting an object o can be parameterized by the ray's four intercepts (a, b, c, d) with two parallel bounded *faces* surrounding o (see Figure 4). This parameterization is similar to some previous schemes [Gortler et al. 1996; Levoy and Hanrahan 1996; Teller et al. 1996].

Six pairs of faces surrounding o are required to represent all rays intersecting o. The principal directions of the six face pairs are $+\hat{x}$, $+\hat{y}$, $+\hat{z}$, $-\hat{x}$, $-\hat{y}$, and $-\hat{z}$. As in the 2D case, the faces are expanded on all sides by the distance between the faces, as shown in Figure 4. The dominant direction and sign of a ray determine which of the six face pairs it is associated with. The ray is parameterized by its four intercepts (a, b, c, d) with the two parallel faces of that face pair.

4.2 Interpolants and linetrees

We discuss how interpolants and linetrees are built and used to approximate radiance when rendering a frame, again considering 2D rays first for simplicity.

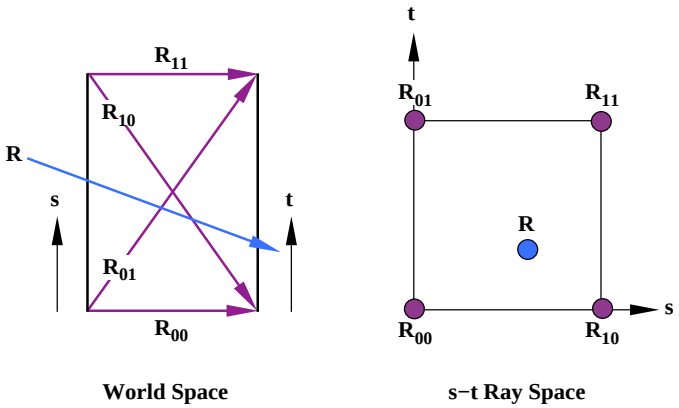

Fig. 5. A segment pair and its associated s-t ray space.

4.2.1 *2D ray space.* Every 2D ray is associated with a segment pair and is parameterized by its (s, t) coordinates. In Figure 5, a segment pair in 2D world space is shown on the left, and the corresponding s-t ray space is shown on the right. All rays associated with the segment pair in world space are points that lie inside a square in s-t space; the ray R, shown in blue, is an example. The extremal points at the four corners of the s-t square, R_{00}, R_{01}, R_{10}, and R_{11}, correspond to the rays in maroon shown on the left.

Radiance for any ray R inside the s-t square can be approximated by bilinearly interpolating radiance samples associated with the four extremal rays. This set of four radiance samples associated with the extremal rays is called an *interpolant.*

Radiance interpolants are stored in a hierarchical data structure called a *linetree*; each segment pair has an associated linetree. In 2D, the linetree is a quadtree built over ray space; the root of the linetree represents all the rays that intersect the segment pair. An interpolant is built at the root of the linetree by computing the radiance along the extremal rays, R_{00}, R_{01}, R_{10}, and R_{11}, that span the s-t square

in ray space. An error bounding algorithm (described in Section 5) determines if the interpolant is valid; i.e., if the interpolant can be used to approximate radiance to within a user-specified error parameter. If the interpolant is valid, it is used to bilinearly interpolate radiance for every eye ray R inside the s-t square. If the interpolant is not valid, the 2D linetree is subdivided at the center of both the s and t axes, as in a quadtree, to produce four children. Subdividing the s and t axes in ray space corresponds to subdividing the front and back line segments of the linetree cell in world space. The rays represented by the linetree cell can be divided into four categories depending on whether the rays enter by the top or bottom half of the front line segment and leave by the top or bottom half of the back line segment. These four categories correspond to the four children of the linetree cell. Therefore, rays that lie in the linetree cell are uniquely associated with one of its four children.

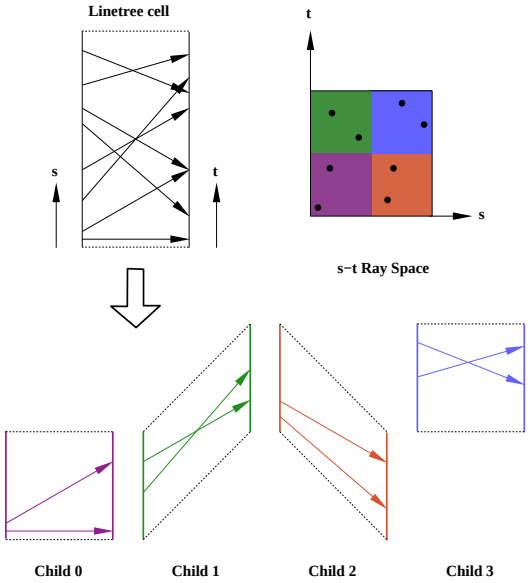

Fig. 6. A 2D segment pair and its children. Every ray that intersects the segment pair (represented as a point in s-t space) lies in one of its four subdivided children.

In Figure 6, a segment pair and its associated s-t ray space are depicted. On the top left, the segment pair is shown with some rays that intersect it. The four children of the subdivided segment pair are shown on the bottom. Each of the four children is represented by the correspondingly colored region of ray space shown in the top right. The dotted lines show the region of world space intersected by the rays represented by that linetree cell.

4.2.2 *4D ray space.* Now consider rays in 3D, which are parameterized by four coordinates (a, b, c, d). Each face pair corresponds to a 4D hypercube in ray space that represents all the rays that pass from the front face to the back face of the face pair. Each face pair has a 4D linetree associated with it that stores radiance

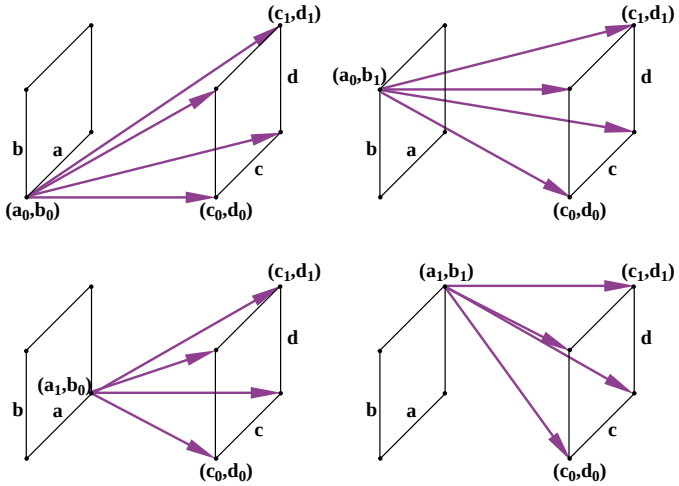

Fig. 7. 4D linetree cell and its sixteen extremal rays.

interpolants. The root of the linetree represents all rays associated with the face pair. When an interpolant is built for a linetree cell, samples for the sixteen extremal rays of the linetree cell are computed. In ray space, these sixteen rays are the vertices of the 4D hypercube represented by the linetree cell, and in world space they are the rays from each of the four corners of the front face of the linetree cell to each of the four corners of its back face. Figure 7 shows a linetree cell and its sixteen extremal rays.

If the error bounding algorithm (see Section 5) determines that an interpolant is valid, the radiance of any eye ray represented by that linetree cell is quadrilinearly interpolated using the stored radiance samples. If the interpolant is not valid, the linetree cell is subdivided adaptively; both the front and back faces of the linetree cell are subdivided along the a, b and c, d axes respectively. Thus, the linetree cell is subdivided into sixteen children; each child represents all the rays that pass from one of its four front subfaces to one of its four back subfaces. A ray that intersects the linetree cell uniquely lies in one of its sixteen children. This subdivision scheme is similar to that in [Teller and Hanrahan 1993].

4.3 Using 4D linetrees

Linetrees are used to store and look up interpolants during rendering. When an eye ray intersects an object, its four intercepts (a, b, c, d) are computed with respect to the appropriate face pair. The linetree stored with the face pair is traversed to find the leaf cell containing the ray. This leaf cell is found by walking down the linetree performing four interval tests, one for each of the ray coordinates. If the leaf cell contains a valid interpolant, radiance for that pixel is quadrilinearly interpolated. If a valid interpolant is not available, an interpolant for the leaf cell is built by computing radiance along the sixteen extremal rays of the linetree cell. The error bounding algorithm determines if the samples collected represent a valid interpolant. If so, the interpolant is stored in the linetree cell.

If the interpolant is not valid, the front and back faces of the linetree cell are

subdivided. Interpolants are lazily built for the child that contains the eye ray. Thus, linetrees are adaptively subdivided; this alleviates the memory problem of representing 4D radiance by using memory only where necessary. More samples are collected in regions with high frequency changes in radiance. Fewer samples are collected in regions with low frequency changes in radiance, saving computation and storage costs. Section 8 discusses a linetree cache management scheme that further bounds the memory usage of the system.

Figure 13 shows the linetree cells that contribute to the image of the sphere. A linetree cell can be shown as a shaft from its front face to its back face; however, this visualization is cluttered. To simplify the visualization, we show only the front and back face of each linetree cell. Each subdivision of a linetree cell corresponds to a subdivision of its front and back face. Therefore, highly subdivided front and back faces in the visualization correspond to highly subdivided linetree cells.

5. BOUNDING ERROR

Rendering systems trade accuracy for speed by using error estimates to determine where computation and memory resources should be expended. Radiosity systems use explicit error bounds to make this trade-off [Hanrahan et al. 1991; Lischinski et al. 1994]. Ray tracers typically use super-sampling and stochastic techniques to estimate error in computed radiance [Cook 1986; Painter and Sloan 1989]. This section describes how linear interval arithmetic can be used to bound interpolation error for the Ward isotropic shading model [Ward 1992]. It is straightforward to extend these results to handle surfaces with non-isotropic BRDFs as well.

Interpolant Validation

The error bounding algorithm receives as input the sixteen radiance samples of a linetree leaf cell and their associated ray trees, and answers the following question conservatively:

> Using quadrilinear interpolation, is the interpolated radiance within ϵ of the base radiance for *every* ray represented by that linetree leaf cell?

where the base radiance for a ray is the radiance computed by the base ray tracer. If the answer is yes, the interpolant is valid; otherwise, the linetree cell is adaptively subdivided.

Interpolation error could arise in two ways:

—Interpolation over a radiance discontinuity (due to shadows, occluding objects or total internal reflection).

—Interpolation over regions of ray space in which radiance varies non-linearly (for example, due to diffuse or specular peaks).

In Section 5.1, we present techniques to detect and avoid interpolation over discontinuities, while in Section 5.2, we discuss how linear interval analysis is used to bound interpolation error. Together these techniques completely specify the error bounding algorithm.

5.1 Radiance discontinuities

An interpolant is invalid if it interpolates radiance over a radiance discontinuity. We first consider the reasons why radiance is discontinuous in an image, and then we present techniques to detect these discontinuities.

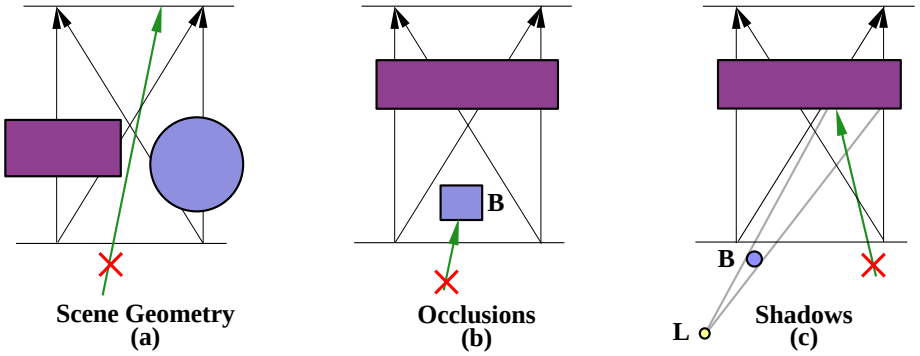

Fig. 8. Radiance discontinuities. Interpolation for the ray marked with the × would be erroneous.

5.1.1 *Causes.* Radiance discontinuities arise because the scene is composed of multiple geometrical objects that occlude and cast shadows on each other. Figures 8 and 9 show interpolants that are invalid due to radiance discontinuities in 2D. The black rays are the four extremal interpolant rays for a 2D linetree cell, and the green ray is a query ray that is represented by the linetree cell. In Figure 8-(a) the extremal (black) rays hit different objects while the query (green) ray misses the objects completely. In Figure 8-(b) the extremal rays all hit the same object, but the query ray hits an occluding object. In Figure 8-(c), the extremal rays and the query ray all hit the same object, but the blue circle **B** casts a shadow on the maroon rectangle. While the four extremal rays are illuminated by the light **L**, the query ray is not. In each of these three cases, it would be incorrect to interpolate radiance using the samples associated with the extremal rays.

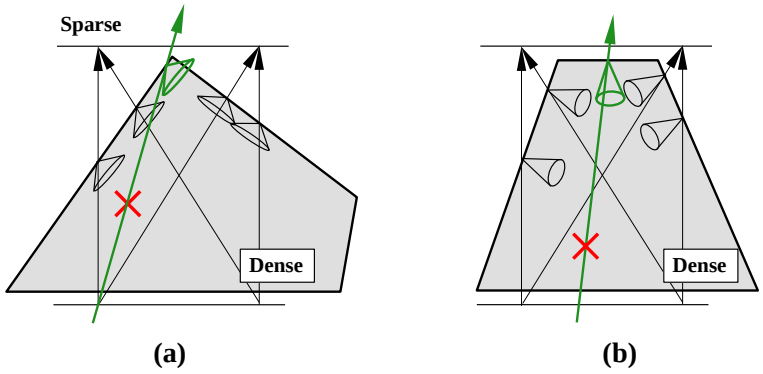

Fig. 9. Erroneous interpolation due to total internal reflection.

Figure 9 depicts discontinuities that arise due to total internal reflection (TIR). For a ray traversing different media, TIR occurs when the angle θ between the incident ray and normal is greater than the critical angle θ_c which is determined by the relative indices of refraction of the media. All rays outside the TIR cone (rays with $\theta > \theta_c$) undergo total internal reflection. In Figure 9-(a), the extremal rays lie in the TIR cone but the query ray does not; in Figure 9-(b), the extremal rays lie outside the TIR cone while the query ray lies in the cone. In both cases, interpolation would produce incorrect results [Teller et al. 1996].

5.1.2 *Detecting discontinuities.* The error bounding algorithm detects discontinuities by maintaining additional geometric information per extremal ray of the interpolant: *ray trees* [Séquin and Smyrl 1989]. These ray trees are used to detect self-shadows, occlusions, and visibility changes which could potentially cause incorrect interpolation.

A ray tree tracks all objects, lights, and occluders that contribute to the radiance of a particular ray: both direct contributions and indirect contributions through reflections and refractions. A ray tree node associated with a ray stores the object intersected by the ray in addition to the lights and occluders visible at the point of intersection. The children of the node are pointers to the ray trees associated with the corresponding reflected and refracted rays.

A crucial observation is that *radiance changes discontinuously over a linetree cell only when the ray trees associated with the rays represented by the linetree cell differ.* Therefore, to guarantee that interpolants do not erroneously interpolate over a radiance discontinuity, the error bounding algorithm must check that *all* rays in the 4D hypercube represented by a linetree cell have the same ray tree.

Textured surfaces are an exception to the observation above because radiance can change discontinuously across the texture. However, the argument above can be made about the *incoming* radiance at the textured surface: it changes discontinuously when the associated ray trees change (see Section 7.5). Therefore, to allow interpolation over textured surfaces, texture coordinates are interpolated separately from incoming radiance at the textured surface.

Erroneous interpolation over object edges is prevented by requiring that each primitive be built of a finite number of smooth (possibly non-planar) faces. For example, a cube has six faces, a cylinder has three faces, and a sphere has one face. This face index is also stored in the ray tree node and used in ray tree comparisons. Note that a ray tree stores only position-independent information, such as the identity of the objects intersected; the exact points of intersection are not stored in the tree [Brière and Poulin 1996].

The error bounding algorithm *conservatively* determines interpolant validity for each possible discontinuity by testing the following conditions:

Geometry changes and shadows. If the ray trees of the sixteen extremal rays are not the same, the interpolant is not valid.

Occluders. The previous condition is necessary but not sufficient: it does not guarantee that *all* rays in the linetree cell have the same ray tree. There could be occluding objects between the sixteen extremal rays. These occluders are detected using a variant of shaft-culling [Haines and Wallace 1994; Teller et al. 1996].

Total internal reflections. A conservative test for TIR is to invalidate an interpolant if its extremal ray trees include an edge representing rays traveling between different media. However, this rule prevents interpolation whenever there is refraction, which is too conservative. The main problem is that the extremal rays do not indicate whether or not a query ray could undergo TIR. In the following section, we describe how linear interval arithmetic can be used to conservatively bound the angle θ between the normal and incident ray to a range $[\theta_-, \theta_+]$. If θ_c (the critical angle at which TIR occurs) is outside this range, the interpolant is valid: either all rays represented by the linetree cell undergo TIR or none do.

Reflections and Refractions. Discontinuities can arise recursively through reflections or refractions. These discontinuities are detected by conservatively computing reflected (or refracted) rays that bound the original reflected (or refracted) rays and then recursively testing for discontinuities against these new rays (for details see [Bala 1999]).

5.2 Non-linear radiance variations

Quadrilinear interpolation approximates radiance well in most regions of ray space that are free of discontinuities. However, quadrilinear interpolation is not sufficient where there are significant higher-order radiance terms; for example, at specular highlights and diffuse peaks. In this section, we show how to *conservatively* bound the deviation between interpolated radiance and base radiance for all rays represented by a linetree cell.

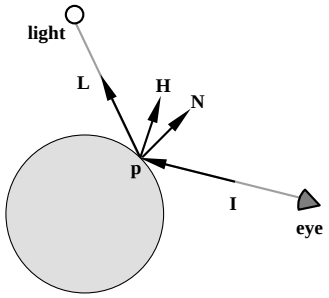

Fig. 10. Ray geometry.

5.2.1 *Shading model.* The total error bound for a linetree cell is computed using its associated ray trees. Since there are no radiance discontinuities in the linetree cell, the sixteen extremal ray trees of the cell are the same; that is, the sixteen rays hit the same objects, are illuminated by the same lights, and are blocked by the same occluders. The radiance associated with a ray tree node is computed as a local shading term plus a weighted sum of the radiance of its children. Therefore, error in the radiance of the ray tree node is bounded by the error in its local shading term plus the weighted sum of the error bounds for its children, which are computed recursively. We now discuss how to bound error in the local shading computation.

Error in the local shading term can arise from the approximation of both diffuse and specular radiance. Given an incident ray **I** (Figure 10) that intersects an object

at a point $\vec{p}$, the diffuse radiance for that ray is:

$$R_d = \rho_d(\mathbf{N} \cdot \mathbf{L}) \tag{1}$$

and the specular radiance, using the Ward isotropic shading model [Ward 1992], is:

$$R_s = \frac{\rho_s}{4\sigma^2} e^{\frac{1}{\sigma^2}(1-\frac{1}{(\mathbf{N}\cdot\mathbf{H})^2})} \sqrt{\frac{\mathbf{N}\cdot\mathbf{L}}{\mathbf{N}\cdot(-\mathbf{I})}} \tag{2}$$

where $\mathbf{N}$ is the normal to the surface at $\vec{p}$, $\mathbf{L}$ is the vector to the light source at $\vec{p}$, $\mathbf{I}$ is the incident ray direction, and $\mathbf{H}$ is the half-vector ($\mathbf{H} = \frac{\mathbf{L}-\mathbf{I}}{\|\mathbf{L}-\mathbf{I}\|}$) [Foley and van Dam 1982]. For an infinite light source, the light vector $\mathbf{L}$ is independent of $\vec{p}$ and is given as $\mathbf{L} = (l_x, l_y, l_z)$, where $l_x^2 + l_y^2 + l_z^2 = 1$. For a local light source, $\mathbf{L} = \frac{\mathbf{L}_p - \vec{p}}{\|\mathbf{L}_p - \vec{p}\|}$, where $\mathbf{L}_p$ is the position of the light source.

5.2.2 *Methodology.* We describe our methodology for computing error bounds by considering a one-dimensional function $f(x)$. A smooth function $f(x)$ can be approximated around the point x_0 by its Taylor expansion:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)\frac{(x-x_0)^2}{2!} + \ldots$$

According to Taylor's Theorem, when n terms of the Taylor series are used to approximate $f(x)$ over a domain $D = [x_0 - \Delta x, x_0 + \Delta x]$, the remainder R_{n+1} is the error that results from using this truncated approximation:

$$\forall_x \exists_{\xi \in D} \tilde{f}(x) = f(x_0) + f'(x_0)(x - x_0) + \ldots + f^n(x_0)\frac{(x-x_0)^n}{n!} + R_{n+1}(x)$$

$$R_{n+1}(x) = f^{n+1}(\xi)\frac{(x-x_0)^{n+1}}{(n+1)!}$$

where ξ is some point in the domain D for each x.

Consider a linear interpolation function $\tilde{f}(x)$ that approximates $f(x)$ by only its constant and linear terms ($n = 1$). Our goal is to bound the maximum difference between $f(x)$ and $\tilde{f}(x)$, as shown in Figure 11. By Taylor's Theorem, the maximal value of the quadratic remainder term $R_2(x)$, where x varies over the domain D, conservatively bounds the error due to this approximation. Assuming $x_0 = 0$ without loss of generality:

$$\forall_x \exists_{\xi \in D} f(x) = f(0) + xf'(0) + \frac{x^2}{2}f''(\xi)$$

$$f(x) - \tilde{f}(x) \leq \max_{x \in D, \xi \in D} \frac{x^2}{2}f''(\xi)$$

$$f(x) - \tilde{f}(x) \geq \min_{x \in D, \xi \in D} \frac{x^2}{2}f''(\xi)$$

There are several ways in which this error can be bounded, including the following: standard interval arithmetic [Moore 1979], Hansen's linear interval arithmetic [Hansen 1975] and variants [Tupper 1996], and affine arithmetic [Andrade et al. 1994]. Standard interval arithmetic can be used to bound the interpolation error by bounding the minimum and maximum value of $f(x)$ over the domain of

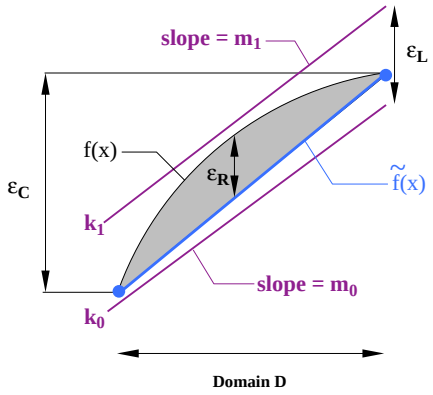

Fig. 11. Interpolation error. The gray curve is the base radiance function $f(x)$. The blue circles at the end-points of the domain D are the samples that are interpolated, and the blue line at the bottom of the curve is the interpolated radiance function $\tilde{f}(x)$. ϵ_R is the interpolation error, ϵ_C is the error computed by standard (constant) interval arithmetic, and ϵ_L is the error computed by linear interval arithmetic.

interpolation D. However, this error bound is typically too conservative because it does not take into account the ability of linear interpolation to approximate the linear component of the function f. Linear interval arithmetic, described in greater detail below, generalizes standard interval arithmetic by constructing linear functions that bound f over its domain D. Because the interpolating function $\tilde{f}(x)$ lies between these linear functions, the maximum difference between these bounding linear functions conservatively bounds the interpolation error $\max_{x \in D} |f(x) - \tilde{f}(x)|$. In Figure 11, the bounds computed by standard and linear interval arithmetic are ϵ_C and ϵ_L respectively, while the real error bound is ϵ_R. As can be seen, linear interval arithmetic can bound error more tightly than standard interval arithmetic, and is never worse.

In this section, we show how Hansen's linear interval arithmetic can be used to bound error for quadrilinear interpolation. We chose Hansen's technique over other variants of interval arithmetic because of its simplicity. In Hansen's linear interval arithmetic, the value of $f(x)$ is bounded at each x by the interval $F(x) = Mx + K$, where F is an interval-valued function and M, K are simple intervals $[m_0, m_1]$ and $[k_0, k_1]$. The addition of Mx and K is performed using interval arithmetic. This representation states that for any $x' \in D$, $f(x')$ lies in the interval $[m_0, m_1]x' + [k_0, k_1]$. The maximum linear interpolation error, $\epsilon_R = \max_{x \in D} |f(x) - \tilde{f}(x)|$, is bounded by the maximum interval computed by $F(x)$, which occurs at either the right-hand or left-hand side of the domain D. In general, the linear interval $F(x)$ represents four linear functions bounding radiance: line $y = m_0 x + k_0$, line $y = m_1 x + k_0$, line $y = m_0 x + k_1$, and line $y = m_1 x + k_1$. As x varies over its domain D, different pairs of these four lines bound $f(x)$. In Figure 11, we have shown a special case where $m_0 = m_1$; therefore, the two linear functions bounding radiance are: the line $y = m_0 x + k_0$ and the line $y = m_0 x + k_1$.

Extending this discussion to functions of several variables, linear intervals must be determined for each of the variables. For example, radiance is a function of

four variables (a, b, c, d), so the interval that bounds interpolation error has four linear intervals (M_a, M_b, M_c, M_d) and a single constant interval K. When radiance $R(a, b, c, d)$ is expressed in terms of its multi-variate Taylor expansion, linear intervals can be used to bound the error of a simple interpolation function $\tilde{R}(a, b, c, d)$ consisting of just the constant and linear terms of the Taylor expansion. Quadrilinear interpolation approximates radiance at least as well as $\tilde{R}(a, b, c, d)$. Therefore, a bound on the error of linear interpolation conservatively bounds the error of quadrilinear interpolation.

The shading computation performed by the ray tracer can be broken down into a series of simple operations, and constant and linear intervals can be propagated through each of these operations. For example, consider the product of two functions $f(\mathbf{x})$ and $g(\mathbf{x})$ where $\mathbf{x}$ is a four-tuple (x_a, x_b, x_c, x_d) and each of the variables x_i varies over the domain $[-\Delta x_i, \Delta x_i]$. The functions f and g are represented conservatively by interval functions $F(\mathbf{x}) = K_f + \mathbf{M}_f \cdot \mathbf{x}$ and $G(\mathbf{x}) = K_g + \mathbf{M}_g \cdot \mathbf{x}$, where both $\mathbf{M}_f$ and $\mathbf{M}_g$ are vectors of four intervals $\mathbf{M}_f = (M_{fa}, M_{fb}, M_{fc}, M_{fd})$ and $\mathbf{M}_g = (M_{ga}, M_{gb}, M_{gc}, M_{gd})$. The product $h(\mathbf{x}) = f(\mathbf{x})g(\mathbf{x})$ is represented by its terms K_h and $\mathbf{M}_h$. These terms are computed as follows [Hansen 1975]:

$$\begin{aligned} K_h &= K_f \cdot K_g + \sum_{i \in [a,d]} [0, \Delta x_i^2] M_{fi} M_{gi} \\ M_{hi} &= K_f \cdot M_{gi} + K_g \cdot M_{fi} + M_{fi} \sum_{j \neq i} [-\Delta x_j, \Delta x_j] \cdot M_{gj} \end{aligned}$$

Similar rules are derived for the other operations needed to compute shading: $\frac{1}{f(x)}$, $\sqrt{f(x)}$, and $e^{f(x)}$. For example, the reciprocal of a linear interval $h(x) = \frac{1}{f(x)}$ is computed as follows:

$$\begin{aligned} K_h &= 1/K_f \\ M_{hi} &= \frac{-M_{fi}}{K_f \cdot (K_f + \sum_j [-\Delta x_j, \Delta x_j] M_{fj})} \end{aligned}$$

For general functions such as e^x, linear intervals can be computed by considering the Taylor expansion of the function and bounding its second-order terms over the domain D. According to Taylor's Theorem,

$$\min_{\xi \in D, x \in D} \frac{x^2}{2} f''(\xi) \leq f(x) - \tilde{f}(x) \leq \max_{\xi \in D, x \in D} \frac{x^2}{2} f''(\xi)$$

Therefore, a conservative linear interval for a general function f is:

$$[f(0) + \min_{\xi \in D, x \in D} \frac{x^2}{2} f''(\xi), f(0) + \max_{\xi \in D, x \in D} \frac{x^2}{2} f''(\xi)] + [f'(0), f'(0)]x$$

For most functions, the minima and maxima in this equation are computed without difficulty. This approach also can be extended straightforwardly to the multivariate case. The mixed second-order terms that result can be folded into either the constant intervals, as above, or into the linear intervals, as in Hansen's formula for multiplication. Both approaches yield similar results for the radiance computation.

5.2.3 *Application to the shading computation.* Linear interval arithmetic can be used to derive error bounds for interpolated radiance. The first step is to compute

linear intervals for the various inputs used in the computation of radiance: the incident ray **I**, the light ray **L**, and the normal **N**. These intervals are propagated by evaluating diffuse and specular radiance, using the operations on linear interval arithmetic described above, to produce a linear interval for radiance. Relative or absolute interpolation error is then computed using this linear interval.

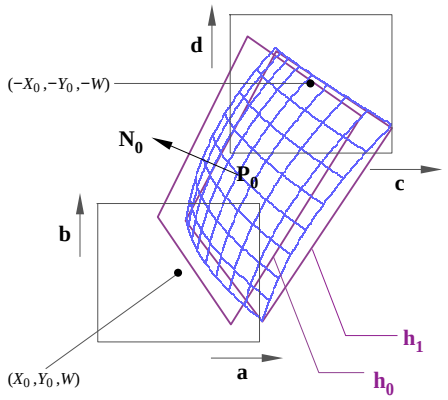

Fig. 12. A linetree cell for a surface patch.

Consider a linetree cell associated with a surface for which an interpolant is being constructed (shown in Figure 12). Without loss of generality, the principal direction of the linetree cell is $-\hat{z}$, and the origin is located in the center of the cell; in other words, the cell's front face is centered on (X_0, Y_0, W) and its back face is centered on $(-X_0, -Y_0, -W)$.

Incident ray **I**. Since the front and back faces of the linetree cell are at $z = W$ and $z = -W$ respectively, a ray parameterized by (a, b, c, d) represents a ray in 3D space from $(a+X_0, b+Y_0, W)$ to $(c-X_0, d-Y_0, -W)$. Therefore, the unnormalized incident ray is:

$$\mathbf{I} = (c - a - 2X_0, d - b - 2Y_0, -2W)$$

Each component of **I** is a simple linear function of the variables (a, b, c, d) and the corresponding linear intervals are computed trivially. For example, the linear interval representation for the x-component of **I** is $[-2X_0, -2X_0]+[-1, -1]a+[1, 1]c$. The incident ray is normalized by computing the linear interval for $\frac{1}{\sqrt{I_x^2+I_y^2+I_z^2}}$, where each of the operations — division, square, square root, and addition — are the linear interval operations defined in the previous section.

Light ray **L**. For an infinite light source, the light vector is $\mathbf{L} = (l_x, l_y, l_z)$, where $l_x^2 + l_y^2 + l_z^2 = 1$. The linear interval representation for the x-component of **L** is computed trivially as $[l_x, l_x]$. For a local light source, $\mathbf{L} = \frac{\mathbf{L}_p - \vec{p}}{\|\mathbf{L}_p - \vec{p}\|}$, where $\mathbf{L}_p$ is the position of the light source. The linear interval representation for **L** is computed from $\vec{p}$ using the interval operations described in the previous section. Now we explain how the linear interval representation for $\vec{p}$ is computed.

Point of intersection. The point of intersection $\vec{p}$ of the incident ray $\mathbf{I}$ with the surface lies on $\mathbf{I}$, and can be parameterized by its distance t from the front face:

$$\vec{p} = (X_0 + a, Y_0 + b, W) + t\ \mathbf{I} \tag{3}$$

A conservative linear interval for the components of $\vec{p}$ can be constructed in several ways with varying degrees of precision. First, we describe a simple way of conservatively bounding the intersection point for any convex surface. Consider the ray $\mathbf{R}_0$ from the middle of the linetree's front face to the middle of its back face. $\mathbf{R}_0$ intersects the surface at the point $\mathbf{P}_0$, and the normal at that point is $\mathbf{N}_0$ (see Figure 12). The plane h_0 tangent to the surface at $\mathbf{P}_0$ is defined by the equation:

$$\mathbf{N}_0 \cdot (x, y, z) - \mathbf{P}_0 \cdot \mathbf{N}_0 = 0$$

Another plane h_1 can be constructed parallel to h_0 that passes through the farthest point of intersection of any ray covered by the interpolant. Since the object is convex, the point of intersection of one of the sixteen extremal rays is guaranteed to be this farthest point. Let $\mathbf{P}_1$ be this farthest point of intersection of the sixteen extremal rays; that is, the point with the most negative projection on the normal. The equation of h_1 is:

$$\mathbf{N}_0 \cdot (x, y, z) - \mathbf{P}_1 \cdot \mathbf{N}_0 = 0$$

If the points of intersection of the incident ray $\mathbf{I}$ with planes h_0 and h_1 are at $t = t_{\text{near}}$ and $t = t_{\text{far}}$ respectively, then we have $t \in [t_{\text{near}}, t_{\text{far}}]$, where:

$$t_{\text{near}} = \frac{\mathbf{N}_0 \cdot (P_0 - (X_0 + a, Y_0 + b, W))}{\mathbf{N}_0 \cdot \mathbf{I}}$$
$$t_{\text{far}} = \frac{\mathbf{N}_0 \cdot (P_1 - (X_0 + a, Y_0 + b, W))}{\mathbf{N}_0 \cdot \mathbf{I}}$$

These equations for t_{near} and t_{far} are converted into a linear interval representation using the rules described in the previous section, and used to compute a linear interval for the parameter t:

$$t = t_{\text{near}} + [0, 1] \cdot (t_{\text{far}} - t_{\text{near}}) \tag{4}$$

The linear interval for $\vec{p}$ is computed by substituting Equation 4 into Equation 3.

There is another option for computing linear intervals for $\vec{p}$ when the point of intersection can be computed analytically. For example, consider a general quadric surface in three dimensions, defined by the following implicit equation:

$$Ax^2 + By^2 + Cz^2 + Dxy + Eyz + Fxz + Gx + Hy + Iz + J = 0 \tag{5}$$

Substituting $\vec{p} = (x, y, z) = (X_0 + a, Y_0 + b, W) + t\ \mathbf{I}$, we obtain a quadratic in t. The solution to the quadratic is converted to a linear interval using the Taylor series technique described earlier, in which the solution is expanded to second order and the second-order term is bounded.

Normal. The vector $\mathbf{N}$ normal to the surface at the intersection point is used in the computation of diffuse and specular radiance: it appears in the terms $\mathbf{N} \cdot \mathbf{I}$, $\mathbf{N} \cdot \mathbf{L}$, and $\mathbf{N} \cdot \mathbf{H}$. The surface normal at the intersection point varies across the linetree cell; it is a function of (a, b, c, d). There are various options for constructing

the linear intervals that conservatively approximate **N**. If little information about the surface within the linetree cell is available, the normal can be bounded using a constant interval bound. While this characterization is not precise, it will bound error conservatively. This approach to bounding the normal is similar to that taken in our earlier work [Teller et al. 1996], where the convexity of the surface allows information about the surface normals at the extremal intersection points to be used to bound the surface normal throughout the linetree cell.

For quadric surfaces and other surfaces that can be characterized analytically, tighter bounds can be obtained for the normal by taking the gradient of the implicit equation defining the surface and normalizing. For the quadric surface described by Equation 5, the unnormalized normal vector is:

$$(2Ax + Dy + Fz + G, 2By + Dx + Ez + H, 2Cz + Ey + Fx + I)$$

As described earlier, the point of intersection $\vec{p} = (x, y, z)$ can be expressed as a linear interval. This solution is substituted into the normal vector formula, allowing the normal vector to be bounded by linear intervals.

This approach can be extended to support spline patches. The surface intersection point can be bounded by the two-plane technique. Normals can be bounded to varying degrees of precision. A simple approach is to use constant intervals to bound the normals; a more accurate approach is to compute linear intervals for the surface parameters (u, v) using interval-based root finding [Hansen 1975]. The normal can then be computed using these parameters.

5.2.4 *Error refinement.* Figures 13 and 14 demonstrate the error refinement process for a specular sphere and a diffuse plane. The top row of each figure shows the sphere and plane rendered without testing for non-linearity; visible artifacts can be seen around the specular and diffuse highlights. The bottom row shows the sphere and plane rendered with non-linearity detection enabled and $\epsilon = 0.2$ and 0.1 respectively. The system automatically detects the need for refinement around highlights and refines interpolants that exceed the user-specified error bound. The image quality improves (bottom row) and the linetrees are increasingly subdivided, as shown in the linetree visualization on the right. Figure 15 shows a more complex scene containing many primitives (described in Section 8). Non-linearity detection substantially reduces error and eliminates visual artifacts (shown in the bottom row). The difference images in the right column show the error in interpolated radiance for the images on the left. Because this error is subtle, the difference images have been scaled by a factor of 4 for the purpose of visualization.

An interesting observation is that the error intervals can be used to subdivide linetree cells more effectively. The error bound computed for an interpolant specifies intervals for each of the four axes; the axes with larger intervals (that is, more error) are subdivided preferentially. This *error-driven subdivision* is discussed further in Section 7.

Figure 16 depicts actual and bounded error for one scanline of the image in Figure 14. This scanline passes through the diffuse peak of that image. The x-axis represents the pixels of the scanline, and the y-axis measures error in radiance. The image was generated with a user-specified error of 0.1, as shown by the horizontal dashed line in the figure. The blue trace is the actual error for each pixel in the

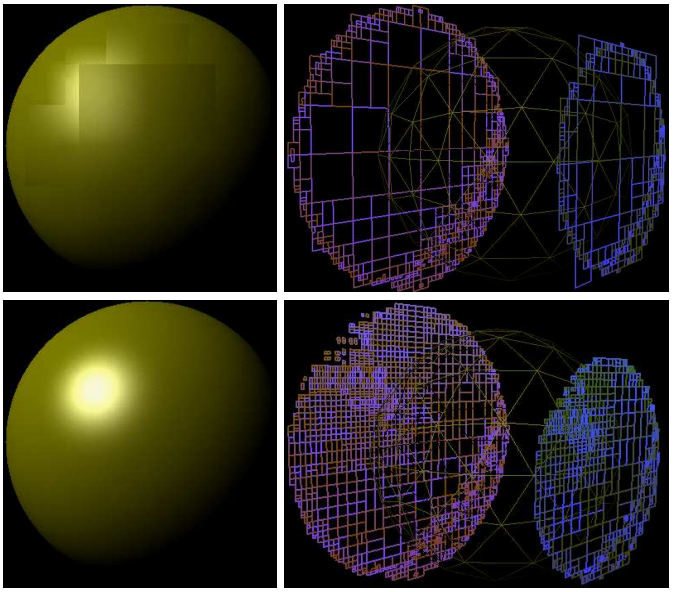

Fig. 13. **Error refinement for specular highlight**. A visualization of the front (blue) and back (pink) faces of the sphere's linetrees is shown on the right (the eye is off-screen to the right). Notice the error-driven adaptive subdivision along the silhouette and at the specular highlight. Top row: without non-linearity detection. Bottom row: with non-linearity detection and $\epsilon = 0.2$.

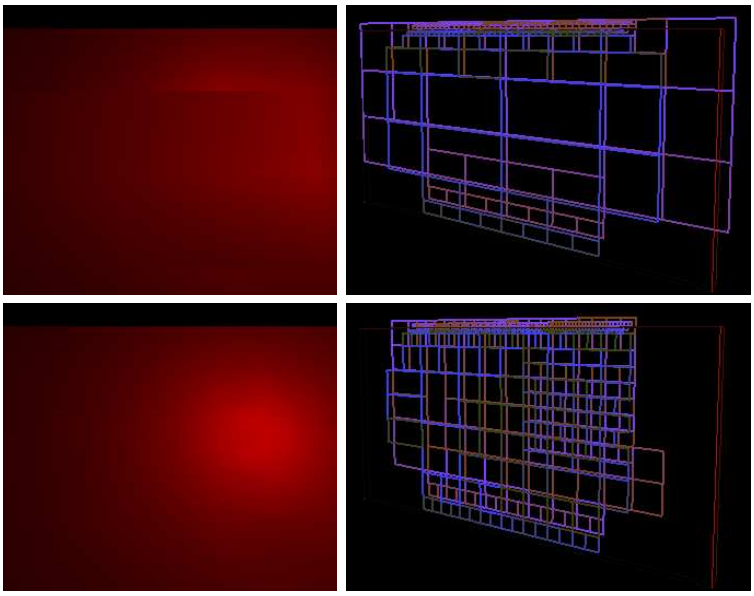

Fig. 14. **Error refinement for diffuse highlight**. A visualization of the front (blue) and back (pink) faces of the plane's linetrees is shown on the right (the eye is off-screen to the left). Notice the error-driven adaptive subdivision along the silhouette and at the diffuse highlight. Top row: without non-linearity detection. Bottom row: with non-linearity detection and $\epsilon = 0.1$.

Fig. 15. **Error refinement for museum scene**. Top row: without non-linearity detection. Bottom row: with non-linearity detection and $\epsilon = 0.5$. Right column: scaled difference images.

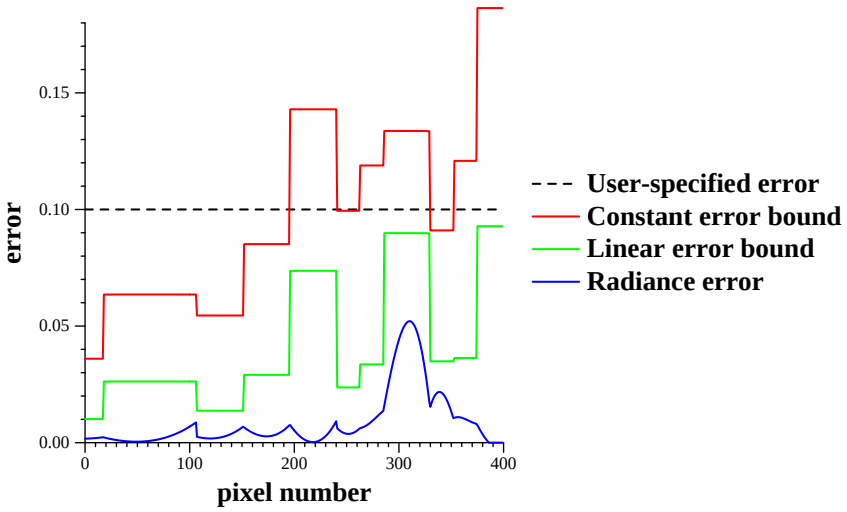

Fig. 16. Actual and conservative error bounds for one scanline of the image in Figure 14.

scanline, computed as the difference between interpolated and base radiance. The green trace is the error bound computed by linear interval analysis for the interpolants contributing to the corresponding pixels in the scanline. The red trace is the error bound computed by constant interval analysis. Since each linetree cell contributes to multiple consecutive pixels, both the linear and constant interval traces are piece-wise constant. Notice that both linear and constant interval analysis conservatively bound error for the pixels. However, linear interval analysis computes tighter error bounds than constant interval analysis.

Several factors make linear interval error bounds more conservative than necessary:

—The error bound is computed for the entire linetree cell, whereas any scanline is a single slice through the linetree cell and usually does not encounter the point of worst error in the interpolant.

—The bound is computed assuming simple linear interpolation, but the actual interpolation technique is quadrilinear interpolation, which interpolates with less error.

—Linear interval analysis is inherently conservative.

Several measures are taken to increase the precision of the linear interval analysis. One important technique is to treat the function $e^{\frac{1}{\sigma^2}(1-\frac{1}{x^2})}$ as a primitive operator. This function, used in the computation of radiance with $x = \mathbf{N} \cdot \mathbf{H}$, is well behaved even though its constituent operators (exponentiation and division) tend to amplify error. The linear interval computation rule for this function is determined using the second-order Taylor expansion technique described in Section 5.2.2. This approach yields a tighter error bound than simple composition of the linear interval rules for exponentiation and division does.

6. ACCELERATING VISIBILITY

Interpolants eliminate a significant fraction of the shading computations and their associated intersections; however, they do not reduce the number of intersections computed for determining visibility. Once interpolants accelerate shading, the cost of rendering a frame is dominated by the following three operations:

—Determining visibility at each pixel, i.e., constructing a ray from the eye through the pixel and intersecting that ray with the scene to find the closest visible object.

—For pixels that can be interpolated, computing the 4D intercepts for the ray and evaluating radiance by quadrilinear interpolation.

—For pixels that cannot be interpolated (because valid interpolants are unavailable), evaluating radiance using the base ray tracer.

In this section, we present techniques to accelerate visibility determination and interpolation by further exploiting temporal and image-space coherence.

6.1 Temporal coherence

For complex scenes, determining visibility at each pixel is expensive. However, for small changes in the viewpoint, objects that are visible in one frame are often visible in subsequent frames. This temporal coherence occurs because rays from the

new viewpoint through the image plane are similar to eye rays from the previous viewpoint. For the same reason, linetree cells that contribute to one frame typically contribute to subsequent frames. In this section, we present a reprojection algorithm that exploits this temporal coherence to accelerate visibility determination, while guaranteeing correctness.

If an object o is visible at some pixel, and the ray from the viewpoint through that pixel lies in a linetree cell with a valid interpolant, the radiance for the pixel is interpolated using the interpolant. In this case, the linetree cell is said to *cover* the pixel. A linetree cell typically covers a number of adjacent pixels on the image plane. Linetree cells that contribute to the previous frame are reprojected to the new viewpoint to determine which pixels in the new frame are covered by these cells. If a pixel in the new frame is covered by a cell, it is not necessary to compute visibility for that pixel: the radiance of the pixel can be computed directly from the interpolant associated with that cell. Since a cell typically covers multiple pixels, the cost of reprojecting the cell is amortized across many pixels. Reprojection also enables scan-line interpolation, further accelerating rendering (see Section 6.2).

The reprojection algorithm never incorrectly assigns a linetree to a pixel, i.e., it is conservative. To guarantee that the correct linetree cell is assigned to a pixel, cell faces are shaft-culled with respect to the current viewpoint. Reprojection accelerates visibility by replacing intersect operations for many pixels by a single shaft cull against the cell that covers those pixels. We will now discuss two important issues for the reprojection algorithm: how to reproject linetree cells efficiently, and how to use shaft-culling to guarantee that a cell is reprojected to a pixel only if it covers that pixel in the new frame; i.e., how to guarantee correctness.

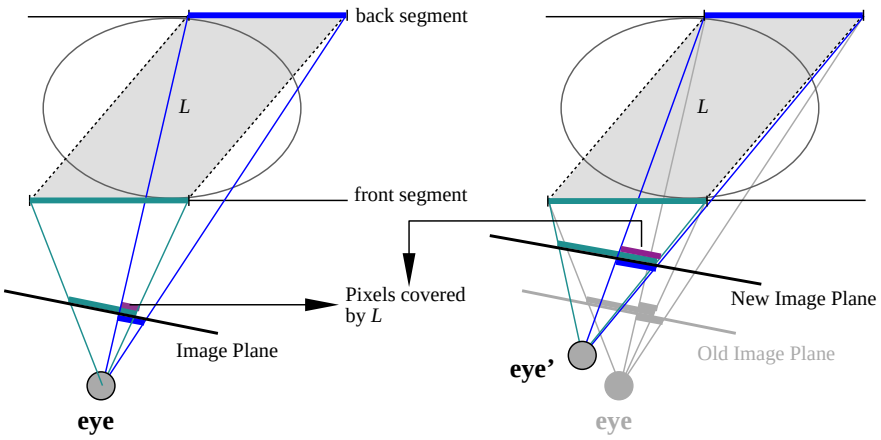

Fig. 17. A linetree cell with its front face (light blue) and back face (dark blue). The projections of the front and back faces on the image plane are shown as thick light and dark blue lines respectively. The linetree cell covers exactly those pixels onto which both its front and back face project (shown in purple). On the right, the viewpoint has changed from **eye** to **eye'**. Different pixels are covered by the cell in the new image plane (shown in purple).

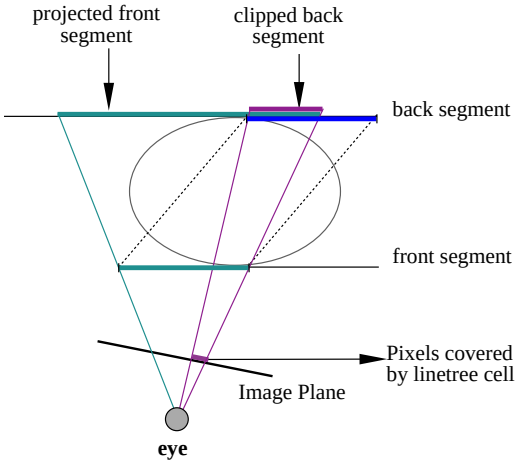

Fig. 18. Reprojection in 2D. The front segment is projected on and clipped against the back segment. The purple segment shows the pixels covered by the linetree cell.

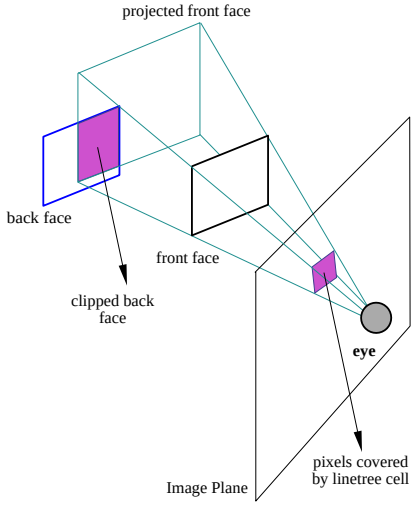

Fig. 19. Reprojection in 3D. The front face is projected on and clipped against the back face. The purple region shows the pixels covered by the linetree cell.

6.1.1 *Reprojecting linetrees.* First, let us consider reprojecting linetrees in 2D. In 2D, each linetree cell represents all the rays that enter its front line segment and leave its back line segment. For a given viewpoint, a cell covers some set of pixels; the rays from the viewpoint through these pixels are a subset of the rays represented by the linetree cell. Given a viewpoint, we will discuss how to efficiently find the subset of rays (and their corresponding pixels) covered by a linetree cell.

On the left in Figure 17, one linetree cell, L, of an object is shown; the front segment of L is shown in light blue and its back segment is shown in dark blue. In this frame the viewpoint is at the point **eye**. A ray from the viewpoint that lies in L must intersect both its front and back segment. Consider the projection of the front segment of L on the image plane with **eye** as the center of projection (shown as the thick light blue line on the image plane). Similarly, the pixels that the back segment projects onto are shown by the thick dark blue line on the image plane. The pixels covered by L are the pixels onto which *both* the front and back segment of L project; this is because a pixel is covered by L only if the eye ray through that pixel intersects the front *and* back segments of L. Therefore, the pixels covered by L are the intersection of the pixels covered by both the front and back segment (shown by the purple segment in the figure). On the right in Figure 17, the same cell is shown projected onto the image plane from a new viewpoint, **eye'**. Notice that L covers different (in fact, more) pixels in the new frame.

To determine which pixels are covered by L we should compute the intersection of pixels covered by its front and back segments. Using the viewpoint as the center of projection, the front segment of L is projected onto the line containing its back segment, and is then clipped against the back segment. When this clipped back segment is projected onto the image plane it covers exactly the same pixels that L covers. This is clear from the geometry in Figure 18.

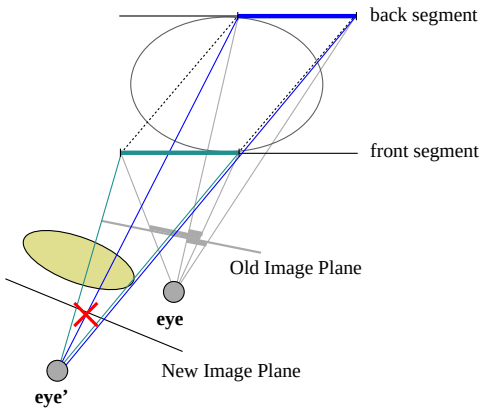

Fig. 20. Yellow ellipse occludes visibility to the linetree cell from the new viewpoint **eye'** but not from **eye**. Therefore, it would be incorrect to reproject the linetree cell to the new viewpoint.

We now extend this discussion to rays in 3D as depicted in Figure 19. Each linetree cell represents all the rays that enter its front face and leave its back face. To determine the pixels covered by a linetree cell L, the front face of L is projected onto the plane of the back face of L and clipped against the back face. This clipping operation is inexpensive because the projected front face and back face are both axis-aligned rectangles. The pixels covered by the projection of the clipped back face onto the image plane are exactly the pixels covered by L: in the figure, the purple shaded region on the image plane.

When a new frame is rendered, the system projects and clips the faces of all the linetree cells visible in the previous frame. To accelerate this process, we exploit the projection capabilities of standard polygon-rendering hardware. These clipped back faces are rendered from the new viewpoint using a unique color for each linetree cell; the color assigned to a pixel then identifies the reprojected linetree cell that covers it. The frame buffer is read back into a *reprojection buffer* which now stores the reprojected linetree cell, if any, for each pixel in the image. A pixel covered by a linetree cell is rendered using the interpolant of that cell; no intersection or shading computation is done for it.

6.1.2 *Reprojection Correctness.* Reprojection, as described in the previous section, is necessary but not sufficient to guarantee correct visibility determination at each pixel. Figure 20 illustrates a problem that can arise due to changes in visibility; an object that was not visible in the previous frame (the yellow ellipse) occludes reprojected linetrees in the current frame.

To guarantee that the correct visible surface is found for each pixel, we conservatively determine visibility by suppressing reprojection of linetree cells that might be occluded. This condition is detected by shaft-culling [Haines and Wallace 1994] each clipped back face against the current viewpoint. The shaft consists of five planes: four planes extend from the eye to each of the edges of the clipped back face, and the fifth plane is the plane of the back face of the linetree cell. If any object is found inside the shaft, the corresponding linetree cell is not reprojected onto the image plane. If no object lies in the shaft, reprojecting the linetree's clipped

back face correctly determines the visible surface for the reprojected pixels. Note that in other systems this visibility determination is complicated by non-planar silhouettes. They are not a problem in our system because while objects may have non-planar silhouettes, the error bounding algorithm guarantees that linetree cells with valid interpolants do not include these silhouettes.

Reprojection accelerates visibility computation by correctly assigning a linetree cell to each pixel. For pixels covered by reprojection, no visibility or shading operation is needed. Only one shaft-cull is required per linetree cell, which, for reasonably-sized linetree cells, is faster than intersecting a ray for each covered pixel. In Section 7, an additional optimization is described that amortizes the cost of shaft-culling over multiple linetree cells by clustering.

6.2 Image-space coherence

Using the reprojected linetree cells, a simple scan-line algorithm further accelerates the ray tracer. The reprojection buffer, which stores the reprojected linetree cell (if any) associated with a pixel, is checked when rendering the pixel. If a reprojected linetree cell is available, a span is identified for all subsequent pixels on that scanline with the same reprojected linetree cell. The radiance for each pixel in the span is then interpolated in screen space. Using screen-space interpolation eliminates almost all of the cost of ray-tracing the pixels in the span. No intersection or shading computations are required, and interpolation can be performed incrementally along the span in a tight loop. One effect of screen-space interpolation is that speedup is greater for higher resolution images because reprojected linetree cells extend over larger spans of pixels.

The current perspective projection matrix can be considered by the error bounding algorithm (see Section 5), yielding an additional *screen-space* error term. For linetree cells that are not close to the viewpoint, the additional error is negligible and can be ignored. In practice, we have not been able to observe any artifacts from screen-space interpolation. Unlike other systems that exploit image coherence [Amanatides and Fournier 1984; Guo 1998], our error bounding algorithm allows us to exploit image coherence while producing correct results.

Figure 21 shows the reprojection buffer and the pixels covered by reprojection for the museum scene. The color-coded image in the top right shows how image pixels are rendered: purple pixels are span-filled (fast path), blue-gray pixels are interpolated (interpolate path), green and yellow pixels are not accelerated (slow path). The pale lines in the span-filled regions mark the beginning and end of spans. Note that objects behind the sculpture are conservatively shaft-culled, resulting in a significant number of pixels around the sculpture not being reprojected. Better shaft-culling techniques would improve the reprojection rate.

7. OPTIMIZATIONS

In this section, we present some important performance optimizations and features of our system.

7.1 Error-driven subdivision

The error analysis presented in Section 5 suggests that uniformly subdividing a linetree cell along all four axes is too aggressive. A better approach is to use the

Fig. 21. **Museum scene**. Top row (left to right): reprojection buffer, color-coded image. Middle row: span-filled pixels, span-filled and interpolated pixels. Bottom: final image.

error bounding algorithm to determine the appropriate axes to split. We have implemented a four-way split algorithm that splits each linetree cell into four children, using the error bounding algorithm to guide subdivision. The split algorithm uses the sixteen ray trees associated with the interpolant to subdivide either the a and c axes or the b and d axes. This error-based adaptive subdivision results in fewer interpolants being built for the same number of successfully interpolated pixels. For the museum scene, building fewer interpolants using error-driven subdivision resulted in a 35% speedup. Currently our algorithm adaptively subdivides only across radiance discontinuities. However, this technique could also be applied to adaptive subdivision across regions with non-linear radiance variation.

7.2 Linetree depth

Our adaptive algorithm builds interpolants only if the benefits of interpolating radiance outweigh the cost of building interpolants. It achieves this by evaluating a simple cost model when deciding whether to subdivide a linetree cell. Linetree cells are subdivided on the basis of the number of screen pixels that they are estimated to cover. This estimate is computed using an algorithm similar to that used for reprojection (see Section 6). The front face of the linetree cell is projected onto its back plane, and clipped against its back face. The area of this clipped back face projected on the image plane estimates the number of pixels covered by the linetree cell. Thus, when the observer zooms in on an object, interpolants for that object are built to a greater resolution if required by the error bounding algorithm; if an observer is far away from an object, the interpolants are coarse, saving memory. This cost model ensures that the cost of building interpolants is amortized over several pixels. For the results in this paper, an interpolant is built only if it is expected to cover at least twelve pixels on the image plane. This setting delivers the best performance because building an interpolant is roughly twelve times more expensive than shooting a single ray (four of the sixteen samples associated with the interpolant have already been computed for its parent linetree cell and are reused).

7.3 Unique interpolant rays

The cost of building interpolants can be decreased by noticing that linetree cells share many rays. In the four-way split algorithm, each child of a linetree cell shares four rays with its parent. Similarly, siblings in the linetree share common rays; for example, two sibling linetree cells that have the same front face but different back faces share eight of the sixteen extremal rays. Therefore, when building interpolants, we use hash tables indexed by the ray's four parameters to guarantee that each ray is only shot once. This optimization reduces the total number of rays used to build interpolants by about 65%.

7.4 Unique ray trees

Storing radiance samples and their associated ray trees could result in substantial memory usage. However, all valid interpolants and a large number of invalid interpolants are associated with similar ray trees. Therefore, we use hash tables to avoid storing duplicate ray trees, resulting in substantial memory savings.

7.5 Textures

Our system supports textured objects by separating the texture coordinate computation from texture lookup. Texture coordinates and incoming radiance at the textured surface are both quadrilinearly interpolated. In general, multiple textures may contribute to the radiance of an interpolant. The texture coordinates of every contributing texture, and the appropriate weighting of each texture by the incoming radiance are recorded in the interpolant for each of the sixteen extremal rays and are separately interpolated. For reflected textures, an additional ray must be shot for each contributing texture from the point of reflection to compute the texture coordinates used in interpolation.

7.6 Adaptive shaft culling

The reprojection algorithm applies shaft culling to linetree cells from the previous frame. However, for large linetree cells that are only partially occluded, these shaft culls are too conservative. On the other hand, for small linetree cells it is wasteful to shaft-cull all the linetree cells of the object if the object is completely visible.

To address this problem, we cluster all the linetree cells of an object and shaft-cull the cluster. If the shaft cull succeeds, a significant performance improvement is achieved by eliminating the shaft cull for each individual cell. If the shaft cull fails, we recursively subdivide the cluster spatially along its two non-principal axes, and shaft-cull the four smaller clusters that result. This recursive subdivision is repeated until the cluster size is reduced to a single linetree cell. Then, using a cost model similar to that used for subdividing linetrees themselves, the contribution of this single cell to the image is evaluated. If the contribution of the cell is significant, it in turn is recursively subdivided and shaft-culled. The portions of the faces that are not blocked are then reprojected. For the museum scene, clustering decreases the number of shaft culls by about a factor of four.

8. PERFORMANCE RESULTS

This section evaluates the speed and memory usage of our system.

8.1 Base ray tracer

We compare our system to a base ray tracer that implements classical Whitted ray tracing with textures and uses the isotropic Ward local shading model. The base ray tracer is also used by our system for non-interpolated pixels.

To make the comparison between the ray tracers fair, optimizations were applied to both the base ray tracer and to our interpolant ray tracer when possible. There were a number of such optimizations. To speed up intersection computations, the ray tracers use kd-trees for spatial subdivision of the scene [Glassner 1989]. Marching rays through the kd-tree is accelerated by associating a quadtree with each face of the kd-tree cell. The quadtrees also cache the path taken by the most recent ray landing in that quadtree; this cache has a 99% hit rate. Therefore, marching a ray through the kd-tree structure is very fast. Also, shadow caches associated with objects accelerate shadow computations for shadowed objects. Other extensions such as adaptive shadow testing [Ward 1994] and Light Buffers [Haines and Greenberg 1986] might improve performance further.

8.2 Test scene

The data reported below was obtained for the museum scene shown in Figures 15, 21, and 22. In Figure 22, rendered images from the scene appear on the left, and on the right error-coded images show the regions of interpolation success and failure. In the error-coded images, interpolation success is indicated by a blue-gray color; other colors indicate various reasons why interpolation was not permitted. Green pixels correspond to interpolant invalidation due to radiance discontinuities such as shadows and occluders. Yellow pixels correspond to interpolants that are invalid because some sample rays missed the object. Pink pixels correspond to interpolant invalidation because of non-linear radiance variations.

The scene has more than 1100 convex primitives such as cubes, spheres, cylinders, cones, disks and polygons, and CSG union and intersection operations on these primitives. A coarse tesselation of the curved primitives requires more than 100k polygons, while more than 500k polygons are required to produce comparably accurate silhouettes. All timing results are reported for frames rendered at 1200×900 resolution. The camera translates and rotates incrementally from frame to frame in various directions. The rate of translation and rotation are set such that the user can cross the entire length of the room in 300 frames, and can rotate in place by 360° in 150 frames. These rates correspond to walking speed.

8.3 Rendering paths

There are three paths by which a pixel is assigned radiance:

(1) *Fast path*: reprojected data is available and used with the span-filling algorithm.
(2) *Interpolate path*: no reprojected data is available, but a valid interpolant exists. A single intersection is performed to find the appropriate linetree cell, and radiance is computed by quadrilinear interpolation.
(3) *Slow path*: no valid interpolant is available, so the cell is subdivided and interpolants are built if deemed cost-effective. If the built interpolant is invalid, the pixel is rendered by the base ray tracer.

8.4 Performance results

Table 1 shows the costs of each of the three rendering paths. The data for this table was obtained by using the cycle counter on a single-processor 194 MHz Reality Engine 2, with 512 MB of main memory.

In a 60-frame walk-through of the museum scene, about 75% of the pixels are rendered through the fast path, which is approximately thirty times faster than the base ray tracer for this scene. In this table, the entire cost of reprojecting pixels to the new frame is assigned to the fast path.

Pixels that are not reprojected but can be interpolated must incur the penalty of determining visibility. This interpolation path accounts for about 17% of the pixels. Quadrilinear interpolation is much faster than shading; as a result, the interpolation path is five times faster than the base ray tracer.

A pixel that is not reprojected or interpolated goes through the slow path, which subdivides a linetree, builds an interpolant, and shades the pixel. This path is approximately 40% slower than the base ray tracer. However, this path is only taken for 8% of the pixels and does not impose a significant penalty on overall

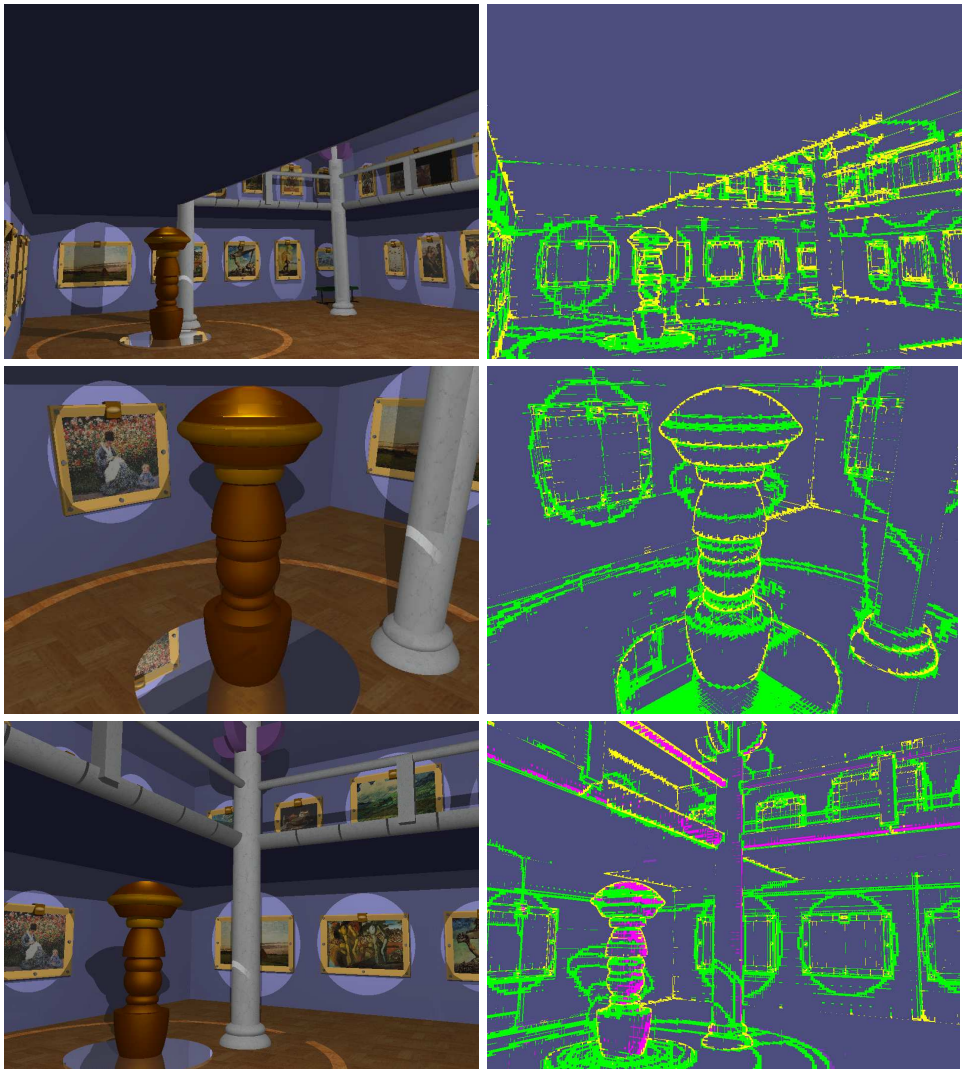

Fig. 22. **Museum scene**. Radiance is successfully interpolated for the blue-gray pixels. The error bounding algorithm invalidates interpolants for the green, yellow, and pink pixels. Non-linearity detection is enabled for the bottom row. Green pixels: occluders/shadows. Yellow pixels: silhouettes. Pink pixels: excessive non-linear radiance variation. Note the reflected textures in the base of the sculpture.

Path		Cost for path (μsecs)	Average fraction of pixels covered
Fast path	Span fill	1.9	75.2%
	Reproject	3.9	75.2%
	Total	**5.8**	**75.2%**
Interpolate path	Intersect object	24.1	16.88%
	Find linetree	4.5	16.88%
	Quad. interpolation	4.2	16.88%
	Total	**32.8**	**16.88%**
Slow path	Intersect object	24.1	7.92%
	Find linetree	4.5	7.92%
	Test subdivision	11.9	7.49%
	Build interpolant	645.5	0.43%
	Shade pixel (base)	160.6	7.92%
	Weighted total	**235.9**	**7.92%**
Interpolant ray tracer		28.5	100.0%
Base ray tracer		166.67	100.0%

Table 1. Average cost and fraction of pixels for each path over a 60 frame walk-through. The total time for the interpolant ray tracer, shown in the second to last row, is the weighted average of the time for each of the three paths. The last row reports the time taken by the base ray tracer.

performance. Much of the added cost results from building interpolants. As explained in Section 7, interpolants are adaptively subdivided only when necessary. On average, interpolants are built for only 0.4% of the pixels (about 5% of the pixels that fail); the cost model is very effective at preventing useless work. This cost could be further reduced in an off-line rendering application through judicious pre-processing.

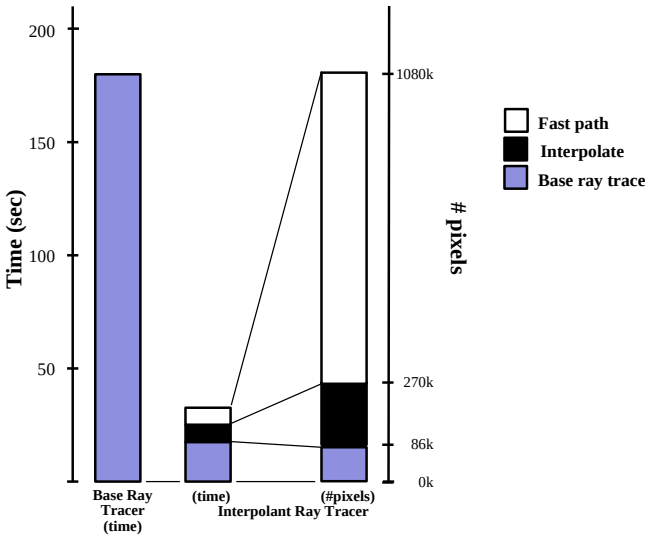

Fig. 23. Performance breakdown by rendering path and time.

In Figure 23, we compare the average performance of the interpolant ray tracer with that of the base ray tracer. The bar on the left shows the time taken by the base ray tracer, which for the museum scene is a nearly constant 180 seconds per frame. The middle bar shows the time taken by the interpolant ray tracer, classified by rendering paths. The bar on the right shows the number of pixels rendered by each path. Note that most of the pixels are rendered quickly by the fast path. Including the cost of reprojection, on average the fast path renders 75% of the pixels in 15% of the time to render the frame. Building interpolants accounts for 19% of the time, and the remaining 66% of the time is spent ray tracing pixels that could not be interpolated.

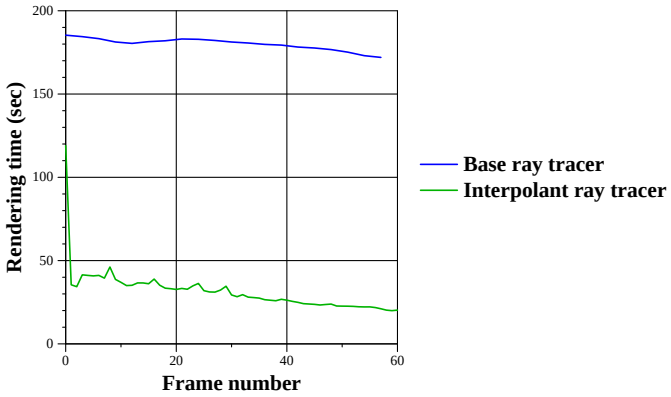

Fig. 24. Timing comparisons for 60 frames.

In Figure 24, the running time of the base ray tracer and interpolant ray tracer are plotted for each frame. After the first frame, the interpolant ray tracer is 4 to 8.5 times faster than the base ray tracer. Note that even for the first frame, with no pre-processing, the interpolant ray tracer exploits spatial coherence in the frame and is faster than the base ray tracer.

The time taken by the interpolant ray tracer depends on the scene and the amount of change in the user's viewpoint. Moving forward, for example, reuses interpolants very effectively, whereas moving backward introduces new non-reprojected pixels on the periphery of the image, for which the fast path is not taken. The museum walk-through included movements and rotations in various directions.

8.5 Memory usage and linetree cache management

One concern about a system that stores and reuses 4D samples is memory usage. The Light Field [Levoy and Hanrahan 1996] and Lumigraph [Gortler et al. 1996] make heavy use of compression algorithms to reduce memory usage. Our work differs in several important respects. Our system uses an on-line algorithm that adaptively subdivides linetrees only when the error bounding algorithm indicates that radiance does not vary smoothly in the enclosed region of ray space. Since we do not store much redundant radiance information, we do not expect our data to be as compressible. However, the memory requirements of our system are quite modest when compared to these 4D radiance systems. During the 60-frame walk-through,

the system allocates about 75 MB of memory. As the walk-through progresses, new memory is allocated at the rate of about 1MB per frame.

Since our system uses an on-line algorithm—it does no pre-processing, and interpolants are built lazily—the system memory usage can be bounded by a least-recently-used (LRU) cache management strategy that reuses memory for linetrees and interpolants. We have implemented a linetree cache management algorithm similar to the UNIX clock algorithm for page replacement [Tanenbaum 1987], though it manages memory at the granularity of linetree cells rather than at page granularity. The system allocates memory for linetrees and interpolants in large blocks. When the system memory usage exceeds some user-specified maximum block count, the cache management algorithm scans through entire blocks of memory at a time to evict any contained interpolants that have not been used recently. Each linetree cell has a counter that stores the last frame in which the cell was touched. If the linetree cell scanned for eviction is a leaf, and it has not been touched for n frames, where n is a user-specified *age* parameter, it is evicted. If all the children of a cell have been evicted, it too is evicted. Once the system recovers a sufficient amount of memory, normal execution resumes. Since scanning operates on coherent blocks of memory, the algorithm has excellent memory locality, which is important for fine-grained cache eviction strategies [Castro et al. 1997].

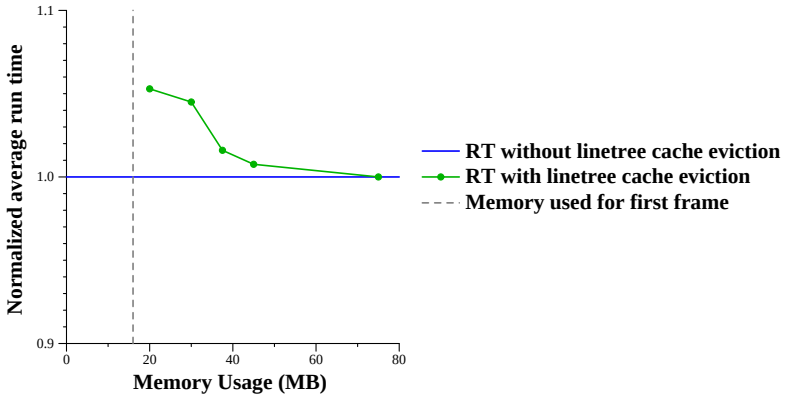

Fig. 25. Impact of linetree cache eviction on performance.

In Figure 25, we present the results of our LRU linetree cache management algorithm. The x-axis shows the user-specified maximum memory limit. The gray vertical dashed line shows the memory required by the first frame (17 MB). The y-axis shows the average run time of the interpolant ray tracer, normalized with respect to the average run time in the absence of memory usage restrictions (shown as a flat blue line). The green trace shows that the cache management algorithm is effective at preventing performance degradation when memory usage is restricted. Even when memory is restricted to 20 MB, the performance penalty is only 5%; at 45 MB, the penalty is only 0.75%. For long walk-throughs, the benefits of using cache management far outweigh this small loss in performance. Furthermore, it should be possible to hide the latency of the cache management algorithm by using idle CPU cycles when the user's viewpoint is not changing.

9. CONCLUSIONS AND FUTURE WORK

Our system exploits object-space, ray-space, image-space and temporal coherence to accelerate ray tracing. We have introduced 4D radiance interpolants that are quadrilinearly interpolated to approximate radiance. A fast, conservative reprojection algorithm accurately determines visibility as the user's viewpoint changes.

Our error bounding algorithm detects radiance discontinuities and prevents interpolation across them. We have introduced the use of linear interval arithmetic to compute conservative error bounds for radiance interpolation, thus guaranteeing the quality of the radiance approximation. The maximum error bound ϵ can be used to control performance-quality tradeoffs.

The combination of lazy interpolants and reprojection achieves significant performance improvements (4 to 8.5 times faster than the base ray tracer) for complex scenes and smoothly varying viewpoints. For the museum scene, 92% of the pixels are accelerated and only 8% of the pixels are rendered using the base ray tracer. An efficient cache management algorithm bounds memory usage, keeping the memory requirements of our system modest.

There are many opportunities for applying the new techniques described here. Ray tracing is often used to produce high-quality imagery in large off-line animations. Our techniques should be useful in accelerating both these pre-programmed animations and interactive walk-throughs. For an animation, the user could specify the camera path completely; for a walk-through, the user could indicate areas of the scene through which the viewpoint could move. Our system could use this information about camera position to guide the sampling of interpolants. The error bounding algorithm, which identifies regions of non-linear radiance variation, could also be used to guide intelligent super-sampling.

Another useful application of these techniques is interactive scene manipulation [Brière and Poulin 1996]. In such systems, the user interactively manipulates material properties, lights and scene geometry with ray-traced imagery. Such systems would be more useful if the user's viewpoint were permitted to change. Integrating radiance interpolants with scene manipulation permits both dynamic scene editing and dynamic viewing [Bala et al. 1999].

We would also like to extend the shading model of our ray tracer and its support for complex primitives; both of these would require extending the error bounding algorithm, though the interpolant-building mechanism described in this paper should be directly applicable. Our system samples the view-dependent component of radiance; we would like to extend it to support a more complete shading model, including diffuse inter-reflection, generalized BRDFs, and area light sources. We would also like to expand the support for textures to include more general texturing techniques such as bump maps and displacement maps. Information about perturbed surface positions and normals can be applied to the linear interval analysis equations in Section 5.2. Finally, our current technique assumes that object surfaces are convex. Relaxing this assumption to support polygon meshes, or parametric patches, would be useful. In Section 5.2, we have described how our linear interval analysis techniques could be extended to support spline patches.

ACKNOWLEDGMENTS

We would like to thank Andrew Myers for many helpful discussions and his fast hash table implementation. We would also like to thank Greg Ward Larson and the anonymous reviewers for their in-depth suggestions, and Michael Capps and Ulana Legedza for their comments on early drafts. This work was supported by an Alfred P. Sloan Research Fellowship (BR-3659), an NSF CISE Research Infrastructure award (EIA-9802220), an ONR MURI Award (SA-15242582386), and a grant from Intel Corporation.

REFERENCES

ADELSON, S. J. AND HODGES, L. F. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications 15*, 3 (May), 43–52.

AMANATIDES, J. 1984. Ray tracing with cones. In *Computer Graphics (SIGGRAPH 1984 Proceedings)* (July 1984), pp. 129–135.

AMANATIDES, J. AND FOURNIER, A. 1984. Ray casting using divide and conquer in screen space. In *Intl. Conf. on Engineering and Computer Graphics*. Beijing, China.

ANDRADE, A., COMBA, J., AND STOLFI, J. 1994. Affine arithmetic. In *Interval* (St. Petersburg, Russia, March 1994).

ARVO, J. AND KIRK, D. 1987. Fast ray tracing by ray classification. In *Computer Graphics (SIGGRAPH 1987 Proceedings)* (July 1987), pp. 196–205.

BADT, S., JR. 1988. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer 4*, 3 (Sept.), 123–132.

BALA, K. 1999. *Radiance Interpolants for Interactive Scene Editing and Ray Tracing*. Ph. D. thesis, Massachusetts Institute of Technology.

BALA, K., DORSEY, J., AND TELLER, S. 1998. Bounded-error interactive ray tracing. Technical Report Laboratory for Computer Science TR-748 (Aug.), Massachusetts Institute of Technology.

BALA, K., DORSEY, J., AND TELLER, S. 1999. Interactive ray-traced scene editing using ray segment trees. In *Tenth Eurographics Workshop on Rendering* (June 1999), pp. 39–52.

BRIÈRE, N. AND POULIN, P. 1996. Hierarchical view-dependent structures for interactive scene manipulation. In *Computer Graphics (SIGGRAPH 1996 Proceedings)* (August 1996), pp. 83–90.

CASTRO, M., ADYA, A., LISKOV, B., AND MYERS, A. C. 1997. HAC: Hybrid adaptive caching for distributed storage systems. In *Symposium on Operating Systems (SOSP) 1997* (Oct. 1997), pp. 102–115.

CHAPMAN, J., CALVERT, T. W., AND DILL, J. 1990. Exploiting temporal coherence in ray tracing. In *Proceedings of Graphics Interface 1990* (Toronto, Ontario, May 1990), pp. 196–204. Canadian Information Processing Society.

CHAPMAN, J., CALVERT, T. W., AND DILL, J. 1991. Spatio-temporal coherence in ray tracing. In *Proceedings of Graphics Interface 1991* (Calgary, Alberta, June 1991), pp. 101–108. Canadian Information Processing Society.

CHEN, S. E., RUSHMEIER, H. E., MILLER, G., AND TURNER, D. 1991. A progressive multi-pass method for global illumination. In *Computer Graphics (SIGGRAPH 1991 Proceedings)* (July 1991), pp. 165–74.

CHEVRIER, C. 1997. A view interpolation technique taking into account diffuse and specular inter-reflections. *The Visual Computer 13*, 7, 330–341.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1 (Jan.), 51–72.

CRAGG, T. 1998. *Pillars of salt*. Sculpture of Goodwood.

DIEFENBACH, P. AND BADLER, N. 1997. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (April 1997), pp. 59–70.

FOLEY, J. AND VAN DAM, A. 1982. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10, 15–22.

GLASSNER, A. S. 1989. *An Introduction to Ray Tracing.* Academic Press, London.

GLASSNER, A. S. 1995. *Principles of Digital Image Synthesis.* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), pp. 213–222.

GORTLER, S., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. 1996. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (Aug. 1996), pp. 43–54.

GUO, B. 1998. Progressive radiance evaluation using directional coherence maps. In *Computer Graphics (SIGGRAPH 1998 Proceedings)* (Aug. 1998), pp. 255–266.

HAINES, E. AND GREENBERG, D. 1986. The light buffer: A shadow-testing accelerator. *IEEE Computer Graphics & Applications 6*, 9 (Sept.), 6–16.

HAINES, E. A. AND WALLACE, J. R. 1994. Shaft culling for efficient ray-traced radiosity. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)* (New York, 1994). Springer-Verlag.

HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. 1991. A rapid hierarchical radiosity algorithm. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), pp. 197–206.

HANSEN, E. R. 1975. A generalized interval arithmetic. In *Interval Mathematics: Proceedings of the International Symposium* (Karlsruhe, West Germany, May 1975), pp. 7–18. Springer-Verlag. Also published in Lecture Notes in Computer Science Volume 29.

HECKBERT, P. S. AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Computer Graphics (SIGGRAPH 1984 Proceedings)* (July 1984), pp. 119–127.

LEVOY, M. AND HANRAHAN, P. 1996. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (Aug. 1996), pp. 31–42.

LISCHINSKI, D. AND RAPPOPORT, A. 1998. Image-based rendering for non-diffuse synthetic scenes. In *Rendering Techniques 1998* (June 1998), pp. 301–314.

LISCHINSKI, D., SMITS, B., AND GREENBERG, D. P. 1994. Bounds and error estimates for radiosity. In *Computer Graphics (SIGGRAPH '94 Proceedings)* (July 1994), pp. 67–74.

MARK, W., MCMILLAN, L., AND BISHOP, G. 1997. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (April 1997), pp. 7–16.

MOORE, R. E. 1979. *Methods and Applications of Interval Analysis.* Studies in Applied Mathematics (SIAM), Philadelphia.

NIMEROFF, J., DORSEY, J., AND RUSHMEIER, H. 1995. A framework for global illumination in animated environments. In *6th Annual Eurographics Workshop on Rendering* (June 1995), pp. 223–236.

PAINTER, J. AND SLOAN, K. 1989. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics (SIGGRAPH 1989 Proceedings)* (July 1989), pp. 281–288.

PIGHIN, F., LISCHINSKI, D., AND SALESIN, D. 1997. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In *Rendering Techniques 1997* (June 1997), pp. 115–126.

ROTH, S. D. 1982. Ray casting for modeling solids. *Computer Graphics and Image Processing 18*, 2 (Feb.).

SÉQUIN, C. H. AND SMYRL, E. K. 1989. Parameterized ray tracing. In *Computer Graphics (SIGGRAPH 1989 Proceedings)* (July 1989), pp. 307–314.

SILLION, F. AND PUECH, C. 1989. A general two-pass method integrating specular and diffuse reflection. In *Computer Graphics (SIGGRAPH 1989 Proceedings)* (July 1989), pp. 335–44.

SILLION, F. AND PUECH, C. 1994. *Radiosity and Global Illumination.* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

TANENBAUM, A. S. 1987. *Operating Systems: Design and Implementation.* Prentice-Hall, Inc.

TELLER, S., BALA, K., AND DORSEY, J. 1996. Conservative radiance interpolants for ray tracing. In *Seventh Eurographics Workshop on Rendering* (June 1996), pp. 258–269.

TELLER, S. AND HANRAHAN, P. 1993. Global visibility algorithms for illumination computations. *Computer Graphics (Proc. Siggraph '93)*, 239–246.

TUPPER, J. A. 1996. Graphing equations with generalized interval arithmetic. Master's thesis, University of Toronto.

WALLACE, J. R., COHEN, M. F., AND GREENBERG, D. P. 1987. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. In *Computer Graphics (SIGGRAPH 1987 Proceedings)* (July 1987), pp. 311–20.

WARD, G. AND HECKBERT, P. 1992. Irradiance gradients. In *Rendering in Computer Graphics (Proceedings of the Third Eurographics Workshop on Rendering)* (May 1992). Springer-Verlag.

WARD, G. J. 1992. Measuring and modeling anisotropic reflection. In *Computer Graphics (SIGGRAPH 1992 Proceedings)* (July 1992), pp. 265–272.

WARD, G. J. 1994. Adaptive shadow testing for ray tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)* (1994). Springer-Verlag.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. In *Computer Graphics (SIGGRAPH 1988 Proceedings)* (Aug. 1988), pp. 85–92.

WHITTED, T. 1980. An improved illumination model for shaded display. *CACM 23*, 6, 343–349.