MIT LCS TR-748

Bounded-Error
Interactive Ray Tracing

Kavita Bala Julie Dorsey Seth Teller

MIT Computer Graphics Group

March 1, 1998

545 TECHNOLOGY SQUARE CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

Bounded-Error Interactive Ray Tracing

KAVITA BALA JULIE DORSEY SETH TELLER
MIT Graphics Group*

Abstract

Ray tracing, which computes radiance, is traditionally regarded as an off-line rendering algorithm that is too slow for interactive use. In this paper, we present an interactive system that uses *4D interpolants* to approximate radiance, while providing *guaranteed error bounds*. Our system exploits the object-space, ray-space, image-space and temporal coherence in radiance to accelerate ray tracing.

Our system explicitly decouples the two operations of the ray tracer — shading computation and visibility determination at each pixel, which we call pixel assignment. Rendering is accelerated by approximating the shading computation while guaranteeing correct pixel assignment. Without any pre-processing, the system lazily collects 4D radiance samples, which are quadrilinearly interpolated to approximate radiance. An error predicate conservatively guarantees that the relative error introduced by interpolation is bound by a user-specified ϵ. The user can change this parameter to trade off performance for quality. The predicate guarantees the error bound by detecting discontinuities and non-linearities in radiance. If error cannot be guaranteed for an interpolant, the system adaptively refines the interpolant.

Pixel assignment is accelerated using a novel algorithm that *reprojects* 4D interpolants to new viewpoints as the user's viewpoint changes. Using reprojection, a fast scan-line algorithm achieves high performance without sacrificing image quality. This lazy interpolant system with reprojection speeds up ray tracers substantially for smoothly varying viewpoints. In this paper, we present the design, implementation and results for our system. We expect our techniques to be useful for interactively rendering general scenes as well as for batch off-line rendering processes.

Keywords: Ray tracing, rendering, error bounds, 4D interpolation, interactive, visibility

*{kaybee,dorsey,seth}@graphics.lcs.mit.edu. This research was funded in part by a grant from Intel Corporation, and MURI award N00014-96-1-1200 sponsored by the Office of Naval Research.

1 Introduction

One primary goal of computer graphics has been the generation of fast, accurate, high quality imagery. Global illumination algorithms generate high quality images by evaluating radiance, a five-dimensional function. To achieve reasonable performance, illumination systems trade freedom of viewer motion for both scene complexity and accuracy in different ways. On one end of the spectrum, ray tracers [39] compute high quality view-dependent images, while compromising on interactivity. Ray tracers typically support specular and diffuse reflectance functions, and generalized primitives, but at the expense of viewing the output image from a fixed viewpoint. At the other end of the spectrum, radiosity algorithms[16] trade scene complexity for interactivity. Radiosity algorithms support interactive viewing, but typically render only diffuse, polygonal environments and require a pre-processing phase. Other hybrid systems [9, 32, 33, 35, 38] have tried to bridge the gap between these two extremes. However, the view-dependent component of radiance has been extremely expensive to compute, and ray tracing image quality is traditionally associated with off-line rendering algorithms that are too slow for interactive use.

In this paper, we present an interactive system that uses interpolants to accelerate classical Whitted ray tracing [39]. The goal of our system is to provide an interactive ray tracing environment in which the user moves around freely as the system renders ray-trace quality images. To achieve this goal, we decouple and independently accelerate the two operations of a ray tracer: the visibility computation that assigns the closest visible object to each pixel (we call this *pixel assignment*), and the radiance computation for that intersection point. A standard ray tracer uses intersection computations for both these operations, which dominate performance [13].

In designing and building an interactive ray-tracing environment we have made several contributions:

- *Radiance approximation using 4D interpolants:* Our system builds interpolants lazily to approximate the radiance computed by a Whitted ray tracer. An *interpolant* is a set of samples used to approximate radiance, and is stored in 4D trees called *linetrees*. As the viewpoint changes, radiance is approximated by *quadrilinearly* interpolating the per-surface interpolants. Interpolants are subdivided adaptively, thereby permitting greater interpolant reuse where radiance varies smoothly, and sampling at a higher frequency where radiance changes rapidly.

- *Guaranteed error bounds:* For every pixel, we guarantee that the relative error between the interpolated radiance and Whitted radiance is bounded by a user-specified error ϵ. Interpolation error arises from both discontinuities and non-linearities in the radiance function. The *error predicate* prevents interpolation over discontinuities and non-linear variations in radiance. If the predicate is false, more detailed interpolants are built. The user can use ϵ to trade off performance for quality.

- *Accurate pixel assignment using linetree reprojection:* Pixel assignment is accelerated using a novel conservative algorithm for reprojection of 4D linetree cells. This algorithm exploits the temporal frame-to-frame coherence in the user's viewpoint, while guaranteeing accurate pixel assignment. A fast scan-line algorithm uses the reprojected linetrees to accelerate rendering.

Our techniques can be integrated with several other approaches to produce faster and better rendering systems: Interpolants could be built in an off-line rendering phase and re-used in an on-line phase to accelerate walkthroughs. They can also be used to accelerate generation of light fields [24] or Lumigraphs [17]. The error predicate, which identifies regions of high radiance variation, could be used to guide intelligent super-sampling.

The rest of the paper is organized as follows: In Section 2 related work is presented, with a discussion of how our system is different. In Section 3, an overview of the system is presented. In Section 4, the interpolant building and storing mechanisms are described in greater detail. We derive analytical expressions for error bounds, and present the error predicate in Section 5. In Section 6 we present the reprojection algorithm, and discuss how it is used. In Section 7, we discuss the system implementation and other optimizations to improve performance, and present performance results in Section 8. Finally, we present future work and conclude in Section 9.

2 Related Work

Several researchers have focused on the problem of improving the performance of global illumination algorithms [13, 33]. For ray tracing, several effective techniques have been developed: adaptive 3D spatial hierarchies [15], beam-tracing for polyhedral scenes [22], cone-tracing [3], and ray classification [4]. Lack of space prevents us from referring to all the work that has been done in this field, but a good summary of these algorithms can be found in [13, 14].

2.1 Acceleration of radiance computation

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

Teller et al. present an algorithm to speed up shading of a Whitted ray tracer (without texturing) by building 4D interpolants that are reused to satisfy radiance queries [34]. While their algorithm identifies radiance discontinuities, it does not correctly deal with radiance non-linearities, nor does it accelerate pixel assignment. Radiance [38] uses ray tracing and computes diffuse inter-reflection lazily, producing very high quality images by sparsely and non-uniformly sampling the slowly-varying diffuse-interreflections [38]. Radiance also includes an accuracy threshold to guide sampling; gradient information is used to guide sampling density [36]. A recent multi-pass rendering system [11] uses multiple passes of the standard graphics hardware to obtain images of higher quality than that ordinarily available using graphics hardware. The drawback of this system is that it approximates pixel assignment by resorting to discretizing the scene into polygons, and has no correctness guarantees.

Image-based rendering (IBR) systems, such as the light field [24] and the Lumigraph [17], have similarities to our system in that they build 4D radiance information that is quadrilinearly interpolated to approximate radiance. However, IBR systems typically have a data acquisition pre-processing phase, and are not intended for interactive rendering. Light fields and Lumigraphs are uniformly subdivided 4D arrays with fixed resolution determined in the pre-processing phase. This fixed sampling rate does not guarantee that enough samples are collected in regions that have high-frequency radiance changes. Additionally, these systems do not compute any bounds on error introduced by approximating radiance or pixel assignment, and they constrain the viewer to lie outside the convex hull of the scene.

Nimeroff et al. use IBR techniques to warp pre-rendered images in animated environments with moving viewpoints [27], but do not guarantee error.

2.2 Algorithms that approximate pixel assignment

Algorithms that exploit temporal coherence to approximate pixel assignment can be categorized by the assumptions they make about the scene, and the correctness guarantees they provide. Chapman et al. [7, 8] restrict the model to be polygonal, and use the trajectory of the viewpoint through the scene to compute continuous intersection information of rays with the polygons. Badt [5], Adelson and Hodges [1], and Mark et al.[26] reuse pixels from the previous frame to render pixels in the current frame. Adelson and Hodges apply a 3D warp to pixels from reference images to the current image. Each pixel receives the radiance that would be computed at some reprojected point guaranteed to lie inside the pixel (though not necessarily at the center). This algorithm speeds up the first level rays from the eye to the screen but does not accelerate shading computation. Mark et al. [26] also apply a 3D warp to pixels, but they treat their reference image as a mesh and warp the mesh triangles to the current image. Their system does not currently deal with occlusion due to multiple objects, and depends on the eye lying on a linear path between the two reference images. Additionally, all of the above algorithms suffer from aliasing effects that arise from the fact that the pixels are not warped to pixel centers in the current frame, i.e. they do not accurately determine pixel assignment.

2.3 Discussion

We differ from previous rendering systems in several respects: we do no pre-processing; instead, interpolants are built lazily and adaptively. Using conservative analytical error bounds, we bound interpolation error, and allow speed and quality to be traded off during rendering. We guarantee correct pixel assignment while exploiting temporal coherence by reprojecting 4D linetree cells. There are no pixel aliasing artifacts because radiance is sampled at the center of each pixel.

3 System Overview

In this section, we present an overview of our interactive rendering system. Figure 1 shows a system block diagram. Our ray tracer implements a classical Whitted ray tracing model with texturing [39]. For each object surface, there is an associated collection of interpolants representing the radiance from that surface. These interpolants are stored in 4D linetrees, which allow an efficient lookup of an interpolant for any particular ray. When the system detects a change in the user's viewpoint, it starts rendering a new frame. If possible, linetrees from the previous frame are reprojected. The system tries to accelerate rendering every pixel by using the reprojected linetrees, or by looking up linetrees for interpolants. If both of these fail, it falls back to rendering the pixel using the base ray tracer.

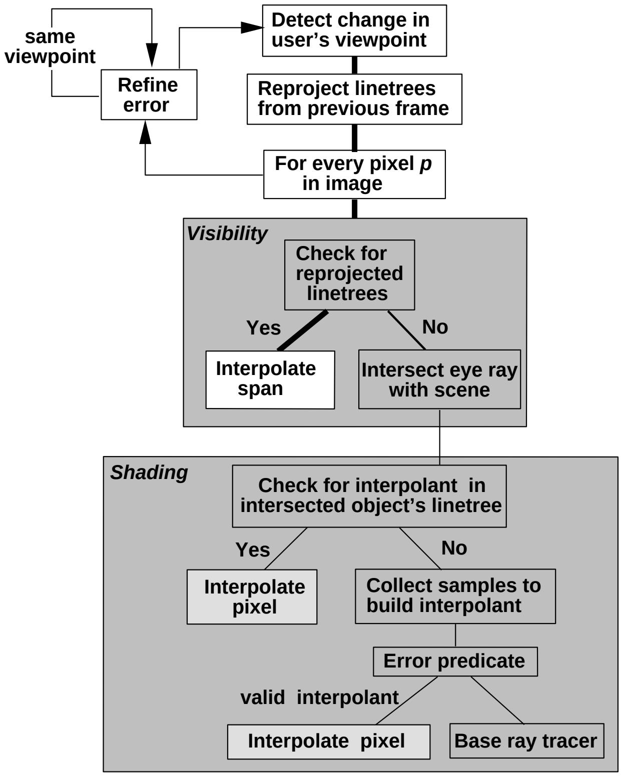

Figure 1: System Overview

For every pixel in the viewport, the system checks if there is a linetree available in the reprojection buffer. If so, radiance is interpolated for all consecutive pixels in the scanline with the same linetree, using screen-space interpolation. This is the fast path indicated by bold lines in Figure 1. This path is about 45 times faster than the base ray tracer (Refer to Section 8 for details).

If no reprojected linetree is available, pixel assignment is determined by the slower path of intersecting the eye ray with objects in the scene. Once the object visible at a pixel is determined, a check determines the availability of a valid interpolant for that pixel. If an interpolant is available, the radiance for the pixel is computed by quadrilinear interpolation. Otherwise, an attempt is made to build an interpolant by collecting radiance samples for that leaf linetree cell. An *error predicate* checks the validity of the new interpolant. If the interpolant is valid, the pixel's radiance can be computed from the new interpolant. If it is not valid, the linetree cell is subdivided, and the ray tracer falls back to shading the pixel using the base ray tracer. In the case where an interpolant *can* be used to shade the pixel the ray tracer is accelerated since no shading operation needs to be invoked for the pixel.

The user's viewpoint is tracked, and when it is stationary, error is refined to produce images of higher quality matching the user-specified error bound. When the user's viewpoint changes, the system renders the scene at the highest speed possible, while guaranteeing that radiance discontinuities are not interpolated over.

4 Radiance Interpolants

In this section, we briefly summarize the mechanisms to build and use interpolants (Refer to [34] for more details). First, we present a parameterization of rays that captures all the rays that intersect some volume of 3D space, then the linetree data structure and the interpolant building mechanism are presented.

4.1 Ray parameterization

Every ray intersecting an object o, can be parameterized by the ray's four intercepts $\langle a, b, c, d\rangle$ with two parallel faces surrounding o [34, 24, 17], as shown in Figure 2. To cover the entire space of rays that could intersect o, six pairs of parallel faces (called facepairs) surrounding the object are considered. Each facepair is built by expanding the corresponding faces of the object's bounding box appropriately. The dominant direction (and sign) of the ray determine which of the 6 facepairs to consider for looking up interpolants.

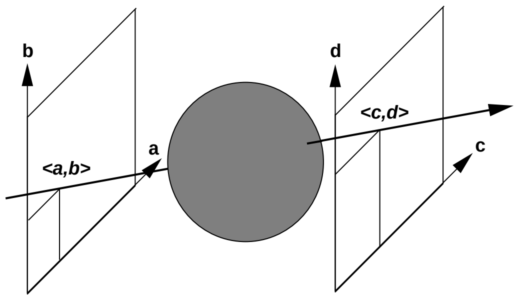

Figure 2: Ray parameterization

4.2 Linetrees

A linetree is a 4D tree that represents all the rays that pass through the 3D volume defined by the linetree's front face and back face. Each leaf cell in a linetree corresponds to a 4D hypercube in the 4D space of rays. The sixteen vertices of the 4D hypercube correspond to the sixteen extremal rays spanning the linetree leaf in 3D. These rays are the sixteen rays from the four corners of the leaf's front face to each of the four corners of its back face (Figure 3 shows eight of these sixteen rays). The 4D hypercube captures a region of ray space for which interpolants are built. Every ray that intersects the front and back face of the linetree in 3D, lies inside the corresponding hypercube in 4D. Given the 4D intercepts of a ray, $\langle a, b, c, d\rangle$, the linetree leaf cell that contains that ray can be found by walking down the tree and performing four interval tests, one for each of the ray coordinates.

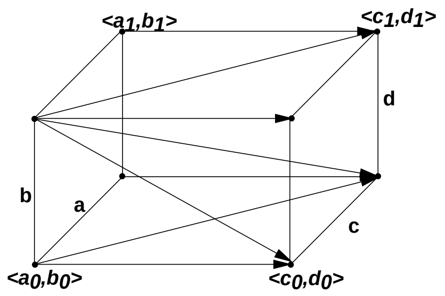

Figure 3: Linetree Cell

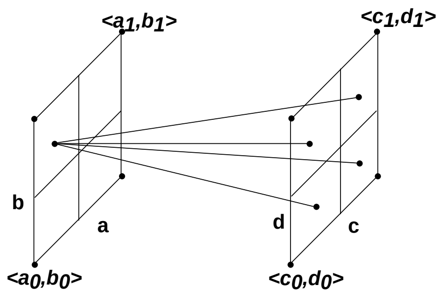

Figure 4: Linetree subdivision

4.3 Building and using interpolants

When an eye ray intersects an object, its four intercepts $\langle a, b, c, d\,\rangle$ are computed, and the linetree leaf cell containing the intercepts is found. If the leaf does not contain a valid interpolant, the system tries to build an interpolant for that leaf cell. Radiance is computed along the sixteen extremal rays from the leaf's front face to its back face. Additional information about the ray trees along each of the sixteen paths is maintained, as in [34, 13]. Using the ray trees, and the computed radiance values, the error predicate determines if the samples collected represent a valid interpolant. If so, the radiance for that pixel is quadrilinearly interpolated from the interpolant.

If the interpolant is not valid, the front and back face of the linetree cell are subdivided, creating sixteen children of the linetree cell (Four of the sixteen children are shown in Figure 4). Interpolants are lazily built for the child that contains the eye ray. Thus, linetrees are adaptively subdivided only when necessary. Building interpolants adaptively alleviates the memory problem of representing 4D radiance, by using memory only where necessary. More samples are collected in regions of detail, accurately representing radiance. Fewer samples are collected in less detailed regions, saving computation and storage costs.

Plate 1 and our demo demonstrate the linetree data structure and interpolant building mechanism. In Plate 1, Figure 1 is a snapshot of the interpolant building process, and the right hand images in Figure 2 are snapshots of all the linetrees built lazily for the sphere in the first and second rendering frames respectively.

5 Error Predicate

Rendering systems trade accuracy against speed by using error estimates to determine where computation and memory resources should be expended. Radiosity systems have developed explicit error bounds [20, 25] to make this trade-off. Ray tracing computations typically use super-sampling and stochastic techniques [28, 10] to estimate and decrease the error in computed radiance. In this paper, we present analytical error bounds for interpolation error.

Given sixteen radiance values and ray trees for the extremal rays of a linetree leaf cell, the error predicate guarantees that the interpolants produce interpolated radiance values within ϵ of Whitted radiance for *every* ray that lies in that linetree leaf cell. Incorrect interpolation arises in two possible ways:

- Interpolation over a discontinuity in the radiance function (due to changes in scene geometry, shadows, or occluding objects)
- Quadrilinear interpolation in regions with higher order radiance terms (due to diffuse or specular peaks)

In this section, we discuss results from [34], and then focus on our new contributions, presenting a complete solution for the error predicate.

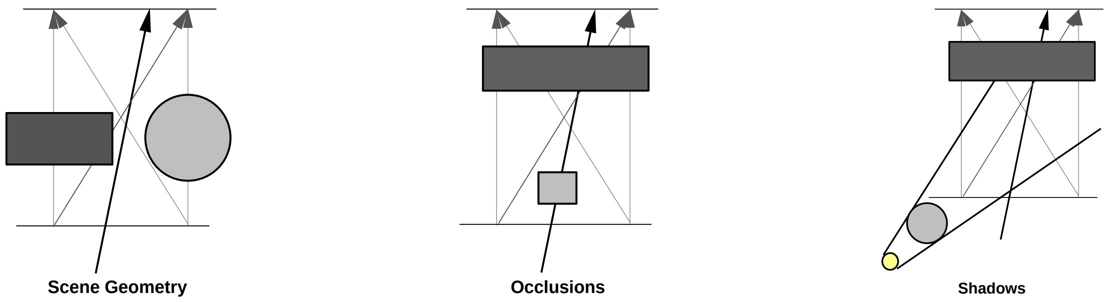

Figure 5: Radiance discontinuities — interpolation for the black ray would be erroneous.

5.1 Radiance discontinuities

An interpolant is invalid if it interpolates radiance over a radiance discontinuity. Radiance discontinuities arise because the scene is composed of multiple geometrical objects that occlude each other and cast shadows on each other. Figures 5, and 6 show invalid interpolation due to radiance discontinuities in 2D. Gray rays are the four extremal interpolant rays for the 2D linetree, and the black ray corresponds to some arbitrary ray that lies in the convex hull of the extremal rays. In Figure 5, interpolation would be incorrect because an internal ray (black) does not necessarily hit the same objects hit by the extremal rays. In Figure 5 on the left and middle, the extremal (gray) rays all hit the same object but interpolation would be incorrect because an internal ray could hit an occluding object; while on the right, interpolation would not capture the shadow cast on the rectangle by the sphere. In Figure 6, discontinuities that arise due to total internal reflection (TIR) are shown. On the left, the interpolant rays lie in the TIR cone but the internal ray does not; on the right the interpolant rays lie outside the TIR cone while the internal ray lies in the cone. In both cases, interpolation would be incorrect (indicated by the cross).

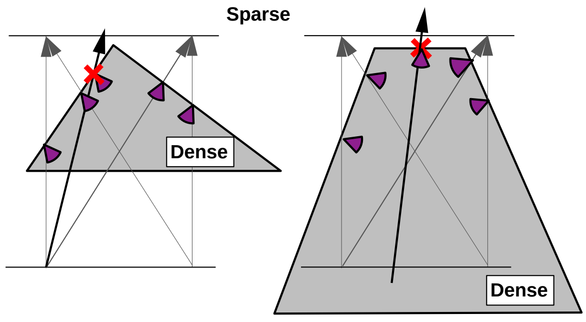

Figure 6: Erroneous interpolation due to total internal reflection.

Ray trees

Ray trees [31, 34] keep track of all objects, lights and occluders that contribute to the radiance of a particular ray, both directly and indirectly through reflections and refractions. The root of a ray tree stores the object intersected by the eye ray (and its face, if appropriate), and the lights and occluders visible at the point of intersection. The children of the root store the ray trees associated with the corresponding reflected and refracted rays. Two ray trees are *isomorphic* and said to have the same topology, if there exists a one-to-one correspondence between their nodes. A crucial observation is that, *radiance changes discontinuously only when the topology of the associated ray trees changes*.

Detecting discontinuities

To guarantee that interpolants do not erroneously interpolate over a radiance discontinuity, *all* rays in the 4D hypercube represented by the linetree cell should have the same ray tree topology. The error predicate conservatively determines validity for each possible discontinuity in the following way:

Geometry changes and Shadows: If the ray trees of the sixteen extremal rays are not isomorphic, the interpolant is not valid.

Occluders: Occluders are detected using a variant of shaft-culling [19, 34].

Total Internal Reflections If a ray tree includes an edge representing rays traveling from a dense to sparse medium the error predicate reports that the interpolant is invalid.

5.2 Radiance non-linearity

Quadrilinear interpolation approximates radiance well in most regions of ray space except where there are significant second-order radiance terms, which are created by specular highlights and diffuse peaks. In this section, we *conservatively* bound the deviation between radiance computed by a Whitted ray tracer and interpolated radiance for *all the rays* represented by a linetree. We derive analytical expressions for the diffuse and specular components of radiance in terms of the linetree parameters (a, b, c, d), and the object's surface parameters. The constant and linear terms of radiance are approximated well by quadrilinear interpolation, therefore, these terms are not considered in the error computation. Instead, we focus on the contribution of the quadratic term (cubic and higher terms are dominated by the quadratic term [30]).

The total error bound for a linetree cell is computed by recursively bounding the error at each node of the ray tree and aggregating these node errors, appropriately weighted by the node's weight. Error at each ray tree node can arise from the approximation of both diffuse and specular radiance. Given an incident ray that intersects an object at a point $\vec{p}_{xn}$, the diffuse radiance for that ray is $D = \mathbf{N} \cdot \mathbf{L}$, and the specular radiance is $S = (\mathbf{N} \cdot \frac{-\mathbf{I}+\mathbf{L}}{2})^n$, where, $\mathbf{N}$ is the normal to the surface at $\vec{p}_{xn}$, and $\mathbf{L}$ is the vector to the light source at $\vec{p}_{xn}$. For an infinite light source, the light vector $\mathbf{L}$ is independent of $\vec{p}_{xn}$ and is given as $\mathbf{L} = (l_1, l_2, l_3)$, where $l_1^2 + l_2^2 + l_3^2 = 1$. For a local light source, $\mathbf{L} = \frac{(L_1,L_2,L_3)-\vec{p}_{xn}}{||\mathbf{L}-\vec{p}_{xn}||}$ where (L_1, L_2, L_3) is the position of the light source.

Our error analysis handles error that arises both from non-linearity in the shading model, and from curvature in objects. First, we derive error for the case of a rotated and scaled paraboloid, since a paraboloid is a representative quadric surface. Using these analytical error bounds for paraboloids and simple transformations to match specific sections of convex objects to the paraboloid, radiance error bounds are derived for the various ray tracing primitives.

5.2.1 Analytical error for a paraboloid

Without loss of generality, we consider a scaled, rotated paraboloid to represent the sections of objects inside a linetree cell centered around the origin, and perpendicular to the z axis (Refer to Figure 7). All points on the paraboloid satisfy the constraint equation $z' + x'^2 + y'^2 = 0$ in the paraboloid's coordinate system (x', y', z').

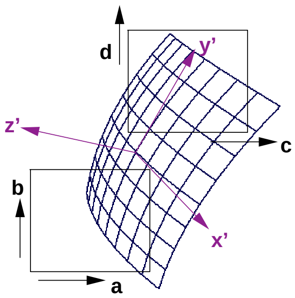

Figure 7: A rotated and scaled paraboloid at origin of linetree cell.

The transformation between the paraboloid's object-space axes, and the linetree's world-space axes is given by a matrix M:

$$(x', y', z') = \mathbf{M}(x, y, z)$$

where $\mathbf{M} = \rho\mathbf{S}$. M combines an arbitrary rotation matrix ρ, and a scaling matrix $\mathbf{S}$. $\mathbf{S}$ scales by s_x on the x' axis, s_y on the y' axis, and 1 on the z' axis, without loss of generality [21]. This scaling and rotation allow the paraboloid to approximate any convex surface near the intersection point. s_x and s_y are equal to half the reciprocal curvature of the represented surface along the x' and y' axes respectively. For a plane, $s_x = s_y = 0$, while for a sphere, $s_x = s_y = \frac{1}{2R}$.

Since the front and back faces of the linetree cell are at $z = H$ and $z = -H$ respectively, a ray parameterized by (a, b, c, d) represents a ray in 3D space from (a, b, H) to $(c, d, -H)$. Therefore, the unnormalized incident ray is $\mathbf{I} = (c - a, d - b, -2H)$. Since the linetree cell is guaranteed to be free of discontinuities (Refer to Section 5.1), every ray intersects the paraboloid at some point $\vec{p}_{xn}$, where $\vec{p}_{xn}$ satisfies $z' + x'^2 + y'^2 = 0$. Using the constraint that $\vec{p}_{xn}$ lies on both the incident ray and the paraboloid we can solve for $\vec{p}_{xn}$ in terms of (a, b, c, d). The constraint equation is a quadratic in z [22], which can be solved to second order. For a sufficiently small linetree cell, the intersection point is close to the origin, and can be approximated as a perturbation to the intersection of the ray with the tangent plane at the origin: $z = -\frac{C}{B}(1 + \frac{AC}{B^2})$, where A, B, C are the coefficients of the quadratic. This approximated z is used to compute the point of intersection and the normal, which are then used to estimate error in diffuse and specular radiance.

We use the notation $\delta^2 R$ to represent the quadratic terms in radiance R. These terms can be used to bound interpolation error. We only consider the terms in a^2, b^2, c^2, d^2, since quadrilinear interpolation can approximate cross-terms such as ab. Using this notation, we have the diffuse radiance error for an infinite light source:

$$\begin{aligned}
\epsilon_{di} &= \delta^2 R \\
&\approx (\frac{\delta^2 \mathbf{N}}{||\mathbf{N}||} + \mathbf{N}\delta^2 \frac{1}{||\mathbf{N}||}) \cdot \mathbf{L}
\end{aligned}$$

$$= \sum_i^3 l_i\left(\frac{(c^2-a^2)s_x\rho_{1i}\rho_{31}+(d^2-b^2)s_y\rho_{2i}\rho_{32}}{2\rho_{33}H} - \rho_{3i}\frac{(a^2+c^2)s_x^2+(b^2+d^2)s_y^2}{2}\right)$$

For a plane, $s_x = s_y = 0$; therefore, ϵ_{di} is 0, which is what we would expect since the diffuse radiance for a plane with an infinite light source is a constant.

Using a similar technique, the error for diffuse radiance with a local light source is:

$$\begin{aligned}\epsilon_{dl} &= \delta^2\widehat{\mathbf{N}}\cdot\mathbf{L} + \widehat{\mathbf{N}}\cdot\delta^2\mathbf{L} \\ &= \epsilon_{di} + \frac{a^2+c^2}{4\|L\|}\left(s_x^2\frac{\rho_{22}^2}{\rho_{33}^2} + s_y^2\frac{\rho_{12}^2}{\rho_{33}^2} - \frac{2s_x\rho_{22}}{\rho_{33}}\right) + \frac{b^2+d^2}{4\|L\|}\left(s_x^2\frac{\rho_{21}^2}{\rho_{33}^2} + s_y^2\frac{\rho_{11}^2}{\rho_{33}^2} - \frac{2s_y\rho_{11}}{\rho_{33}}\right)\end{aligned}$$

where ϵ_{di} is the diffuse error term with light $L = (\frac{L_1}{\|L\|}, \frac{L_2}{\|L\|}, \frac{L_3}{\|L\|})$. Note that the $\|L\|$ term in the denominator implies that the error term for distant lights is essentially the same as the diffuse error for infinite light sources, which is exactly what we would expect intuitively.

In the interest of brevity, we do not present the error terms for specular radiance; however, the solution technique is of interest. The error for specular radiance is more complicated because $(\mathbf{N}\cdot\frac{-\mathbf{I}+\mathbf{L}}{2})$ is raised to the exponent n, the shininess of the object. Using a Taylor expansion of specular radiance, the quadratic error term for specular radiance is :

$$\epsilon \approx K_0^n\left[n\frac{K_2}{K_0} + \frac{n(n-1)}{2}\frac{K_1}{K_0}^2\right]$$

where, K_0, K_1, K_2 are the terms of the expression $(\mathbf{N}\cdot\frac{-\mathbf{I}+\mathbf{L}}{2})$ that are constant, linear and quadratic in a, b, c, d respectively. For an infinite light source $K_0 = \widehat{L'}\cdot(\rho_{31}, \rho_{32}, \rho_{33})$, where $L' = (l_x, l_y, 1+l_z)$.

5.2.2 Evaluating the error estimate

The analytical expressions derived in Section 5.2.1 can be used to bound the error for interpolants. For a linetree cell spanning (a_0, b_0, c_0, d_0) to (a_1, b_1, c_1, d_1), this error bound is evaluated conservatively. For example, $(c^2 - a^2)$ is evaluated as $(c_1^2 - a_0^2)$, and $(c^2 + a^2)$ is evaluated as $c_1^2 + a_1^2$. In addition, incorrect cancellation is avoided by adding absolute values of error components to compute error. The computed error value is compared against a user-specified error ϵ; the error predicate signals the interpolant as valid only if the analytical error is less than ϵ. The analytical error bounds show that subdividing linetree cells decreases error by roughly a factor of 4 per subdivision.

Plate 1-Figure 2 demonstrates the error refinement process for $\epsilon = (40, 0.5)$. On the left, the progressively refined rendered images are shown; on the right are the linetrees corresponding to the images. When ϵ is high, visible artifacts can be seen around the specular highlight. The system automatically detects that refinement is necessary around the specular highlight and refines interpolants that exceed the user-specified error bound. As ϵ is decreased the image quality is improved, and the linetrees get increasingly subdivided. At a relative error of 0.5, the region of the specular highlight is no longer interpolated, as shown by the hole in the linetree cells.

An interesting observation is that the analytical error expressions can be used to determine which axes to split the linetree on: the a and c axes, or the b and d axes. Error-driven subdivision increases the effectiveness of interpolants at approximating linear components of radiance. We discuss error-driven subdivision in Section 8.6.

5.2.3 Matching the paraboloid to ray tracing primitives

Given an object, and a potential interpolant for a linetree cell, we would like to map that object and linetree cell to the scaled and rotated paraboloid in Section 5.2.1. Matching the scaling and rotation of the paraboloid with each of the primitives of the ray tracers — spheres, cones, cubes, cylinders — is straightforward; the normal and curvature of each primitive are matched with that of the paraboloid, yielding s_x, s_y, and ρ. The same process can be carried out easily for more complex surfaces. Once a mapping is established, we can use the error terms in Section 5.2.1 to bound the error for the interpolant.

6 Exploiting Coherence

Lazy interpolants eliminate a significant fraction of the shading computations, and their associated intersections; however, they do not reduce the number of intersections computed for pixel assignment. With lazy interpolants, the following three operations become dominant costs:

- pixel assignment
- for pixels that can be interpolated, computing the 4D intercepts for the ray and evaluating 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 the techniques we use to accelerate pixel assignment and interpolation by further exploiting temporal and image-space coherence.

6.1 Temporal coherence

For scenes with any complexity, pixel assignment is expensive. However, it is possible to accelerate pixel assignment by exploiting temporal coherence in the user's viewpoint. In this section, we present our reprojection algorithm, which accelerates pixel assignment while guaranteeing correctness. The intuition behind this algorithm is that for small movements of the viewpoint, the visibility of a large number of pixels in the frame is similar to that in the previous frame.

Our algorithm reprojects linetrees from a previous frame to the new viewpoint; for most pixels this quickly determines the linetree that covers the pixel. Our algorithm is conservative: it never incorrectly assigns a linetree to a pixel. To guarantee correct pixel assignment, linetree faces are shaft-culled [19] with respect to the current viewpoint. Reprojection accelerates pixel assignment by replacing multiple intersect operations for pixels by a single shaft cull against the corresponding linetree. Figure 8 shows how reprojection works when the viewpoint moves from **eye** in frame 0 to **eye'** in frame 1; linetree cells L1 and L2 are reprojected to the new image plane.

An additional benefit of the reprojection algorithm is that in assigning a linetree to a pixel, it replaces all intersect and shading computations for that pixel by an interpolation.

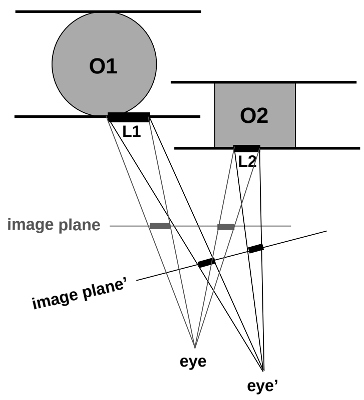

Figure 8: Linetree cells L_1 and L_2 can be reprojected when the eye moves from **eye** to **eye'**

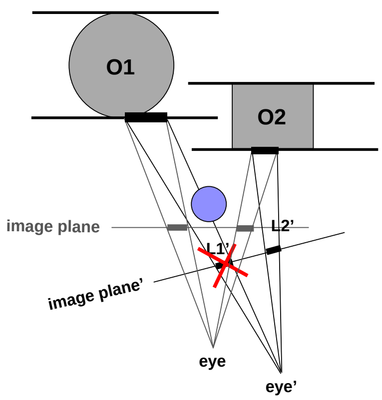

Figure 9: Sphere occluding visibility to the reprojection of L_1

Reprojection is based on the observation that if a linetree cell has a valid interpolant, it is guaranteed that there are no discontinuities, such as occluders, in the volume of space between its faces. This invariant helps ensure that reprojection determines *correct* pixel assignment. Note that in most systems pixel assignment is complicated by non-planar silhouettes. They are not a problem in our algorithm because the linetree cells do not contain discontinuities.

We will now discuss two important issues for the reprojection algorithm: how to efficiently reproject 4D linetrees in 3D space, and how to guarantee correct pixel assignment.

6.1.1 Reprojecting 4D linetrees in 3D

Each linetree cell corresponds to a 4D hypercube in ray space; however, in 3D, a linetree cell can be represented by its front and back faces. Therefore, to reproject a linetree cell, we reproject the front and back faces of the linetree cell in 3D. A pixel is covered by a linetree cell *iff* both its front and back faces cover the pixel, and no other linetree cell is projected onto that pixel. If two linetree cells reproject onto the same pixel, it indicates a potential change in visibility that invalidates the use of the reprojected linetrees for that pixel.

A fast way to identify the 4D linetrees that contribute to a pixel is to use the polygon-drawing hardware available in a standard graphics workstation. The front and back faces of linetree cells from the previous frame can be projected to the screen from the new viewpoint as quadrilaterals. In addition, alpha-blending hardware can be used both to efficiently determine which linetree covers a pixel, and to detect conflicts between reprojected linetrees.

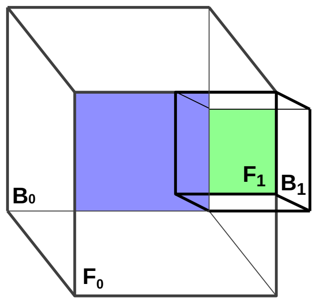

Figure 10: Linetree overlap

Figure 10 shows a subtle problem that arises from the fact that linetrees are 4D entities. Faces F_0 and B_0 define a linetree cell L_0, and faces F_1 and B_1 define another linetree cell L_1. L_0 lies one level higher within the linetree than L_1 does. If the linetrees are naively reprojected, F_1 will overlap with F_0, indicating a conflict. The overlap is not really a problem because when the front faces overlap, it is the back face that uniquely determines which linetree cell contributes to that pixel. In the figure, the two gray regions correspond to the screen-space projections of the two linetree cells. Note that the regions do not overlap, indicating that there is no conflict. Since this overlap problem arises for a significant number of pixels, we have devised an efficient algorithm to solve the problem. The algorithm walks up the linetree and clips faces to avoid spurious overlap between linetree cells at different depths. This algorithm takes $O(\log\ d)$ time per linetree cell, where d is the cell depth.

6.1.2 Occlusion

Reprojection alone is not enough to guarantee correct visibility. In Figure 9, an object that was not visible in the previous frame (the small sphere) occludes objects in the current frame.

We conservatively determine visibility by not reprojecting linetrees when they might be occluded. This condition is detected by shaft-culling [19] each reprojected linetree cell against the current viewpoint. 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.

If only one linetree is reprojected to a pixel, the shaft-cull tests guarantee that that linetree is the correct assignment for the pixel. Thus, no visibility computation or shading operation needs to be done for that pixel, since the linetree cell can be used for interpolation directly.

6.2 Image-space coherence

Using the reprojected linetrees, a simple scan-line algorithm further accelerates the ray tracer. Information about which reprojected linetree is associated with a pixel is stored in a *reprojection buffer*, which is checked when rendering the pixel. If a reprojected linetree is available, a span is identified for all subsequent pixels on that scanline that have the same linetree in their reprojection buffer. The radiance for each pixel in the span is interpolated in screen space between the endpoints of the span. Using screen-space interpolation eliminates almost all of the cost of ray-tracing the span. No intersection or shading computations are performed, and interpolation can be performed incrementally along the span in a tight loop.

The current perspective projection matrix can be folded into the error bound computation (See Section 5), yielding a different *screen-space* error guarantee, but for linetrees 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.

Several researchers have accelerated ray tracing by exploiting image coherence [3, 2, 23]. The traditional problem with screen-space interpolation is that it is difficult to predict the accuracy of interpolation. Our error predicate solves this problem of interpolation accuracy, allowing us to exploit image coherence while producing correct results.

7 Other Optimizations

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

Linetree depth

In our adaptive algorithm, interpolants are built only if the benefits of interpolating radiance outweigh the costs of building interpolants. We achieve this with a simple cost model that is evaluated when deciding whether to subdivide a linetree cell. Linetrees are subdivided on the basis of number of screen pixels that they are estimated to cover. Thus, when the observer zooms in closer to an object, interpolants for that object are built to a greater resolution; if an observer is far away from an object, the interpolants are coarse, saving memory.

This cost model also ensures that the cost of building interpolants is amortized over several pixels. This is important for performance, because building interpolants is much more expensive than ray tracing a single pixel.

Unique interpolant rays

The cost of building interpolants can be decreased by noticing that the 4D linetree structure uniformly subdivides 4D space. The 4D hypercubes represented by siblings in the linetree share common vertices in 4D, corresponding to common rays in 3D. For example, two sibling linetrees that both have the same front face but different back faces share eight of the sixteen extremal rays. Therefore, when building interpolants, we use a hash table indexed by the ray's parameters to guarantee that each ray is only shot once.

Unique ray trees

Storing interpolants and their associate ray trees could result in substantial memory usage. However, all successful interpolants and a large number of failed interpolants are associated with similar ray trees. Therefore, hash tables can be used to avoid storing duplicate ray trees. As a result, virtually all the memory allocated by the system is used to store interpolants.

A compression technique similar to that presented by Levoy and Hanrahan [24] might be useful to compress interpolants. However, we do not expect to achieve the same compression rates, because our adaptive subdivision effectively compresses regions of ray space already.

Recursive shaft culls

The reprojection algorithm shaft-culls linetrees from the previous frame. However, for large linetrees these shaft-culls often fail but are only partly blocked by an occluding object. To ameliorate this problem, we recursively subdivide and shaft-cull linetree faces, using a cost model similar to that used for subdividing linetrees themselves. The portions of the faces that are not blocked are then reprojected.

Textures

Our system supports textured objects by separating the texture coordinate computation from texture lookup. Texture coordinates are quadrilinearly interpolated along with radiance. However, since we do not recursively interpolate radiance, this solution does not interpolate reflections of textured objects in textures. These regions fail and are ray traced.

8 Performance Results

This section evaluates the performance of our system, both in terms of speed and memory usage. To evaluate the speed of the system, we use a simple cost model based on Figure 1.

8.1 Base ray tracer

For performance, we compare our system to a ray tracer that implements classical Whitted ray tracing with textures [39]. This ray tracer is in fact the same ray tracer that our system uses for non-interpolated pixels. Both our system and the base ray tracer support the same Whitted shading model, augmented with textures. They support convex primitives such as cubes, spheres, cylinders, cones, disks and polygons.

To make the comparison between the ray tracers fair, optimizations were applied to both the base ray tracer and to our interactive ray tracer when possible. There were a number of such optimizations. To speed up intersect computations, the base ray tracer uses kd-trees for spatial subdivision of the scene [13]. 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 last ray landing in that quadtree, and this cache has a 85% 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. Extensions such as the adaptive shadow testing [37], and Light Buffers [18] would improve performance but have not been implemented yet in our ray tracer.

8.2 Test scene

The data reported below was obtained for the art-room scene shown in Plate 2. Rendered images from the scene appear on the left, and on the right error-coded images show the regions of interpolation success and failure.

The scene has about 80 primitives: cylinders, cubes, cones and spheres, and three local light sources. All timing results are reported for frames rendered at 800x600 resolution; i.e. each frame has 48k pixels. The user's viewpoint changes from frame to frame in arbitrary directions. The rate of translation and rotation of the room are set such that the user can cross the entire length of the room in 400 frames, and can rotate in place by 360deg in 150 frames.

In the error-coded images, success is indicated by a gray-blue color; other colors indicate various reasons why interpolation was not permitted. Bright green regions correspond to subdivision due to radiance discontinuities: for example, shadows or occluders. The yellow regions correspond to subdivision because the interpolant rays did not hit the object. Red regions correspond to blocker objects, which occur when interpolating reflected radiance. Magenta regions show where the error predicate detected non-linear radiance.

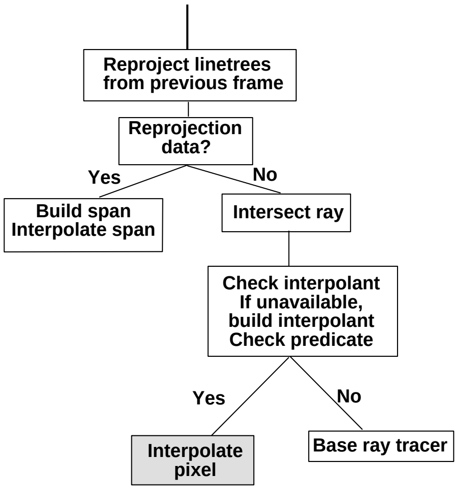

Figure 11: Cost Model

8.3 Cost model

As shown in Figure 1, there are three paths by which a pixel is assigned radiance:

1. Fast path: reprojected data is available, and is 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 if the interpolant is valid. If the interpolant is invalid, the cell is subdivided.
3. Slow path: no valid interpolant is available, so rendering is done by the base ray tracer.

In the interactive ray tracer, every frame includes the basic cost of reprojection, in addition to the cost of rendering pixels using one of the three paths.

8.4 Performance results

Path	Cost for path
Fast path	0.022
Interpolate and build interpolants	0.84 — 1.33
Base ray tracer	1.00

Table 1: Relative costs normalized with respect to the base ray tracer.

In Table 1 the relative costs of each of the three paths is shown, where the cost of the base ray tracer is assumed to be 1. The data for this table was obtained by averaging the results obtained from our system over a 30-frame walkthrough of the art-room scene. The timing results were obtained on a single-processor 194 MHz Reality Engine 2, with 512 MB of main memory. The costs in the table do not reflect the time required to shaft-cull and reproject linetrees, which adds a roughly constant overhead to each frame of about 2 seconds.

The fast path is approximately 45 times faster than the base ray tracer. However, the interpolate building path is somewhat slow because we pay a penalty for doing no pre-processing. In an off-line rendering application of this system, this penalty would not be significant. On the average, the fast path accounts for 65-75% of the pixels, and the build interpolant and base ray tracing paths each account for about 15% of the pixels.

In Figure 12, we compare the average performance of the interactive ray tracer with that of the base ray tracer. The graph shows three bars. The bar on the left shows the time required by the base ray tracer, which for the art-room scene is a nearly constant 28.4 seconds per frame. The middle bar shows the time required by the interactive ray tracer, broken down by rendering path. Notice how little time is spent on span fills. The interpolation path (2) is a significant part of the time because it builds new interpolants on the fly. If the user stays in the same position or moves slowly, fewer interpolants are built and rendering is faster (less than 8 seconds per frame). However, we selected a movement speed that was consistent with generating a walkthrough animation.

The bar on the right shows the number of pixels rendered by each path. Note that most of the pixels were rendered quickly by the fast path. Including the reprojection costs, the fast path renders 75% of the pixels in 25% of the time. The rest of the time is spent in building interpolants and ray tracing pixels that failed. Overall, the system is 250% to 300% faster than the base ray tracer during the walkthrough sequence.

The time required by the interactive ray tracer depends on the complexity of the image shown 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 paths 2 or 3 must be followed. The art-room walkthrough included movements in various directions.

8.5 Memory Usage

During the 30-frame walkthrough, the system allocated about 50 megabytes of memory to contain linetree cells. The first frame of the walkthrough allocates about 40% of this total memory; each subsequent frame then allocates 900k

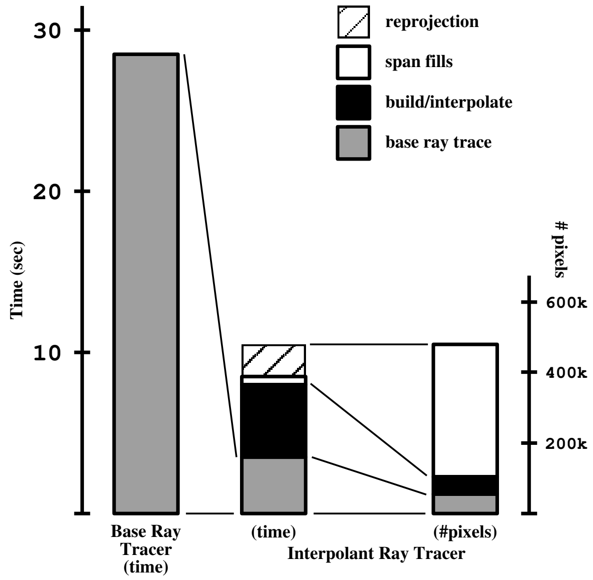

Figure 12: Average performance and pixel breakdown

bytes more on average, though there is considerable fluctuation; more memory is allocated for linetrees as new objects move into view. We have not tried seriously to reduce memory usage, though there are some optimizations for memory usage. For example, because of the 16-way branching factor of linetrees, almost all the memory is taken up by leaves of the tree. For this reason, we allocate memory for leaves lazily.

8.6 Discussion

We have presented a system that uses radiance interpolation and reprojection to accelerate rendering. As Figure 12 indicates, rendering time is dominated by the cost of ray tracing failure regions, and the cost of building interpolants.

It is clear that further significant improvements can only come by increasing reprojection and interpolant success rates. The analytical error expressions derived in Section 5 suggest that uniformly subdividing a linetree cell into its 16 children is overly agressive. A better approach would be to use the error predicate to determine the appropriate axes to split. We have implemented a 4-way split algorithm that splits each linetree cell into 4 children and uses the error predicate to guide subdivision. For the same scene and using the same number of rays to build interpolants, the 4-way split achieves an interpolation success rate of 91%. This nearly halves the interpolation failure rate. However, 4-way splitting complicates the reprojection algorithm. We are currently implementing reprojection with 4-way splits and expect performance to improve substantially.

Another way to further improve performance is to decrease the cost of building interpolants. This can be done by using an off-line preprocessing phase, or predictively building interpolants while the user is idle.

9 Conclusions and Future Work

In this paper, we have presented a system for interactive ray tracing. Our system explicitly decouples shading computations and pixel assignment, and accelerates both independently. We use 4D interpolants to approximate radiance, and a fast conservative reprojection algorithm to accurately determine pixel assignment. An error predicate detects radiance discontinuities and computes analytic error bounds to guarantee that interpolation does not approximate radiance erroneously. The system interactively renders ray traced images and refines error.

There are many opportunities for extending and applying this work. Our system samples the view-dependent base ray tracing component of radiance, while systems such as Radiance [38] sample view-independent diffuse inter-reflection. It would be interesting to extend our system to support a more complete shading model, including diffuse inter-reflection and generalized BRDFs.

One advantage of the linetree representation for radiance is that it explicitly keeps track of the ray trees used to generate the radiance function. These ray trees are very similar to those used for interactive scene manipulation [31, 6]. Integrating such systems would allow dynamic editing of the model being viewed while also allowing dynamic viewing.

References

[1] ADELSON, S. J., AND HODGES, L. F. Generating Exact Ray-Traced Animation Frames by Reprojection. *IEEE Computer Graphics and Applications 15*, 3 (May 1995), 43–52.

[2] AMANATIDES, J. Ray Tracing with Cones. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), p. ray tracing spheres and polygons with circular conical rays. summary in Graphics Interface '84, p. 97-8Published as Computer Graphics (SIGGRAPH '84 Proceedings), volume 18, number 3.

[3] AMANATIDES, J., AND FOURNIER, A. Ray Casting using Divide and Conquer in Screen Space. In *Intl. Conf. on Engineering and Computer Graphics*. Beijing, China, Aug. 1984.

[4] ARVO, J., AND KIRK, D. Fast Ray Tracing by Ray Classification. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), p. five dimensional space subdivision. Published as Computer Graphics (SIGGRAPH '87 Proceedings), volume 21, number 4also in Tutorial: Computer Graphics: Image Synthesis, Computer Society Press, Washington, 1988, pp. 196-205.

[5] BADT, JR., S. Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing. *The Visual Computer 4*, 3 (Sept. 1988), 123–132.

[6] BRIERE, N., AND POULIN, P. Hierarchical View-dependent Structures for Interactive Scene Manipulation. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (Aug. 1996), pp. 83–90.

[7] CHAPMAN, J., CALVERT, T. W., AND DILL, J. Exploiting Temporal Coherence in Ray Tracing. In *Proceedings of Graphics Interface '90* (Toronto, Ontario, May 1990), Canadian Information Processing Society, pp. 196–204.

[8] CHAPMAN, J., CALVERT, T. W., AND DILL, J. Spatio-Temporal Coherence in Ray Tracing. In *Proceedings of Graphics Interface '91* (Calgary, Alberta, June 1991), Canadian Information Processing Society, pp. 101–8.

[9] CHEN, S. E., RUSHMEIER, H., MILLER, G., AND TURNER, D. A Progressive Multi-Pass Method for Global Illumination. *Computer Graphics (SIGGRAPH '91 Proceedings) 25*, 4 (July 1991), 165–174.

[10] COOK, R. L. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics 5*, 1 (Jan. 1986), 51–72. also in Tutorial: Computer Graphics: Image Synthesis, Computer Society Press, Washington, 1988, pp. 283-304. Comments on article in ACM TOG, v. 9, n. 2, p 233-243.

[11] DIEFENBACH, P., AND BADLER, N. Multi-Pass Pipeline Rendering: Realism for Dynamic Environments. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (1997), pp. 59–70,186–7.

[12] DRETTAKIS, G., AND SILLION, F. X. Interactive Update of Global Illumination Using a Line-Space Hierarchy. In *Computer Graphics (SIGGRAPH '97 Proceedings)* (August 3–8 1997), pp. 57–64.

[13] GLASSNER, A., Ed. *Introduction to Ray Tracing*. Academic Press, Palo Alto, CA, 1989.

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

[15] GLASSNER, A. S. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[16] GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics (Proc. Siggraph '84) 18*, 3 (1984), 213–222.

[17] GORTLER, S., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (August 4–9 1996).

[18] HAINES, E., AND GREENBERG, D. The Light Buffer: A Shadow-Testing Accelerator. *IEEE Computer Graphics & Applications 6* (September 1986), 6–16.

[19] HAINES, E. A., AND WALLACE, J. R. 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, p. also in SIGGRAPH '91 Frontiers in Rendering course notes and via FTP from princeton.edu:/pub/Graphics/Papers.

[20] HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. A Rapid Hierarchical Radiosity Algorithm. *Computer Graphics (Proc. Siggraph '91) 25*, 4 (1991), 197–206.

[21] HECKBERT, P. The Mathematics of Quadric Surface Rendering and SOID. Tech. Rep. 3-D Technical Memo No. 4, New York Institute of Technology, July 1984.

[22] HECKBERT, P., AND HANRAHAN, P. Beam Tracing Polygonal Objects. *Computer Graphics (Proc. Siggraph '84) 18*, 3 (1984), 119–127.

[23] KOLB, C. RayShade Homepage `http://www-graphics.stanford.edu/~cek/rayshade/`.

[24] LEVOY, M., AND HANRAHAN, P. Light Field Rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (August 4–9 1996).

[25] LISCHINSKI, D., SMITS, B., AND GREENBERG, D. P. Bounds and Error Estimates for Radiosity. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)* (July 1994), A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 67–74. ISBN 0-89791-667-0.

[26] MARK, W., MCMILLAN, L., AND BISHOP, G. Post-Rendering 3D Warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (1997), pp. 7–16,180.

[27] NIMEROFF, J., DORSEY, J., AND RUSHMEIER, H. A Framework for Global Illumination in Animated Environments. In *6th Annual Eurographics Workshop on Rendering* (June 12–14 1995), pp. 223–236.

[28] PAINTER, J., AND SLOAN, K. Antialiased Ray Tracing by Adaptive Progressive Refinement. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), p. adaptive stochastic sampling. Published as Computer Graphics (SIGGRAPH '89 Proceedings), volume 23, number 3.

[29] PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Computer Graphics (SIGGRAPH '97 Proceedings)* (August 3–8 1997), pp. 101–8.

[30] PHILLIPS, G. M., AND TAYLOR, P. J. *Theory and Application of Numerical Analysis*. Academic Press, London, 1996.

[31] SEQUIN, C. H., AND SMYRL, E. K. Parameterized Ray Tracing. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), p. store ray tree data to allow quick material changes. Published as Computer Graphics (SIGGRAPH '89 Proceedings), volume 23, number 3.

[32] SILLION, F., AND PUECH, C. A General Two-Pass Method Integrating Specular and Diffuse Reflection. *Computer Graphics (SIGGRAPH '89 Proceedings) 23*, 3 (July 1989), 335–344.

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

[34] TELLER, S., BALA, K., AND DORSEY, J. Conservative Radiance Interpolants for Ray Tracing. In *Seventh Eurographics Workshop on Rendering* (June 15–17 1996).

[35] WALLACE, J., COHEN, M., AND GREENBERG, D. A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. *Computer Graphics (SIGGRAPH '87 Proceedings) 21*, 4 (July 1987), 311–320.

[36] WARD, G., AND HECKBERT, P. Irradience Gradients. In *Rendering in Computer Graphics (Proceedings of the Third Eurographics Workshop on Rendering)* (Bristol, United Kingdom, May 1992), Springer-Verlag.

[37] WARD, G. J. Adaptive Shadow Testing for Ray Tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)* (New York, 1994), Springer-Verlag, p. avoid shooting rays at lights determined to not affect perception of image.

[38] WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. A Ray Tracing Solution for Diffuse Interreflection. In *Computer Graphics (SIGGRAPH '88 Proceedings)* (Aug. 1988), pp. 85–92. Published as Computer Graphics (SIGGRAPH '88 Proceedings), volume 22, number 4.

[39] WHITTED, T. An Improved Illumination Model for Shaded Display. *CACM 23*, 6 (1980), 343–349.