# Conservative Radiance Interpolants for Ray Tracing

Kavita Bala*
Synthetic Imagery and Visual Simulation Groups
MIT Laboratory for Computer Science

## 1 Introduction

Ray tracing is a well-known graphics technique used to generate images. Ray tracers compute radiance (units POWER per source AREA per receiver STERADIAN) for each ray originating at the eye and passing through each pixel of the image. Ray traced images achieve realism by considering diffuse and specular effects locally, as well as reflection, and refraction effects globally. Thus, the radiance function for a scene is a complex 4D function. Ray tracers sample this 4D space sparsely with respect to a fixed viewpoint, and output a 2D image array of radiance values.

In this abstract, we present a technique to exploit the temporal and spatial coherence in ray traced images to accelerate rendering. Our scheme calculates radiance samples lazily for each surface in the scene, controlled by a user-specified error bound $\epsilon$. Radiance values for rays that lie within the 4D convex hull of the sample rays are interpolated from the radiance samples. Thus, our scheme accelerates ray tracing by replacing expensive radiance computations by fast interpolation when possible.

## 2 Algorithm

We first present a brief overview of classical *backward* ray tracing. For every pixel in the scene, the ray tracer shoots a ray from the eye into the scene, to find which object is intersected. If there is no intersection, the background color is returned; otherwise, the ray tracer shades the point locally using the diffuse and specular properties of the material. The ray tracer then adds in the global component of radiance associated with reflected and refracted rays. Thus, the ray tracer uses two primitive operations: `Intersect`, which computes the intersection of a ray with the objects in the scene, and `Shade`, which computes the radiance at a point by local and global recursive computations of the radiance associated with reflected and refracted rays. Thus, `Shade` recursively calls `Intersect` and `Shade`, and these calls make ray tracing expensive.

Our algorithm for lazy radiance interpolation is as follows: For every pixel in the image, the ray tracer invokes `Intersect`. If an object is intersected, the ray tracer tries to find radiance samples to interpolate the radiance for the current ray. If it cannot find samples, the ray tracer calls `Shade` recursively. The techniques used to build interpolants and check their validity are described in the next two sections.

## 3 Building Interpolants

Our scheme builds radiance samples lazily and interpolates radiance values using these samples. In order to do this, we require a way to classify and index all rays.

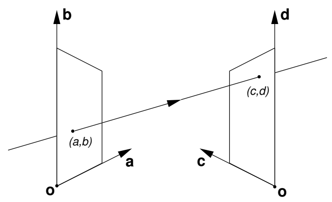


Figure 1: Ray coordinatization.

A ray can be *coordinatized* by a pair of 2D intercepts on two reference planes (e.g., *(a,b)* and *(c,d)* in Figure 1). Thus, every ray can be expressed in terms of its 4D coordinates in line-space. Given a ray (in 5D space), the ray's 4D ray-space coordinates can be defined with reference to a box around the object that the ray intersects.

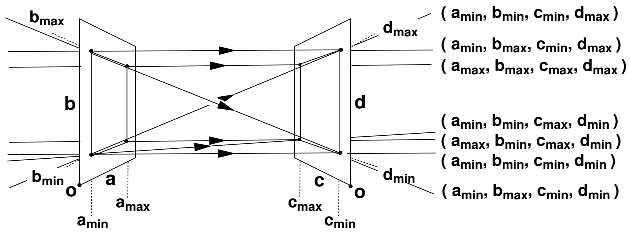


Figure 2: Ray subspace is a 4D hypercube.

All rays in a subspace 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, we can 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 2 depicts seven of the sixteen extreme rays through a pair of bounded planes. Thus, all the rays within the 4D convex hull of these extreme rays can be interpolated using the 16 sample values depicted

*Supervised by Prof. Teller and Prof. Dorsey. MIT office: NE43-219. E-mail: kaybee@graphics.lcs.mit.edu. Web: http://graphics.lcs.mit.edu/kaybee.

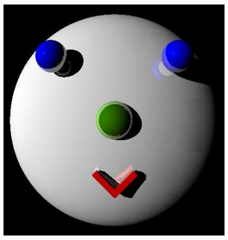

Figure 3: Output of algorithm with $\epsilon = 0.5$.

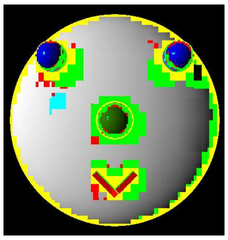

Figure 4: Color-coded failure regions.

| Frame # | Intersect Ratio | Shade Ratio |
|---|---|---|
| 1 | 1.12 | 0.92 |
| 2 | 0.65 | 0.45 |
| 3 | 0.59 | 0.39 |
| 4 | 0.65 | 0.45 |
| 5 | 0.64 | 0.43 |

Table 1: Clown face model ratios, $\epsilon = 0.5$.

in Figure 2. We store the 16 radiance samples associated with an interpolant in 4D trees called linetrees.

Our algorithm first identifies which object is intersected by the current ray, and finds its $(a, b, c, d)$ intercepts. We then find the appropriate linetree associated with that *object* with intercepts $(a, b, c, d)$. If samples exist for that linetree cell, we perform quadrilinear interpolation, and return a radiance value for the current ray. Otherwise, the ray tracer builds interpolants by shooting 16 rays and storing their associated radiance values in the linetree. In this fashion, interpolants are lazily constructed by our algorithm.

## 4 Avoiding Erroneous Interpolation

Our algorithm guarantees that the radiance we compute for a ray does not not have more than $\epsilon$-relative error with respect to the radiance computed using classical ray tracing. Therefore, we have to avoid the traditional problem with sampling, called *aliasing*, of missing important detail by sampling too coarsely. We have identified regions where aliasing can arise, and have developed efficient methods to detect them. Aliasing errors arise if interpolant sample rays intersect different objects, or miss specular highlights, or differ by greater than $\epsilon$ relative error etc. More details are available on how to detect and handle aliasing problems in [1].

When the ray tracer builds interpolants it tests that each of the 16 sample rays satisfies all the aliasing tests. If any of the aliasing tests fails it recursively subdivides the linetree and lazily builds interpolants for the children.

## 5 Results

We have implemented our algorithm within the context of a Whitted ray-tracer, `ivray`[1]. This section discusses the algorithm's performance on the clown face model in Figure 3, which comprises 7 primitives. We instrumented both the initial and modified ray tracer to record the number of calls to `Intersect` and `Shade`. Table 1 presents the results for the clown model with $\epsilon = 0.5$. Figure 4 presents the regions where the aliasing tests fail and cause finer subdivision of the linetree[2].

In Table 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. 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.

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 samples by shooting interpolant rays. The first row shows that our algorithm increases the intersection computations by 12%, but decreases the shade computations by 8%. Thus, the algorithm uses spatial coherence to decrease the amount of work done by the ray tracer even while the database is being populated.

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. Even so, we find that moving forward makes significant use of existing interpolants reducing the number of `Intersects` and `Shades` by up to 41% and 61% respectively, while moving sideways also decreases them by up to 36% and 57% respectively.

## 6 Conclusions

We have presented a scheme for lazy construction of interpolants for ray tracing. We have also presented an algorithm to build, store and use the interpolants to accelerate ray tracing. Our results indicate that our algorithm improves performance by decreasing `Intersect` calls by up to 40%, and `Shade` calls by up to 60%. We plan to explore the use of this algorithm for interactive walkthroughs.

## References

[1] Teller, S., Bala, K., and Dorsey, J. Conservative Radiance Interpolants for Ray Tracing. In *Seventh Eurographics Workshop on Rendering* (June 15–17 1996).

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

[2] Refer to http://graphics.lcs.mit.edu/kaybee for color plates.