Multiresolution Radiosity Caching for Efficient Preview and Final Quality Global Illumination in Movies

Per H. Christensen George Harker Jonathan Shade Brenden Schubert Dana Batali

Pixar Technical Memo #12-06 — July, 2012

Pixar Animation Studios

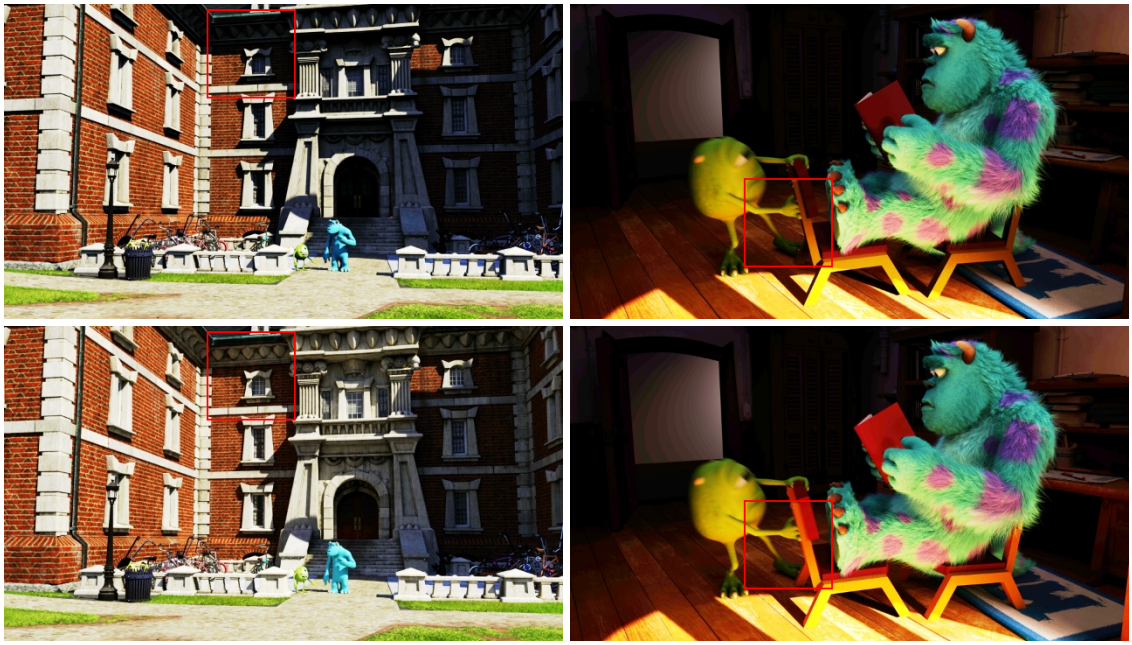

Figure 1: *Two scenes from the CG movie 'Monsters University'. Top row: direct illumination. Bottom row: global illumination — images such as these render more than 30 times faster with radiosity caching.* © *Disney/Pixar.*

Abstract

We present a multiresolution radiosity caching method that allows global illumination to be computed efficiently in a single pass in complex CG movie production scenes.

For distribution ray tracing in production scenes, the bottleneck is the time spent evaluating complex shaders at the ray hit points. We speed up this shader evaluation time for global illumination by separating out the view-independent component and caching its result — the radiosity. Our cache contains three resolutions of the radiosity; the resolution used for a given ray is selected using the ray's differential. The resulting single-pass global illumination method is fast and flexible enough to be used in movie production, both for interactive lighting design and final rendering. It is currently being used in production at several studios.

The multiresolution cache is also used to store shader opacity results for faster ray-traced shadows, ambient occlusion and volume extinction, and to store irradiance for efficient ray-traced subsurface scattering.

Keywords: Global illumination, color bleeding, distribution ray tracing, complex geometry, complex shaders, multiresolution cache, ray differentials, radiosity, opacity, volume extinction, subsurface scattering, interactive rendering, movie production.

1 Introduction

Recently there has been a surge in the use of global illumination ("color bleeding") in CG movies and special effects, motivated by the faster lighting design and more realistic lighting that it enables. The most widely used methods are distribution ray tracing, path tracing, and point-based global illumination; for an overview of these please see the course notes by Křivánek et al. [2010].

The ray-traced global illumination methods (distribution ray tracing and path tracing) are slow if the scenes have not only very complex base geometry but also very complex and programmable displacement shaders, light source shaders, and surface shaders that have to be executed at every ray hit point.

Point-based global illumination (PBGI) reduces this computation bottleneck since the shaders are only evaluated at the surface tessellation vertices and only twice: at point cloud generation time and for the final render. Other advantages of PBGI are that it is fast and has no noise. Disadvantages are bias and aliasing if coarse settings are used, having to shade the entire scene (including off-camera parts) during point cloud generation, file I/O overhead, pipeline management, and non-interactivity. The goal of our multiresolution radiosity cache is to get the same reduction in shader evaluations, but with a single-pass distribution ray-tracing method more suitable for interactive rendering and movie production pipelines.

The motivating observation is that for ray-traced global illumination in production scenes with complex geometry and shaders, the bottleneck is not the "raw" ray tracing time (spatial acceleration data structure traversal and ray–surface intersection tests), but the time spent evaluating the displacement, light source, and surface shaders at the ray hit points. This shader evaluation time includes texture map lookups, procedural texture generation, shadow calculation, BRDF evaluation, shader set-up and execution overhead, calls to external plug-ins, etc. We reduce this time by separating out the view-independent shader component — radiosity — needed for global illumination and caching it. During distribution ray tracing global illumination these radiosities are computed on demand and reused many times. As a by-product of caching these shading results, the number of shadow rays is reduced. For further efficiency, the radiosity is computed for an entire grid (a coherent batch of shading points) at a time, allowing coherency in texture map lookups, coherent shadow rays, etc.

The radiosity cache is implemented in Pixar's PhotoRealistic RenderMan renderer that supports both progressive ray tracing and REYES-style micropolygon rendering. The cache contains multiple resolutions of the radiosity on the surface patches in the scene. The appropriate resolution for each cache lookup is selected using ray differentials.

The resulting single-pass global illumination method is fast and flexible enough to be used in movie production, both for interactive material and lighting design and for final rendering. Radiosity caching gives speed-ups of 3–12 for simple scenes and more than 30 for production scenes. Figure 1 shows examples of images computed with the method.

We originally developed the multiresolution caching method in order to accelerate global illumination, but it turns out that the caching method can be applied in other parts of the shading pipeline as well — wherever there are view-independent shading results that can be reused. We take advantage of this by also caching surface and volume opacity and caching irradiance for ray-traced subsurface scattering.

2 Related Work

Our method builds on prior global illumination work, particularly in rendering of very complex scenes for movie production. For a general introduction to global illumination, please see e.g. the textbooks by Pharr and Humphreys [2010] and Dutré et al. [2003].

2.1 Global Illumination in Movies

The first use of global illumination in a feature-length movie was for the movie 'Shrek 2' [Tabellion and Lamorlette 2004]. They computed direct illumination and stored it as 2D texture maps on the surfaces, and then used distribution ray tracing to compute single-bounce global illumination. The use of 2D textures requires the surfaces to have a parameterization. The irradiance atlas method [Christensen and Batali 2004] is similar, but uses 3D texture maps ("brick maps") so the surfaces do not need a 2D parameterization. Both methods use two passes: one pass to compute the direct illumination and store it (as 2D or 3D texture maps), and one pass for final rendering.

Path tracing is a brute-force global illumination method that has been used on 'Monster House' and several other movies [Fajardo 2010]. The advantages of path tracing are that the algorithm is simple, runs in a single pass, and provides fast feedback during interactive lighting design. Its disadvantages are that the results are noisy, many shader evaluations are required, and it has an inherently slow convergence to low-noise final-quality images. It does not lend itself to caching and interpolation of (view-independent) shading results: each shading result is extremely noisy so cannot be reused without introducing significant bias.

Point-based global illumination is a fairly new, but widely used global illumination method in feature film production [Christensen 2008; Kontkanen et al. 2011]. It is fast and produces noise-free results. It was first used on the movies 'Pirates of the Caribbean 2' and 'Surf's Up', and has since been used for more than 40 other feature films. It is a multi-pass method: In the first pass, a point cloud is generated from directly illuminated micropolygons. In the second pass, $n-1$ bounces of global illumination are computed for the point cloud. (The second pass can be skipped if only a single bounce is needed.) In the third pass, the indirect illumination from the point cloud is computed and rendered. Due to its multipass nature, this method is not suitable for interactive lighting design. Also, its large point cloud files can be cumbersome to manage and require significant disk I/O.

The method of Meyer and Anderson [2006] computes a sequence of noisy distribution ray-traced global illumination images and filters them spatially and temporally to remove the noise. Since it requires working on an entire sequence of images at a time, it is not suitable for interactive rendering. We also believe our method fits better into the traditional CG movie production pipeline which usually renders one image at a time.

2.2 Other Related Global Illumination Methods

Our multiresolution radiosity representation is reminiscent of Heckbert's adaptive radiosity textures [Heckbert 1990]. His method traces rays from the light sources and stores the resulting radiosities in textures that are adaptively refined where there is large variation in the radiosity.

We consider our method a variation of Ward's irradiance cache [Ward et al. 1988; Ward Larson and Shakespeare 1998]. Both methods are based on distribution ray tracing, and both cache partial results to increase performance. Ward's method stores incident indirect illumination (irradiance) to individual points, while our method stores post-shading radiosity values from grids of micropolygon vertices. By caching an exitant quantity (radiosity), we capture and reuse both direct and indirect illumination and reduce the number of shader evaluations (including texture lookups and other calculations). By shading a grid of points together rather than shading individual points, our method is more suitable for REYES-style SIMD shader execution [Cook et al. 1987]; it amortizes shader start-up and execution overhead and ensures coherent texture map lookups, coherent shadow rays, etc. The two caching methods actually work well together: Ward's irradiance cache reduces the number of diffuse and shadow rays, and our radiosity cache reduces the number of shader evaluations and further reduces the shadow rays.

2.3 Shading Result Reuse

Recent work by Hou and Zhou [2011] shares our goal of reducing the number of shader evaluations, but for ray-traced specular and glossy reflections and refractions. Their method stores shading results in kd-trees and achieves speed-ups of 3.5. They reuse view-dependent shading results for any ray direction and texture filter size, giving arbitrarily wrong results; we only reuse view-independent results and distinguish between three different filter sizes. Their method is for specular and glossy reflections and refractions, while our method is for diffuse global illumination.

3 The Multiresolution Radiosity Cache

3.1 Cache Structure and Data

With the REYES rendering algorithm [Cook et al. 1987], large object surfaces are split into smaller surface patches. Each patch is then tessellated into a micropolygon grid. Shading is done at the grid vertices, and the shading results are interpolated over the micropolygons.

The radiosity cache contains representations of the radiosity on each surface patch at up to three different resolutions. The finest resolution contains radiosities for all the micropolygon vertices on the patch (the REYES shading points). The medium resolution contains radiosities at a subset of those vertices, for example every fourth vertex in u and v. The coarse resolution contains radiosities at only the four corners of the patch. See figure 2 for an example of the radiosity cache entries for a grid with 15×12 points. (This example shows a rectangular surface patch; we use a similar multiresolution representation for triangular patches.)

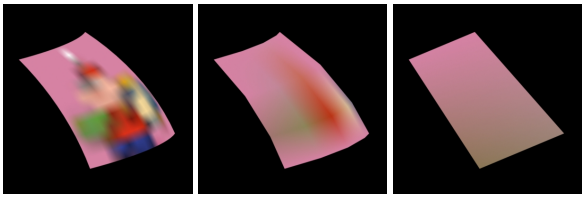

Figure 2: *Radiosity on a surface patch at three different resolutions: fine (15×12 points), medium (5×4 points), and coarse (2×2 points).*

Since a medium cache entry contains fewer points than a fine cache entry it uses less memory; since a coarse cache entry contains only four points it uses even less memory. So if cache memory is divided evenly between the three resolutions, the coarse cache has many more entries than the medium cache, which in turn has many more entries than the fine cache.

The multiresolution radiosity cache data structures are very similar to the multiresolution tessellation cache used for efficient ray–surface intersection in geometrically complex scenes [Christensen et al. 2003; Christensen et al. 2006]. One difference is that the vertex positions stored in the tessellation cache need full floating point precision, whereas the radiosities in the radiosity cache can be stored with less precision — we use half floats, but could instead use Ward's rgbe format [Ward 1991], at least for the coarse cache. Another difference is for instanced geometry: while the instances can share a master tessellation, each instance needs its own radiosity cache entry since the illumination, textures, and surface properties usually differ between instances.

The size of the radiosity cache can be defined by the user; the default size is 100 MB. If the memory were divided evenly with 33.3% for each resolution, the default capacities would be 336,000 coarse, 111,000 medium, and 19,000 fine grids. However, we have empirically found that enlarging the size of the fine cache to 50% (a default capacity of 252,000 coarse, 83,000 medium, and 29,000 fine grids) gives faster render times in most cases since the fine cache entries are more expensive to recompute if needed again after having been evicted from the cache.

We use an efficient cache implementation with low memory overhead and good multithreaded performance. Each thread has a local cache with pointers to a shared cache, which keeps a reference count for each of its entries. The cache replacement strategy is standard least-recently-used (LRU) replacement within each resolution.

3.2 Cache Lookups

We use ray differentials [Igehy 1999] to select the appropriate cache resolution for a ray hit: if the ray has only traveled a short distance or comes from specular reflection or refraction from a relatively flat surface, the differential is small and the fine cache resolution will be selected; if the ray comes from a distant highly curved surface and/or from a distant diffuse reflection the differential will be large and the medium or coarse cache resolution will be selected. A key observation in Christensen et al. [2003] is that for distribution ray tracing with a multiresolution cache, the lookups in the fine cache are coherent. The lookups in the medium and coarse caches are less coherent, but the capacity of those caches is higher since each entry is smaller. Consequently, high cache hit rates are obtained for all three cache resolutions.

Once we have determined which resolution to use for a ray hit, the cache for that resolution is queried to see whether it contains an appropriate entry for the radiosity on that side of the surface patch. If not, the view-independent part of the surface shader is run (along with the displacement shader and light source shaders) at the grid vertices and the resulting radiosities are stored in the cache.

Given the ray–patch intersection point and the grid of cached radiosities, the nearest radiosity values are then interpolated (using bilinear interpolation on quadrilateral micropolygon grids or barycentric interpolation on triangular micropolygon grids) to get the approximate radiosity at the intersection point.

A cache entry identifier consists of patch number, diffuse ray depth (to keep results from multiple bounces separate), motion segment (for multi-segment motion blur), surface side, and a timestamp. The time is updated when the illumination or shader parameters change in an interactive application; then the old cache entries which are no longer valid will gradually be evicted from the cache since their timestamps are too old.

4 Shading System Interface

The original RenderMan Shading Language (RSL) [Hanrahan and Lawson 1990; Apodaca and Gritz 2000] did not distinguish between view-independent and view-dependent surface shading results, instead relying on a relatively monolithic shader which produced two final combined results: color Ci and opacity Oi.

The original shading pipeline, which separates only displacement shading and surface shading (including lighting), is:

```
displacement()
surface()
```

If the renderer only needs geometric information then only the displacement shader will be executed.

While simple and unrestrictive from a shader author's perspective, the lack of further atomization of the shaders prevents the renderer from being able to single out certain parts of the shading process for special treatment, such as caching results that can be meaningfully reused.

4.1 Shader Methods

RSL was extended in 2007 to divide the shading pipeline into more components. Shader objects, with multiple methods — each of

which can be separately invoked by the renderer — changed the pipeline to:

```
displacement()
opacity()
prelighting()
lighting()
postlighting()
```

Typically, the prelighting() method contains surface calculations which are independent of the illumination (e.g. texture lookups). The primary motivation for this more complex pipeline was the ability to store the outputs of prelighting() so that they can be used as inputs when re-running only the lighting() method when the lighting changes during interactive rerendering.

4.2 Diffuselighting Method

We now present a new shader method which permits the view-independent results to be cached. The view-independent part of the lighting() method is duplicated in the new diffuselighting() method. Note our somewhat loose nomenclature where "diffuse lighting" is used to describe view-independent lighting. Although diffuse reflections can be view dependent [Oren and Nayar 1994], we found this nomenclature easier to describe to users.

The purpose of the diffuselighting() method is to provide the renderer with a view-independent radiosity value (Ci). (It may also optionally produce a view-independent irradiance value for use in subsurface scattering computations, as discussed in section 6.2.) It is not important what calculations are performed to arrive at the view-independent shading result, only that they are indeed view independent.

Here is an example of matching lighting() and diffuselighting() methods:

```
class simplesurface() {

  public void
  lighting(output color Ci, Oi) {
    // direct/indirect, specular/diffuse
    color ds, is, dd, id;
    // integrate direct and indir lighting
    directlighting(...,
                   ds, dd);  // outputs
    is = indirectspecular(...);
    id = indirectdiffuse(...);
    Ci = ds + is + dd + id;
    Oi = ...;
  }

  public void
  diffuselighting(output color Ci, Oi) {
    // no view dependency permitted here
    color dd, id; // direct/indir diff
    // integrate direct and indir diffuse
    directlighting(...,
                   dd);  // output
    id = indirectdiffuse(...);
    Ci = dd + id;
    Oi = ...;
  }
}
```

This simple factoring of the lighting computation provides the renderer with a method which it can invoke in order to get the view-independent ("diffuse") lighting.

In such a simple case as the above shader, separating out the diffuse lighting computation is relatively straightforward and mechanical. In cases where pattern generation which sets the BRDF parameters is shared between view-independent and view-dependent lighting computation, some additional factoring of code may be required. Not shown in the example above are the callbacks that the renderer might make to have the surface provide evaluations of light samples — the BRDF weighting — or for generating material samples. These are clearly important for the shading process but do not affect the mechanism by which radiosity caching operates.

It may be possible to automate this factoring of the lighting computation. The "Lightspeed" interactive rendering system [Ragan-Kelley et al. 2007] has an automatic analysis to distinguish static versus dynamic variables; it is likely that a similar analysis could be done for view-independent versus view-dependent shader computations. However, many shader writers find that manual separation of the diffuse lighting is both intuitive and expressive.

4.3 Ray Types, Shader Execution, and Caching

Our renderer makes a distinction between different types of rays, which drives differing shading demands on the surfaces those rays hit.

Diffuse rays, which are collecting indirect diffuse energy for color bleeding purposes, do not require specular computations. For such rays, the radiosity cache itself is sufficient and an interpolated result can be returned if the cache entry already exists. In other words, diffuselighting() will not be run if its result is already in the cache. (This is the primary source of our global illumination speed-ups.)

For specular rays we need both view-independent and view-dependent shading results. For these rays we run the lighting() method.

4.4 Caching Tricks and Pitfalls

The identifiers for the radiosity cache entries include the ray's diffuse depth. This is primarily to distinguish between different illumination bounces, but also allows caching of shaders that compute a different accuracy result depending on the diffuse ray depth. For efficiency, it is common for shaders to reduce the number of diffuse rays used at deeper diffuse ray depths.

Clearly, for a shading result to be view-independent, the shader must not use the incident ray direction (vector I in RSL) in its computation.

The importance of a ray is a measure of its ultimate contribution to the image. A common optimization is to use approximate shader calculations for rays with low importance. However, with caching we can't use importance to simplify shading if there's a risk that the cached result will be reused by a more important ray later on. In section 7 we discuss some ideas for overcoming this limitation.

5 Global Illumination Results

In this section we first use results from a simple scene to illustrate the benefits of radiosity caching for computing single- and multi-bounce global illumination, and then analyze renders of three complex CG movie scenes for more "real-world" statistics.

The images in sections 5.1– 5.3 are 1024 pixels wide and are rendered on an Apple PowerMac computer with two 2.26 GHz quad-core Intel Xeon processors and 8 GB memory. For these images, we observe speed-ups of 6.3–6.6 with 8 threads. The radiosity cache size is the default 100 MB.

5.1 Preview Quality Global Illumination

Figure 3 shows four images of a Cornell box containing two teapots. The left teapot is reflective chrome, while the right teapot is matte and has procedural displacements. The scene is tessellated into approximately 18,000 micropolygon grids with a total of 3.9 million grid vertices.

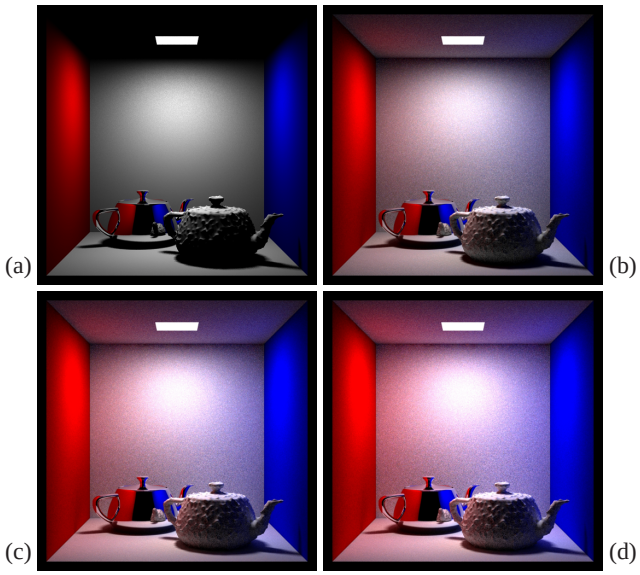

Figure 3: *Preview quality global illumination in box: 0–3 bounces (4, 10, 11, 13 sec).*

Figure 3(a) is rendered with preview-quality direct illumination: the soft shadow from the area light source in the ceiling is sampled with 4 shadow rays per shading point. Rendering this image uses 7.3 million shading points and 14 million rays and takes 4 seconds.

Figure 3(b) shows the same scene, but with preview-quality global illumination. Here the shadows are again sampled with 4 shadow rays and the indirect illumination is sampled with 4 diffuse rays per shading point. This preview quality can be used with progressive ray tracing to dynamically render global illumination images while the illumination and shader parameters are being adjusted. Figures 3(c) and (d) show two- and three-bounce preview-quality global illumination.

Table 1 shows the number of shading points, rays, and render time with and without radiosity caching for 1, 2, and 3 bounces b. The table also shows the speed-ups resulting from radiosity caching.

	w/o caching			w/ caching			speed
b	sh.pts	rays	time	sh.pts	rays	time	-up
1	46M	66M	38s	8.1M	32M	10s	3.8
2	162M	164M	2m20s	8.9M	37M	11s	12.7
3	528M	721M	7m36s	9.8M	42M	13s	35.1

Table 1: *Statistics for rendering the global illumination images in figure 3. (M = million.)*

The time saving is dominated by the reduction in shader evaluations and shadow rays for computing direct illumination at diffuse ray hit points. Note that with radiosity caching the rendering time becomes sublinear in the number of bounces.

5.2 Final Quality Global Illumination

Consider the Cornell box scene again. Now we will render it in final quality. In figure 4(a), with direct illumination, the soft shadow is sampled with 64 shadow rays per shading point. Rendering this image uses 7.3 million shading points and 211 million rays and takes 22 seconds.

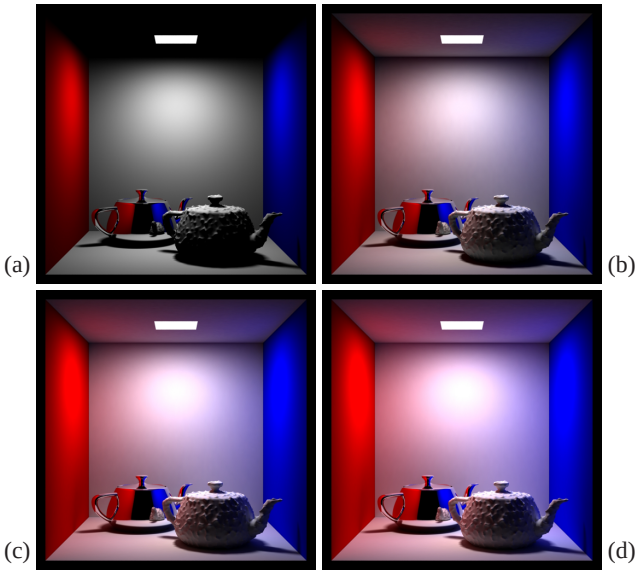

Figure 4: *Final quality global illumination in box: 0–3 bounces (22, 64, 99, 127 sec).*

Final-quality global illumination is shown in figure 4(b). This image is rendered with up to 1024 diffuse rays per shading point. We use irradiance interpolation with irradiance gradients [Ward and Heckbert 1992; Tabellion and Lamorlette 2004; Křivánek et al. 2008] to reduce the number of shading points from which diffuse rays need to be traced. For final-quality multibounce images, we still shoot up to 1024 diffuse rays at diffuse depth 0 and up to 64 diffuse rays at higher diffuse depths. Two- and three-bounce final-quality images are shown in figures 4(c) and (d).

Table 2 shows the number of shading points, rays, and render time with and without radiosity caching for 1, 2, and 3 bounces, and shows the speed-ups resulting from radiosity caching. (There are no data for three bounces without radiosity caching since that case is prohibitively slow.)

	w/o caching			w/ caching			speed
b	sh.pts	rays	time	sh.pts	rays	time	-up
1	209M	3.4B	13.2m	10.0M	410M	64s	12.3
2	2.3B	39B	2.3h	11.6M	569M	99s	83
3	–	–	–	13.1M	691M	127s	–

Table 2: *Statistics for rendering the global illumination images in figure 4. (M = million, B = billion.)*

Note that the single-bounce speed-up is 12.3 and that two and three bounces take only 1.5 and 2 times as long as a single bounce when radiosity caching is on. This sublinear growth is due to the ray count reduction at diffuse depths above 0 and the wider ray differentials after multiple bounces allowing rays to use coarser levels of the radiosity cache.

5.3 Comparison with Point-Based Global Illumination

For comparison, the same scene with single-bounce point-based approximate global illumination is shown in figure 5(a). Point cloud generation takes 20 seconds and the point cloud contains 3.6 million points. Rendering time is 42 seconds (with irradiance gradients and interpolation).

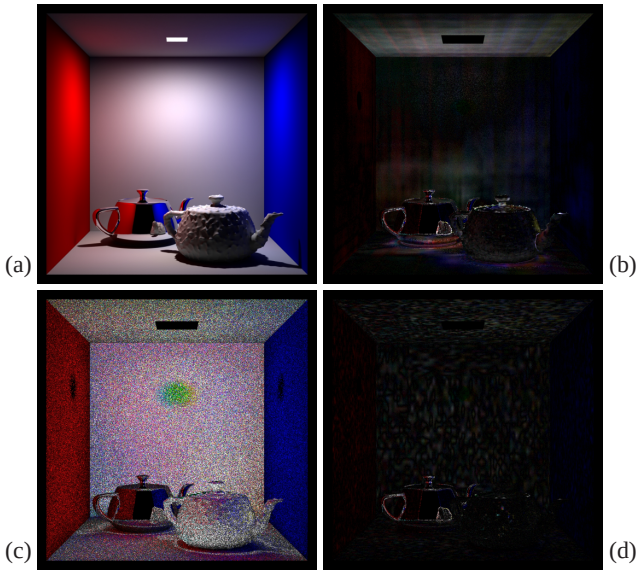

Figure 5: *Top row: Point-based single-bounce global illumination in box (20 sec + 42 sec); 20× difference from reference image. Bottow row: 20× difference between ray-traced preview image and reference image; same for ray-traced final image.*

Figure 5(b) shows 20 times the difference between the point-based image in figure 5(a) and a reference image computed with 4096 rays per shading point and no radiosity caching.

For comparison, the bottom row of figure 5 shows 20 times the difference between the ray-traced preview quality image in figure 3(b) and the reference image, as well as 20 times the difference between the final quality image figure 4(b) and the reference image. The point-based image has more error than the ray-traced final-quality image for comparable total render times. (Depending on the intended use, the difference in accuracy may or may not be important.)

5.4 Movie Production Results

The radiosity caching speed-ups of around 3.8 and 12 reported above for one bounce are for very simplistic shaders and only one light source. The more complex the shaders are, and the more direct light sources there are, the higher the speed-up from caching will be. For more realistic results, we have compiled data from production shots of the future Pixar movie 'Monsters University'.

Figure 1 shows direct illumination and single-bounce global illumination in two movie scenes, an exterior and an interior. The images are 1920×1080 pixels and rendered with motion blur. The number of micropolygon grids is 0.7 and 15.2 million and the number of shading points is 13 and 64 million, respectively. (Much of the geometric complexity in the latter scene is from the fur.) The radiosity cache size is set to 256 MB. These images were rendered using 4 threads on a six-core 2.9 GHz machine with 12 GB memory.

The direct illumination consists of 2 light sources (sun and dome light) for the exterior scene and 20 light sources for the interior scene. The indirect illumination is rendered with 256–1024 diffuse rays from each shading point. The close-ups in figure 6 show parts of the two images (indicated with red squares in figure 1) with strong indirect illumination.

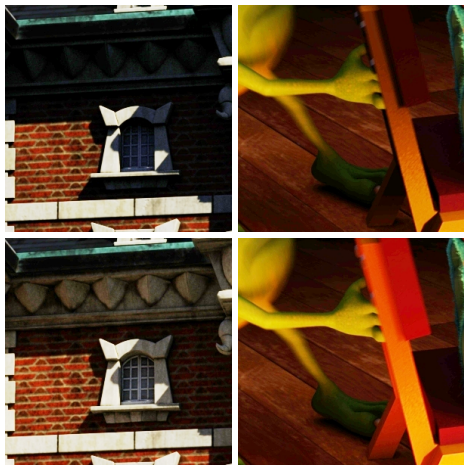

Figure 6: *Close-up of parts of the 'Monsters University' images. Top row: direct illumination. Bottom row: global illumination.* © *Disney/Pixar.*

Table 3 shows the render time, peak total memory, and number of shadow, diffuse, and specular rays traced during rendering of the scenes with direct and global illumination.

scene	illum	shadow	diff	spec	memory	time
exter	direct	784M	0	0	11.4GB	25m
exter	global	1.3B	1.2B	0	16.6GB	1h 31m
inter	direct	4.5B	0	28k	10.5GB	2h 16m
inter	global	6.9B	3.1B	28k	11.0GB	3h 58m

Table 3: *Statistics for rendering the images in figure 1.*

We rendered the same two global illumination images with radiosity caching turned off, and the render times were 32.7 and 41.4 times longer, respectively.

Figure 7 shows another image from the same movie. Table 4 shows the cache hit rates (coarse, medium, fine cache), number of shading points and rays, and render time for various cache sizes. As it can be

Figure 7: *Another global illumination image from 'Monsters University'.* © *Disney/Pixar.*

seen, the cache hit rates are very high for a cache size of 256MB and do not improve by increasing the cache size further. For this scene, the overall peak memory use is 6.4GB. Caching reduces render time from 102 hours to just over one hour, a speed-up of 94. In general, for these types of scenes, *rendering is more than 30 times faster with radiosity caching* than without it.

cache size	cache hit rates (%)			sh.pts	rays	time
0	0	0	0	7.2B	50B	102h
16MB	97.8	97.5	99.5	5.3B	73B	36h
64MB	99.7	99.8	99.9	30M	2.7B	1h22m
256MB	99.9	99.9	99.9	15M	2.3B	1h05m
1GB	99.9	99.9	99.9	15M	2.3B	1h05m

Table 4: *Statistics for rendering figure 7 with varying cache sizes.*

Rendering these scenes with two bounces of global illumination takes about 30%–40% longer than with a single bounce. Even though two bounces give a slightly more realistic appearance, the lighting supervisors on this movie have chosen to use just one bounce to stay within the rendering budget.

Lighting rigs on previous movies usually included hundreds of light sources in each scene. With global illumination, it has been possible to reduce the number of lights dramatically: in interior scenes there are typically around 10–20 lights, in exterior scenes even fewer. The use of global illumination has also halved the master lighting budget; traditional master lighting usually took four weeks or more per sequence and now it is typically done in two weeks. For this movie, the lighting supervisors chose ray-traced global illumination with radiosity caching over point-based global illumination because it is much more interactive, particularly when re-rendering a cropped part of the image.

6 Other Caching Applications

The multiresolution caching scheme can be extended to uses other than global illumination. In general, caching any expensive and reusable shading results can significantly reduce render time. We have already used a tessellation cache for displacement() results [Christensen et al. 2003] and in the previous sections introduced the radiosity cache for diffuselighting() Ci results. In this section we show caching of opacity() Oi results for shadows, ambient occlusion, and volume extinction, and caching of diffuselighting() Irradiance results for ray-traced subsurface scattering.

We could store the opacity and irradiance values in separate multiresolution caches. In practice, however, we have chosen to store the opacities and irradiances in the same cache as radiosity and augment the cache entry identifiers with data type (radiosity, opacity, or irradiance).

6.1 Opacity

Shaders are often used to compute the opacity of surfaces, for example for stained-glass windows, stencils, cuculoris ("cookies") or semitransparent surfaces casting shadows, and to compute extinction in volumes.

For caching, we assume that opacity is independent of which side the surface is seen from, and independent of ray depth. So there is only one opacity value per shading point. (Opacity caching can be turned off if this assumption does not hold.)

Figure 8(a) shows colored soft shadows from a curved stained-glass object. The image was rendered with 25 million shadow rays. Without caching, every ray hit requires running the shader to evaluate the opacity; the render time is 7 seconds. With opacity caching the render time is 4 seconds — a modest speed-up of 1.7.

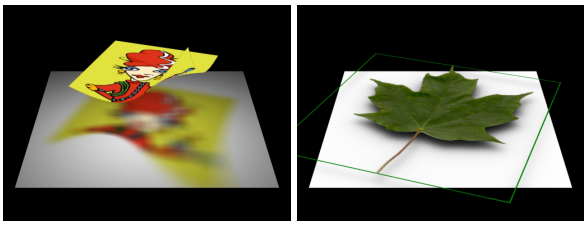

Figure 8: *Opacity stored in the cache: (a) Colored opacity for shadows. (b) Opacity for ambient occlusion.*

Figure 8(b) shows ambient occlusion [Zhukov et al. 1998; Landis 2002] from an opaque leaf texture on an otherwise transparent square. Rendering the image was done with 48 million rays. The render time without opacity caching is 14 seconds, and with opacity caching it is 8 seconds — again a speed-up of 1.7.

If the surface shaders' opacity calculations had been more complicated, the speed-ups would obviously have been higher. But even in these simple cases, opacity caching shifts the computational bottleneck from shader evaluation to raw ray tracing.

Figure 9 shows illumination and shadows in an inhomogeneous volume. For volumes we use the convention that the "opacity" returned by a shader is interpreted as the volume extinction coefficients. In this image, the volume extinction (opacity) is computed by a shader calling a turbulence function with five octaves of noise. The image was rendered with 49 million shadow (transmission) rays. Without caching, the volume extinction has to be evaluated many times along every ray and the render time is 20 minutes. With opacity caching the render time is 4.7 minutes — a speed-up of 4.3.

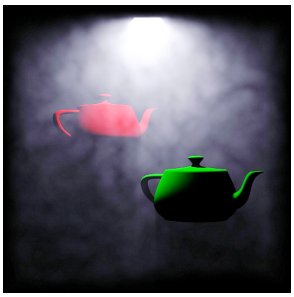

Figure 9: *Extinction in an inhomogeneous volume.*

6.2 Subsurface Scattering

Subsurface scattering gives translucent materials such as wax, marble, and skin their distinctive appearance. We have adopted Jensen et al.'s [2001] formulation of modeling multiple subsurface scattering as diffusion. In this model, the exitant radiance at a given point is the integral over the surface of the product of a diffusion kernel and the irradiance (incident illumination) on the surface.

Previously, our implementation of the diffusion approximation closely followed the two-pass point-based formulation of Jensen and Buhler [2002]. In the first pass, irradiance is computed at each REYES micropolygon and written to disk as a point cloud. In the second pass, these points are organized into a hierarchical representation that is traversed when computing the subsurface integral of irradiance.

Our new single-pass implementation uses the multiresolution cache instead of the point-based hierarchical representation. We use ray tracing to distribute points across the surface of an object. (Ray tracing is remarkably efficient in this application because subsurface rays only need to be tested for intersections with the object they were shot from.) An advantage over point-based subsurface scattering is that internal blockers — for example bones under skin — are more readily simulated with ray tracing.

Given a point on the surface, a ray is traced into the object along the inverse of the normal, finding the distance to the back side of the object. A ray probe point is placed along this ray, either one mean free path length into the object or half the distance to the back side — which ever is shorter. From this probe point we cast a shader-supplied number of rays distributed uniformly over the sphere of directions. The hit points of these rays give a distribution of points on the object surface. At each ray hit point we tessellate the surface in the same multiresolution fashion as for radiosity caching and run a shader that computes irradiance. The grid of irradiance results are stored in the cache for reuse by subsequent subsurface computations. The irradiance value needed for a given hit point is interpolated from those in the grid. The surface area associated with the irradiance value is computed as 4π divided by the number of rays times the squared hit distance and divided by the dot product of the ray direction and the surface normal. Finally, we compute the distance between the irradiance sample and the point to which we are scattering. The irradiance, area, and distance are the inputs to the diffusion approximation.

To facilitate shaders passing irradiance to the renderer, the diffuselighting() method of RSL can be extended with an optional third parameter, Irradiance:

```
class sss_surface() {

  public void
  diffuselighting(output color Ci, Oi;
                  output color Irradiance) {
    // integrate direct and indir lighting
    directlighting(...,
                   Irradiance);  // output
  }
}
```

If the shader uses the built-in integrator directlighting(), irradiance will be automatically computed as a side effect.

Figure 10 shows two images rendered with this approach — one in preview quality with 4 shadow and 4 subsurface rays per shading point, and one in final quality with 64 shadow and 64 subsurface rays per shading point. The render time for the preview image is 64 seconds without caching and 19 seconds with caching; a speed-up of 3.4. For the final quality image the render times are 54 minutes versus 3.2 minutes, a speed-up of 17. In these images we have used the dipole diffusion approximation [Jensen et al. 2001], but more advanced formulas could be used as well, for example a multipole model or the quantized-diffusion model of d'Eon and Irving [2011].

7 Discussion and Future Work

Biased versus unbiased rendering The radiosity caching method proposed here is inherently biased since we reuse and interpolate results. There is some debate in the global illumination community about the pros and cons of unbiased rendering. Some rendering techniques (for example pure path tracing) are unbiased, so they are guaranteed to be correct on average. This can be useful when computing e.g. "ground truth" reference images. However,

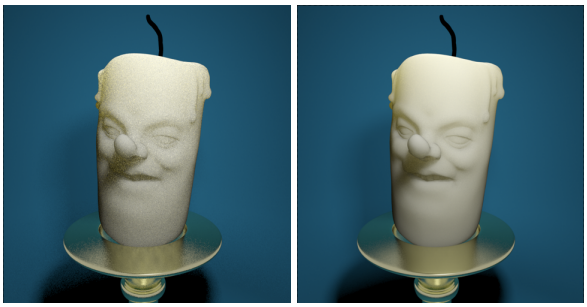

Figure 10: *Candle face with subsurface scattering: (a) Preview quality (19 sec). (b) Final quality (3.2 min).*

the price is noisy images and very slow convergence to a final image with no visible noise. In contrast, the success of point-based global illumination techniques indicates that biased solutions can be very useful in practical movie production. We believe that biased results are perfectly acceptable as long as the results are noise-free, stable in animation, and close to the ground truth.

Non-physical global illumination Even though global illumination is based on a physical simulation of light transport, there is still room for artistic choices. For example, different sets of lights can be used for direct illumination, global illumination, and subsurface scattering. It is also possible to use different lights at specular ray hit points since these are computed at diffuse depth 0 — this allows for lights which contribute to directly visible and specularly reflected objects but not to diffuse interactions and visa versa. Some objects can have global illumination turned off entirely. These global illumination tricks are in addition to the "classic" rendering tricks such as lights having non-physical fall-off, casting no shadow, or have shadows cast from a position different from the actual light source position [Barzel 1997], bent specular reflection rays, etc.

The global illumination images in figure 1 (bottom row) were rendered with only one bounce of global illumination, but the indirect illumination was boosted to compensate for the missing additional bounces. This is a good example of "photosurrealistic" global illumination used in movie production: visually pleasing and controllable bounce light is much more important than physical reality.

Specularlighting shader method At specular ray hit points we need both view-independent and view-dependent shading results. Currently this is done by calling the lighting() method. We have also experimented with a separate specularlighting() method that is executed after the diffuselighting() method for specular rays. More precisely: If an appropriate radiosity cache entry exists we seed Ci with the radiosity from the cache; otherwise we run diffuselighting(). Then we run specularlighting() which adds to Ci.

However, we have found that in practice it is unclear whether there is an actual advantage to this approach. Specular ray hit points are always at diffuse depth 0, so they can't share diffuse results with diffuse ray hit points (which are always at diffuse depth higher than 0). The only potential sharing is between REYES shading grids and specular ray hits, and between specular ray hits. If a REYES grid is shaded, it seems appealing to cache the diffuse result for possible reuse just in case some specular rays hit the same patch later. But even this case isn't clear-cut: polluting the fine radiosity cache with results that may never be used can lead to eviction of more useful radiosity values. If a specular ray hits a patch before it is REYES shaded, it is unclear whether it is worthwhile paying the

up-front cost of computing diffuse illumination on the entire patch — we don't know if any other specular rays will hit the patch later, or whether the patch is visible to the camera (causing REYES shading). We are currently investigating this surprisingly complex issue.

Caching view-dependent shading results In this paper, we have focused only on caching of view-independent shading results, but sometimes it is perfectly acceptable to cache view-dependent results as well. Such a technique was used to speed up sharp and glossy reflections in movies such as 'Ratatouille' and 'Cars 2'. Shah et al. [2007] precompute the direct illumination, including specular reflections and highlights, as seen from a dominant viewing direction. (The dominant direction is not necessarily from the camera position, but could be, for example, from below the ground plane if it is shiny or wet, or from another prominent highly reflective object.) When tracing reflection rays, instead of running the shader at the ray hit points they just look up the precomputed shading results there. Obviously the specular highlights seen in the reflections will often be in the wrong places, but they are stable, plausible, and fast.

We could extend out multiresolution cache to store view-dependent shading results and add a way to specify the dominant viewing direction. We leave this as future work.

Importance As mentioned in section 4.4 we can't use importance to simplify shading if there's a risk that the cached result will be reused by a more important ray later on. One idea to relax this restriction is to store with each shading result the importance at which it was computed, and only allow reuse by a subsequent ray if that ray's importance is lower or equal to the importance of the cached result. This ensures that we never use an approximate shading result when an accurate result is needed. Unfortunately, this strategy introduces multi-threaded non-repeatability: a low-importance ray will reuse a cached shading result with higher importance if it happens to be in the cache, but otherwise compute an approximate shading result (and store it in the cache). The shading result can therefore be either accurate or approximate depending on the execution order. Classifying importance into ranges and only reusing shading results with importance in the same range as requested could alleviate some of the undesired variation — albeit at the cost of more cache entries and more shader evaluations.

8 Conclusion

We have presented a new multiresolution caching scheme targeted at interactive preview and final-quality images for movies. The caching reduces the number of shader executions, texture lookups, and shadow rays, and provides significant speed-ups for ray-traced rendering of global illumination, shadows, ambient occlusion, volumes, and subsurface scattering.

An advantage over Ward's irradiance caching method is that our method also caches direct illumination and surface shading results and uses SIMD shader execution over multiple points at a time. The primary advantages over point-based global illumination are that our new method is more interactive, has no file I/O, uses only a single pass, and is more accurate. However, we do not see our method as a replacement of point-based global illumination, but as complimentary — another tool in the toolbox.

Our method is implemented in Pixar's RenderMan renderer and is currently being used in the production of upcoming movies from several studios.

Supplemental Material

The supplemental material contains two videos. They illustrate interactive changes of viewpoint, illumination, and shader parameters while the scene is rendered continuously with progressive ray tracing. The first video shows global illumination with radiosity caching, while the second video shows subsurface scattering with irradiance caching.

Acknowledgements

Many thanks to our colleagues in Pixar's RenderMan team for all their help in this project. Brian Smits implemented the latest generation of the multiresolution tessellation cache — his implementation has efficient multithreading and memory use, and was used as the basis for our multiresolution radiosity cache. Andrew Kensler implemented most of the progressive ray-tracing front-end and the fast float-to-half conversions. Julian Fong did most of the work related to volume rendering. Huge thanks to Chris Harvey who created the scene in figure 10 and the videos in the supplemental materials, and to Wayne Wooten for video editing and voice-over.

Also many thanks to Bill Reeves, Jean-Claude Kalache, Christophe Hery and others for early testing and feedback, and to Guido Quaroni, Sanjay Bakshi, and the entire 'Monsters U.' lighting team for adopting this method in production. Apurva Shah suggested the idea of caching view-dependent shading results, as described in section 7. Thanks to Charlie Kilpatrick and Tony DeRose for reading early drafts of this memo and for providing very helpful feedback.

References

APODACA, A. A., AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers.

BARZEL, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools 2*, 1, 1–20.

CHRISTENSEN, P. H., AND BATALI, D. 2004. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques (Proc. Eurographics Symposium on Rendering 2004)*, 133–141.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proc. Eurographics Conference 2003) 22*, 3, 543–552.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray tracing for the movie 'Cars'. In *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, 1–6.

CHRISTENSEN, P. H. 2008. Point-based approximate color bleeding. Tech. Rep. 08-01, Pixar Animation Studios. (Available at graphics.pixar.com/library/PointBasedColorBleeding).

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. *Computer Graphics (Proc. SIGGRAPH '87) 21*, 4, 95–102.

D'EON, E., AND IRVING, G. 2011. A quantized-diffusion model for rendering translucent materials. *ACM Transactions on Graphics (Proc. SIGGRAPH 2011) 30*, 4.

DUTRÉ, P., BEKAERT, P., AND BALA, K. 2003. *Advanced Global Illumination*. A K Peters.

FAJARDO, M., 2010. Ray tracing solution for film production rendering. In [Křivánek et al. 2010].

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. *Computer Graphics (Proc. SIGGRAPH '90) 24*, 4, 289–298.

HECKBERT, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proc. SIGGRAPH '90) 24*, 4, 145–154.

HOU, Q., AND ZHOU, K. 2011. A shading reuse method for efficient micropolygon ray tracing. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2011) 30*, 6.

IGEHY, H. 1999. Tracing ray differentials. *Computer Graphics (Proc. SIGGRAPH '99) 33*, 179–186.

JENSEN, H. W., AND BUHLER, J. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics (Proc. SIGGRAPH 2002) 21*, 3, 576–581.

JENSEN, H. W., MARSCHNER, S., LEVOY, M., AND HANRAHAN, P. 2001. A practical model for subsurface light transport. *Computer Graphics (Proc. SIGGRAPH 2001) 35*, 511–518.

KONTKANEN, J., TABELLION, E., AND OVERBECK, R. S. 2011. Coherent out-of-core point-based global illumination. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering 2011) 30*, 4, 1353–1360.

KŘIVÁNEK, J., GAUTRON, P., WARD, G., JENSEN, H. W., TABELLION, E., AND CHRISTENSEN, P. H. 2008. Practical global illumination with irradiance caching. *SIGGRAPH 2008 Course Note #16*. (Available at cgg.mff.cuni.cz/˜jaroslav/-papers/2008-irradiance_caching_class).

KŘIVÁNEK, J., FAJARDO, M., CHRISTENSEN, P. H., TABELLION, E., BUNNELL, M., LARSSON, D., AND KAPLANYAN, A. 2010. Global illumination across industries. *SIGGRAPH 2010 Course Notes*. (Available at cgg.mff.cuni.cz/˜jaroslav/-gicourse2010).

LANDIS, H. 2002. Production-ready global illumination. In *SIGGRAPH 2002 Course Note #16: RenderMan in Production*, 87–102. (Available at www.renderman.org/RMR/Publications/-sig02.course16.pdf).

MEYER, M., AND ANDERSON, J. 2006. Statistical acceleration for animated global illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH 2006) 25*, 3, 1075–1080.

OREN, M., AND NAYAR, S. K. 1994. Generalization of Lambert's reflectance model. *Computer Graphics (Proc. SIGGRAPH '94) 28*, 239–246.

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*, 2nd. ed. Morgan Kaufmann Publishers.

RAGAN-KELLEY, J., KILPATRICK, C., SMITH, B. W., EPPS, D., GREEN, P., HERY, C., AND DURAND, F. 2007. The Lightspeed automatic interactive lighting preview system. *ACM Transactions on Graphics (Proc. SIGGRAPH 2007) 26*, 3.

SHAH, A., RITTER, J., KING, C., AND GRONSKY, S. 2007. Fast, soft reflections using radiance caches. Tech. Rep. 07-04, Pixar Animation Studios. (Available at graphics.pixar.com/library/-SoftReflections).

TABELLION, E., AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics (Proc. SIGGRAPH 2004) 23*, 3, 469–476.

WARD, G. J., AND HECKBERT, P. S. 1992. Irradiance gradients. In *Proc. 3rd Eurographics Workshop on Rendering*, 85–98.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. *Computer Graphics (Proc. SIGGRAPH '88) 22*, 4, 85–92.

WARD LARSON, G., AND SHAKESPEARE, R. 1998. *Rendering with Radiance*. Morgan Kaufmann Publishers.

WARD, G. 1991. Real pixels. In *Graphics Gems II*. 80–83.

ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An ambient light illumination model. In *Rendering Techniques (Proc. Eurographics Workshop on Rendering 1998)*, 45–55.