Surface Integrators

Digital Image Synthesis

Yung-Yu Chuang

12/24/2008

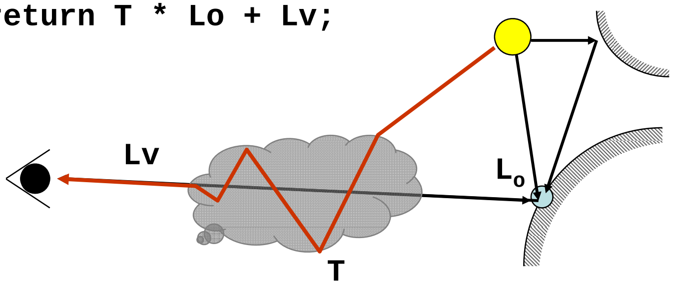
with slides by Peter Shirley, Pat Hanrahan, Henrik Jensen, Mario Costa Sousa and Torsten Moller

Main rendering loop

```
void Scene::Render() {
  Sample *sample = new Sample(surfaceIntegrator,
                              volumeIntegrator,
                              this);
  ...
  while (sampler->GetNextSample(sample)) {
      RayDifferential ray;
      float rW = camera->GenerateRay(*sample, &ray);
      <Generate ray differentials for camera ray>
      float alpha;
      Spectrum Ls = 0.f;
      if (rW > 0.f)
            Ls = rW * Li(ray, sample, &alpha);
      ...
      camera->film->AddSample(*sample,ray,Ls,alpha);
      ...
  }
  ...
  camera->film->WriteImage();
}
```

Scene::Li

```
Spectrum Scene::Li(RayDifferential &ray,
                   Sample *sample, float *alpha)
{
  Spectrum Lo=surfaceIntegrator->Li(…);
  Spectrum T=volumeIntegrator->Transmittance(…);
  Spectrum Lv=volumeIntegrator->Li(…);
  return T * Lo + Lv;
}
```



Surface integrators

- Responsible for evaluating the integral equation
- `core/transport.*  integrator/*`

  Whitted, directlighting, path, bidirectional, irradiancecache, photonmap igi, exphotonmap

```
class COREDLL Integrator {
  Spectrum Li(Scene *scene, RayDifferential
        &ray, Sample *sample, float *alpha);
  void Proprocess(Scene *scene)
  void RequestSamples(Sample*, Scene*)
};
class SurfaceIntegrator : public Integrator
```

Surface integrators

- **`void Preprocess(const Scene *scene)`**

  Called after scene has been initialized; do scene-dependent computation such as photon shooting for photon mapping.

- **`void RequestSamples(Sample *sample, const Scene *scene)`**

  Sample is allocated once in Render(). There, sample's constructor will call integrator's RequestSamples to allocate appropriate space.

```
Sample::Sample(SurfaceIntegrator *surf,
    VolumeIntegrator *vol, const Scene *scene) {
    // calculate required number of samples
    // according to integration strategy
    surf->RequestSamples(this, scene);
    ...
```

Direct lighting

Rendering equation

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

If we only consider direct lighting, we can replace L_i by L_d.

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_d(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

- simplest form of equation
- somewhat easy to solve (but a gross approximation)
- kind of what we do in Whitted ray tracing
- Not too bad since most energy comes from direct lights

Direct lighting

- Monte Carlo sampling to solve

$$\int_\Omega f(p,\omega_o,\omega_i) L_d(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

- Sampling strategy A: sample only one light
  - pick up one light as the representative for all lights
  - distribute N samples over that light
  - Use multiple importance sampling for f and L_d

$$\frac{1}{N}\sum_{j=1}^{N} \frac{f(p,\omega_o,\omega_j) L_d(p,\omega_j) |\cos\theta_j|}{p(\omega_j)}$$

  - Scale the result by the number of lights N_L

$E[f+g]$ Randomly pick f or g and then sample, multiply the result by 2

Direct lighting

- Sampling strategy B: sample all lights
  - do A for each light
  - sum the results
  - smarter way would be to sample lights according to their power

$$\sum_{j=1}^{N_L} \int_\Omega f(p,\omega_o,\omega_i) L_{d(j)}(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

$E[f+g]$ sample f or g separately and then sum them together

DirectLighting

```
enum LightStrategy {
  SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM,
                      SAMPLE_ONE_WEIGHTED
};
```

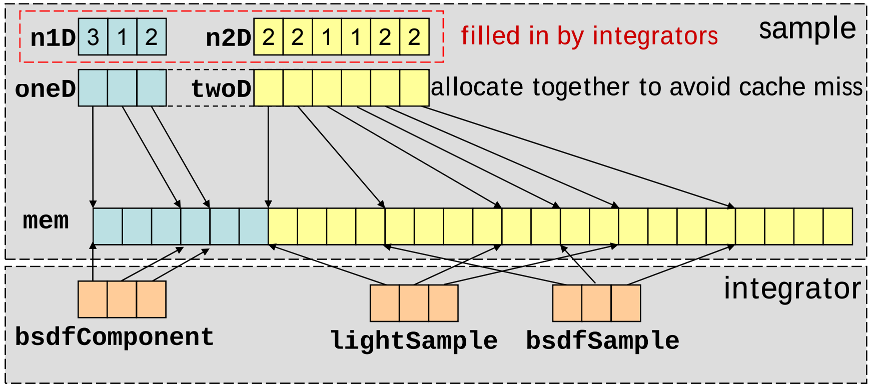
two possible strategies; if there are many image samples for a pixel (e.g. due to depth of field), we prefer only sampling one light at a time. On the other hand, if there are few image samples, we prefer sampling all lights at once.

```
class DirectLighting : public SurfaceIntegrator {
public:
  DirectLighting(LightStrategy ls, int md);
  ...
}
```

maximal depth

RequestSamples

- Different types of lights require different number of samples, usually 2D samples.
- Sampling BRDF requires 2D samples.
- Selection of BRDF components requires 1D samples.



DirectLighting::RequestSamples

```
void RequestSamples(Sample *sample, const Scene *scene) {
  if (strategy == SAMPLE_ALL_UNIFORM) {
    u_int nLights = scene->lights.size();
    lightSampleOffset = new int[nLights];
    bsdfSampleOffset = new int[nLights];
    bsdfComponentOffset = new int[nLights];
    for (u_int i = 0; i < nLights; ++i) {
      const Light *light = scene->lights[i];
      int lightSamples
          = scene->sampler->RoundSize(light->nSamples);
      lightSampleOffset[i] = sample->Add2D(lightSamples);
      bsdfSampleOffset[i] = sample->Add2D(lightSamples);
      bsdfComponentOffset[i] = sample->Add1D(lightSamples);
    }
    lightNumOffset = -1;
  }
```

gives sampler a chance to adjust to an appropriate value

DirectLighting::RequestSamples

```
  else {
    lightSampleOffset = new int[1];
    bsdfSampleOffset = new int[1];
    bsdfComponentOffset = new int[1];

    lightSampleOffset[0] = sample->Add2D(1);
    bsdfSampleOffset[0] = sample->Add2D(1);
    bsdfComponentOffset[0] = sample->Add1D(1);

    lightNumOffset = sample->Add1D(1);
  }
}
```

which light to sample

DirectLighting::Li

```
Spectrum DirectLighting::Li(Scene *scene,
  RayDifferential &ray, Sample *sample, float *alpha)
{
  Intersection isect;
  Spectrum L(0.);
  if (scene->Intersect(ray, &isect)) {
    // Evaluate BSDF at hit point
    BSDF *bsdf = isect.GetBSDF(ray);
    Vector wo = -ray.d;
    const Point &p = bsdf->dgShading.p;
    const Normal &n = bsdf->dgShading.nn;
    <Compute emitted light; see next slide>
  }
  else {
    // handle ray with no intersection
  }
  return L;
}
```

DirectLighting::Li

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{\Omega} f(p,\omega_o,\omega_i) L_d(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

```
L += isect.Le(wo);
if (scene->lights.size() > 0) {
  switch (strategy) {
    case SAMPLE_ALL_UNIFORM:
      L += UniformSampleAllLights(scene, p, n, wo, bsdf,
        sample, lightSampleOffset, bsdfSampleOffset,
        bsdfComponentOffset);
        break;
    case SAMPLE_ONE_UNIFORM:
      L += UniformSampleOneLight(scene, p, n, wo, bsdf,
        sample, lightSampleOffset[0], lightNumOffset,
        bsdfSampleOffset[0], bsdfComponentOffset[0]);
        break;
```

DirectLighting::Li

```
    case SAMPLE_ONE_WEIGHTED:  sample according to power
      L += WeightedSampleOneLight(scene, p, n, wo, bsdf,
        sample, lightSampleOffset[0], lightNumOffset,
        bsdfSampleOffset[0], bsdfComponentOffset[0], avgY,
        avgYsample, cdf, overallAvgY);
        break;
  }
}
if (rayDepth++ < maxDepth) {
  // add specular reflected and transmitted contributions
}
```

This part is essentially the same as Whitted integrator.

The main difference between Whitted and DirectLighting is the way they sample lights. Whitted uses sample_L to take one sample for each light. DirectLighting uses multiple Importance sampling to sample both lights and BRDFs.

Whitted::Li

```
...
// Add contribution of each light source
Vector wi;
for (i = 0; i < scene->lights.size(); ++i)
{
  VisibilityTester visibility;
  Spectrum Li = scene->lights[i]->
           Sample_L(p, &wi, &visibility);
  if (Li.Black()) continue;
  Spectrum f = bsdf->f(wo, wi);
  if (!f.Black() &&
       visibility.Unoccluded(scene))
    L += f * Li * AbsDot(wi, n) *
         visibility.Transmittance(scene);
}
...
```

UniformSampleAllLights

```
Spectrum UniformSampleAllLights(...)
{
  Spectrum L(0.);
  for (u_int i=0;i<scene->lights.size();++i) {
    Light *light = scene->lights[i];
    int nSamples =
      (sample && lightSampleOffset) ?
       sample->n2D[lightSampleOffset[i]] : 1;
    Spectrum Ld(0.);
    for (int j = 0; j < nSamples; ++j)
      Ld += EstimateDirect(...);
    L += Ld / nSamples;
  }
  return L;
}
```

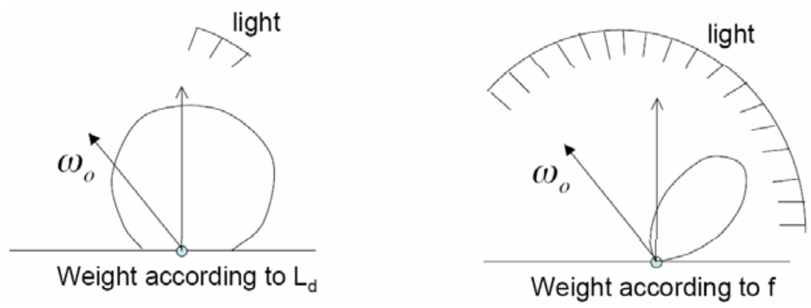
compute contribution for one sample for one light

$$\frac{f(p,\omega_o,\omega_j)L_d(p,\omega_j)|\cos\theta_j|}{p(\omega_j)}$$

UniformSampleOneLight

```
Spectrum UniformSampleOneLight (...)
{
  int nLights = int(scene->lights.size());
  int lightNum;
  if (lightNumOffset != -1)
    lightNum =
      Floor2Int(sample->oneD[lightNumOffset][0]*nLights);
  else
    lightNum = Floor2Int(RandomFloat() * nLights);
  lightNum = min(lightNum, nLights-1);
  Light *light = scene->lights[lightNum];
  return (float)nLights * EstimateDirect(...);
}
```

Multiple importance sampling



$$\frac{1}{n_f}\sum_{i=1}^{n_f}\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)}+\frac{1}{n_g}\sum_{j=1}^{n_g}\frac{f(Y_j)g(Y_j)w_g(Y_i)}{p_g(Y_j)}$$

$$w_s(x)=\frac{(n_s p_s(x))^\beta}{\sum_i (n_i p_i(x))^\beta}$$

EstimateDirect

```
Spectrum EstimateDirect(Scene *scene, Light *light, Point
  &p, Normal &n, Vector &wo, BSDF *bsdf, Sample *sample,
  int lightSamp, int bsdfSamp, int bsdfComponent,
  u_int sampleNum)
{
  Spectrum Ld(0.);
  float ls1, ls2, bs1, bs2, bcs;
  if (lightSamp != -1 && bsdfSamp != -1 &&
      sampleNum < sample->n2D[lightSamp] &&
      sampleNum < sample->n2D[bsdfSamp]) {
      ls1 = sample->twoD[lightSamp][2*sampleNum];
      ls2 = sample->twoD[lightSamp][2*sampleNum+1];
      bs1 = sample->twoD[bsdfSamp][2*sampleNum];
      bs2 = sample->twoD[bsdfSamp][2*sampleNum+1];
      bcs = sample->oneD[bsdfComponent][sampleNum];
  } else {
      ls1 = RandomFloat();
      ls2 = RandomFloat();
      ...
  }
```

$$\frac{f(p,\omega_o,\omega_j)L_d(p,\omega_j)|\cos\theta_j|}{p(\omega_j)}$$

Sample light with MIS

```
Spectrum Li = light->Sample_L(p, n, ls1, ls2, &wi,
                              &lightPdf, &visibility);
if (lightPdf > 0. && !Li.Black()) {
  Spectrum f = bsdf->f(wo, wi);
  if (!f.Black() && visibility.Unoccluded(scene)) {
    Li *= visibility.Transmittance(scene);
    if (light->IsDeltaLight())
      Ld += f * Li * AbsDot(wi, n) / lightPdf;
    else {
      bsdfPdf = bsdf->Pdf(wo, wi);
      float weight = PowerHeuristic(1,lightPdf,1,bsdfPdf);
      Ld += f * Li * AbsDot(wi, n) * weight / lightPdf;
    }
  }
}
```

$$\frac{f(p,\omega_o,\omega_j)L_d(p,\omega_j)|\cos\theta_j|w_L(\omega_j)}{p(\omega_j)}$$

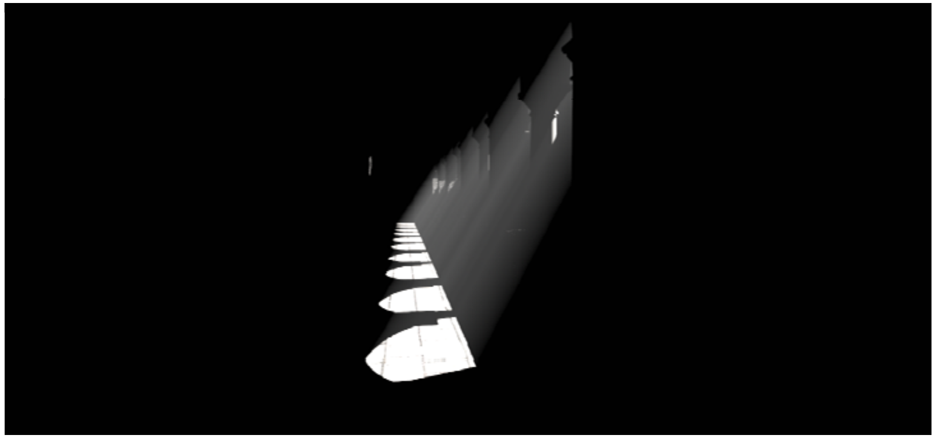
Sample BRDF with MIS

Only for non-delta light and BSDF

```
if (!light->IsDeltaLight()) {
  BxDFType flags = BxDFType(BSDF_ALL & ~BSDF_SPECULAR);
  Spectrum f = bsdf->Sample_f(wo, &wi, bs1, bs2, bcs,
                              &bsdfPdf, flags);
  if (!f.Black() && bsdfPdf > 0.) {
    lightPdf = light->Pdf(p, n, wi);
    if (lightPdf > 0.) {
      // Add light contribution from BSDF sampling
      float weight = PowerHeuristic(1,bsdfPdf,1,lightPdf);
      Spectrum Li(0.f);
      RayDifferential ray(p, wi);
      if (scene->Intersect(ray, &lightIsect)) {
        if (lightIsect.primitive->GetAreaLight() == light)
          Li = lightIsect.Le(-wi);
      } else Li = light->Le(ray);
      if (!Li.Black()) {
        Li *= scene->Transmittance(ray);
        Ld += f * Li * AbsDot(wi, n) * weight / bsdfPdf;
      }
    }
  }
}
```

for infinite area light

Direct lighting



The light transport equation

- The goal of integrator is to numerically solve the light transport equation, governing the equilibrium distribution of radiance in a scene.

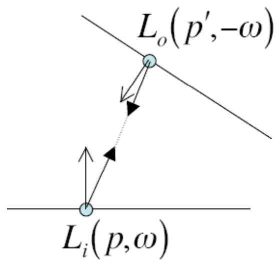
$$L_o(x,\omega_o) = L_e(x,\omega_o) + L_r(x,\omega_o)$$
$$= L_e(x,\omega_o) + \int_{H^2} f_r(x,\omega_i \to \omega_o) L_i(x,\omega_i)\cos\theta_i \, d\omega_i$$

The light transport equation

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{S^2} f_r(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| d\omega_i$$

- If no participating media - express incoming in terms of outgoing radiance:

$$L_i(p,\omega) = L_o(t(p,\omega),-\omega)$$



- Need to solve for L (only one unknown)

$$L(p,\omega_o) = L_e(p,\omega_o) + \int_{S^2} f_r(p,\omega_o,\omega_i) L(t(p,\omega_i),-\omega_i) |\cos\theta_i| d\omega_i$$

Analytic solution to the LTE

- In general, it is impossible to find an analytic solution to the LTE because of complex BRDF, arbitrary scene geometry and intricate visibility.
- For an extremely simple scene, e.g. inside a uniformly emitting Lambertian sphere, it is however possible. This is useful for debugging.

$$L(p,\omega_o) = L_e + c\int_{H^2} L(t(p,\omega_i),-\omega_i) |\cos\theta_i| d\omega_i$$

- Radiance should be the same for all points

$$L = L_e + c\pi L$$

Analytic solution to the LTE

$$L = L_e + c\pi L$$

$$L = L_e + \rho_{hh} L$$

$$= L_e + \rho_{hh}(L_e + \rho_{hh} L)$$

$$= L_e + \rho_{hh}(L_e + \rho_{hh}(L_e + \ldots$$

$$= \sum_{i=0}^{\infty} L_e \rho_{hh}^i$$

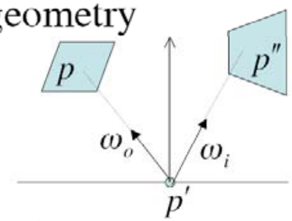
$$L = \frac{L_e}{1-\rho_{hh}} \qquad \rho_{hh} \le 1$$

Surface form of the LTE

- Expressing LTE in terms of geometry within the scene

$$L(p',\omega_o) = L(p' \to p)$$

$$f(p',\omega_o,\omega_i) = f(p'' \to p' \to p)$$



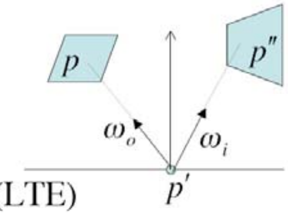
- Replacing the integrand ($d\omega_i$) with an area integrator over the whole scene geometry and remembering: $d\omega_i = \frac{|\cos\theta''|}{\|p'-p''\|^2} dA(p'')$
- $V(p \leftrightarrow p')$ - visibility term (either one or zero)

Surface form of the LTE

- Geometry coupling term

$$G(p'' \leftrightarrow p') = V(p'' \leftrightarrow p')\frac{|\cos\theta''||\cos\theta'|}{\|p'-p''\|^2}$$

- New (geometric) formulation of the Light Transport Equation (LTE)



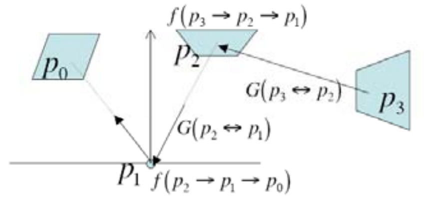
$$L(p' \to p) = L_e(p' \to p) + \int_A f_r(p'' \to p' \to p) L(p'' \to p') G(p'' \leftrightarrow p') dA(p'')$$

- Randomly pick points in the scene and create a path vs. (previously)
- randomly pick directions over a sphere

These two forms are equivalent, but they represent two different ways of approaching light transport.

Surface form of the LTE

$$L(p_1 \to p_0) = L_e(p_1 \to p_0)$$
$$+ \int_{A_2} L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) dA(p_2)$$
$$+ \iint_{A_2 A_3} L_e(p_3 \to p_2) f(p_3 \to p_2 \to p_1) G(p_3 \leftrightarrow p_2)$$
$$f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) dA(p_2) dA(p_3)$$
$$+ \ldots$$

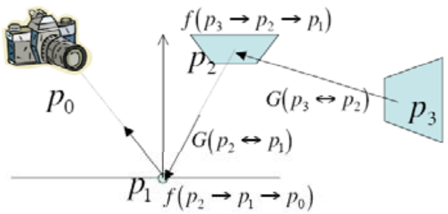


Surface form of the LTE

- compact formulation:

$$L(p_1 \to p_0) = \sum_{i=1}^{\infty} P(\bar{p}_i)$$

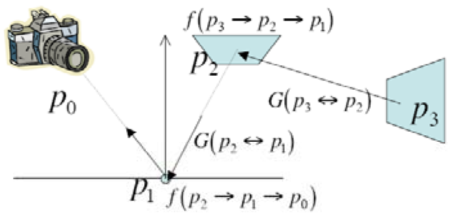
- For a path $\bar{p}_i = p_0 p_1 \ldots p_i$
- Where p_0 is the camera and p_i is a light source



Surface form of the LTE

- with: $P(\bar{p}_i) = \int_{A_2}\int_{A_3} \ldots \int_{A_i} L_e(p_i \to p_{i-1}) T(\bar{p}_i) dA(p_2) \ldots dA(p_i)$
- Where $T(\bar{p}_i) = \prod_{j=1}^{i-1} f(p_{j+1} \to p_j \to p_{j-1}) G(p_{j+1} \leftrightarrow p_j)$
- Is called the *throughput*
- Special case:

$$P(\bar{p}_1) = L_e(p_1 \to p_0)$$



Delta distribution

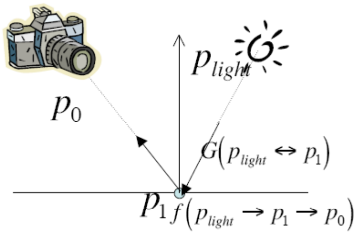
- Again - handle with care (e.g. point light):

$$P(\bar{p}_2) = \int_A L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) dA(p_2)$$

$$= \frac{\delta(p_{light} - p_2) L_e(p_2 \to p_1)}{p(p_{light})} f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1)$$

$$= L_e(p_{light} \to p_1) f(p_{light} \to p_1 \to p_0) G(p_{light} \leftrightarrow p_1)$$

- E.g. Whitted ray tracing only uses specular BSDF's



Partition the integrand

- Many different algorithms proposed to deal with $\sum_{i=0}^{\infty} P(\bar{p}_i)$
- Most energy in the first few bounces:

$$L(p_1 \to p_0) = P(\bar{p}_1) + P(\bar{p}_2) + \sum_{i=3}^{\infty} P(\bar{p}_i)$$

- $P(\bar{p}_1)$ emitted radiance at p_1
- $P(\bar{p}_2)$ one bounce to light (direct lighting)

Partition the integrand

- Simplify according to *small* and *large* light sources: $L_e = L_{e,s} + L_{e,l}$

$$P(\bar{p}_i) = \int_A \int_A \cdots \int_A L_e(p_i \to p_{i-1}) T(\bar{p}_i) dA(p_2) \ldots dA(p_i)$$

$$= \int_A \int_A \cdots \int_A L_{e,s}(p_i \to p_{i-1}) T(\bar{p}_i) dA(p_2) \ldots dA(p_i)$$

$$+ \int_A \int_A \cdots \int_A L_{e,l}(p_i \to p_{i-1}) T(\bar{p}_i) dA(p_2) \ldots dA(p_i)$$

- Can be handled separately (different number of samples)

Partition the integrand

- Similarly, we can split BxDF into delta and non-delta distributions:

$$f = f_\Delta + f_{\bar{\Delta}}$$

$$T(\bar{p}_i) = \prod_{j=1}^{i-1} (f_\Delta + f_{\bar{\Delta}}) G(p_{j+1} \leftrightarrow p_j)$$

Rendering operators

Scattering operator

$$L_o(x,\omega_o) = \int_{H^2} f_r(x,\omega_i \to \omega_o) L_i(x,\omega_i) \cos\theta_i \, d\omega_i$$

$$\equiv S \circ L_i$$

Transport transport

$$L_i(x,\omega_i) = L_o(x^*(x,\omega_i), -\omega_i)$$

$$\equiv T \circ L_o$$

Solving the rendering equation

Rendering Equation

$$K \equiv S \circ T$$

$$L = L_e + K \circ L$$

$$(I - K) \circ L = L_e$$

Solution

$$L = (I - K)^{-1} \circ L_e$$

$$(I - K)^{-1} = \frac{1}{I - K} = I + K + K^2 + \ldots$$

Successive approximation

Successive approximations

$$L^1 = L_e$$

$$L^2 = L_e + K \circ L^1$$

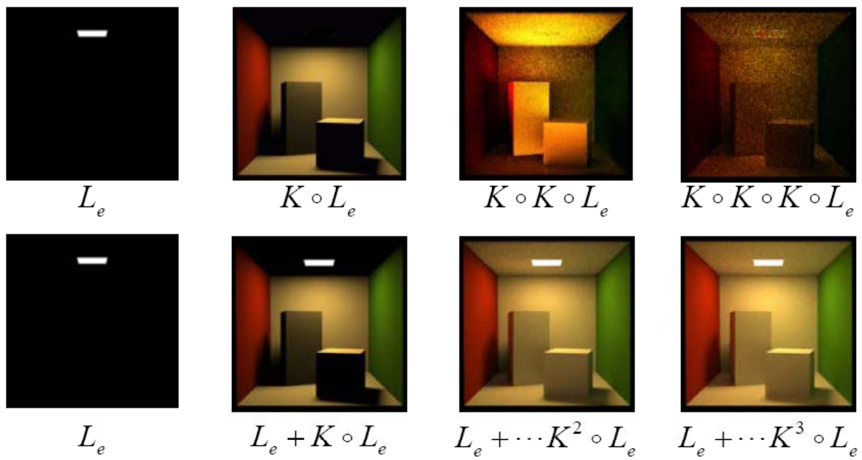
$$\ldots$$

$$L^n = L_e + K \circ L^{n-1}$$

Converged

$$L^n = L^{n-1} \quad \therefore \quad L^n = L_e + K \circ L^n$$

Successive approximation

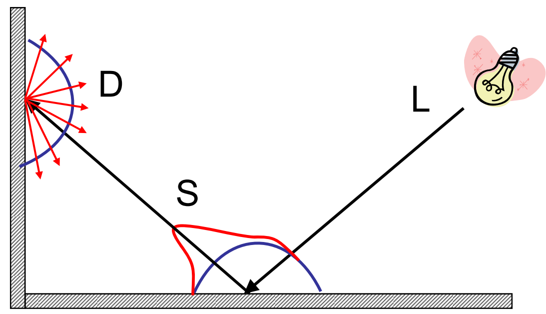


Light Transport Notation (Hekbert 1990)

- Regular expression denoting sequence of events along a **light path alphabet: {L,E,S,D,G}**
  - **L** a light source (emitter)
  - **E** the eye
  - **S** specular reflection/transmission
  - **D** diffuse reflection/transmission
  - G glossy reflection/transmission
- operators:
  - **$(k)^+$** one or more of k
  - **$(k)^*$** zero or more of k (iteration)
  - **(k|k')** a k or a k' event

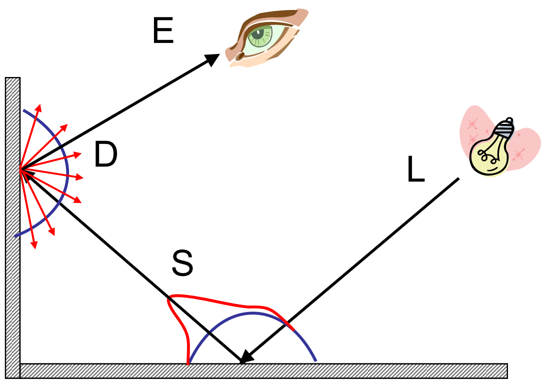
Light Transport Notation: Examples

- LSD
  - a path starting at a light, having one specular reflection and ending at a diffuse reflection



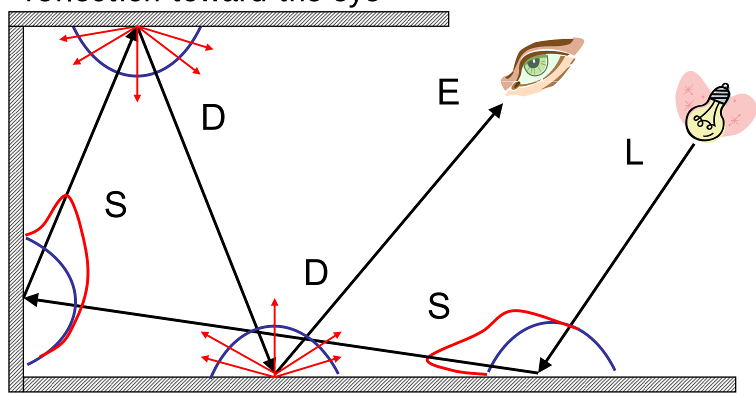
Light Transport Notation: Examples

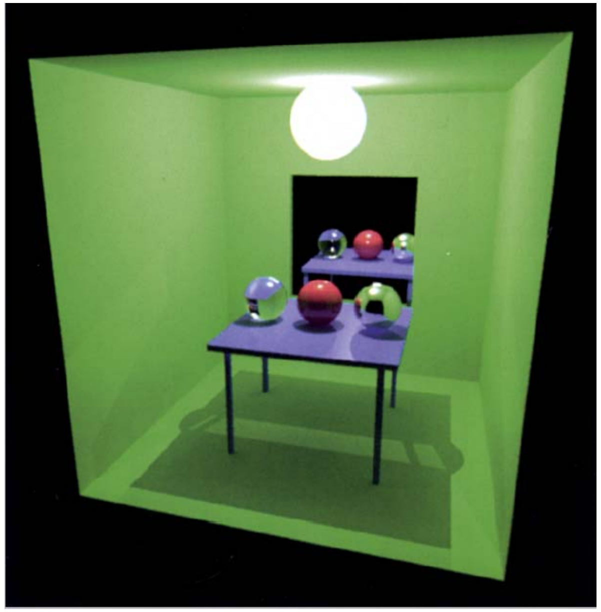
- $L(S|D)^+DE$
  - a path starting at a light, having one or more diffuse or specular reflections, then a final diffuse reflection toward the eye

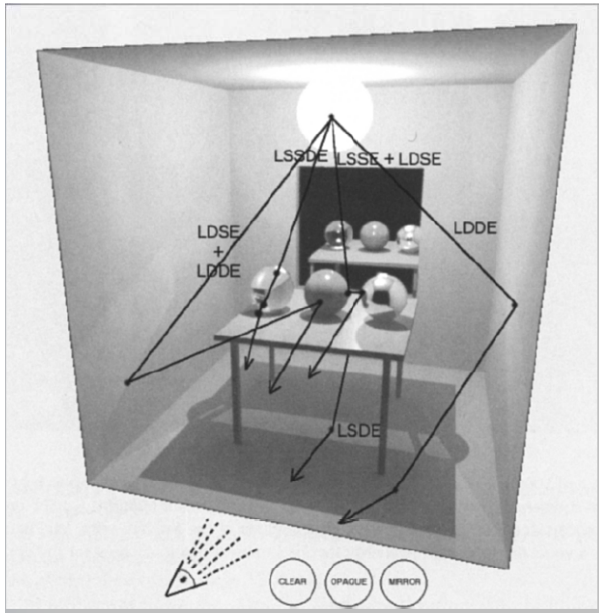


Light Transport Notation: Examples

- $L(S|D)^+DE$
  - a path starting at a light, having one or more diffuse or specular reflections, then a final diffuse reflection toward the eye







Rendering algorithms

- Ray casting: E(D|G)L
- Whitted: E[S*](D|G)L
- Kajiya: E[(D|G|S)$^+$(D|G)]L
- Goral: ED*L

The rendering equation

Directional form

$$L(x,\omega) = L_e(x,\omega) + \int_{H^2} f_r(x,\omega' \rightarrow \omega)\, L(x^*(x,\omega'), -\omega') \cos\theta' \, d\omega'$$

Integrate over hemisphere of directions

Transport operator i.e. ray tracing

The rendering equation

Surface form

$$L(x',x) = L_e(x',x) + \int_{M^2} f_r(x'',x',x)\, L(x'',x')\, G(x'',x')\, dA''(x'')$$

Geometry term

Integrate over all surfaces

$$G(x'',x') = \frac{\cos\theta_i''\cos\theta_o'}{\left\|x''-x'\right\|^2} V(x'',x')$$

Visibility term

$$V(x'',x') = \begin{cases} 1 & \text{visible} \\ 0 & \text{not visible} \end{cases}$$

The radiosity equation

Assume diffuse reflection

1. $f_r(x,\omega_i \rightarrow \omega_o) = f_r(x) \Rightarrow \rho(x) = \pi f_r(x)$
2. $L(x,\omega) = B(x)/\pi$

$$B(x) = B_e(x) + \rho(x)E(x)$$

$$B(x) = B_e(x) + \rho(x)\int_{M^2} F(x,x')B(x')\,dA'(x')$$

$$F(x,x') = \frac{G(x,x')}{\pi}$$

Radiosity

- formulate the basic radiosity equation:

$$B_m = E_m + \rho_m \sum_{n=1}^{N} B_n F_{mn}$$

- B_m = radiosity = total energy leaving surface m (energy/unit area/unit time)
- E_m = energy emitted from surface m (energy/unit area/unit time)
- ρ_m = reflectivity, fraction of incident light reflected back into environment
- F_{mn} = form factor, fraction of energy leaving surface n that lands on surface m
- (A_m = area of surface m)

Radiosity

- Bring all the B's on one side of the equation

$$E_m = B_m - \rho_m \sum_m B_n F_{mn}$$

- this leads to this equation system:

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \dots & 1-\rho_N F_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix}$$

$$S \circ B = E$$

Path tracing

- Proposed by Kajiya in his classic SIGGRAPH 1986 paper, rendering equation, as the solution for

$$L(p_1 \to p_0) = \sum_{i=1}^{\infty} P(\overline{p}_i)$$

- Incrementally generates path of scattering events starting from the camera and ending at light sources in the scene.
- Two questions to answer
  - How to do it in finite time?
  - How to generate one or more paths to compute $P(\overline{p}_i)$

Infinite sum

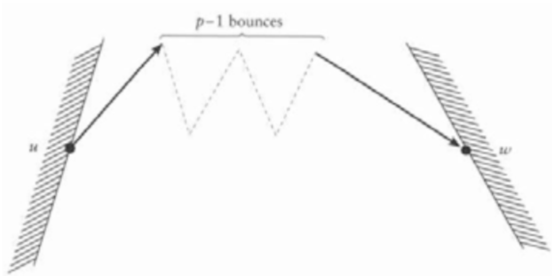
- In general, the longer the path, the less the impact.
- Use Russian Roulette after a finite number of bounces
  - Always compute the first few terms
  - Stop after that with probability q

$$L(p_1 \to p_0) \approx P(\overline{p}_1) + P(\overline{p}_2) + P(\overline{p}_3) + \frac{1}{1-q}\sum_{i=4}^{\infty} P(\overline{p}_i)$$

Infinite sum

- Take this idea further and instead randomly consider terminating evaluation of the sum at each term with probability q_i

$$L(p_1 \to p_0) \approx \frac{1}{1-q_1}\left(P(\overline{p}_1) + \frac{1}{1-q_2}\left(P(\overline{p}_2) + \frac{1}{1-q_3}\left(P(\overline{p}_3) + \ldots\right)\right)\right)$$



Path generation (first trial)

- First, pick up surface i in the scene randomly and uniformly $p_i = \frac{A_i}{\sum_j A_j}$
- Then, pick up a point on this surface randomly and uniformly with probability $1/A_i$
- Overall probability of picking a random surface point in the scene:

$$p_A(p_i) = \frac{A_i}{\sum_j A_j} \cdot \frac{1}{A_i} = \frac{1}{\sum_j A_j}$$

Path generation (first trial)

- This is repeated for each point on the path.
- Last point should be sampled on light sources only.
- If we know characteristics about the scene (such as which objects are contributing most indirect lighting to the scene), we can sample more smartly.
- Problems:
  - High variance: only few points are mutually visible, i.e. many of the paths yield zero.
  - Incorrect integral: for delta distributions, we rarely find the right path direction

Incremental path generation

- For path $\bar{p}_i = p_0 p_1 \cdots p_j p_{j+1} \cdots p_i$
  - At each p_j, find p_{j+1} according to BSDF
  - At p_{i-1}, find p_i by multiple importance sampling of BSDF and L
- This algorithm distributes samples according to solid angle instead of area. So, the distribution p_A needs to be adjusted

$$p_A(p_i) = p_\omega \frac{\left\| p_i - p_{i+1} \right\|^2}{|\cos\theta_i|}$$

Incremental path generation

- Monte Carlo estimator

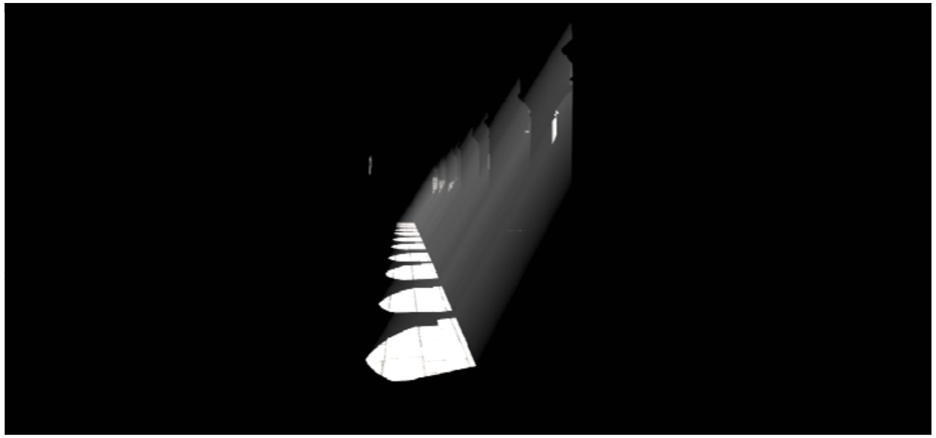
$$\frac{L_e(p_i \to p_{i-1})}{p_A(p_i)} \left(\prod_{j=1}^{i-1} \frac{f(p_{j+1} \to p_j \to p_{j-1}) |\cos\theta_i|}{p_\omega(p_{j+1} \to p_j)} \right)$$

- Implementation re-uses path $\bar{p}_{i-1}$ for new path $\bar{p}_i$ This introduces correlation, but speed makes up for it.

Path tracing

```
Step 1. Choose a camera ray r given the
   (x,y,u,v,t) sample
      weight = 1;
Step 2. Find ray-surface intersection
Step 3.
   if light
      return weight * Le();
   else
      weight *= reflectance(r)
      Choose new ray r' ~ BRDF pdf(r)
      Go to Step 2.
```

Direct lighting



Path tracing

8 samples per pixel

Path tracing

1024 samples per pixel

Bidirectional path tracing

- Compose one path $\bar{p}$ from two paths
  - $p_1 p_2 \ldots p_i$ started at the camera p_0 and
  - $q_j q_{j-1} \ldots q_1$ started at the light source q_0

$$\bar{p}_i = p_1 p_2 \ldots p_i, q_j q_{j-1} \ldots q_1$$

- Modification for efficiency:
  - Use all paths whose lengths ranging from 2 to i+j

$$\begin{array}{ll} p_1 \ldots p_i, q_j \ldots q_1 & p_1 \ldots p_i, q_j \ldots q_1 \\ p_1 \ldots p_{i-1}, q_j \ldots q_1 & p_1 \ldots p_i, q_{j-1} \ldots q_1 \\ p_1 \ldots p_{i-2}, q_j \ldots q_1 & p_1 \ldots p_i, q_{j-2} \ldots q_1 \\ \vdots & \vdots \\ p_1, q_j \ldots q_1 & p_1 \ldots p_i, q_1 \end{array}$$

Helpful for the situations in which lights are difficult to reach and caustics

Bidirectional path tracing

Bidirectional path tracing

Path tracing

Noise reduction/removal

- More samples (slow convergence)
- Better sampling (stratified, importance etc.)
- Filtering
- Caching and interpolation

Biased approaches

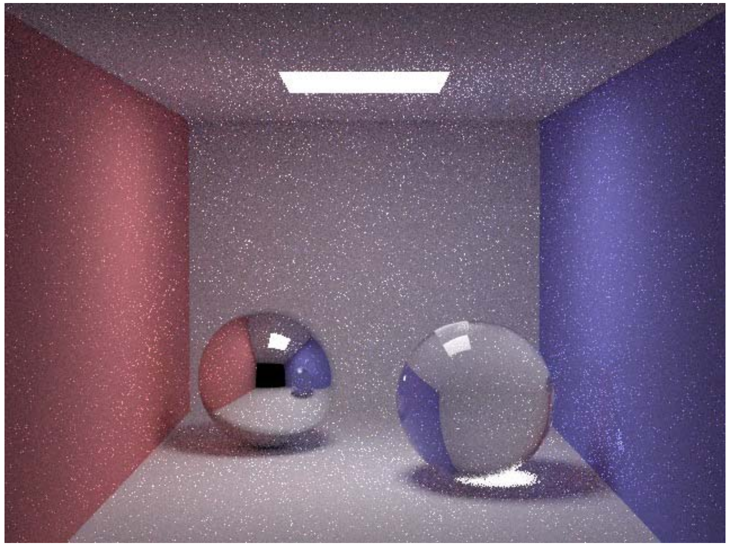
- By introducing bias (making smoothness assumptions), biased methods produce images without high-frequency noise
- Unlike unbiased methods, errors may not be reduced by adding samples in biased methods
- On contrast, when there is little error in the result of an unbiased method, we are confident that it is close to the right answer
- Three biased approaches
  - Filtering
  - Irradiance caching
  - Photon mapping

The world is more diffuse!

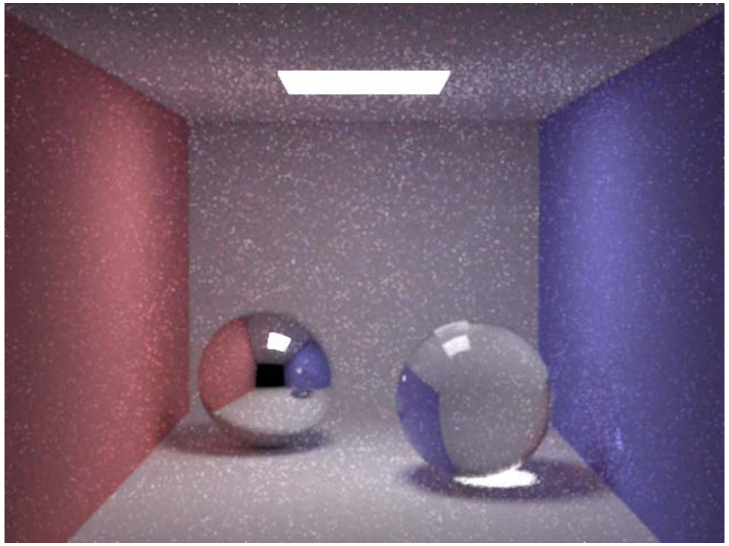
Filtering

- Noise is high frequency
- Methods:
  - Simple filters
  - Anisotropic filters
  - Energy preserving filters
- Problems with filtering: everything is filtered (blurred)

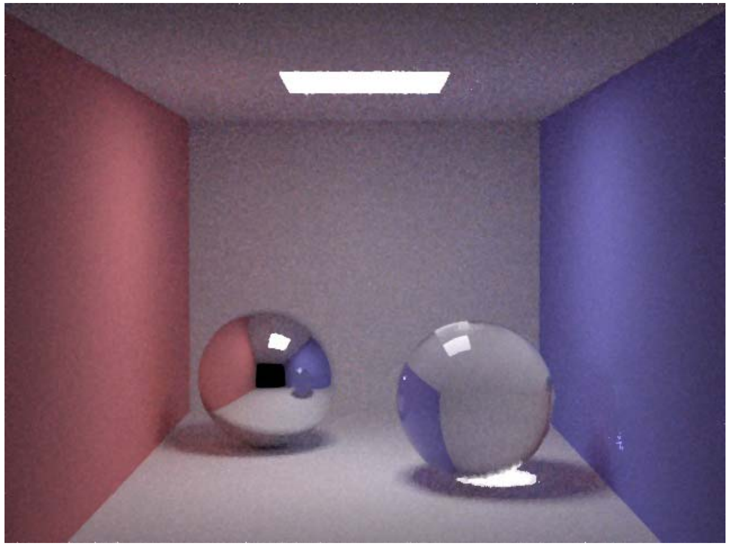
Path tracing (10 paths/pixel)



3x3 lowpass filter



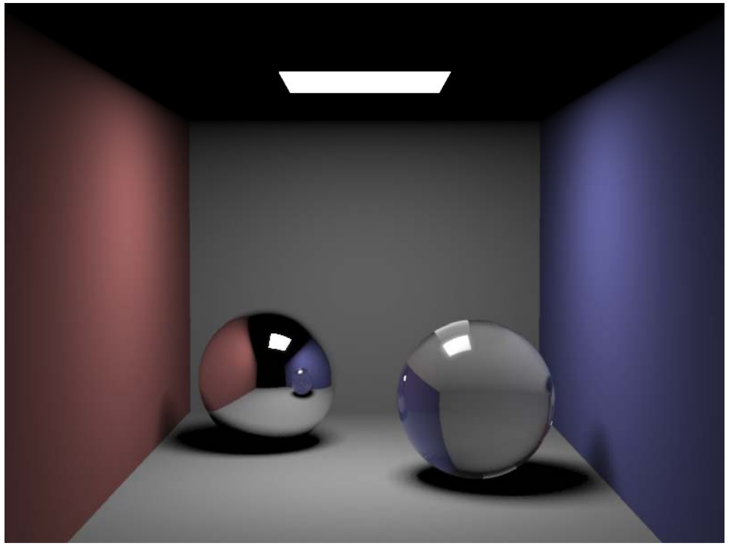
3x3 median filter



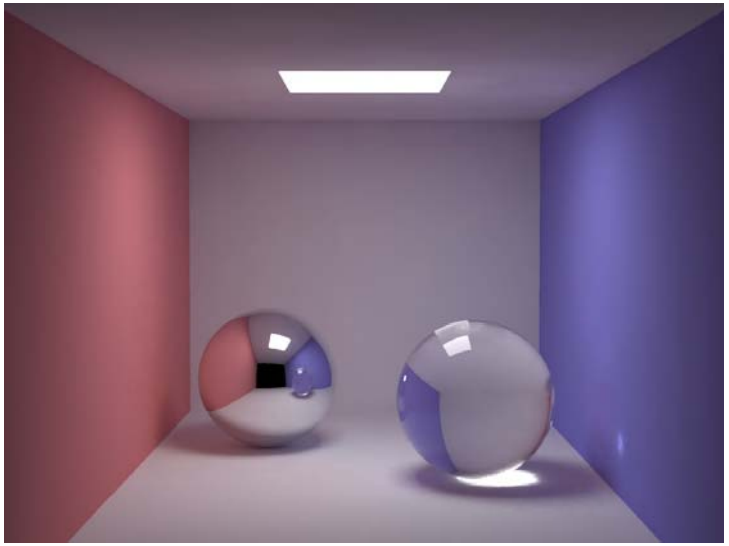
Caching techniques

- Irradiance caching: compute irradiance at selected points and interpolate
- Photon mapping: trace photons from the lights and store them in a photon map, that can be used during rendering

Direct illumination



Global illumination



Indirect irradiance

Irradiance caching

- Introduced by Greg Ward 1988
- Implemented in Radiance renderer
- Contributions from indirect lighting often vary smoothly → cache and interpolate results

Irradiance caching

- Compute indirect lighting at sparse set of samples
- Interpolate neighboring values from this set of samples
- Issues
  - How is the indirect lighting represented
  - How to come up with such a sparse set of samples?
  - How to store these samples?
  - When and how to interpolate?

Set of samples

- Indirect lighting is computed on demand, store *irradiance* in a spatial data structure. If there is no good nearby samples, then compute a new irradiance sample
- Irradiance (radiance is direction dependent, expensive to store)

$$E(p) = \int_{H^2} L_i(p,\omega_i)\,|\cos\theta_i|\,d\omega_i$$

- If the surface is Lambertian,

$$L_o(p,\omega_o) = \int_{H^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i)\,|\cos\theta_i|\,d\omega_i$$
$$= \int_{H^2} \rho L_i(p,\omega_i)\,|\cos\theta_i|\,d\omega_i$$
$$= \rho E(p)$$

Set of samples

- For diffuse scenes, irradiance alone is enough information for accurate computation
- For nearly diffuse surfaces (such as Oren-Nayar or a glossy surface with a very wide specular lobe), we can view irradiance caching makes the following approximation

$$L_o(p,\omega_o) \approx \left(\int_{H^2} f(p,\omega_o,\omega_i)\,d\omega_i\right)\left(\int_{H^2} L_i(p,\omega_i)\,|\cos\theta_i|\,d\omega_i\right)$$
$$\approx \left(\tfrac{1}{2}\rho_{hd}(\omega_o)\right)E(p)$$

↑ directional reflectance

Set of samples

- Not a good approximation for specular surfaces
- specular → Whitted integrator
- Diffuse → irradiance caching
  - Interpolate from known points
  - Cosine-weighted
  - Path tracing sample points

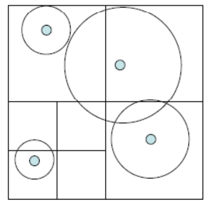
$$E(p) = \int_{H^2} L_i(p,\omega_i) |\cos\theta_i| \, d\omega_i$$

$$E(p) = \frac{1}{N} \sum_j \frac{L_i(p,\omega_j) |\cos\theta_j|}{p(\omega_j)} \qquad p(\omega) = \cos\theta / \pi$$

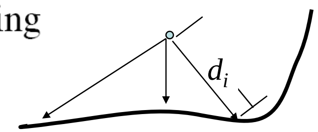
$$E(p) = \frac{\pi}{N} \sum_j L_i(p,\omega_j)$$

Storing samples

- Octree data structure
  - Each node stores samples that influence this node (each point has a radius of influence!)
- Radius of influence
  - determined by harmonic mean $\frac{N}{\sum_i^N 1/d_i}$
  - d_i is the distance that the ith ray (used for estimating the irradiance) traveled before intersecting an object
  - Computed during path tracing

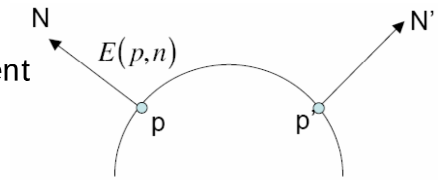


{E,p,n,d}



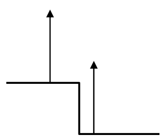
Interpolating from neighbors

- Skip samples
  - Normals are too different
  - Too far away
  - In front



- Weight (ad hoc)

$$w_i = \left(1 - \frac{d}{d_{max}} \frac{1}{N \cdot N'}\right)^2$$



- Final irradiance estimate is simply the weighted sum

$$E = \frac{\sum_i w_i E_i}{\sum_i w_i}$$

IrradianceCache

```
class IrradianceCache : public SurfaceIntegrator {
  ...
  float maxError;  how frequently irradiance samples are
                   computed or interpolated
  int nSamples;   how many rays for irradiance samples
  int maxSpecularDepth, maxIndirectDepth;
  mutable int specularDepth;  current depth for specular
}
```

IrradianceCache::Li

```
L += isect.Le(wo);
L += UniformSampleAllLights(scene, p, n, wo,...);
if (specularDepth++ < maxSpecularDepth) {
  <Trace rays for specular reflection and
   refraction>
}
--specularDepth;

// Estimate indirect lighting with irradiance cache
...
BxDFType flags = BxDFType(BSDF_REFLECTION |
                 BSDF_DIFFUSE | BSDF_GLOSSY);
L+=IndirectLo(p, ng, wo, bsdf, flags, sample, scene);

flags = BxDFType(BSDF_TRANSMISSION |
                 BSDF_DIFFUSE | BSDF_GLOSSY);
L+=IndirectLo(p, -ng, wo, bsdf, flags, sample,scene);
```

IrradianceCache::IndirectLo

```
if (!InterpolateIrradiance(scene, p, n, &E)) {
  ... // Compute irradiance at current point
  for (int i = 0; i < nSamples; ++i) {
    <Path tracing to compute radiances along ray
     for irradiance sample>
    E += L;
    float dist = r.maxt * r.d.Length(); // max distance
    sumInvDists += 1.f / dist;
  }
  E *= M_PI / float(nSamples);
  ... // Add computed irradiance value to cache
  octree->Add(IrradianceSample(E,p,n,nSamples/sumInvDists),
                               sampleExtent);
}
return .5f * bsdf->rho(wo, flags) * E;
```

Octree

- Constructed at **`Preprocess()`**

```
void IrradianceCache::Preprocess(const Scene *scene)
{
 BBox wb = scene->WorldBound();
 Vector delta = .01f * (wb.pMax - wb.pMin);
 wb.pMin -= delta;
 wb.pMax += delta;
 octree=new Octree<IrradianceSample,IrradProcess>(wb);
}

struct IrradianceSample {
  Spectrum E;
  Normal n;
  Point p;
  float maxDist;
};
```

InterpolateIrradiance

```
Bool InterpolateIrradiance(const Scene *scene,
      const Point &p, const Normal &n, Spectrum *E)
{
  if (!octree) return false;
  IrradProcess proc(n, maxError);
  octree->Lookup(p, proc);
   Traverse the octree; for each node where the query point is inside, call
   a method of proc to process for each irradiacne sample.

  if (!proc.Successful()) return false;
  *E = proc.GetIrradiance();
  return true;
}
```

IrradProcess

```
void IrradProcess::operator()(const Point &p,
      const IrradianceSample &sample)
{
 // Skip if surface normals are too different
 if (Dot(n, sample.n) < 0.01f) return;
 // Skip if it's too far from the sample point
 float d2 = DistanceSquared(p, sample.p);
 if (d2 > sample.maxDist * sample.maxDist) return;
 // Skip if it's in front of point being shaded
 Normal navg = sample.n + n;
 if (Dot(p - sample.p, navg) < -.01f) return;
 // Compute estimate error and possibly use sample
 float err=sqrtf(d2)/(sample.maxDist*Dot(n,sample.n));
 if (err < 1.) {
   float wt = (1.f - err) * (1.f - err);
   E += wt * sample.E;  sumWt += wt;
 }
}
```

$$w_i = \left(1 - \frac{d}{d_{\max}} \frac{1}{N \cdot N'}\right)^2$$

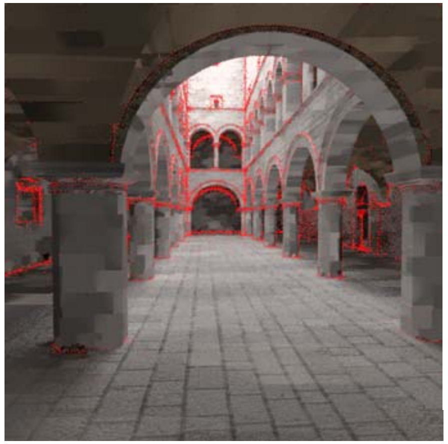
Comparison with same limited time

Irradiance caching
Blotch artifacts

Path tracing
High-frequency noises

Irradiance caching

Irradiance caching



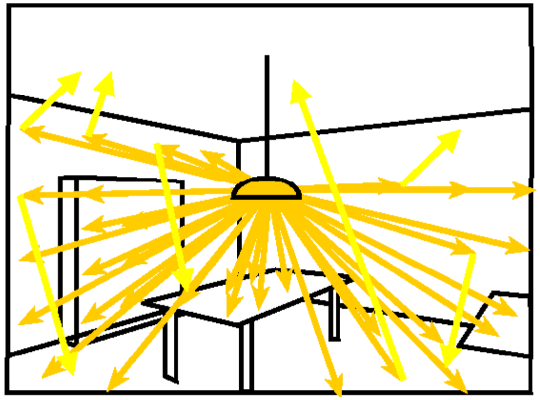
Irradiance sample positions

Photon mapping

- It can handle both diffuse and glossy reflection; specular reflection is handled by recursive ray tracing
- Two-step particle tracing algorithm
- Photon tracing
  - Simulate the transport of individual photons
  - Photons emitted from source
  - Photons deposited on surfaces
  - Photons reflected from surfaces to surfaces
- Rendering
  - Collect photons for rendering

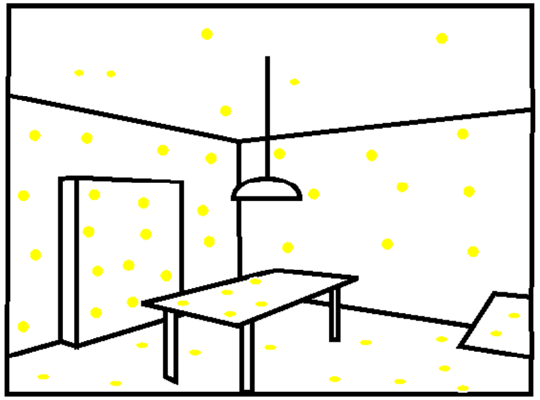
Photon tracing

- Preprocess: cast rays from light sources



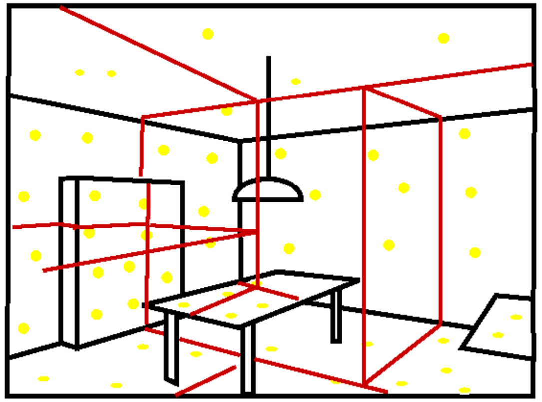
Photon tracing

- Preprocess: cast rays from light sources
- Store photons (position + light power + incoming direction)



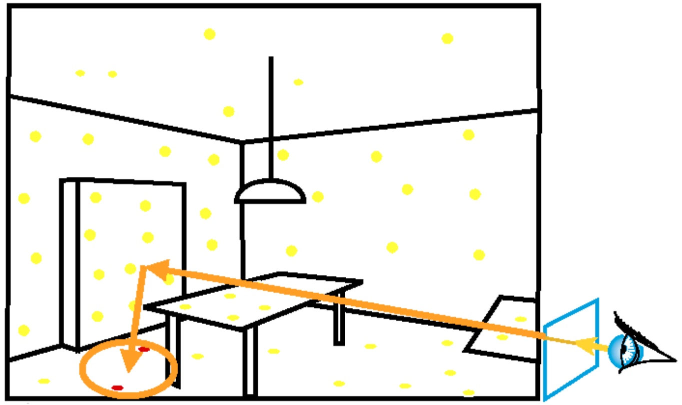
Photon map

- Efficiently store photons for fast access
- Use hierarchical spatial structure (kd-tree)



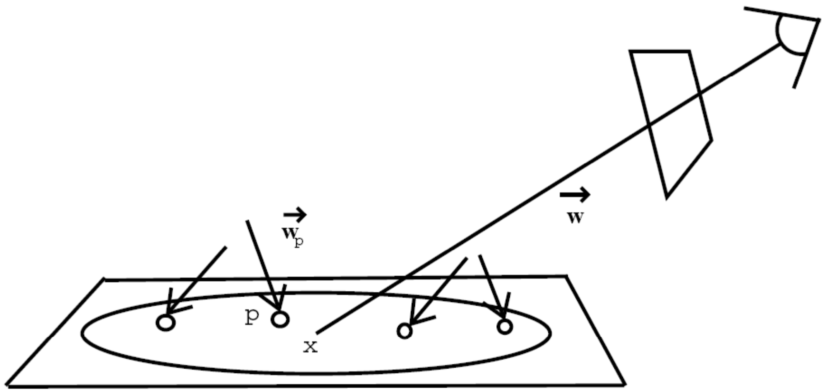
Rendering (final gathering)

- Cast primary rays; for the secondary rays, reconstruct irradiance using the k closest stored photon

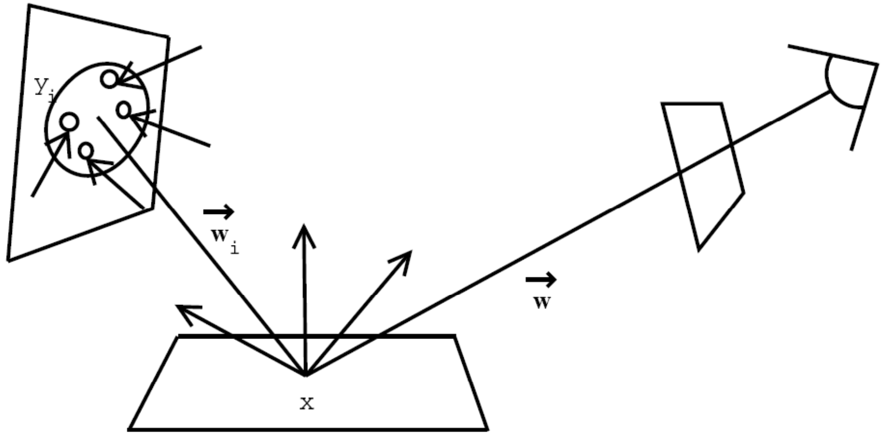


Rendering (without final gather)

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| \, d\omega_i$$



Rendering (with final gather)

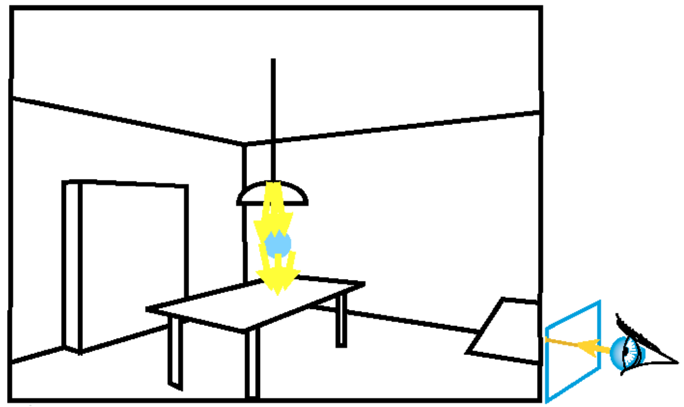


Photon mapping results

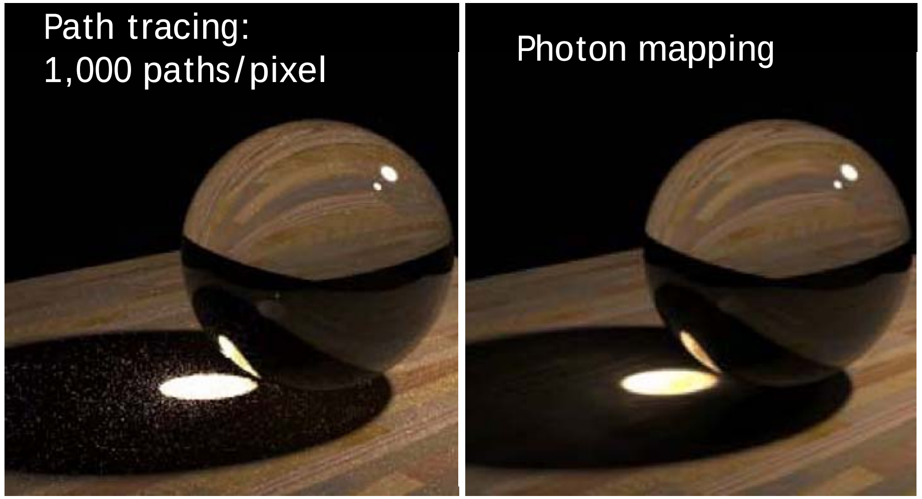
photon map rendering

Photon mapping - caustics

- Special photon map for specular reflection and refraction



Caustics

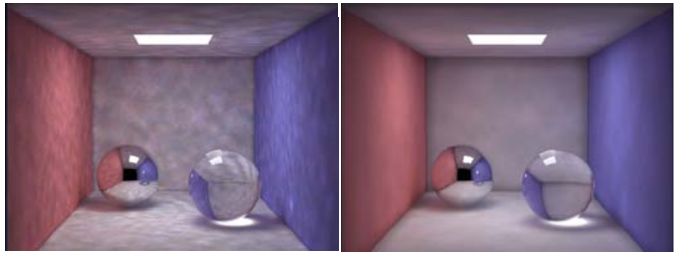


PhotonIntegrator

```
class PhotonIntegrator : public SurfaceIntegrator {
  int nCausticPhotons,nIndirectPhotons,nDirectPhotons;
  int nLookup; number of photons for interpolation (50~100)
  int specularDepth, maxSpecularDepth;
  float maxDistSquared; search distance; too large, waste
                        time; too small, not enough samples
  bool directWithPhotons, finalGather;
  int gatherSamples;
}
```

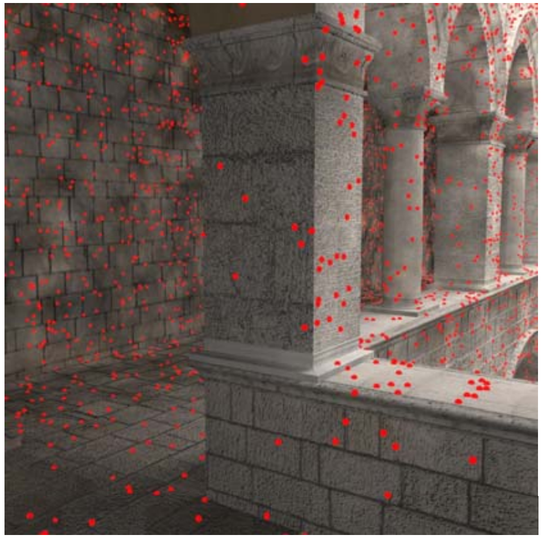
Left:
100K photons
50 photons in
radiance estimate

Right:
500K photons
500 photons in
radiance estimate



Photon map

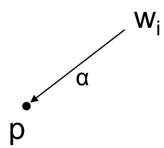
Kd-tree is used to store photons, decoupled from the scene geometry

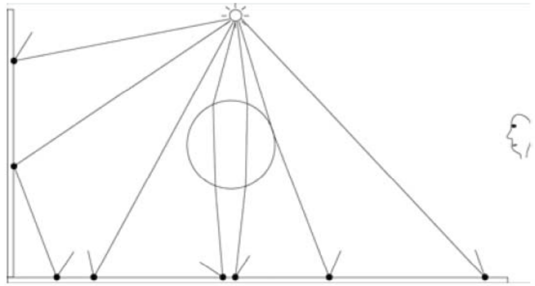


Photon shooting

- Implemented in **`Preprocess`** method
- Three types of photons (caustic, direct, indirect)

```
struct Photon {
  Point p;
  Spectrum alpha;
  Vector wi;
};
```

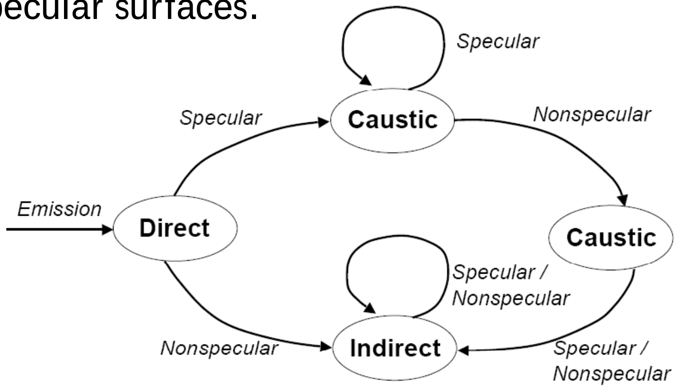




For 100 photons emitted from 100W source, each photon initially carries 1W.

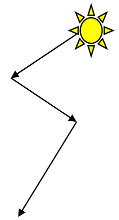
Photon shooting

- Use Halton sequence since number of samples is unknown beforehand, starting from a sample light with energy $\frac{L_e(p_0,\omega_0)}{p(p_0,\omega_0)}$. Store photons for non-specular surfaces.



Photon shooting

```
void PhotonIntegrator::Preprocess(const Scene *scene)
{
  vector<Photon> causticPhotons;
  vector<Photon> directPhotons;
  vector<Photon> indirectPhotons;
  while (!causticDone || !directDone|| !indirectDone)
  {
    ++nshot;
    <trace a photon path and store contribution>
  }
}
```



Photon shooting

```
Spectrum alpha = light->Sample_L(scene, u[0], u[1],
                     u[2], u[3], &photonRay, &pdf);
alpha /= pdf * lightPdf;
While (scene->Intersect(photonRay, &photonIsect)) {
 alpha *= scene->Transmittance(photonRay);
 <record photon depending on type>
 <sample next direction>
 Spectrum fr = photonBSDF->Sample_f(wo, &wi, u1, u2,
                       u3, &pdf, BSDF_ALL, &flags);
 alpha*=fr*AbsDot(wi, photonBSDF->dgShading.nn)/ pdf;
 photonRay = RayDifferential(photonIsect.dg.p, wi);
 if (nIntersections > 3) {
   if (RandomFloat() > .5f) break;
   alpha /= .5f;
 }
}
```

Rendering

- Partition the integrand

$$\int_{S^2} f(p,\omega_o,\omega_i)L_i(p,\omega_i)|\cos\theta_i|\,d\omega_i$$

$$= \int_{S^2} f_\Delta(p,\omega_o,\omega_i)L_i(p,\omega_i)|\cos\theta_i|\,d\omega_i$$

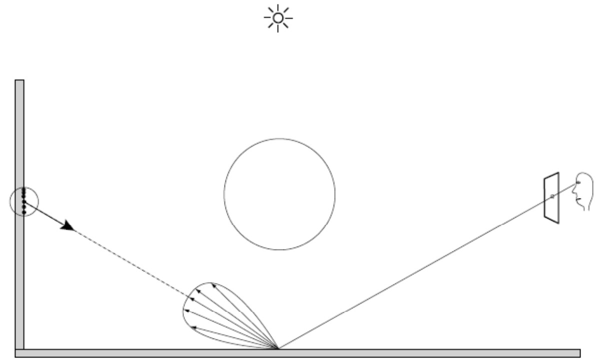
$$+ \int_{S^2} f_{\neg\Delta}(p,\omega_o,\omega_i)\left(L_{i,d}(p,\omega_i)+L_{i,i}(p,\omega_i)+L_{i,c}(p,\omega_i)\right)|\cos\theta_i|\,d\omega_i$$

Rendering

```
L += isect.Le(wo);

// Compute direct lighting for photon map integrator
if (directWithPhotons) L += LPhoton(directMap,...);
else L += UniformSampleAllLights(...);

// Compute indirect lighting for photon map integrator
L += LPhoton(causticMap, ...);

if (finalGather) {
  <Do one-bounce final gather for photon map>
} else
  L += LPhoton(indirectMap, ...);
// Compute specular reflection and refraction
```

Final gather

```
for (int i = 0; i < gatherSamples; ++i) {
  <compute radiance for a random BSDF-sampled
   direction for final gather ray>
}
L += Li/float(gatherSamples);
```



Final gather

```
BSDF *gatherBSDF = gatherIsect.GetBSDF(bounceRay);
Vector bounceWo = -bounceRay.d;
Spectrum Lindir =
  LPhoton(directMap, nDirectPaths, nLookup,
   gatherBSDF, gatherIsect, bounceWo, maxDistSquared)
+ LPhoton(indirectMap, nIndirectPaths, nLookup,
   gatherBSDF, gatherIsect, bounceWo, maxDistSquared)
+ LPhoton(causticMap, nCausticPaths, nLookup,
   gatherBSDF, gatherIsect, bounceWo,maxDistSquared);
Lindir *= scene->Transmittance(bounceRay);
Li += fr * Lindir * AbsDot(wi, n) / pdf;
```

Rendering

50,000 direct photons

shadow rays are traced for direct lighting

Rendering

500,000 direct photons

caustics

Photon mapping

Direct illumination

Photon mapping

Photon mapping + final gathering

Photon mapping
+final gathering

Photon mapping

Photon interpolation

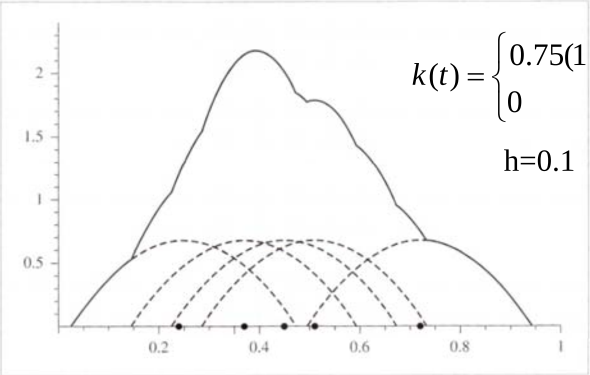
- **`LPhoton()`** finds the **`nLookup`** closest photons and uses them to compute the radiance at the point.
- A kd-tree is used to store photons. To maintain the **`nLookup`** closest photons efficiently during search, a heap is used.
- For interpolation, a statistical technique, density estimation, is used. Density estimation constructs a PDF from a set of given samples, for example, histogram.

Kernel method

$$\hat{p}(x) = \frac{1}{Nh}\sum_{i=1}^{N} k\left(\frac{x - x_i}{h}\right) \quad \text{where} \int_{-\infty}^{\infty} \mathrm{k(x)dx} = 1$$

window width (↑ h)

h too wide→too smooth
h too narrow→too bumpy



$$k(t) = \begin{cases} 0.75(1 - 2t^2)/\sqrt{5} & t < \sqrt{5} \\ 0 & \text{otherwise} \end{cases}$$

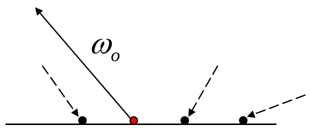
h=0.1

Generalized nth nearest-neighbor estimate

$$\hat{p}(x) = \frac{1}{Nd_n(x)}\sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right)$$

distance to nth nearest neighbor (↑ $d_n(x)$)

2D constant kernel $k(x) = \begin{cases} \frac{1}{\pi} & |x| < 1 \\ 0 & \text{otherwise} \end{cases}$




```
float scale=1.f/(float(nPaths)*maxDistSquared* M_PI);
```

LPhoton

```
if (bsdf->NumComponents(BxDFType(BSDF_REFLECTION |
            BSDF_TRANSMISSION | BSDF_GLOSSY)) > 0) {
// exitant radiance from photons for glossy surface
for (int i = 0; i < nFound; ++i) {
  BxDFType flag=Dot(Nf, photons[i].photon->wi)> 0.f ?
         BSDF_ALL_REFLECTION : BSDF_ALL_TRANSMISSION;
  L += bsdf->f(wo, photons[i].photon->wi, flag) *
              (scale * photons[i].photon->alpha);
}} else {
// exitant radiance from photons for diffuse surface
Spectrum Lr(0.), Lt(0.);
for (int i = 0; i < nFound; ++i)
  if (Dot(Nf, photons[i].photon->wi) > 0.f)
    Lr += photons[i].photon->alpha;
  else Lt += photons[i].photon->alpha;
L+=(scale*INV_PI)*(Lr*bsdf->rho(wo,BSDF_ALL_REFLECTION)
            +Lt*bsdf->rho(wo, BSDF_ALL_TRANSMISSION));
}
```

Results