INSTANT GLOBAL ILLUMINATION

Matt Pharr

© 2004 Matt Pharr

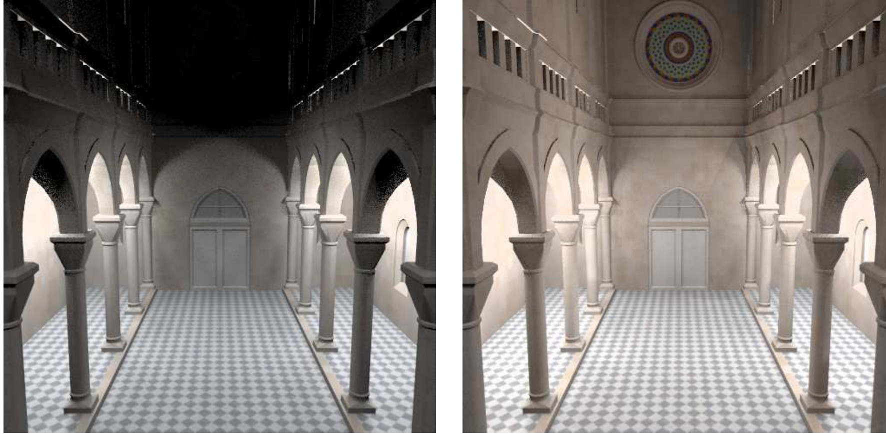

Figure 1: Images of the Sibenik cathedral model with direct lighting only (left) and with global illumination computed with the "instant global illumination" algorithm implemented here (right). The IGI image took only 10% longer than the one with direct lighting only to be rendered, though has few artifacts and represents the indirect lighting well.

0.1 Overview

This note describes an implementation of the "instant global illumination" (IGI) algorithm as an integrator for `pbrt`. Instant global illumination was developed by Wald, Benthin, and collaborators (Wald, Kollig, Benthin, Keller, and Slusallek 2002; Wald, Benthin, and Slusallek 2003; Benthin, Wald, and Slusallek 2003), building on the "instant radiosity" algorithm introduced by Keller (Keller 1997). The basic idea behind both of these approaches is to follow a small number of light carrying paths from the light sources and to construct a number of point light sources at the points where these paths intersect surfaces in a way such that the lights approximate the indirect radiance distribution in the scene. After these lights have been created, when the integrator later needs to compute exitant radiance at a point, it only needs to loop over these point light sources and trace shadow rays to see if they are visible, accumulating the indirect illumination that they represent if so.

Figure 1 shows two images of the Sibenik cathedral model, one rendered with direct lighting only, and the other using IGI. IGI renders indirect lighting effects very efficiently, with only a 10% increase in computation time, and without the noise characteristic of algorithms like path tracing or bidirectional path tracing. It can be implemented as an unbiased light transport algorithm, though because a small set of virtual lights is used for all pixels in the scene, error shows up as systemic error due to this correlation as opposed to noise as would be the case for path tracing algorithms. While this is a desirable property for single images, this error can be distracting in animations, where subsequent frames may have noticeably different brightnesses due to a different set of lights being generated for each one.

In this note, we will show how IGI can be understood as a variant of bidirectional path tracing (see Section 16.3.5 on page 753). This formulation both makes it easier to understand which cosine terms and such to include in the values computed and

also makes it easier to see how the approach could be modified. However, while we will use this theory as the foundation for the derivation of the approach, the implementation will still be in terms of the "virtual light source" approach.

0.2 Implementation

As usual, most of the implementation of the integrator will be elided here, and we will just focus on the parts that are different than standard integrators.

The `IGIIntegrator` constructor takes four parameters; the first, `nLightPaths`, gives the number of paths to follow from lights to create the virtual lights (the actual number of virtual lights created will generally be larger than this, as each path usually creates multiple lights).

The second parameter, `nLightSets`, describes the number of light sets to create; instead of computing just one set of virtual lights, this integrator actually computes `nLightSets` independent sets of them. To see doing so is useful, consider an image being rendered at a rate of sixteen image samples per pixel: in general, we will expect a better result if each image sample uses a different set of virtual lights, thus incorporating more information about indirect illumination in the scene at little incremental computational cost. Thus, for best results, this value should be roughly as large as the number of samples taken per pixel.

Figure 2 shows the Sibenik scene rendered again with just one set of lights (left), and with 256 sets (right), both images rendered with sixteen samples per pixel. In both cases, image quality suffers. With just one set, all sixteen image samples use the same set of virtual lights. As this set only has ten or so lights in it, indirect lighting can not accurately be approximated, and the virtual lights cast noticeable shadows. (With sixteen sets, as in Figure 1, these shadows aren't noticeable because there are so many lights.)[1] With 256 sets, as in the right image, the result is noisy, as different pixels will naturally use different lights.

`minDist2` is the maximum squared distance to a light; its use will also be explained shortly. Finally, `rrThreshold` is a Russian roulette threshold used to skip tracing rays to virtual lights that make a small contribution to a particular point where exitant radiance is being computed.

⟨*IGIIntegrator Implementation*⟩≡ used: none

```
IGIIntegrator::IGIIntegrator(int nl, int ns, float md,
        float rrt, float is) {
    nLightPaths = RoundUpPow2((u_int)nl);
    nLightSets = RoundUpPow2((u_int)ns);
    minDist2 = md * md;
    rrThreshold = rrt;
    indirectScale = is;
    maxSpecularDepth = 5;
    specularDepth = 0;
    virtualLights = new vector<VirtualLight>[nLightSets];
}
```

[1] The integrator depends on a well-written `Sampler` to generate a stratified set of samples for the usual case where the number of pixel samples is the same as the number of virtual light sets. Otherwise, the one-to-one relationship between the two may be lost.

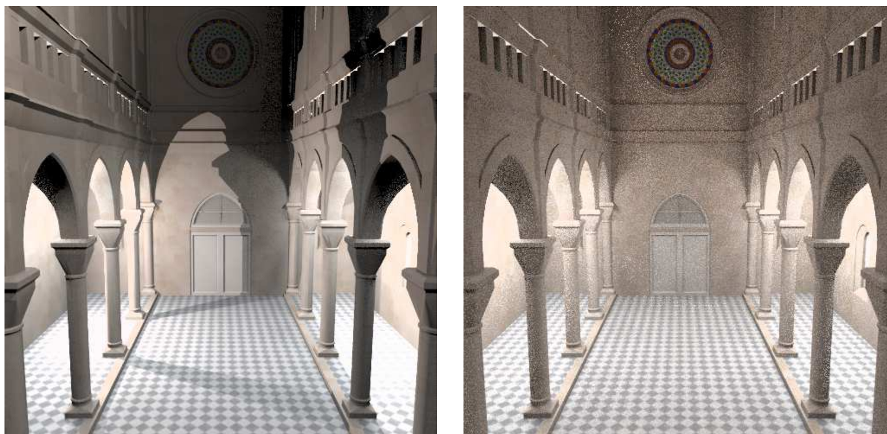

Figure 2: Rendered images of the Sibenik cathedral with sixteen samples per pixel and one set of virtual lights (top) and 256 sets (bottom). With just one set, all sixteen pixel samples use the same set of virtual lights, leading to a lower quality result than if more are used. Furthermore, this image happened to be much brighter than a more accurate one with more virtual lights, since the set of lights generated here happened to follow paths with more indirect lighting contribution than average. With substantially more sets than pixel samples, however, the result has noise. While more sets gives a result with less correlation between pixels due to using different sets of virtual lights, noise is the price to pay for this improvement.

⟨*IGI Private Data*⟩≡ used: none
```
u_int nLightPaths, nLightSets;
vector<VirtualLight> *virtualLights;
mutable int specularDepth;
int maxSpecularDepth;
float minDist2, rrThreshold, indirectScale;
int vlSetOffset;
```

In the integrator's `Li()` method, the typical sampling approach is used for the direct lighting computation, where samples are taken from all light sources. In addition to the usual samples needed for that computation, this integrator also needs a well-distributed 1D sample to choose among the multiple sets of virtual lights; `RequestSamples()` handles notifying the `Sampler` of all of these needed sets of samples.

⟨*IGIIntegrator Implementation*⟩+≡ used: none
```
void IGIIntegrator::RequestSamples(Sample *sample,
                                   const Scene *scene) {
    ⟨Request samples for area light sampling def?⟩
    vlSetOffset = sample->Add1D(1);
}
```

The task of the `Preprocess()` routine is to follow light-carrying paths from the light sources and to create the virtual point sources. Before discussing its implementation, we will first review the relevant ideas from bidirectional path tracing that provide the foundation for IGI.

Recall from Section 16.2.4 on page 738 that we can write the exitant radiance

Figure 3: bidir path

from a point $\mathrm{p_1}$ to a point on the film plane $\mathrm{p_0}$ as an infinite sum over light carrying paths,

$$L(\mathrm{p_1} \to \mathrm{p_0}) = \sum_i^\infty P(\bar{\mathrm{p}}_i)$$

where each path is an integral over points on the surfaces of objects in the scene,

$$P(\bar{\mathrm{p}}_i) = \int_{A^{i-1}} L_\mathrm{e}(\mathrm{p_i} \to \mathrm{p_{i-1}}) \left(\prod_{j=1}^{i-1} f(\mathrm{p_{j+1}} \to \mathrm{p_j} \to \mathrm{p_{j-1}}) G(\mathrm{p_{j+1}} \leftrightarrow \mathrm{p_j}) \right) \mathrm{d}A(\mathrm{p_2}) \cdots \mathrm{d}A(\mathrm{p_i}).$$

With standard path tracing, we computed estimates of these integrals by following paths from the camera into the scene, sampling from probability densities over direction given by the BSDF at each vertex (Section 16.3.3). Bidirectional path tracing follows a similar approach, instead starting paths from both the eye and the lights and then connecting the two paths.

From section 16.3.3, the Monte Carlo estimate for a general path is

$$P(\bar{\mathrm{p}}_i) \approx \frac{L_\mathrm{e}(\mathrm{p_i} \to \mathrm{p_{i-1}})}{p_A(\mathrm{p_i})} \left(\prod_{j=1}^{i-1} \frac{f(\mathrm{p_{j+1}} \to \mathrm{p_j} \to \mathrm{p_{j-1}}) |\cos\theta_j|}{p_\omega(\mathrm{p_{j+1}} - \mathrm{p_j})} \right).$$

To understand IGI, consider the case of bidirectional path tracing where the path from the eye has only one segment and the light path has one or more segments. (Note that direct lighting thus isn't accounted for, and must be handled separately.) Figure 3 shows an example where the light path has three vertices and has been connected to an eye path of one segment with a shadow ray (dotted line). The eye segment and the connecting segment are uniquely determined given the camera ray, its intersection point, and the vertex of the light path it is being connected to.

Note that there is only one way to generate any path of a particular length of n segments: 1 eye segment, 1 connecting segment, and $n-2$ light segments. As such, the Monte Carlo estimate of a path's contribution is

$$P(\bar{p}_i) \approx \left[\frac{L_e(p_i \to p_{i-1})}{p_A(p_i)} \left(\prod_{j=3}^{i-1} \frac{f(p_{j+1} \to p_j \to p_{j-1}) |\cos\theta_j|}{p_\omega(p_{j+1} - p_j)} \right) \right] f(p_0 \to p_1 \to p_2) G(p_1 \leftrightarrow p_2) f(p_1 \to p_2 \to p_3). \quad (0.2.1)$$

Note that the term in square brackets is independent of the segment from the eye to the first visible point p_1 or the segment from that point to the last vertex of the light path p_2. Its value can thus be precomputed and stored with the virtual light.

The task of the `Preprocess()` method is to create `nLightSets` sets of paths from the lights in the scene, where each set has `nLightPaths` paths from lights. Each such light path will generally lead to multiple virtual light sources, as one virtual light is created for each vertex in the path after the one on the light source. (Russian roulette is used to terminate paths in an unbiased manner.)

⟨*IGIIntegrator Implementation*⟩+≡ used: none

```
void IGIIntegrator::Preprocess(const Scene *scene) {
    if (scene->lights.size() == 0) return;
    ⟨Compute samples for emitted rays from lights 5⟩
    ⟨Precompute information for light sampling densities 6⟩
    for (u_int s = 0; s < nLightSets; ++s) {
        for (u_int i = 0; i < nLightPaths; ++i) {
            ⟨Follow path i from light to create virtual lights 6⟩
        }
    }
}
```

To ensure a good sampling of indirect light, it's helpful to use well-distributed samples to choose the initial points on the light sources and their directions. The `LDShuffleScrambled*D()` functions work well to compute samples to compute the initial rays the paths from the lights. Using them in this manner ensures not only that the samples used for each particular set of virtual lights are well-distributed but that in the aggregate over all of the paths, the samples are globally well-distributed as well.

⟨*Compute samples for emitted rays from lights*⟩≡ used: 5

```
float *lightNum = new float[nLightPaths * nLightSets];
float *lightSamp0 = new float[2 * nLightPaths * nLightSets];
float *lightSamp1 = new float[2 * nLightPaths * nLightSets];
LDShuffleScrambled1D(nLightPaths, nLightSets, lightNum);
LDShuffleScrambled2D(nLightPaths, nLightSets, lightSamp0);
LDShuffleScrambled2D(nLightPaths, nLightSets, lightSamp1);
```

The efficiency of the algorithm is also improved if the probability of starting a path from a light source is related to the amount of power it emits in comparison to the power emitted by the other lights. Increasing the probability of generating paths from bright lights naturally leads to more virtual lights being created for those lights, leading to higher-quality results. (The intensity of the virtual lights must

be reduced correspondingly to counterbalance the fact that more virtual lights are created for bright light sources.) Here we compute a discrete probability density for sampling each light and use the `Compute1dCDF()` to compute a discrete CDF which will be used to select among lights.

⟨*Precompute information for light sampling densities*⟩≡ used: 5

```
int nLights = int(scene->lights.size());
float *lightPower = (float *)alloca(nLights * sizeof(float));
float *lightCDF = (float *)alloca((nLights+1) * sizeof(float));
for (int i = 0; i < nLights; ++i)
    lightPower[i] = scene->lights[i]->Power(scene).y();
float totalPower;
ComputeStep1dCDF(lightPower, nLights, &totalPower, lightCDF);
```

The process for computing the paths is generally similar to the approach used in the `PathIntegrator`. Here, the `alpha` variable tracks the current weight of the path, accounting for radiance emitted by the light source along the initial ray and the product of BSDF terms and sampling densities (i.e. the current value of the term in brackets in Equation 0.2.1.)

⟨*Follow path* `i` *from light to create virtual lights*⟩≡ used: 5

```
int sampOffset = s*nLightPaths + i;
⟨Choose light source to trace path from 7⟩
⟨Sample ray leaving light source 7⟩
Intersection isect;
while (scene->Intersect(ray, &isect) && !alpha.Black()) {
    alpha *= scene->Transmittance(ray);
    Vector wo = -ray.d;
    BSDF *bsdf = isect.GetBSDF(ray);
    ⟨Create virtual light at ray intersection point 7⟩
    ⟨Sample new ray direction and update weight 8⟩
}
BSDF::FreeAll();
```

The `SampleStep1d()` function returns a sample from the piecewise-constant light sampling distribution that was previously computed by `ComputeStep1dCDF()`. Because the value returned is over the range $[0,1]$, it must be scaled by the total number of lights to give the appropriate light number to sample.[2]

⟨*Choose light source to trace path from*⟩≡ used: 6

```
float lightPdf;
int lNum = Floor2Int(SampleStep1d(lightPower, lightCDF,
    totalPower, nLights, lightNum[sampOffset], &lightPdf) * nLights);
fprintf(stderr, "samp %f -> num %d\n", lightNum[sampOffset], lNum);
Light *light = scene->lights[lNum];
```

[2] A slightly more complex approach is described by Wald et al (Wald, Benthin, and Slusallek 2003). For densely occluded environments, many of the lights may have little or no contribution to the points visible from the camera. They suggest rendering an image with path tracing and a very low sampling rate (e.g. one path per pixel) before generating the virtual lights. As this image is rendered, information is recorded about which of the light sources made some contribution to the image. This information is then used to set probabilities for starting paths from each light so that few if any paths are started from lights that didn't make any contribution.

Given a particular light and the probability density for starting a path from it, `lightPdf`, the `Light::Sample_L()` routine gives the ray leaving the light, the radiance it is carrying, and the probability density for sampling the ray.

⟨*Sample ray leaving light source*⟩≡ used: 6

```
RayDifferential ray;
float pdf;
Spectrum alpha =
    light->Sample_L(scene, lightSamp0[2*sampOffset],
            lightSamp0[2*sampOffset+1],
            lightSamp1[2*sampOffset],
            lightSamp1[2*sampOffset+1],
            &ray, &pdf);
if (pdf == 0.f || alpha.Black()) continue;
alpha /= pdf * lightPdf;
```

Given the intersection a surface in the scene, a `VirtualLight` object is created to represent the corresponding virtual light source. Given that `alpha` holds the path's contribution up to its arrival at the current point on the surface (the bracketed term in Equation 0.2.1), we just need to store enough information so that the geometry term and two BSDF terms outside the brackets can be evaluated given a one-segment path from the eye.

One of the BSDF terms is from the BSDF at the last vertex of the light path, $f(p_1 \to p_2 \to p_3)$. In a sense, the BSDF at p_2 can be thought of as defining the directionally-varying radiant intensity of the virtual point light. We will introduce an approximation here by representing this term with a Lambertian BSDF based on the hemispherical-directional reflectance of the surface. Doing so makes it possible to include this term in the partial path contribution stored with the virtual light. (The main motivation for making this approximation is that `pbrt`'s BSDF memory allocation optimizations, described in Section 10.1.1 on page 466, doesn't make it easy to prevent these `BSDF`s from being deallocated well before rendering of the image was complete, which would thus lead to runtime errors and memory access violations.) As long as the surfaces in the scene aren't too specular, this approximation works well.

⟨*Create virtual light at ray intersection point*⟩≡ used: 6

```
Spectrum contrib = alpha * bsdf->rho(wo) / M_PI;
virtualLights[s].push_back(VirtualLight(isect.dg.p, isect.dg.nn, contrib));
```

In addition to the partial path contribution, the `VirtualLight` structure needs to store the virtual light source's position and normal in order to evaluate the geometric term in lighting computations below.

⟨*IGI Local Structures*⟩≡ used: none

```
struct VirtualLight {
    VirtualLight() { }
    VirtualLight(const Point &pp, const Normal &nn, const Spectrum &c)
        : p(pp), n(nn), pathContrib(c) { }
    Point p;
    Normal n;
    Spectrum pathContrib;
};
```

The computation to sample the path's outgoing direction leaving an intersection at a surface and compute its updated weight is generally similar to the path sampling computation in the `PathIntegrator` particle tracing computation in the `PhotonIntegrator`.

The one important difference here is how Russian roulette is used for path termination. In those other integrators, the termination probability in the Russian roulette test was constant and Russian roulette was only applied after a fixed number of bounces. Here, the termination probability is one minus the luminance of the product of the BSDF's value and the cosine term divided by the sampling PDF for the outgoing direction. Thus, if the surface as a high albedo–it reflects most of the incident illumination–the probability of continuing the path is high, and if the albedo is small, it is likely that the path will be terminated. This is an intuitively efficient property; if the path is carrying a large amount of light, it should be likely to continue.

An interesting implication of this approach is that the luminance of the path contribution value remains constant after each surface intersection. For example, consider a surface where the luminance of `contribScale` is 0.1. The path will be terminated with 90% probability. For the 10% where the path continues, `alpha` will be scaled by `contribScale`, to account for the light reflection, and divided by 0.1, to account for Russian roulette. Because the luminance of `contribScale` is 0.1, the net result will be that the luminance of the path is unchanged.

Thus, surfaces that reflect little light lead to fewer paths leaving them, rather than paths with a lower contribution. Thus, all of the virtual lights created will have similar path contributions and will thus tend to make an equal contribution to the final exitant radiance values computed. This is a more efficient state of affairs than if some of them were thousands of times brighter than others, for example.

⟨*Sample new ray direction and update weight*⟩≡ used: 6

```
Vector wi;
float pdf;
BxDFType flags;
Spectrum fr = bsdf->Sample_f(wo, &wi, RandomFloat(),
                             RandomFloat(), RandomFloat(),
                             &pdf, BSDF_ALL, &flags);
if (fr.Black() || pdf == 0.f)
    break;
Spectrum contribScale = fr * AbsDot(wi, bsdf->dgShading.nn) / pdf;
float rrProb = min(1.f, contribScale.y());
if (RandomFloat() > rrProb)
    break;
alpha *= contribScale / rrProb;
ray = RayDifferential(isect.dg.p, wi);
```

Most of the `IGIIntegrator::Li()` implementation isn't included here; it samples direct lighting in the usual manner, recursively traces rays to account for perfect specular reflection and refraction, as integrators like `DirectLighting` do. Here we will only focus on the parts that are different for this integrator.

After the direct lighting contribution has been computed at the point being

Figure 4: When the minimum distance test isn't used as was done for this image, there are unsightly bright spots around the points where the virtual light sources are on surfaces, due to the $1/r^2$ term taking on a large value there.

shaded, the contribution from the virtual light sources is found. First, the appropriate sample value from `sample` is used to choose which set of virtual lights to use for this point.

⟨*Compute indirect illumination with virtual lights*⟩≡ used: none

```
u_int lSet = min(u_int(sample->oneD[vlSetOffset][0] * nLightSets),
                 nLightSets-1);
for (u_int i = 0; i < virtualLights[lSet].size(); ++i) {
    const VirtualLight &vl = virtualLights[lSet][i];
    ⟨Add contribution from VirtualLight vl 9⟩
}
```

⟨*Add contribution from* `VirtualLight vl`⟩≡ used: 9

```
⟨Ignore light if it's too close to current point 10⟩
⟨Compute virtual light's tentative contribution Llight 10⟩
⟨Possibly skip shadow ray with Russian roulette 10⟩
if (!scene->IntersectP(Ray(p, vl.p - p, RAY_EPSILON,
                           1.f - RAY_EPSILON)))
    L += Llight;
```

One implementation detail is that IGI ignores the contribution of virtual lights that are closer to the current point than a minimum distance given as a user-defined value. Figure 4 shows why this rejection test is helpful; if nearby lights are included, the image will have bright splotches in areas that are very close to virtual light sources. Recall that there is a one over squared distance term in the geometric term $G(\mathrm{p}_1 \leftrightarrow \mathrm{p}_2)$ for the path's overall contribution; this value becomes arbitrarily large for points very close to the light. If the integrator ignores such light sources, the results are more visually pleasing, although this test introduces systemic error into the values computed. (And thus, the images will be slightly dimmer than they should be.)

In order to hide the transitions from points where a given virtual light is included to points where it isn't, the `Smoothstep()` function is used to set a scale value over

the range $[0,1]$ the smoothly fades in the light's contribution.

⟨*Ignore light if it's too close to current point*⟩≡ used: 9

```
float d2 = DistanceSquared(p, vl.p);
float distScale = SmoothStep(.8 * minDist2, 1.2 * minDist2, d2);
```

Since the BSDF of the last vertex of the light path was approximated by a constant Lambertian term, there are two factors to evaluate from Equation 0.2.1 in order to compute a light path's contribution to a given eye path:

$$f(\mathrm{p}_0 \to \mathrm{p}_1 \to \mathrm{p}_2) G(\mathrm{p}_1 \leftrightarrow \mathrm{p}_2).$$

These are easily computed with the values available to the integrator and in the `VirtualLight` structure.

⟨*Compute virtual light's tentative contribution* `Llight`⟩≡ used: 9

```
Vector wi = Normalize(vl.p - p);
Spectrum f = distScale * bsdf->f(wo, wi);
if (f.Black()) continue;
float G = AbsDot(wi, n) * AbsDot(wi, vl.n) / d2;
Spectrum Llight = indirectScale * f * G * vl.pathContrib /
    virtualLights[lSet].size();
Llight *= scene->Transmittance(Ray(p, vl.p - p));
```

If the light's contribution to outgoing radiance at the point is low, Russian roulette is used to sometimes avoid tracing the shadow ray for it. Thus, less time is spent tracing shadow rays for unimportant lights. The rays that are candidates for Russian roulette termination but are traced anyway are reweighted so that the final result is unbiased (at least, from the termination done here–the minimum distance test and the approximation of the BRDF with a Lambertian term both make this algorithm biased already.)

⟨*Possibly skip shadow ray with Russian roulette*⟩≡ used: 9

```
if (Llight.y() < rrThreshold) {
    float continueProbability = .1f;
    if (RandomFloat() > continueProbability)
        continue;
    Llight /= continueProbability;
}
```

Exercises

0.1 Using the formulation of this approach in terms of bidirectionl path tracing introduced here, discuss whether this formulation leads to other improvements that can be made to the algorithm (in particular to eliminating the bright spots in corners in an unbiased manner)?

Bibliography

Benthin, C., I. Wald, and P. Slusallek (2003). A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum 22*(3), 621–630. (Proceedings of Eurographics).

Keller, A. (1997, August). Instant radiosity. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 49–56. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-896-7.

Wald, I., C. Benthin, and P. Slusallek (2003, June). Interactive global illumination in complex and highly occluded environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pp. 74–81.

Wald, I., T. Kollig, C. Benthin, A. Keller, and P. Slusallek (2002, June). Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pp. 15–24.