LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-549

Conservative Interpolants for Ray Tracing

Seth Teller   Kavita Bala   Julie Dorsey

April, 1996

This document has been made available free of charge via ftp
from the MIT Laboratory for Computer Science.

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

Conservative Interpolants for Ray Tracing

Seth Teller Kavita Bala Julie Dorsey

Imagery and Simulation Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

Classical ray-tracing algorithms compute the quantity RADIANCE (in units of POWER per AREA per STERADIAN) returning to the eye along one or more sample rays through each pixel of an image. Radiance is a four-dimensional quantity over each surface, whose computation involves diffuse and specular effects, as well as refraction, transmission, and secondary emission effects. The radiance computation is therefore extremely complex – so complex, in fact, that ray-tracing algorithms typically *fix* the synthetic viewpoint, requiring it as an algorithm input, and compute radiance values only for this viewpoint. Thus, the output of a ray-tracing algorithm, although potentially photorealistic, is a two-dimensional quantity – an image array of radiance values – and is not directly useful from any viewpoint other than the one for which it was computed. Moreover, classical ray-tracing algorithms make no explicit mention or use of solution error, nor can they avoid aliasing artifacts except by supersampling.

This paper makes several contributions. First, it directly incorporates the notion of *error* into classical ray-tracing, by lazy construction of *conservative radiance interpolants* in *ray space*. These interpolants blend only radiance values that arise from ray trees of identical *topologies*. We show that these interpolants can be constructed so as to return, for any query ray, radiance values that differ by at most ϵ from those that would be computed by classical (e.g., Whitted) ray-tracing, where ϵ is a user-adjustable relative error tolerance. The second contribution of the paper is an explication of the four sources of aliasing inherent in classical ray tracing – termed *gaps, blockers, funnels, and peaks* – and an adaptive subdivision algorithm for identifying regions in ray space *guaranteed* to be free of these phenomena. Finally, we describe a novel data structure that exploits both screen-space and object-space coherence in the radiance function, accelerating not only the generation of single images but the generation of images from a sequence of smoothly varying eyepoints.

We describe a preliminary implementation of our algorithm, and discuss its application to the usual speed-quality tradeoff. If image *quality* is to be optimized, our data structures allow the generation of images that have no unrecognized sources of aliasing error. If rendering *speed* is to be optimized, our data structures allow dynamic specification of an error tolerance ϵ, whose increase yields a direct decrease in rendering time.

©Massachusetts Institute of Technology 1996

This research was supported by Light Control, Inc.

1 Introduction

One long-standing goal of visual simulation is to allow users to traverse realistically illuminated, geometrically detailed environments at interactive rates. Given a scene description, comprised of surface geometry and reflectance properties, and some specification of light sources, a class of techniques known as "global illumination algorithms" computes equilibrium distributions of radiant energy throughout the scene. That is, they compute some representation of radiant energy incident upon, or emerging from, every surface element in the scene. Radiosity, Ray-Tracing, and hybrid algorithms span a spectrum, in the way that they trade freedom of the synthetic viewpoint against the degree of complexity of the light sources, surface geometry, and surface reflectance properties.

1.1 Solution Complexity vs. Viewing Freedom

Radiosity algorithms, at one extreme, compute RADIOSITY (in units of POWER per AREA), over every input surface [11, 29]. Radiosity algorithms are usually formulated for polyhedral scenes, and can correctly handle only ideal diffuse emitters and receivers. Since the human visual system is very good at detecting and interpreting highlights that result from the directional variation of radiance, this representation is restrictive. However, since the computed solution (essentially an annotation of the input scene) is *view-independent*, it is meaningful from any synthetic viewpoint. Thus radiosity algorithms do not, in general, produce photorealistic results, but are widely applied in interactive applications requiring free viewer motion.

Ray tracing algorithms, at the other end of the spectrum, compute the quantity RADIANCE (with units of POWER per source AREA per receiver STERADIAN) for a ray through each pixel of an image [33, 10]. Radiance is a four-dimensional quantity over each surface, whose computation involves diffuse and specular effects, as well as refraction, transmission, and secondary emission effects. The radiance computation is therefore extremely complex – so complex, in fact, that ray tracing algorithms *fix* the synthetic viewpoint and viewport, requiring them as additional inputs, and compute radiance *only for the specified viewpoint.* Thus, the output of a ray tracing algorithm, although potentially photorealistic, is a two-dimensional quantity – an image array of radiance values – and is not directly applicable to any viewpoint other than the one for which it was computed.

A class of hybrid techniques, known as Radiance, Two-Pass, or Specular algorithms, has arisen which compute more general representations of the radiance function – typically, some discretized approximation to the radiance and/or irradiance field at each surface, or in space [30, 31, 28, 27, 4]. However, these representations have typically been enormously expensive to compute and store. Thus, the important visual cues provided by these algorithms (shadows, smooth shading, transparency, reflections, etc.) have not been achievable in interactive visual simulation applications. For a discussion of interactivity versus rendering accuracy, see [22].

1.2 Solution Error vs. Running Time

Rendering is a numerical simulation problem, and as such is subject to various sources of error. Given a fixed scene geometry and reflectance model, another long-standing problem in computer graphics is to achieve a controllable tradeoff between the time expended computing a solution, and the fidelity of that solution. Hierarchical radiosity techniques [14] explicitly incorporate solution error via lazy enumeration of coupling coefficients according to a per-coefficient error estimate. Sophisticated error bounds for radiosity solutions have also been described (e.g. in [19]). Ray-tracing and hybrid algorithms have historically employed supersampling or distribution sampling to decrease solution error.

1.3 Exploiting Coherence, and Piecewise Smoothness, of Radiance

Given the computational resources available, even on today's most powerful workstations, it is overambitious to attempt to represent the radiance distribution for *all* surfaces, from all viewing directions, for a non-trivial scene. However, we show that it is feasible to compute radiance *lazily* for a small *predictive* neighborhood around the synthetic viewpoints exercised in a real visual simulation application. We do so by exploiting both the *visibility coherence* of an observer moving through the scene, and the *smoothness* of the radiance field.

This paper introduces a number of ideas:

- A **relative error tolerance** ϵ directly controls the tradeoff between solution error and rendering time;

- An **adaptive, per-surface radiance interpolant** captures the radiance (angular flux density) leaving the surface to within ϵ, for a lazily-constructed envelope of directions;

- Radiance interpolation is performed by **quadrilinear interpolation in a four-dimensional ray space**;

- The four causes of aliasing in classical ray tracing algorithms – gaps, blockers, funnels, and peaks – are identified, and ray tree **topologies** and **isotopies** defined. Only ray trees of like topology are interpolated, and only when they form an isotopy;

- The fundamental visibility operation in ray tracing, ray casting, is **explicitly decoupled** from the fundamental radiometric operation, integration of irradiance, yielding a sampling scheme that suffers absolutely no unrecognized geometric aliasing, and strictly bounded radiometric aliasing, at every pixel.

2 Classical Ray Tracing

We review classical "backward" ray tracing [33], so called because it identifies only those paths along which radiance reaches the eye from the specified light sources or background [10]. Given a scene description comprised of implicit primitives (each with diffuse and specular reflectance coefficients k_d and k_s, and transmission coefficient k_t, respectively), one or more local or infinite point light sources, an ambient light level, and a synthetic eyepoint, view frustum, and viewport, the ray tracing algorithm computes a representation of the radiance returning to the eye along a set of sample "eye rays" through each pixel.

Radiance values are computed by recursively spawning a **ray tree** for each eye ray. The following pseudocode, adapted from [15], computes radiance at each pixel center as viewed from the eye. `EyeRay(E, F, x, y)` returns the normalized world-space ray from the eye through the center of pixel `x, y`. `Visibility(P, N, L)` returns a fraction $[0..1]$ that multiplies the computed radiance from a light in the direction **L** upon a differential surface element with normal **N** at point **P** (if $N \cdot L < 0$, the fraction is zero; otherwise, if an object of transmissivity t intervenes, the fraction is t; otherwise, the light is fully visible, and the fraction is one). The vector-valued functions `ReflectionDirection(A,N)` and `TransmissionDirection(A,N,eta1,eta2)` compute reflection and transmission directions according to Snell's law.

```
global int maxD;     // trace at most maxD bounces, and
global float minW;   // while subtree weight exceeds minW

void RayTrace (      // spawn one eye ray through each pixel:
    eye E,           //   from specified eye position
    frustum F,       //   field of view, near & far clipping planes
    viewport V,      //   V.height rasters, each with V.width pixels
    scene S,         //   database of scene objects
    framebuffer B )  //   storage for computed RGB radiance values
{
  For 0 <= y < V.height  // for each raster
    For 0 <= x < V.width // for each pixel of raster
      B[y][x] = Trace ( E, EyeRay(E, F, x, y), 1.0, maxD, 1.0, S );
}

Radiance Trace (     // sample radiance field:
    Point P,         //   observed at point P
    Vector R,        //   along direction R
    float eta,       //   medium refraction index
    int D,           //   current call stack depth
    float W,         //   weight of computed radiance
    scene S )        //   database of scene objects
{
  HitStructure H;    // intersection record
  if ( Intersect ( P, R, S, &H ) )
    return Shade ( H.P, R, eta, H.M, H.N, D, W );
  else
    return Background ( R, D )
}
```

```
Radiance Shade (     // evaluate radiance:
    Point P,         //   at surface point P
    Vector A,        //   viewed along direction A
    float eta,       //   in refractive medium eta
    Object M,        //   on the surface of object M
    Vector N,        //   where the surface normal is N
    int D,           //   with current ray tree depth D
    float W )        //   to contribute with weight W
{
  Radiance radiance = 0;

  if ( ambient light present ) // accumulate ambient irradiance
    radiance += ambient;

  For ( each point light L ) { // accumulate direct irradiance
    float vis = Visibility ( P, N, L );
    if ( vis > 0 ) {
       radiance += vis * DiffuseRadiance ( P, N, L, M.k_d );
       radiance += vis * SpecularRadiance ( P, N, L, M.k_s, M.s );
       }
    }

  // recurse no deeper than maxD
  if ( D == 0 ) return radiance;

  if ( M.k_s * W >= minW ) {   // accumulate reflected irradiance
    Vector rR = ReflectionDirection ( A, N );
    radiance += M.k_s * Trace ( P, rR, eta, D - 1, M.k_s * W );
    }

  if ( M.k_t * W >= minW ) {   // accumulate transmitted irradiance
    Vector tR = TransmissionDirection ( A, N, eta, M.eta );
    if ( not total internal reflection )
       radiance += M.k_t * Trace ( P, tR, M.eta, D - 1, M.k_t * W );
    }

  return radiance;
}
```

The function `Intersect(P,R,S,H)` determines which, if any, scene object is first intersected by the ray with origin **P** and direction **R**, and records this information in a "hit structure" **H** along with the intersection point and surface normal, as described in [10]. Finally, for a light source L, the functions DiffuseRadiance(P, N, L, k_d) and SpecularRadiance(P, N, L, k_s, s) compute radiance from a surface with diffuse and specular reflection coefficients k_d and k_s, respectively, and shininess parameter s, using the Phong shading model [23].

By inspection, recursive ray spawning terminates when either (1) the ray leaves the scene; (2) the call stack depth exceeds maxD; or (3) repeated reflection and/or transmission attenuate to less than minW the weight with which the computed radiance will contribute at the root of the ray tree.

3 Previous Work

Ray tracing algorithms are the subject of an extensive literature [10, 7]. Ray casting was introduced in 1968 as a technique for visibility determination and the rendering of hard shadows [2]. Recursive ray tracing was introduced in 1980 to simulate reflection and refraction phenomena [33]. A host of additional algorithms have been developed in the intervening years. In particular, a variety of strategies have been employed to accelerate the ray tracing process; we briefly describe the most relevant approaches below.

3.1 Ray Tracing Acceleration

Ray tracing algorithms perform an expensive operation – finding the first intersection of a ray with some scene object – tens or hundreds of times per image pixel, and are therefore extremely compute intensive. These algorithms can be accelerated by making individual intersection tests faster via spatial subdivision or creative use of a z-buffer; by amortizing the per-ray intersection cost with a more general ray representation; or by spawning fewer rays via adaptive sampling or importance sampling (see [10], pp. 204-5 for a taxonomy of such techniques current to 1989).

Most ray tracing acceleration methods have concentrated on reducing or amortizing the cost of this fundamental operation. This has been done by casting bundles of rays, as did beam-tracing for polyhedral scenes [16] and cone-tracing [1]; employing a fixed-grid [6] or adaptive [8] 3D spatial hierarchy of scene objects; and employing a 5D hierarchy associating ray sets and the first object encountered by every ray in the set [3].

While many researchers have reduced the per-ray intersection cost, relatively few published algorithms have the intent of reducing the number of rays cast. Adaptive, Warnock-style [32] subdivision can be used in screen space, although nowhere is a non-trivial condition for subdivision termination stated. Popular public domain ray tracers such as RAYSHADE [18] use an adaptive subdivision technique based on variance of screen space radiance samples.

Others approaches reduce the computation time for a sequence of images. For example, Glassner [9] introduced an approach for ray tracing animated scenes. For a fixed viewpoint, one method stored ray trees at each pixel and reevaluated them under changes in light intensities and or surface material properties [25]. Others have parallelized ray-tracing algorithms via implementation on shared-memory or message-passing [10] architectures. To the authors' knowledge, no existing work explicitly optimizes the generation of ray-traced images from a sequence of smoothly varying eyepoints.

3.2 Truncation Error

When gathering irradiance on a given surface, classical ray tracing accounts for irradiance only along a finite set of "likely" directions: from any point lights in the scene; along the reflection direction of the ray; and along the transmission direction of the ray. Moreover, recursive ray spawning is suppressed when the ray tree depth (weight) exceeds (falls below) some prespecified threshold [13]. This truncated integration can give rise to error by excluding ray paths along which significant radiance may flow (e.g., light bouncing among many surfaces before reaching the eye). Researchers have addressed this source of error using distribution ray tracing [5], forward ray tracing [10] and path tracing [17]. This paper does not address truncation error, as our system is designed to compute an ϵ approximation to the radiance values computed by classical ray tracing.

4 Aliasing Error

Classical ray tracing samples the radiance field only along a discrete set of eye rays through each pixel. This discretization gives rise to sampling errors, in which the algorithm does not account for radiance contributions that would only be found by interstitial rays. Reconstruction based on insufficient samples gives rise to *aliasing* – differences between computed radiance averages and those which would be obtained with a sufficiently denser set of samples.

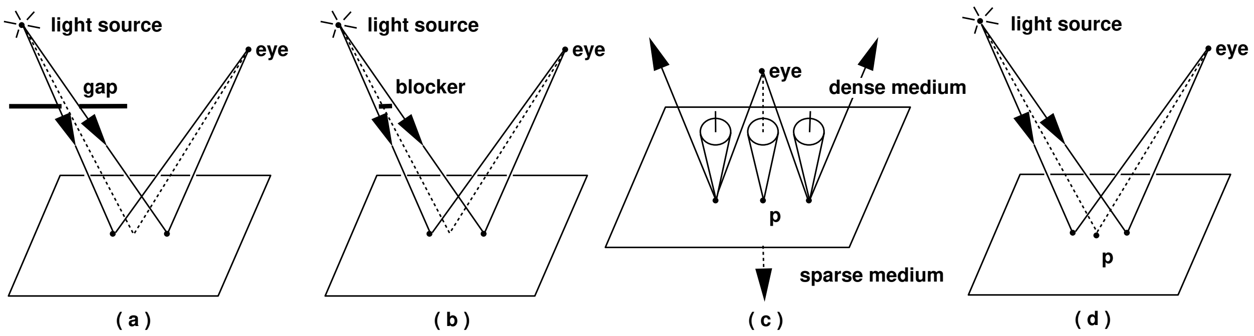

Figure 1: Aliasing errors due to insufficient radiance sampling.

There are four ways in which aliasing errors can arise in classical ray tracing. They are depicted in Figure 1, with sample rays shown as solid lines, and dashed lines depicting significant radiance paths/nonpaths that can be missed through insufficient sampling. First, small **gaps** might not admit sample ray paths, yet admit intervening rays (Figure 1-a). Second, **blockers** might admit sample rays, yet block intervening rays (Figure 1-b). (Horizons or terminators are a special case in which an object "self-blocks," obscuring paths from its own surface to a light source.) Third, when traversing an interface from a dense to a sparse medium (e.g., from water to air), sample rays might be totally internally reflected, whereas an intervening ray is transmitted (Figure 1-c). We call this source of aliasing a **funnel**, because the set of transmitted rays through a dense/sparse boundary have directions which fall within the Gauss map of all normals on the boundary, Minkowski-producted (roughly, expanded) by a cone whose opening angle is determined by the ratio of indices of refraction of the involved media. Finally, an intervening ray might encounter a **peak** in radiance centered on a specular highlight (for example, at point **p** in Figure 1-d). Radiance peaks occur at extremal paths, that is, those of locally minimal or maximal optical length (see, e.g., [21] for a discussion of such paths in the context of ray tracing algorithms).

The only way in which two invocations of `Trace` may be distinguished is through distinct return values for `Intersect` (case **a**; case **b**) or `Shade` (self-blocking; case **c**; case **d**). Thus, when texture maps are not present, these four phenomena are the only sources of aliasing in classical ray tracing.

5 Ray Trees and Isomorphism

Each invocation of `Trace()` computes a radiance value, and can be thought of as inducing a *ray tree* [10] (Figure 2). Each tree leaf corresponds to an aggregation of direct illumination from a light source or the background, is labeled with a unique identifier for that source, and stores the radiance collected from the source. Each internal node, including the root, corresponds to a recursive invocation of `Trace()` to gather radiance reflected by, or transmitted through, some scene object. Such nodes are labeled with a unique identifer for the intervening object, and store the radiance gathered from the object (in the figure, ray paths are marked with arrows indicating the direction along which radiance is gathered). Each ray tree edge corresponds to a call to `Intersect()`; we will find it useful to consider *ordered* lists of edges as representing a sequence of such calls.

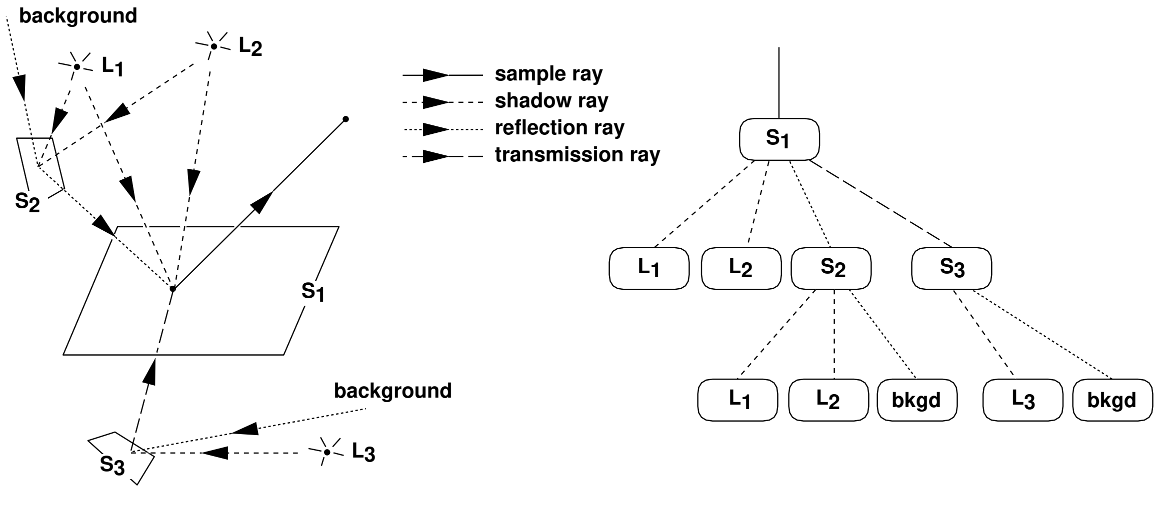

Figure 2: The ray tree induced by a call to `Trace()`.

Two ray trees $\mathcal{A}$ and $\mathcal{B}$ are *isomorphic* if there exists a one-to-one correspondence Π between their nodes, and if, whenever some node $\mathcal{N}_\mathcal{A}$ in $\mathcal{A}$ has as its k^{th} subtree $\mathcal{M}_\mathcal{A}$, its corresponding node $\Pi(\mathcal{N}_\mathcal{A})$ in $\mathcal{B}$ has as its k^{th} subtree the node $\Pi(\mathcal{M}_\mathcal{A})$. We say that isomorphic ray trees have the same *topology*. Together, every surface point and viewing direction of that point (for a fixed surface, a 4D range) induce a ray tree of some topology. A crucial observation is that **radiance or its derivatives change discontinuously only when the topology of the ray tree changes** (Figure 3). Imagine a 2D pencil of rays through a neighborhood of points on some directly illuminated surface (Figure 3-a). An observer experiences a radiance discontinuity when viewing the hard shadow edge cast by the occluder. Radiance discontinuities occur under purely directional variation as well (Figure 3-b). Consider a 2D pencil of rays through a fixed point **p** on some indirectly illuminated surface. An observer viewing **p** from varying directions will experience a radiance discontinuity when the reflection of object S_2 starts (or stops) containing **p**.

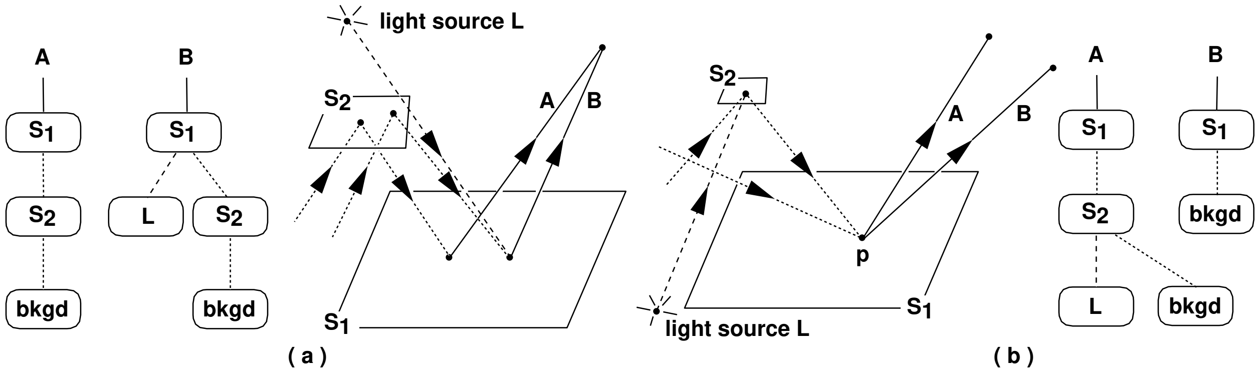

Figure 3: Radiance discontinuities occur where the ray tree topology changes.

6 Radiance Interpolants

We wish to generate ray traced imagery using far fewer radiance samples than would classical ray tracing, while retaining the following "correctness" property: for any ray, the radiance value computed along the ray should have no more than ϵ relative error with respect to the radiance that would be computed by spawning a recursive ray tree, and gathering radiance as in §2. To do so, we lazily construct *radiance interpolants in ray space*, then judiciously substitute queries to such interpolants for most (but not all) radiance sampling operations.

6.1 Ray Space

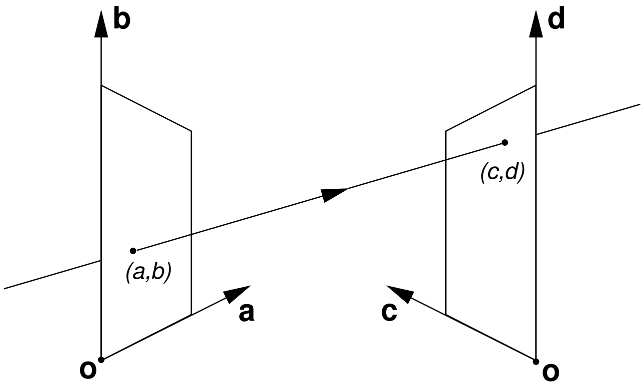

Figure 4: A ray coordinatized by its 2D intercepts with two reference planes.

Given a pair of reference planes, those rays parallel to neither plane can be *coordinatized* by a pair of 2D intercepts (e.g., *(a,b)* and *(c,d)* in Figure 4). Thus, the rays through a pair of bounded regions (e.g., squares) on two reference planes form a four-dimensional set.

Given a ray described in 3D, the ray's 4D ray-space coordinates can be defined with reference to a box around the object which ray intersects. Every such box has six "face-pairs" of opposing (parallel) faces. To represent a ray, we select the face-pair that is maximally perpendicular to the ray; the two faces involved must be expanded to capture all rays that could intersect the object.

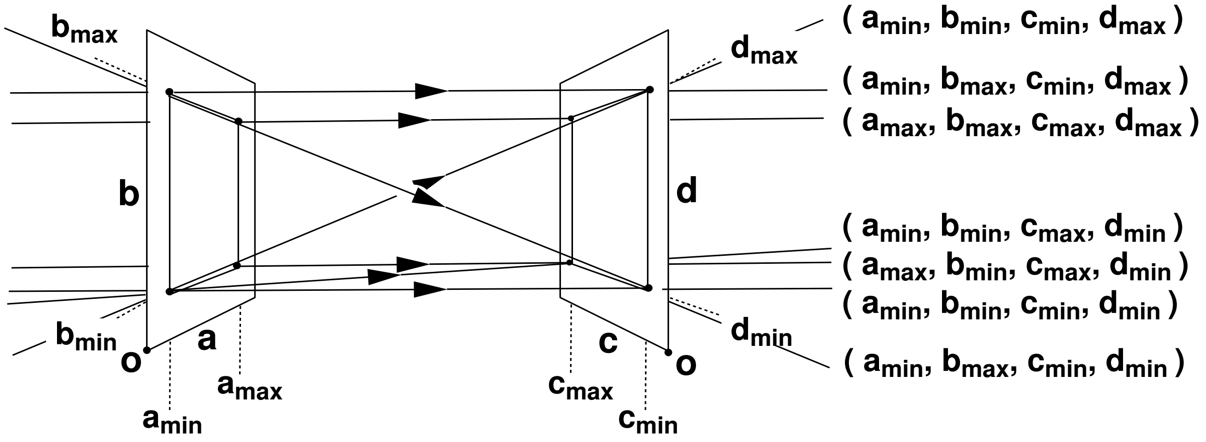

Figure 5: A ray subregion corresponds to a 4D hypercube.

All rays in a subregion of ray space can be generated by allowing (a, b, c, d) to range over some interval $[a_{min}..a_{max}] \times [b_{min}..b_{max}] \times [c_{min}..c_{max}] \times [d_{min}..d_{max}]$. Conversely, one may determine whether a given ray (a, b, c, d) lies inside a region in ray space by performing four interval tests, one for each of the ray coordinates a, b, c, d. (Figure 5 depicts seven of the sixteen extreme rays through a pair of squares.) Observe that the set of rays through a pair of convex objects, although convex in 4D rayspace, has an hourglass nature in 3D; when considered only inside the convex hull of the objects, this set has been called a "shaft" [12].

6.2 Radiance Interpolation

We now have a mechanism by which all rays through some volume of space can be represented, and by which a sample or "query" ray can be checked for inclusion in a represented set. However, rays spawned by a ray tracing algorithm not only have geometric attributes, but can also be thought of as "carrying" radiance, a physical quantity. Moreover, as we have seen that as a sample ray changes position or direction, the radiance along that ray (as computed by classical ray tracing) changes smoothly, except at discontinuities caused by gaps, blockers, or funnels. Thus, it should be possible to interpolate between radiance samples to produce values which are averages of those associated with the samples. When is it sensible to interpolate radiance samples?

Intuitively, if one is to avoid interpolating across a radiance discontinuity, one must average only samples which have been derived from isomorphic ray trees. This is a necessary, but not sufficient, condition for discontinuity-free radiance interpolation since intermediate rays may induce ray trees that are not isomorphic to those of the sample rays (cf. Figures 1-a, b and c). A stronger condition is that *all rays in the ray space convex hull* of the sample set induce isomorphic ray trees, banishing any radiance discontinuities there. We call such a set an *isotopy* of rays.

It remains to elucidate an interpolant mechanism that, given an isotopic region of rayspace, produces for every query ray in the region radiance values within ϵ of those that would be computed by classical ray tracing. The following subsections detail the identification of isotopic regions, and the construction of radiance interpolants.

6.3 Identifying Ray Isotopies

Imagine a scene comprised of convex objects. Recall that ray isotopies can be interrupted by only three configurations: gaps, blockers, and funnels (cf. Figure 1).

6.3.1 Detecting Gaps

Since convex objects cannot have holes, any two sample ray trees which span a gap must encounter distinct objects, and can not be isomorphic. Thus, the presence of gaps inside a ray subregion can be detected by comparing the topologies of the region's extremal ray trees.

6.3.2 Detecting Blockers

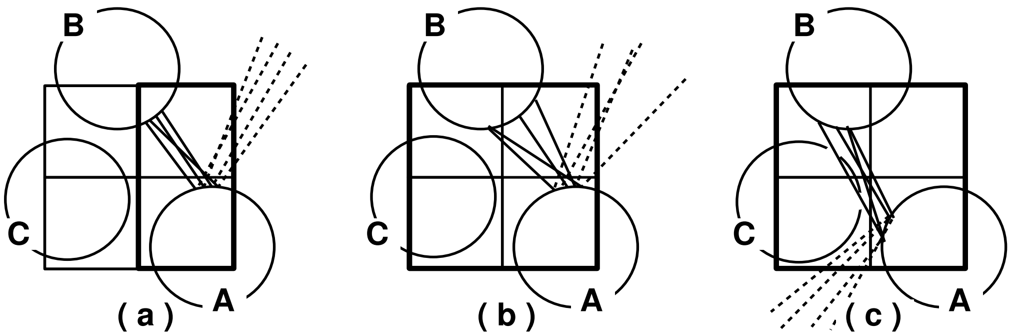

Figure 6: Establishing the absence (a) or potential presence (b, c) of blockers.

Detecting blockers is slightly more involved. To do so, we employ a dynamic version of *shaft culling* [12], which was originally developed for use in a radiosity algorithm based on ray-casting. The pseudocode of §2 invokes `Intersect()` to determine the first intersection of a spawned ray with a scene object. For efficiency, `Intersect()` is implemented as a query on a hierarchical spatial partition [10]. Now, every ray between two points A and B propagates through some well-defined set of cells in the spatial subdivision. Every two cells have some "least common ancestor" in the subdivision; that is, the deepest (smallest) cell which contains them both. Thus, given the extreme rays in some convex ray space region R, all of which connect scene objects A and B, we search the least common ancestor (shown in bold in Figure 6-b and c) of all cells touched by the rays for any objects C other than A or B. If none are found, then by convexity it must be true that no ray in the interior of R meets any object other than A or B.

Note that this blocker identification technique is *conservative*, in that it may sometimes report irrelevant objects as blockers (as in Figure 6-b, in which object C is reported as a blocker even though it is disjoint from the depicted shaft of sample rays from A to B). However, it always identifies objects that are truly blockers (such as the sphere C in Figure 6-c), even though they may be intersected by none of the sample rays.

6.3.3 Detecting Funnels

Finally, we address the issue of detecting funnels; that is, regions of ray space in which extremal samples are (are not) totally internally reflecting across a dense/sparse interface, whereas interpolated samples are not (are) totally internally reflecting (cf. Figure 1). Unfortunately, it is easy to exhibit instances of both positive and negative funnels (Figure 7); there appears to be no simple, general method for detecting their presence. However, a *conservative* subdivision strategy preserves the correctness of our approach: we simply label any raytree containing a dense/sparse interface as non-isotopic. This may cause excessive sampling; however, it will never allow interpolation across a funnel – that is, across a region in which some samples are totally internally reflecting and some are not.

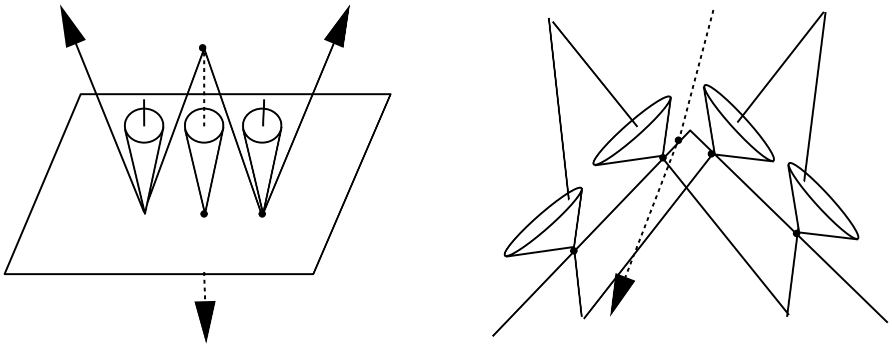

Figure 7: Totally-internally reflecting (TIR) samples may interpolate to non-TIR samples (bold and dashed, at left), and vice-versa (right), even for convex primitives.

We note that existing beam-tracing [16] cone-tracing [1], and pencil-tracing [26] algorithms encounter this problem in a different guise; refraction causes linear descriptions of ray bundles to become nonlinear. Each of these algorithms employed geometric approximation techniques to trace refracted ray bundles, but showed no bounds for the geometric and radiometric errors so incurred.

6.4 Bounding Interpolant Error

Ray tree leaves correspond to a recursion termination (zero gathered radiance), a ray's escape from the scene (to background, collecting constant gathered radiance), or the evaluation of radiance from a point light source (Phong shading, non-constant radiance). The latter cases, appropriately weighted, are the only sources of radiance at the root of the ray tree. Thus, to bound the radiance that could arise from any query ray in some isotopy, we must bound the radiance computed at the ray tree leaves, then multiply these bounds by the aggregate weight of the subtree (note that this weight is the same for all ray trees in the isotopy).

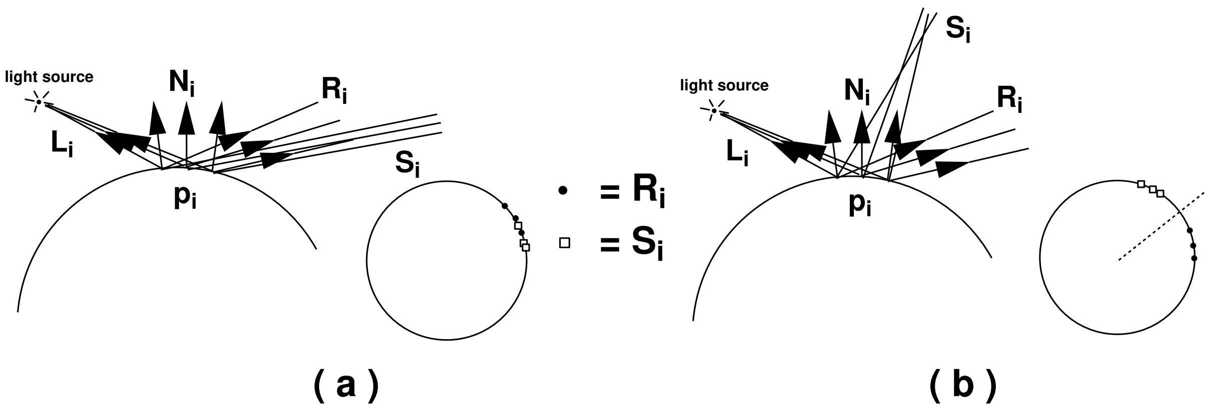

Figure 8: A radiance peak exists if and only if the reflection directions R_i and the sample directions S_i are not separable on the Gauss map.

The radiance function is unimodal for a fixed viewing position or a fixed viewing direction. However, when the sample direction varies, radiance can have many maxima. Bounding the radiance exactly within some ray isotopy could be done by identifying a ray within the region for which $N \cdot H$ (or $N \cdot L$) is maximized, then evaluating radiance along that ray. However, again we formulate a simpler, conservative strategy. We simply subdivide until the extreme value $N \cdot H$ is **known to occur at one of the samples**; that is, until any maxima are banished from the interior of the isotopy. We do so by analyzing the Gauss map of the surface normals (Figure 8).

For an isotopic sample set S_i, define the sample rays' points of intersection with some primitive as the p_i, and the induced light vectors L_i and normals N_i. A radiance peak can occur only if there exists some interpolated sample direction which is the reflection of its induced light source direction. This can occur if and only if the Gauss map of the sample directions intersects that of the reflection directions (Figure 8-a). Otherwise, the maximum radiance arises at one of the S_i. A simple conservative test for non-intersection of two Gauss maps is that of strict linear separability; that is, two sets A_i and B_j of vectors are strictly linearly separable iff there exists a vector D such that $D \cdot A_i < 0 \; \forall i$ and $D \cdot B_j > 0 \; \forall j$. This is a 3D linear programming problem, and can be solved in time linear in the total number of vectors [20, 24].

7 Algorithm Overview

In this section, we present a detailed description of our algorithm for lazy radiance interpolation (refer to the pseudocode below). We have implemented the algorithm within the context of a Whitted ray-tracer, `ivray`[1].

We modified the initial ray-tracer of §2 to call `IpTrace` for each pixel in the image. `IpTrace` first checks for an object intersection. If there is none, the background color is returned. If an object is intersected, `TryInterpolate` attempts to interpolate radiance for the given ray. If it succeeds, the interpolated value is returned. Otherwise, `Shade` is invoked to perform an "analytic" radiance computation.

`TryInterpolate` first invokes `FindLTIntercept` to identify which face-pair is intercepted by the sample ray, and the ray's (a, b, c, d) intercepts on that face-pair. It then finds the appropriate leaf linetree associated with that *(object, facepair)* with intercepts (a, b, c, d). If the interpolation predicate for that leaf is `TRUE`, interpolation is performed and a radiance value returned for the given sample. Otherwise, if the (leaf) linetree's depth does not exceed LT_MAXDEPTH, it is split. Construction of an interpolant is then attempted for the (single) new child which contains the original sample ray. In this fashion, interpolants are "lazily" constructed by the algorithm.

`BuildInterpolant` shoots 16 interpolant rays. If all 16 hit the object for which the interpolant is being built, `Shade` is called for each interpolant ray. If all 16 `Shade` calls produce isomorphic raytrees, and all 16 raytrees satisfy the blocker, specularity, and epsilon tests, the predicate associated with the linetree cell is set to `TRUE`. Subsequent calls to this linetree cell will be successfully interpolated.

[1] `ivray` was developed for the MIT undergraduate graphics course 6.837.

```
struct LTIntercept {
   int   facepair; // [0, 5]
   float in[4];    // 4 coords
};

// Computes radiance values
// for Ray R and scene S
Radiance IpTrace (
     Ray R, scene S)
{
  HitStructure H;
  Radiance     col;

  if (Intersect (R, S, &H))
    if (TryInterpolate (H, R, &col, S))
      return col;
    else
      return Shade (H, R, eta, maxD, 1.0)
  else
    return Background (R, D)
}

// Tries to interpolate radiance
// for Ray R with Intersection H
// If success, place result in col
//             return TRUE
bool TryInterpolate (HitStructure H,
    Ray R, Radiance *col, Scene S)
{
  Intercept in;
  LineTree  *l, *lchild;

  FindLTIntercept (H, &in, R)
  l = LTFind (H.obj, in.facepair, in.in)

  if (l->predicate) { // success
     Interpolate (in, col)
     return TRUE
  }
  else {
     if (l->depth < LT_MAXDEPTH) {
        lchild = Split (l, in)
        BuildInterpolant (H, lchild, S)
     }
     return FALSE
  }
}
```

```
// Tries to build interpolant for
// LineTree lt associated with object
// Hin.obj
// lt->predicate is set to TRUE/FALSE
// depending on whether or not an
// interpolant is successfully built
void BuildInterpolant (HitStructure Hin,
      LineTree *lt, Scene S)
{
  Ray ray[16];
  RayTree raytree[16];
  Radiance col[16];

  for (i = 0; i < 16; i++) {
    Generate ith interpolant ray of lt
    if (!Intersect (ray, S, &H[i]) ||
        (H[i].obj != Hin.obj)) {
          lt->predicate = FALSE;
          return;
    }
  }

  col[0] = Shade (..., &raytree[0]);
  for (i = each of 16 interpolant rays) {
      Shade (..., &raytree[i]);
      if (CmpRayTree (raytree[i], raytree[0]))
        continue;
      else {
        lt->predicate = FALSE;
        return;
      }
   }

   if (!CheckBlockers (raytree, S) ||
       !CheckSpecularity (raytree, S) ||
       !TestEpsilon (raytree, epsilon)) {
          lt->predicate = FALSE;
          return;
   }

   lt->predicate = TRUE;
   SetInterpolantCol (lt, col);
}
```

We employ a sphere illuminated by a single light source to illustrate the algorithm's operation. Figure 9 depicts how our algorithm isolates all possible sources of aliasing. The yellow and green areas correspond to regions of subdivision due to non-isomorphic raytrees – due to silhouetting and self-shadowing, respectively. The cyan region corresponds to subdivision around the specular highlight on the sphere (i.e., failure of the specularity test). The purple region corresponds to subdivision due to the relative error between the samples exceeding epsilon (i.e., failure of the epsilon test). Figure 10 shows the aliasing artifacts that can arise if

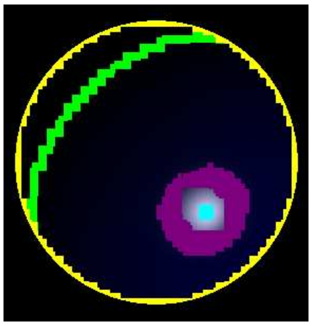

Figure 9: Color-coded subdivision regions.

Figure 10: Sphere with aliasing artifact.

Figure 11: Sphere image, $\epsilon = .25$.

specular highlights are not detected. The image was generated by deactivating the specularity test in the algorithm. As can be seen, this causes interpolation to be incorrectly performed across the specular highlight. The final result of our algorithm in the case of this simple scene is depicted in Figure 11. These images are also shown in Color Plate 1.

A multiple-object scene is required to illustrate failure of the blocker test. Figure 12 depicts a face model with 7 primitives, rendered with $\epsilon = 0.5$. Figure 13 is color-coded as before. The red regions in the scene correspond to regions of subdivision due to potential blockers. Both images are reproduced in Color Plate 1.

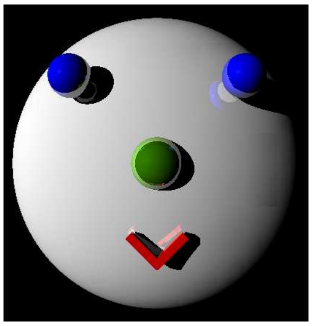

Figure 12: Output of algorithm with $\epsilon = 0.5$.

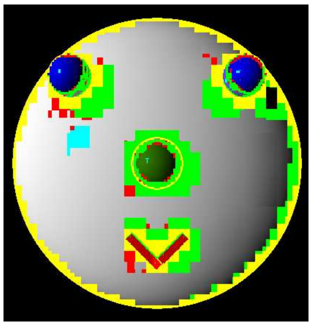

Figure 13: Color-coded failure regions.

8 Results

This section discusses the algorithm's performance on two models (depicted in Figures 12 and 15). The clown-face model comprises 7 primitives. The room model is more complex, with 18 primitives. All images were rendered using the radiance interpolant algorithm, and are presented as well in Color Plates 1 and 2. The color-coded failure regions for the room and face model are presented in Figures 13 and 16 respectively.

8.1 Coherence

We instrumented both the initial and modified ray-tracer to record the number of calls to `Intersect` and `Shade` (§2). Table 1 presents the results for the room model (Plate 2), from which images were generated with $\epsilon = 0.1$.

The second column presents the ratio of intersect calls made by the interpolant algorithm to those made by the unmodified ray-tracer. The third column presents the ratio of shade calls made by the two ray-tracers.

Figure 14: Table, $\epsilon = 0.5$.

Figure 15: Table, $\epsilon = 0.25$.

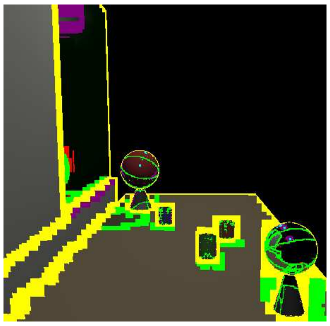

Figure 16: Color-coded subdivision.

Frame #	Intersect Ratio	Shade Ratio	Interpolation Success
1	1.28	1.19	76%
2	0.78	0.67	77%
3	0.66	0.52	78%
4	0.81	0.69	77%
5	0.74	0.60	78%

Table 1: Room model ratios, $\epsilon = 0.1$.

The last column presents the percentage of pixels for which interpolation was successful. The first row presents results for the initial rendering of the scene. The next 2 rows correspond to small movements in the eye forward (along the viewing direction). The last 2 rows correspond to small movements of the eye sideways. Similarly, Table 2 presents the results for the clown face image depicted in Plate 1, with $\epsilon = 0.5$.

The results demonstrate the algorithm's use of spatial and temporal coherence. The first time the scene is traced, our algorithm builds up a database of interpolant information by shooting interpolant rays. In Table 1, the first row shows that the first time the scene is rendered our algorithm increases the intersection and shade computations by 28% and 19% respectively. Subsequent small, coherent movements of the eye along the viewing direction reduce the number of intersections and shades by up to 34% and 48% respectively. Moving forward or sideways can cause hitherto unexplored parts of the scene to be exposed, and interpolants have to be built for these exposed regions. In this scene, moving along the viewing direction tends to use interpolants that are already built, while moving sideways exposes more of the scene. Therefore, the ratio of intersects and shades increase when moving sideways. However, we find that moving sideways makes significant use of the interpolants, decreasing the intersect and shades by up to 26% and 40% respectively.

The results in Table 2 correspond to the face model and they agree with those of the room model. However, since the face model has smaller error regions than the room model, the benefits are more substantial. The number of intersects decreases by up to 40%, and the number of shades decreases by up to 55%. We also see that the first time the scene is rendered, successful interpolation actually causes the number of shades to decrease by 8% (even though the number of intersect calls increases).

8.2 Error Bounds

We also varied ϵ to observe its impact on image quality. Figure 14 depicts an image generated with $\epsilon = 0.5$. Some artifacts arise in the mirror and along the bottom of the mirror where interpolation takes place across the edge of the wall. However, reducing ϵ to 0.25 produces an image with no visible artifacts (Figure 15).

Frame #	Intersect Ratio	Shade Ratio	Interpolation Success
1	1.12	0.92	71%
2	0.65	0.45	72%
3	0.59	0.39	72%
4	0.65	0.45	71%
5	0.64	0.43	72%

Table 2: Face model ratios, $\epsilon = 0.5$.

Epsilon	Intersect Ratio	Shade Ratio	Interpolation Success
1.00	1.35	1.32	77%
0.50	1.40	1.38	76%
0.25	1.41	1.38	76%
0.00	2.42	3.3	0%

Table 3: Ratios as ϵ is varied.

Table 3 summarizes the quantitative impact a range of ϵ values have on the results. As ϵ values decrease, the number of intersect and shade operations increase, and the success rates drop. However, we find that the change is not very dramatic, and good performance can be achieved in a scene with no visible artifacts. It is also useful to note that decreasing ϵ from 1.0 to 0.25 does not increase the work substantially, while achieving good image quality.

9 Conclusion

We described a novel scheme for lazy computation of information about radiance, organized by a per-surface, hierarchical subdivision of linespace. Our scheme first attempts to satisfy radiance queries by searching for a bounded-error interpolant containing the query ray. The generation of a new interpolant is attempted only if no such interpolant exists.

We built a prototype implementation of this strategy, and showed that it accelerates a classical ray-tracing computation by reducing the number of primitive geometric and radiometric operations performed. Our scheme exploits both screen-space, object-space, and temporal coherence, and is the first to directly trade off guaranteed error bounds for rendering time in a ray tracing operation. Finally, the approach accelerates not only the generation of single images but also the generation of images for a sequence of smoothly changing eyepoints.

In the future, we intend to extend this approach in a number of ways. First, we will examine its application to environments that exhibit a full range of BRDFs. Second, since it treats geometric (ray-casting) and radiometric (shading) computations as decoupled primitive operations, it can easily incorporate improved ray-casting algorithms as they arise. Third, we plan to examine the problem of rendering images for a sequence of smoothly varying viewpoints at interactive rates. Finally, we will investigate algorithms for persistently storing the radiance values for portions of the scene that have already been traversed, and discarding such values when they have outlived their usefulness.

10 Acknowledgments

Michael Werman's helpful comments and advice are gratefully acknowledged. We would also like to thank Andrew Myers for his useful suggestions.

References

[1] Amanatides, J. Ray Tracing with Cones. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), H. Christiansen, Ed., vol. 18, pp. 129–135.

[2] Appel, A. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of SJCC* (1968), Thompson Books, Washington, D.C., pp. 37–45.

[3] Arvo, J., and Kirk, D. B. Fast Ray Tracing by Ray Classification. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), M. C. Stone, Ed., vol. 21, pp. 55–64.

[4] 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.

[5] Cook, R. L., Porter, T., and Carpenter, L. Distributed Ray Tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), vol. 18, pp. 137–45.

[6] Fujimoto, A., and Iwata, K. Accelerated Ray Tracing. *Computer Graphics: Visual Technology and Art (Proc. Computer Graphics Tokyo '85)* (1985), 41–65.

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

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

[9] Glassner, A. S. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications 8*, 2 (Mar. 1988), 60–70.

[10] Glassner, A. S., Ed. *An Introduction to Ray Tracing.* Academic Press, 1989.

[11] 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.

[12] Haines, E., and Wallace, J. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proc. 2^{nd} Eurographics Workshop on Rendering* (May 1991).

[13] Hall, R. A., and Greenberg, D. P. A Testbed for Realistic Image Synthesis. *IEEE Computer Graphics and Applications 3* (Nov. 1983), 10–20.

[14] Hanrahan, P., Salzman, D., and Aupperle, L. A Rapid Hierarchical Radiosity Algorithm. *Computer Graphics (Proc. Siggraph '91) 25*, 4 (1991), 197–206.

[15] Heckbert, P. *Introduction to Ray Tracing.* Academic Press, 1989, ch. Writing a Ray Tracer, pp. 263 – 294.

[16] Heckbert, P., and Hanrahan, P. Beam Tracing Polygonal Objects. *Computer Graphics (Proc. Siggraph '84) 18*, 3 (1984), 119–127.

[17] Kajiya, J. T. The Rendering Equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 143–150.

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

[19] 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.

[20] Megiddo, N. Linear-time algorithms for Linear Programming in R^3 and Related Problems. *SIAM Journal Computing 12* (1983), 759–776.

[21] Mitchell, D. P., and Hanrahan, P. Illumination From Curved Reflectors. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (July 1992), E. E. Catmull, Ed., vol. 26, pp. 283–291.

[22] 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.

[23] Phong, B.-T. Illumination for Computer Generated Pictures. *Communications of the ACM 18*, 6 (June 1975), 311–317.

[24] Seidel, R. Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry* (1991), 423–434.

[25] Séquin, C. H., and Smyrl, E. K. Parameterized Ray Tracing. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), J. Lane, Ed., vol. 23, pp. 307–314.

[26] Shinya, M., Takahashi, T., and Naito, S. Principles and Applications of Pencil Tracing. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), M. C. Stone, Ed., vol. 21, pp. 45–54.

[27] Sillion, F., Arvo, J., Westin, S., and Greenberg, D. A Global Illumination Solution for General Reflectance Distributions. *Computer Graphics (SIGGRAPH '91 Proceedings) 25*, 4 (July 1991), 187–196.

[28] 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.

[29] Sillion, F., and Puech, C. *Radiosity and Global Illumination.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1994.

[30] 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.

[31] 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), J. Dill, Ed., vol. 22, pp. 85–92.

[32] Warnock, J. A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures. Tech. Rep. TR 4–15, NTIS AD-733 671, University of Utah, Computer Science Department, 1969.

[33] Whitted, T. An Improved Illumination Model for Shading Display. *CACM 23*, 6 (1980), 343–349.