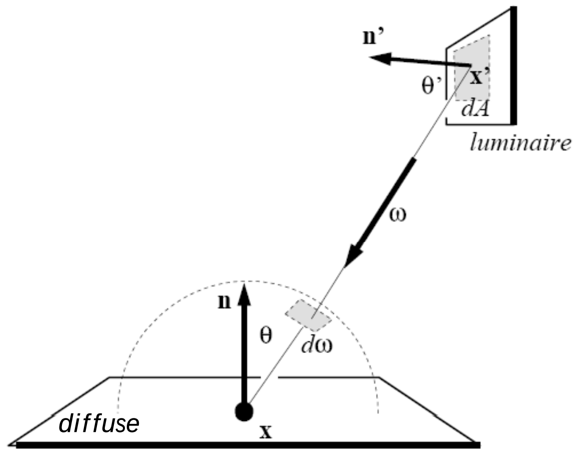
Surface Integrators

Digital Image Synthesis

Yung-Yu Chuang

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

Direct lighting via Monte Carlo integration



$$L(x) = L_e(x) + \frac{R(x)}{\pi} \int_{\text{all } \vec{\omega}'} L_e(x, \vec{\omega}') \cos\theta \, d\vec{\omega}'$$

Direct lighting via Monte Carlo integration

parameterization over hemisphere

$$L(x) = L_e(x) + \frac{R(x)}{\pi} \int_{\text{all } \vec{\omega}'} L_e(x, \vec{\omega}') \cos\theta \, d\vec{\omega}'$$

$$d\vec{\omega}' = \frac{dA \cos\theta'}{\|x' - x\|^2}$$

parameterization over surface

$$L(x) = L_e(x) + \frac{R(x)}{\pi} \int_{\text{all } x'} L_e(x') \cos\theta \frac{dA \cos\theta'}{\|x' - x\|^2}$$

have to add visibility

$$L(x) = L_e(x) + \frac{R(x)}{\pi} \int_{\text{all } x'} L_e(x') \cos\theta \frac{s(x,x') dA \cos\theta'}{\|x' - x\|^2}$$

Direct lighting via Monte Carlo integration

take one sample according to a density function $x' \sim p$

$$L(x) \approx L_e(x) + \frac{R(x)}{\pi} L_e(x') \cos\theta \frac{s(x,x') \cos\theta'}{p(x') \|x' - x\|^2}$$

let's take $p = 1/A$

$$L(x) \approx L_e(x) + \frac{R(x)}{\pi} L_e(x') \cos\theta \frac{A \, s(x,x') \cos\theta'}{\|x' - x\|^2}$$

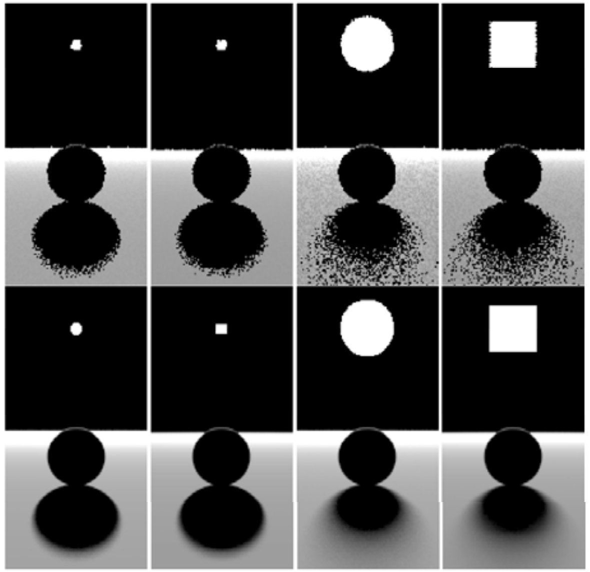
spectrum directLight(x, $\vec{n}$ *)*
pick random point x' *with normal vector* $\vec{n}'$ *on light*
$\vec{d} = (x' - x)$
if *ray* $x + t\vec{d}$ *hits at* x' ***then***
  return $A L_e(x')(\vec{n} \cdot \vec{d})(-\vec{n}' \cdot \vec{d}) / \|\vec{d}\|^4$
else
  return 0

Direct lighting via Monte Carlo integration

1 sample/pixel

100 samples/pixel

Lights' sizes matter more than shapes.
Noisy because
- *x' could be on the back*
- *cos varies*



Noise reduction

$$L(x) \approx L_e(x) + \frac{R(x)}{\pi} L_e(x') \cos\theta \frac{s(x,x')\cos\theta'}{p(x')\|x'-x\|^2}$$

choose better density function $p(x') \propto \cos\theta' / \|x'-x\|^2$

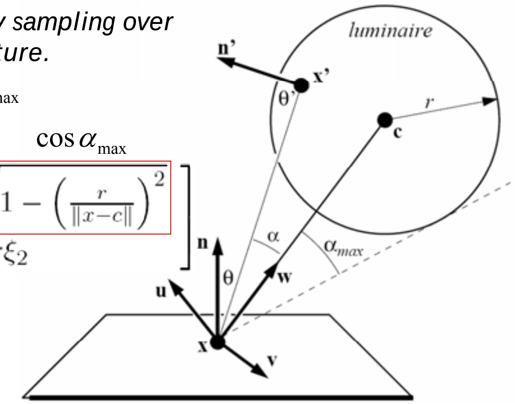
It is equivalent to uniformly sampling over the cone cap in the last lecture.

$$\cos\theta = (1-\xi_1) + \xi_1 \cos\theta_{\max}$$

$$\phi = 2\pi\xi_2$$

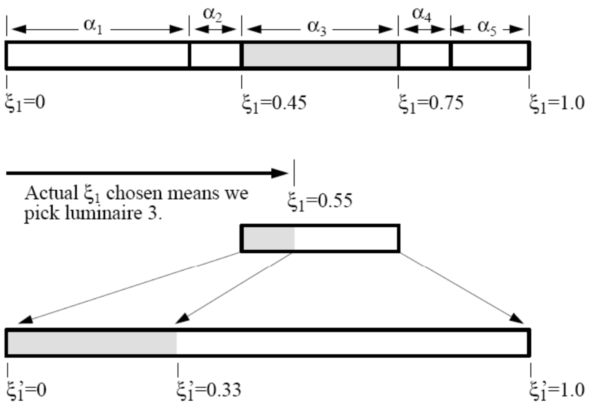
$\cos\alpha_{\max}$

$$\begin{bmatrix} \cos\alpha \\ \phi \end{bmatrix} = \begin{bmatrix} 1 - \xi_1 + \xi_1 \boxed{\sqrt{1 - \left(\frac{r}{\|x-c\|}\right)^2}} \\ 2\pi\xi_2 \end{bmatrix}$$



Direct lighting from many luminaries

- Given a pair (ξ_1, ξ_2), use it to select light and generate new pair (ξ_1', ξ_2) for sampling that light.
- α could be constant for proportional to power



Rendering

- Rendering is handled by **`Renderer`** class.

```
class Renderer {
  …    given a scene, render an image or a set of measurements
  virtual void Render(Scene *scene) = 0;

       computer radiance along a ray
  virtual Spectrum Li(Scene *scn, RayDifferential &r,
for MC sampling Sample *sample, RNG &rng,
            MemoryArena &arena, Intersection *isect,
 transmittance Spectrum *T) const = 0;
       return transmittance along a ray
  virtual Spectrum Transmittance(Scene *scene,
            RayDifferential &ray, Sample *sample,
            RNG &rng, MemoryArena &arena) const = 0;
};    The later two are usually relayed to Integrator
```

SamplerRenderer

```
class SamplerRenderer : public Renderer {
  …
  private:
    // SamplerRenderer Private Data
    Sampler *sampler;
```

choose samples on image plane and for integration

```
    Camera *camera;
```

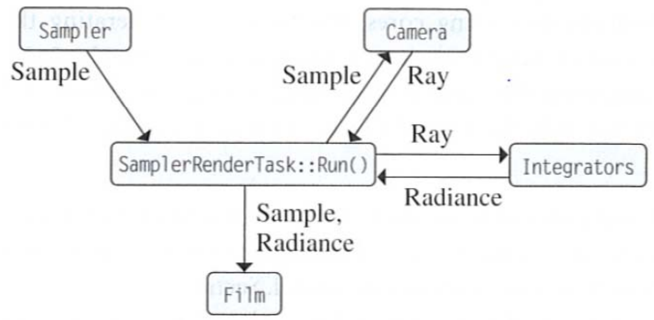
determine lens parameters (position, orientation, focus, field of view) with a film

```
    SurfaceIntegrator *surfaceIntegrator;
    VolumeIntegrator *volumeIntegrator;
};
```

calculate the rendering equation

The main rendering loop

- After **scene** and **Renderer** are constructed, **Renderer:Render()** is invoked.



Renderer:Render()

```
void SamplerRenderer::Render(const Scene *scene) {
  …
```

scene-dependent initialization such photon map

```
  surfaceIntegrator->Preprocess(scene,camera,this);
  volumeIntegrator->Preprocess(scene,camera,this);
```

sample structure depends on types of integrators

```
  Sample *sample = new Sample(sampler,
        surfaceIntegrator, volumeIntegrator, scene);
```

We want many tasks to fill in the core (see histogram next page). If there are too few, some core will be idle. But, threads have overheads. So, we do not want too many either.

```
  int nPixels = camera->film->xResolution
              * camera->film->yResolution;
  int nTasks = max(32 * NumSystemCores(),
              nPixels / (16*16));
  nTasks = RoundUpPow2(nTasks);
```

at least 32 tasks for a core

a task is about 16x16

power2 easier to divide

Renderer:Render()

```
  vector<Task *> renderTasks;
  for (int i = 0; i < nTasks; ++i)
    renderTasks.push_back(new
    SamplerRendererTask(scene,this,camera,reporter,
      sampler, sample, nTasks-1-i, nTasks));
```

all information about renderer must be passed in

task id

total tasks

```
  EnqueueTasks(renderTasks);
  WaitForAllTasks();
  for (int i = 0; i < renderTasks.size(); ++i)
    delete renderTasks[i];

  delete sample;
  camera->film->WriteImage();
}
```

SamplerRenderTask::Run

- When the task system decided to run a task on a particular processor, **SamplerRenderTask::Run()** will be called.

```
void SamplerRendererTask::Run() {
  // decided which part it is responsible for
  …
  int sampleCount;
  while ((sampleCount=sampler ->
    GetMoreSamples(samples, rng)) > 0) {
    // Generate camera rays and compute radiance
```

SamplerRenderTask::Run

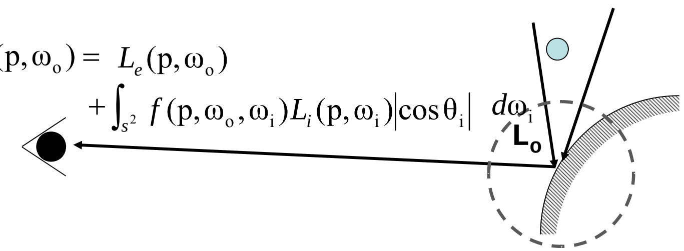
```
  for (int i = 0; i < sampleCount; ++i) {
             for vignetting
  float rayWeight = camera->        ray differential
                                    for antialiasing
      GenerateRayDifferential(samples[i], &rays[i]);
  rays[i].ScaleDifferentials(
                1.f / sqrtf(sampler->samplesPerPixel));

  if (rayWeight > 0.f)
    Ls[i] = rayWeight * renderer->Li(scene, rays[i],
      &samples[i], rng, arena, &isects[i], &Ts[i]);
  else { Ls[i] = 0.f; Ts[i] = 1.f; }

  for (int i = 0; i < sampleCount; ++i)
    camera->film->AddSample(samples[i], Ls[i]);
}
```

SamplerRender::Li

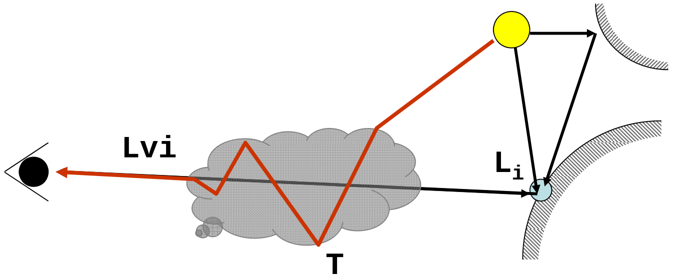
```
Spectrum SamplerRender::Li(Scene *scene,
  RayDifferential &ray, Sample *sample,
  …, Intersection *isect, Spectrum *T)
{ Spectrum Li = 0.f;
  if (scene->Intersect(ray, isect))
    Li = surfaceIntegrator->Li(scene,this,
                ray, *isect, sample,rng, arena);
  else { // ray that doesn't hit any geometry
    for (i=0; i<scene->lights.size(); ++i)
      Li += scene->lights[i]->Le(ray);
  }
  Spectrum Lvi = volumeIntegrator->Li(scene, this,
                 ray, sample, rng, T, arena);
  return *T * Li + Lvi;
}
```

Surface integrator's Li

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



SamplerRender::Li



Integrators

- `core/integrator.* integrator/*`

```
Class Integrator {
  virtual void Preprocess(Scene *scene,
    Camera *camera, Renderer *renderer){}
  virtual void RequestSamples(Sampler
    *sampler, Sample *sample, Scene *scene){}
};
```

Integrators

- `void Preprocess(…)`

  Called after scene has been initialized; do scene-dependent computation such as photon shooting for photon mapping.
- `void RequestSamples(…)`

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

```
Sample::Sample(Sampler *sampler, SurfaceIntegrator
*surf, VolumeIntegrator *vol, Scene *scene) {
  if (surf)
    surf>RequestSamples(sampler,this,scene);
  if (vol)
    vol->RequestSamples(sampler, this, scene);
  …
```

Surface integrators

- Responsible for evaluating the integral equation

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

```
class SurfaceIntegrator:public Integrator {
public:
  virtual Spectrum Li(Scene *scene, Renderer
    *renderer, RayDifferential &ray,
    Intersection &isect, Sample *sample,
    RNG &rng, MemoryArena &arena) const = 0;
};
```

We could call Renderer's Li or Transmittance

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)
- major contribution to the final radiance
- not too bad since most energy comes from direct lights
- kind of what we do in Whitted ray tracing

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

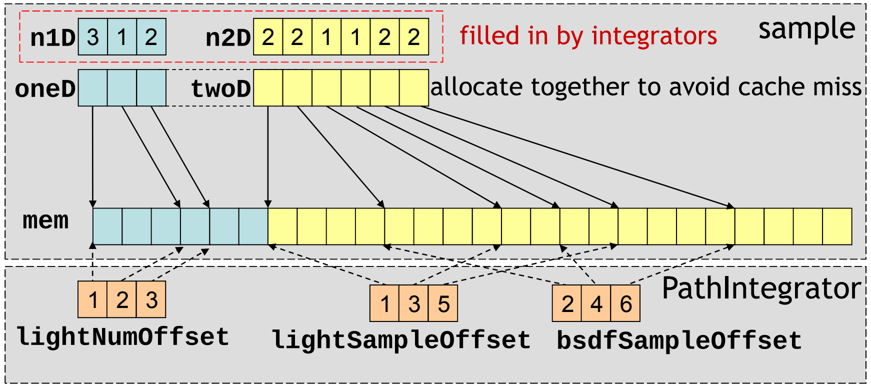
```
enum LightStrategy {
  SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM
};
```

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 often prefer sampling all lights at once.

```
class DirectLighting : public SurfaceIntegrator {
public:
  DirectLighting(
    LightStrategy ls = SAMPLE_ALL_UNIFORM,
    int md=5 maximal depth
  );
  ...
}
```

Data structure

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



DirectLighting::RequestSamples

```
void DirectLightingIntegrator::RequestSamples(
  Sampler *sampler, Sample *sample, Scene *scene) {
 if (strategy == SAMPLE_ALL_UNIFORM) {
   uint32_t nLights = scene->lights.size();
   lightSampleOffsets=new LightSampleOffsets[nLights];
   bsdfSampleOffsets = new BSDFSampleOffsets[nLights];
   for (uint32_t i = 0; i < nLights; ++i) {
     const Light *light = scene->lights[i];
     int nSamples = light->nSamples;
                 gives sampler a chance to adjust to an appropriate value
     if (sampler) nSamples=sampler->RoundSize(nSamples);
     lightSampleOffsets[i]
       = LightSampleOffsets(nSamples, sample);
     bsdfSampleOffsets[i]
       = BSDFSampleOffsets(nSamples, sample);
   }
   lightNumOffset = -1;
 }
```

DirectLighting::RequestSamples

```
 else {
   lightSampleOffsets = new LightSampleOffsets[1];
   lightSampleOffsets[0]
     = LightSampleOffsets(1, sample);
         which light to sample
   lightNumOffset = sample->Add1D(1);
   bsdfSampleOffsets = new BSDFSampleOffsets[1];
   bsdfSampleOffsets[0] = BSDFSampleOffsets(1, sample);
 }
}
```

lightSampleOffsets records where the samples are in the sample structure. With this information, we can drive the required random numbers for generating light samples and store all random numbers required for one sample in LightSample. Similar for bsdfSample.

DirectLighting::Li

```
Spectrum DirectLighting::Li(…)
{
  Spectrum L(0.f);
  BSDF *bsdf = isect.GetBSDF(ray, arena);
  Vector wo = -ray.d;
  const Point &p = bsdf->dgShading.p;
  const Normal &n = bsdf->dgShading.nn;
  L += isect.Le(wo);

  if (scene->lights.size() > 0) {
    switch (strategy) {
      case SAMPLE_ALL_UNIFORM:
        L += UniformSampleAllLights(scene, renderer,
               arena, p, n, wo, isect.rayEpsilon,
               ray.time, bsdf, sample, rng,
               lightSampleOffsets, bsdfSampleOffsets);
        break;
```

DirectLighting::Li

```
      case SAMPLE_ONE_UNIFORM:
        L += UniformSampleOneLight(scene, renderer,
               arena, p, n, wo, isect.rayEpsilon,
               ray.time, bsdf, sample, rng,
               lightNumOffset, lightSampleOffsets,
               bsdfSampleOffsets);
        break;
    }
  }
  if (ray.depth + 1 < maxDepth) {
    Vector wi;
    L += SpecularReflect(…);
    L += SpecularTransmit(…);
  }
  return L;
}
```

This part is essentially the same as Whitted integrator. The main difference 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
for (int i = 0; i < scene->lights.size(); ++i) {
  Vector wi;
  float pdf;
  VisibilityTester visibility;
  Spectrum Li = scene->lights[i]->Sample_L(…);
  if (Li.IsBlack() || pdf == 0.f) continue;
  Spectrum f = bsdf->f(wo, wi);
  if (!f.IsBlack() && visibility.Unoccluded(scene))
      L += f * Li * AbsDot(wi, n) *
           visibility.Transmittance(…) / pdf;
}
```

UniformSampleAllLights

```
Spectrum UniformSampleAllLights(...)
{
  Spectrum L(0.);
  for (u_int i=0;i<scene->lights.size();++i) {
    Light *light = scene->lights[i];
    int nSamples = lightSampleOffsets ?
                   lightSampleOffsets[i].nSamples : 1;
    Spectrum Ld(0.);
    for (int j = 0; j < nSamples; ++j) {
      <Find light and BSDF sample values>
[[lightSample=LightSample(sample,lightSampleOffsets[i],j);]]
      Ld += EstimateDirect(...); }
    L += Ld / nSamples;
  }
  return L;
}
```

compute contribution for one sample for one light

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

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

UniformSampleOneLight

```
Spectrum UniformSampleOneLight (...)
{
  int nLights = int(scene->lights.size());
  if (nLights == 0) return Spectrum(0.);
  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];
  <Find light and BSDF sample values>
  return (float)nLights * EstimateDirect(...);
}
```

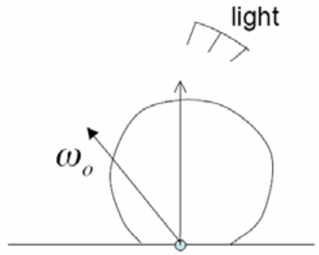
EstimateDirect

```
Spectrum EstimateDirect(Scene *scene, Renderer *renderer,
  Light *light, Point &p, Normal &n, Vector &wo,
  float rayEpsilon, float time, BSDF *bsdf, RNG &rng,
  LightSample &lightSample, BSDFSample &bsdfSample,
  BxDFType flags)
{
  Spectrum Ld(0.);
  Vector wi;
  float lightPdf, bsdfPdf;
  VisibilityTester visibility;
```

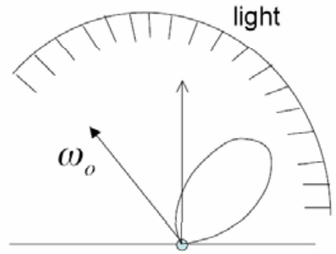
$$\frac{f(p,\omega_o,\omega_j)L_d(p,\omega_j)\,|\cos\theta_j\,|}{p(\omega_j)}$$

Here, we use multiple importance sampling to estimate the above term by taking one sample according to the light and the other according to BSDF.

Multiple importance sampling



Weight according to L_d



Weight according to f

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

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

Here, $n_f=n_g=1$

Sample light with MIS

```
  Spectrum Li = light->Sample_L(p, rayEpsilon, lightSample,
                     time, &wi, &lightPdf, &visibility);
  if (lightPdf > 0. && !Li.IsBlack()) {
    Spectrum f = bsdf->f(wo, wi, flags);
    if (!f.IsBlack() && visibility.Unoccluded(scene)) {
      Li *= visibility.Transmittance(…);
      if (light->IsDeltaLight())
        Ld += f * Li * (AbsDot(wi, n) / lightPdf);
      else {
        bsdfPdf = bsdf->Pdf(wo, wi, flags);
        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

```
  if (!light->IsDeltaLight()) {
    BxDFType sampledType;
    Spectrum f = bsdf->Sample_f(wo, &wi, bsdfSample,
                   &bsdfPdf, flags, &sampledType);
    if (!f.IsBlack() && bsdfPdf > 0.) {
      float weight = 1.f;
      if (!(sampledType & BSDF_SPECULAR)) {
        lightPdf = light->Pdf(p, wi);
        if (lightPdf == 0.) return Ld;
        weight = PowerHeuristic(1, bsdfPdf, 1, lightPdf);
      }
```

If it is delta light, no need to sample BSDF

weight=1 is for specular lights

We need to test whether we can see the light along the sampled direction

```
      Intersection lightIsect;
      Spectrum Li(0.f);
      RayDifferential ray(p, wi, rayEpsilon, INFINITY, time);
```

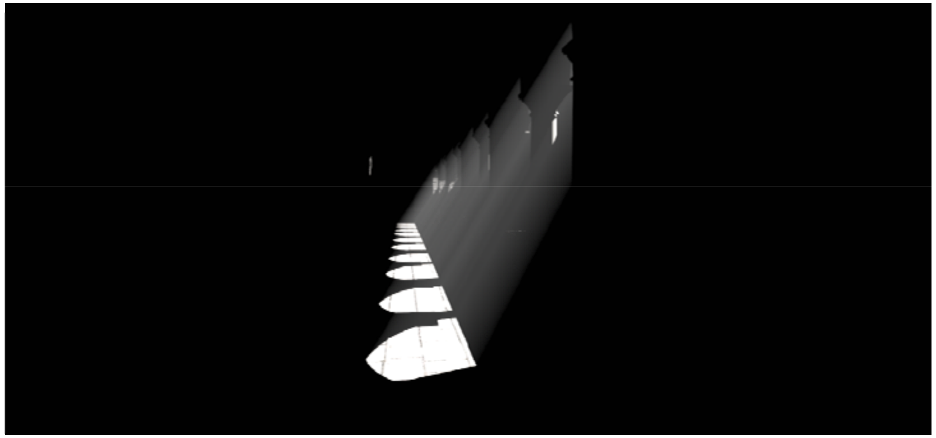
Sample BRDF with MIS

```
      if (scene->Intersect(ray, &lightIsect)) {
         If we can see it, record its Li
        if (lightIsect.primitive->GetAreaLight() == light)
             Li = lightIsect.Le(-wi);
      } else
        Li = light->Le(ray);
```

No intersection, but it could be an infinite area light. For non-infinite-area lights, Le return 0.

```
      if (!Li.IsBlack()) {
        Li *= renderer->Transmittance(…);
        Ld += f * Li * AbsDot(wi, n) * weight / bsdfPdf;
      }
   }
   return Ld;
}
```

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.

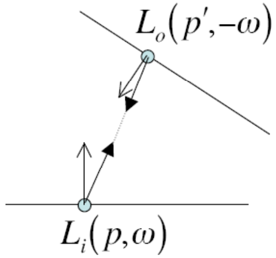
$$L_o(x,\omega_o) = L_e(x,\omega_o) + L_r(x,\omega_o)$$
$$= L_e(x,\omega_o) + \int_{H^2} f_r(x,\omega_i \to \omega_o) L_i(x,\omega_i) \cos\theta_i \, d\omega_i$$

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

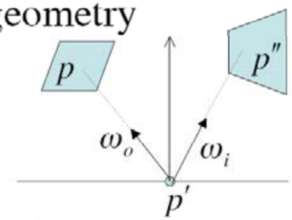
$$\begin{aligned}
L &= L_e + c\pi L \\
L &= L_e + \rho_{hh} L \\
&= L_e + \rho_{hh}(L_e + \rho_{hh} L) \\
&= L_e + \rho_{hh}(L_e + \rho_{hh}(L_e + \ldots \\
&= \sum_{i=0}^{\infty} L_e \rho_{hh}^i
\end{aligned}$$

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

Surface form of the LTE

- Expressing LTE in terms of geometry within the scene

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

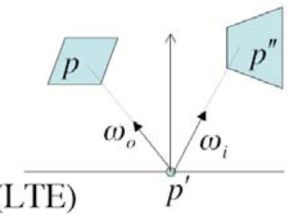


- Replacing the integrand ($d\omega_i$) with an area integrator over the whole scene geometry and remembering: $d\omega_i = \frac{|\cos\theta''|}{\|p'-p''\|^2} dA(p'')$
- $V(p \leftrightarrow p')$ - visibility term (either one or zero)

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)

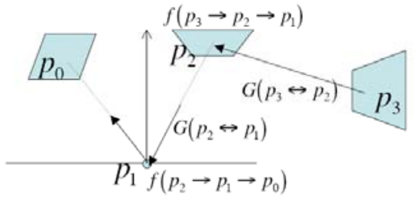
$$L(p' \to p) = L_e(p' \to p) + \int_A f_r(p'' \to p' \to p) L(p'' \to p') G(p'' \leftrightarrow p') dA(p'')$$

- 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

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

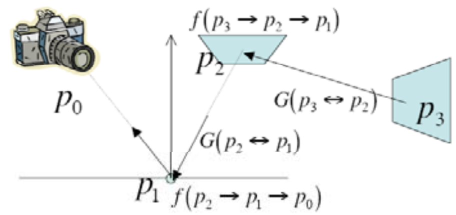


Surface form of the LTE

- compact formulation:

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

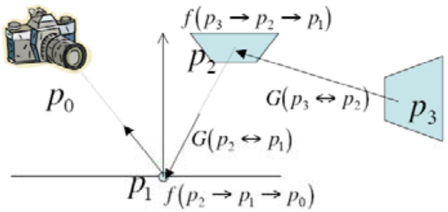
- For a path $\overline{p}_i = p_0 p_1 \ldots p_i$
- Where p_0 is the camera and p_i is a light source



Surface form of the LTE

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

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

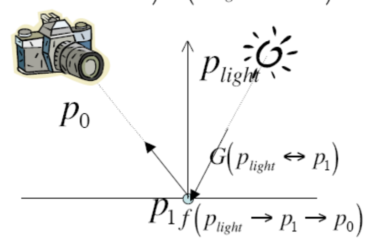


Delta distribution

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

$$P(\overline{p}_2) = \int_A L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) dA(p_2)$$
$$= \frac{\delta(p_{light} - p_2) L_e(p_2 \to p_1)}{p(p_{light})} f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1)$$
$$= L_e(p_{light} \to p_1) f(p_{light} \to p_1 \to p_0) G(p_{light} \leftrightarrow p_1)$$

- 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 operator

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

$$\equiv T \circ L_o$$

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

$$L^1 = L_e$$

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

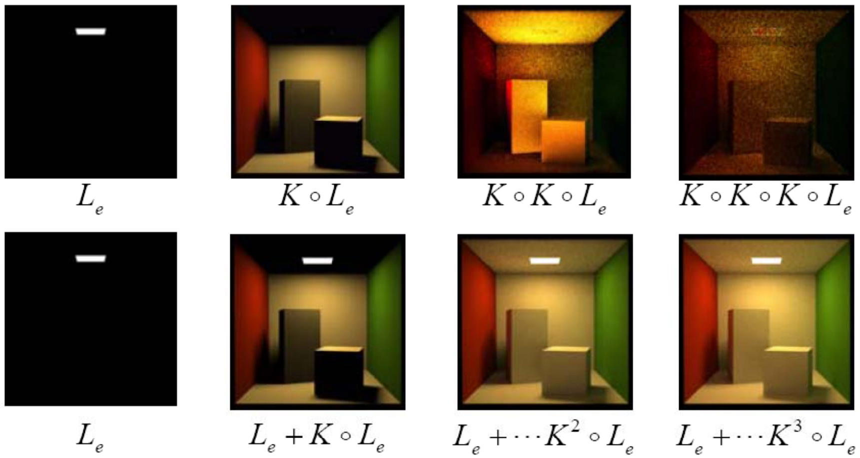
$$\ldots$$

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

Converged

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

Successive approximation

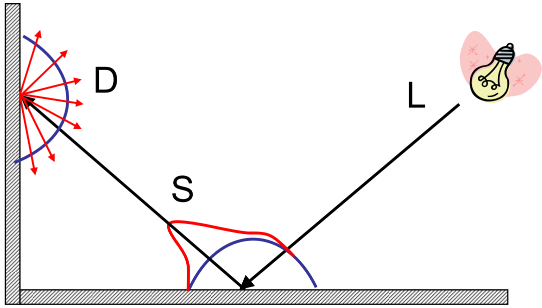


Light transport notation (Hekbert 1990)

- 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

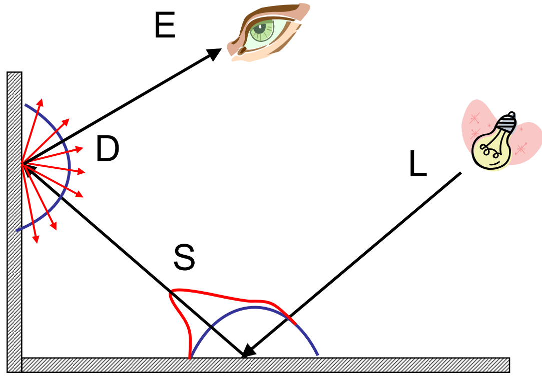
Light transport notation: examples

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



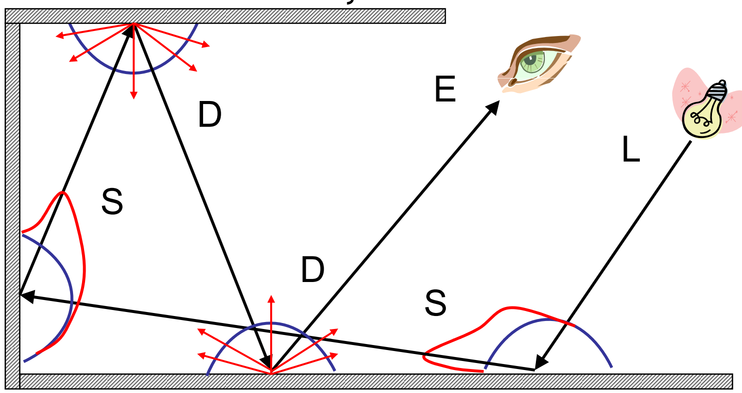
Light transport notation: examples

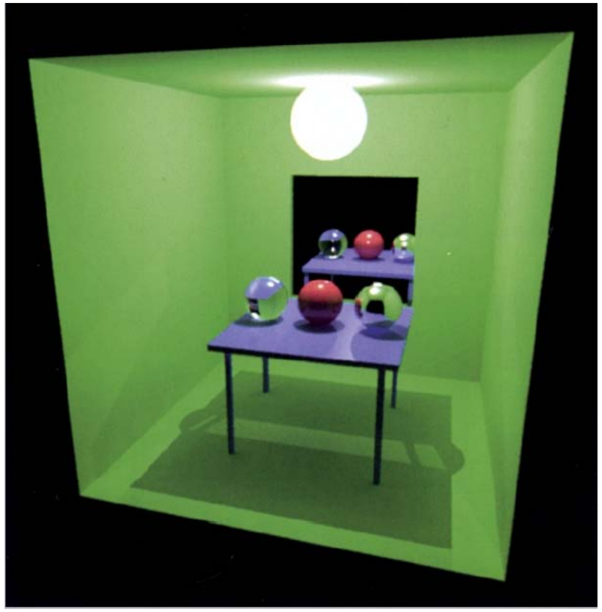
- L(S|D)$^+$DE
  - a path starting at a light, having one or more diffuse or specular reflections, then a final diffuse reflection toward the eye

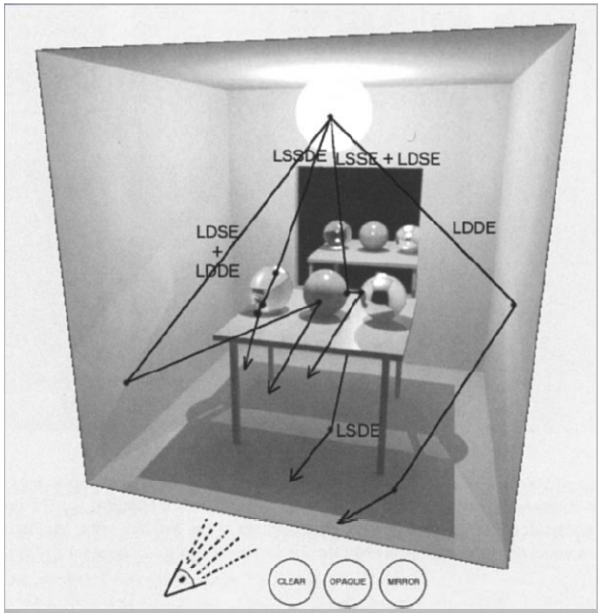


Light transport notation: examples

- 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

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

Integrate over all surfaces

Geometry term

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

Visibility term

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

The radiosity equation

Assume diffuse reflection

1. $f_r(x, \omega_i \to \omega_o) = f_r(x) \Rightarrow \rho(x) = \pi f_r(x)$
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:

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

$$S \circ B = E$$

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

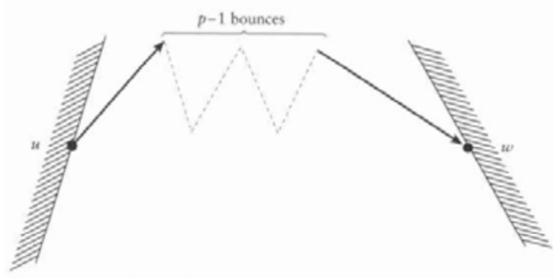
- In general, the longer the path, the less the impact.
- Use Russian Roulette after a finite number of bounces
  - Always compute the first few terms
  - Stop after that with probability q

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

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 $\bar{p}_i = p_0 p_1 \ldots p_j p_{j+1} \ldots p_i$
  - At each p_j, find p_{j+1} according to BSDF (in this way, they are guaranteed to be mutually visible)
  - 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

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

Incremental path generation

- Monte Carlo estimator

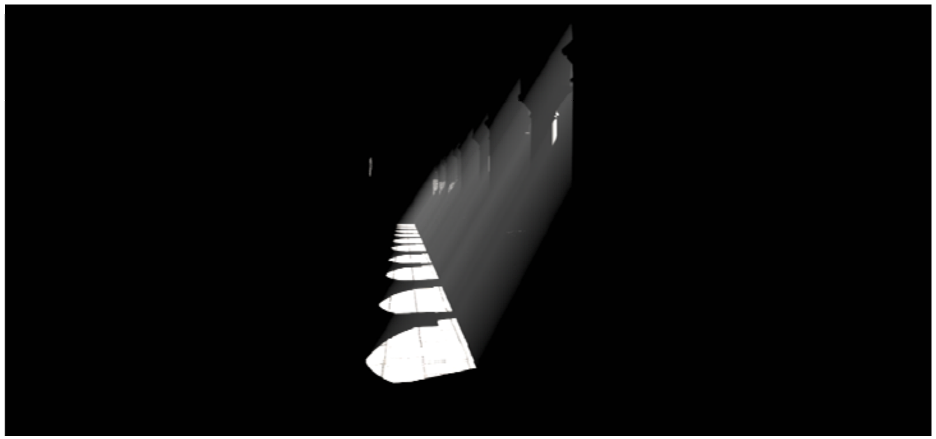
$$\underbrace{\frac{L_e(p_i \to p_{i-1}) f(p_i \to p_{i-1} \to p_{i-2})|\cos\theta_{i-1}|}{p_A(p_i)}}_{\text{MIS}} \left(\prod_{j=1}^{i-2} \underbrace{\frac{f(p_{j+1} \to p_j \to p_{j-1})|\cos\theta_j|}{p_\omega(p_{j+1} - p_j)}}_{\text{sampled by BSDF}}\right)$$

- 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 $\overline{p}$ 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

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

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

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

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

Bidirectional path tracing

Bidirectional path tracing

Path tracing

PathIntegrator

```
class PathIntegrator : public SurfaceIntegrator {
public:
  Spectrum Li(…) const;
  void RequestSamples(…);
  PathIntegrator(int md) { maxDepth = md; }
private:
  int maxDepth;
```

Use samples from Sampler for the first SAMPLE_DEPTH vertices of the path. After that, the advantage of well-distributed samples are greatly reduced, And it switches to using uniform random numbers.

```
#define SAMPLE_DEPTH 3
  LightSampleOffsets lightSampleOffsets[SAMPLE_DEPTH];
  int lightNumOffset[SAMPLE_DEPTH];
  BSDFSampleOffsets bsdfSampleOffsets[SAMPLE_DEPTH];
  BSDFSampleOffsets pathSampleOffsets[SAMPLE_DEPTH];
};
```

RequestSamples

```
class PathIntegrator::RequestSamples(…)
{
  for (int i = 0; i < SAMPLE_DEPTH; ++i) {
```

Path is reused. Thus, for each vertex, we need to perform MIS as it serves as the terminated point for some path. Therefore, we need both light and brdf samples

```
    lightSampleOffsets[i]=LightSampleOffsets(1,sample);
    lightNumOffset[i] = sample->Add1D(1);
    bsdfSampleOffsets[i] = BSDFSampleOffsets(1,sample);
```

```
    pathSampleOffsets[i] = BSDFSampleOffsets(1,sample);
  }
```

Another bsdf sample is used for extending the path

```
}
```

PathIntegrator::Li

```
class PathIntegrator::Li(…) const
{
  Spectrum pathThroughput = 1., L = 0.;
  RayDifferential ray(r);
  bool specularBounce = false;
  Intersection localIsect;
  const Intersection *isectp = &isect;
  for (int bounces = 0; ; ++bounces) {
    <possibly add emitted light at vertex>
    <sample from lights to find path contributions>
    <sample BSDF to get new path direction>
    <possibly terminate the path>
    <find next vertex of path>
  }
  return L;
}
```

PathIntegrator::Li

```
<possibly add emitted light at vertex>
if (bounces == 0 || specularBounce)
  L += pathThroughput * isectp->Le(-ray.d);
```

PathIntegrator::Li

```
<sample from lights to find path contributions>
BSDF *bsdf = isectp->GetBSDF(ray, arena);
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
if (bounces < SAMPLE_DEPTH)
  L += pathThroughput *
       UniformSampleOneLight(scene, renderer, arena,
         p, n, wo, isectp->rayEpsilon, ray.time,
         bsdf, sample, rng, lightNumOffset[bounces],
         &lightSampleOffsets[bounces],
         &bsdfSampleOffsets[bounces]);
else
  L += pathThroughput *
       UniformSampleOneLight(scene, renderer, arena,
         p, n, wo, isectp->rayEpsilon, ray.time,
         bsdf, sample, rng);
```

PathIntegrator::Li

```
<sample BSDF to get new path direction>
BSDFSample outgoingBSDFSample;
if (bounces < SAMPLE_DEPTH)
  outgoingBSDFSample = BSDFSample(sample,
                         pathSampleOffsets[bounces], 0);
else
  outgoingBSDFSample = BSDFSample(rng);
Vector wi;
float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(wo, &wi,
        outgoingBSDFSample, &pdf, BSDF_ALL, &flags);
if (f.IsBlack() || pdf == 0.)
  break;
specularBounce = (flags & BSDF_SPECULAR) != 0;
pathThroughput *= f * AbsDot(wi, n) / pdf;
ray = RayDifferential(p, wi, ray, isectp->rayEpsilon);
```

PathIntegrator::Li

```
<possibly terminate the path>
if (bounces > 3) {
  float continueProbability =
                    min(.5f, pathThroughput.y());
  if (rng.RandomFloat() > continueProbability)
    break;
  pathThroughput /= continueProbability;
}
if (bounces == maxDepth)
  break;
```

PathIntegrator::Li

```
<find next vertex of path>
if (!scene->Intersect(ray, &localIsect)) {
  if (specularBounce)
    for (int i = 0; i < scene->lights.size(); ++i)
      L += pathThroughput*scene->lights[i]->Le(ray);
  break;
}
if (bounces > 1)
  pathThroughput *= renderer->Transmittance(scene,
                             ray, NULL, rng, arena);
isectp = &localIsect;
```

Noise reduction/removal

- Path tracing is unbiased and often taken as a reference. The problem is that it has high variances.

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

Biased approaches

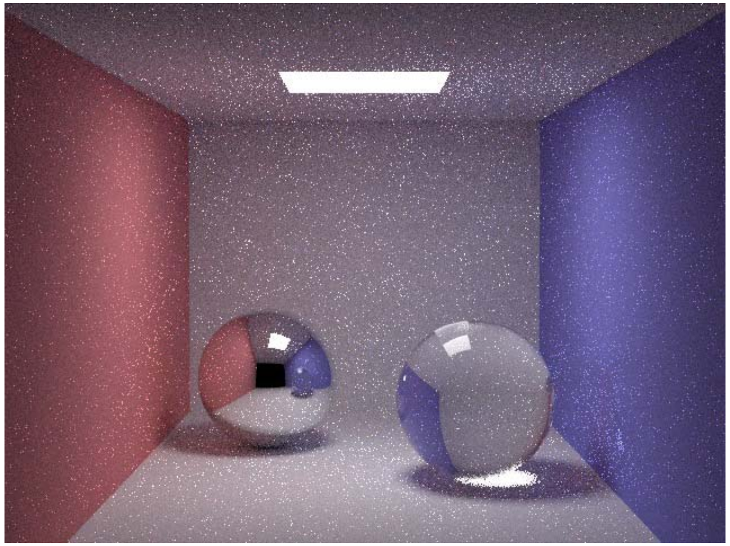
- By introducing bias (making smoothness assumptions), biased methods produce images without high-frequency noise
- Unlike unbiased methods, errors may not be reduced by adding samples in biased methods
- On contrast, when there is little error in the result of an unbiased method, we are confident that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

The world is more diffuse!

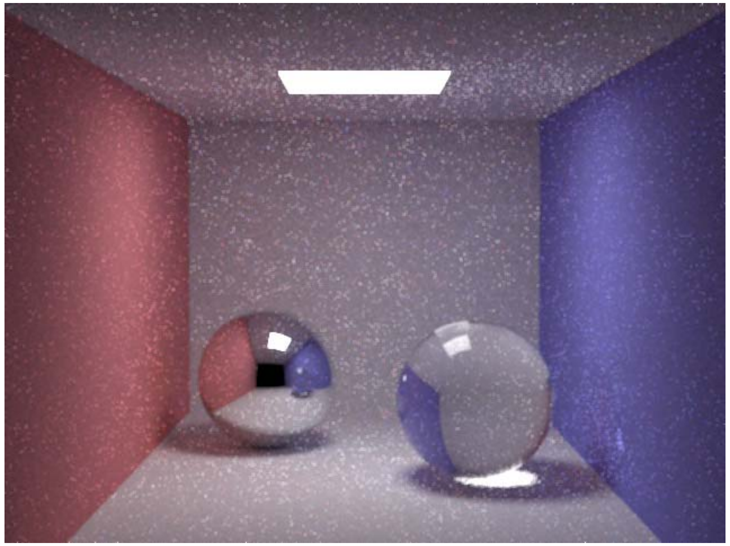
Filtering

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

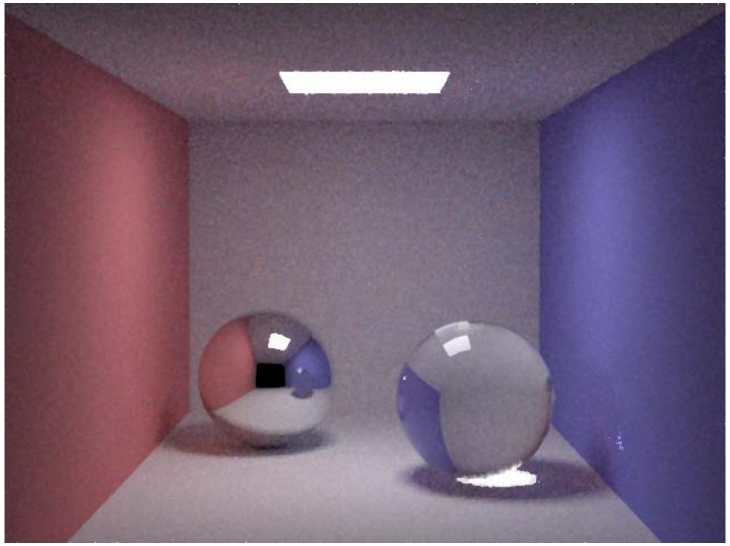
Path tracing (10 paths/pixel)



3x3 lowpass filter



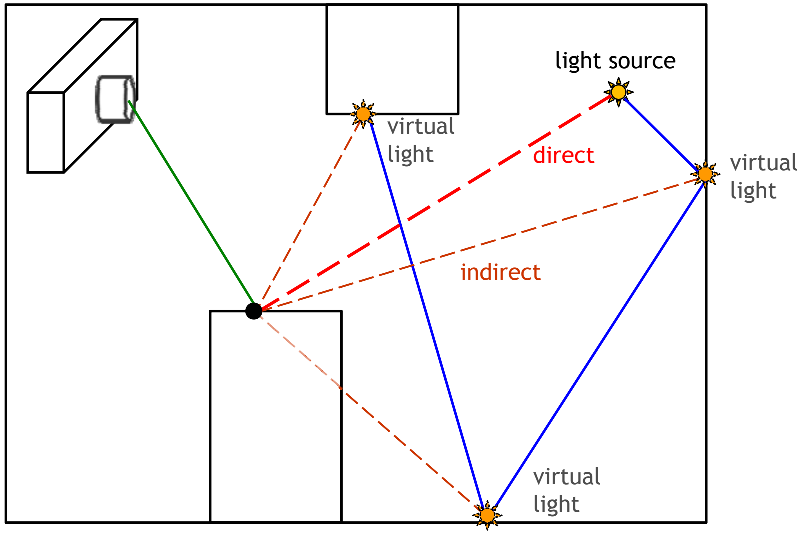
3x3 median filter



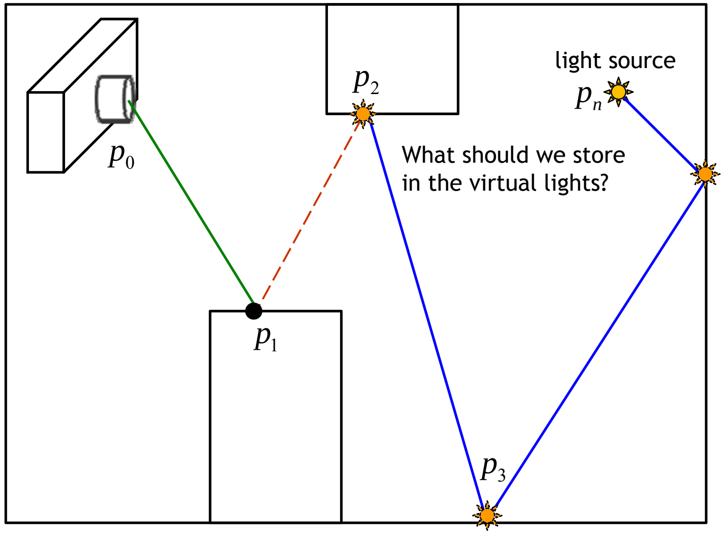
Instant global illumination

- Preprocess: follows some light-carrying paths from the light sources to create virtual light sources.
- Rendering: use only the virtual lights to compute the indirect contributions.
- Since only a set of virtual lights are used, there will be systemic error due to correlation rather than noise due to variance. Similar artifacts for your project #3.

Instant global illumination



Instant global illumination



Instant global illumination

$$P(\overline{p}_n) = \alpha f(p_3 \to p_2 \to p_1) G(p_2 \to p_1) f(p_2 \to p_1 \to p_0)$$

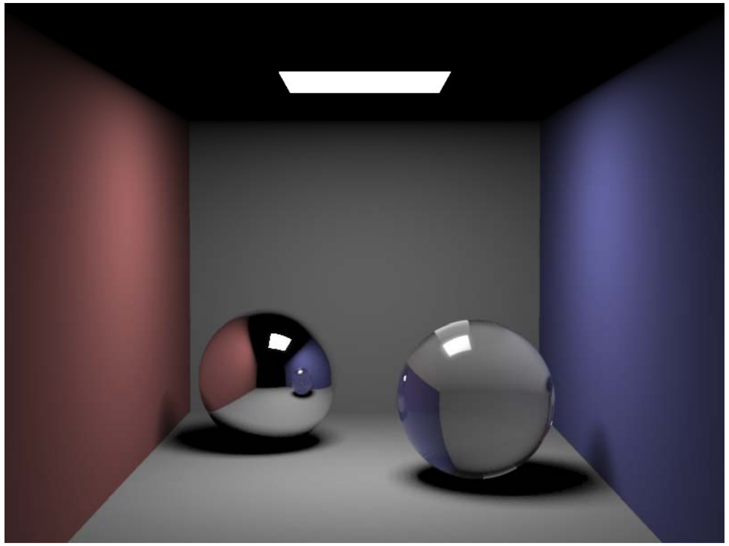
$$\alpha = \frac{L_e(p_n \to p_{n-1}) f(p_n \to p_{n-1} \to p_{n-2}) |\cos\theta_{n-1}|}{P_A(p_n)} \times \left(\prod_{i=3}^{n-2} \frac{f(p_{i+1} \to p_i \to p_{i-1}) |\cos\theta_i|}{P_\omega(p_{i+1} \to p_i)} \right)$$

It is independent to the camera and the first visible point p_1.
It is what we should pre-compute and store at the virtual lights.
During rendering, for each shading point, we need to evaluate the two remaining BRDFs and the geometric term.

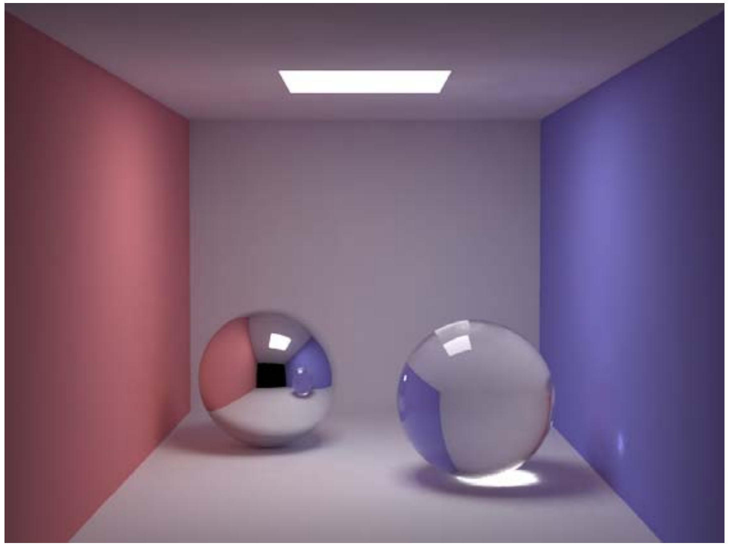
Caching techniques

- 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,

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

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

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

directional reflectance

Set of samples

$$E(p,n)=\int_{H^2} L_i(p,\omega_i)\delta(\omega_i-\omega_{avg})\,|\cos\theta_i|\,d\omega_i=L_{avg}\,|\cos\theta_{avg}|$$

$$L_{avg}=\frac{E}{|\cos\theta_{avg}|}$$

$$\begin{aligned}L_o(p,\omega_o)&=\int_{H^2} f(p,\omega_o,\omega_i)\delta(\omega_i-\omega_{avg})\frac{E}{|\cos\theta_{avg}|}\,|\cos\theta_i|\,d\omega_i\\&=f(p,\omega_o,\omega_{avg})E(p,n)\end{aligned}$$

makes it directional

Set of samples

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

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

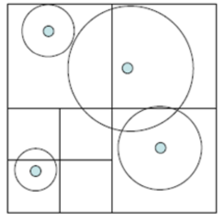
$$E(p)=\frac{1}{N}\sum_j \frac{L_i(p,\omega_j)\,|\cos\theta_j|}{p(\omega_j)} \qquad p(\omega)=\cos\theta/\pi$$

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

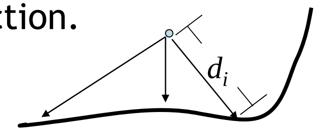
Storing samples

- Samples are stored in an octree.
- Each sample stores the following information

  $\{E,p,n,w_{avg},d_{max}\}$

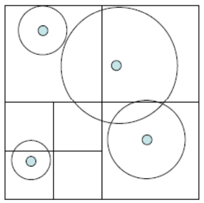


- Maximal distance is kept during path tracing for computing the sample. d_i is the distance that the ith ray hit an intersection.

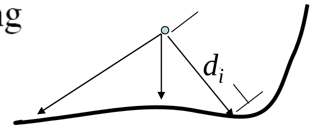


Storing samples

- 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 $\dfrac{N}{\sum_i^N 1/d_i}$
  - d_i is the distance that the ith ray (used for estimating the irradiance) traveled before intersecting an object
  - Computed during path tracing

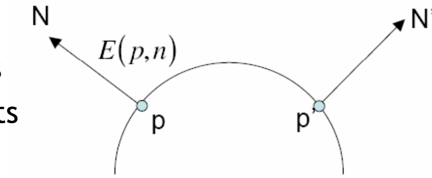


$\{E,p,n,d\}$



Interpolating from neighbors

- Weights depend on
  - Angle between normals
  - Distance between points



- Weight (ad hoc)

$$w_i = 1-\max\left(\frac{d}{d_{\max}},\sqrt{\frac{1-N\cdot N'}{1-\cos\theta_{\max}}}\right)$$

- Final irradiance estimate is simply the weighted sum

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

IrradianceCacheIntegrator

```
class IrradianceCacheIntegrator : public
  SurfaceIntegrator {
  ...
  float minSamplePixelSpacing, maxSamplePixelSpacing;
  float minWeight, cosMaxSampleAngleDifference;
  int nSamples; how many rays for computing irradiance samples
  int maxSpecularDepth, maxIndirectDepth;
}
```

Preprocess() allocates the octree for storing irradiance samples

Li

```
L += isect.Le(wo);
L += UniformSampleAllLights(...);
if (ray.depth+1 < maxSpecularDepth) {
  <Trace rays for specular reflection and refraction>
} Current implemetation uses Whitted style for specular; irradiance cache for
  Both diffuse and glossy. It could lead to errors for glossy.
// Estimate indirect lighting with irradiance cache
...                                 the project area of a pixel
float pixelSpacing =                in the world space
  sqrtf(Cross(isect.dg.dpdx, isect.dg.dpdy).Length());
BxDFType flags =
  BxDFType(BSDF_REFLECTION|BSDF_DIFFUSE|BSDF_GLOSSY);
L += indirectLo(...);
Flags =
  BxDFType(BSDF_TRANSMISSION|BSDF_DIFFUSE|BSDF_GLOSSY);
L += indirectLo(...);
```

IndirectLo

```
if (!InterpolateE(scene, p, n, &E, &wi)) {
  ... // Compute irradiance at current point
  for (int i = 0; i < nSamples; ++i) {
    <Path tracing to compute radiances along ray
     for irradiance sample>
    LiSum += L;
    wAvg += r.d * L.y();
    minHitDistance = min(minHitDistance, r.maxt);
  }
  E = (M_PI / float(nSamples)) * LiSum;
  ... // Add computed irradiance value to cache
 IrradianceSample *sample =
   new IrradianceSample(E, p, ng, wAvg, contribExtent);
  octree->Add(sample, sampleExtent);
}
return bsdf->f(wo, Normalize(wi), flags) * E;
```

Octree

```
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 *>(wb);
  <prefill the irradiacne cache>
}
struct IrradianceSample {
  Spectrum E;
  Normal n;
  Point p;
  Vector wAvg;
  float maxDist;
};
```

InterpolateIrradiance

```
Bool InterpolateE(Scene *scene, Point &p, Normal &n,
  Spectrum *E, Vector *wi)
{
  if (!octree) return false;
  IrradProcess proc(p, n, minWeight,
                    cosMaxSampleAngleDifference);
  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();
  *wi = proc.GetAverageDirection();
  return true;
}
```

IrradProcess

```
void IrradProcess::operator()(IrradianceSample &sample)
{
  float perr = Distance(p, sample->p)
                            / sample->maxDist;
  float nerr = sqrtf((1.f - Dot(n, sample->n))
                     / (1.f - cosMaxSampleAngleDifference));
  float err = max(perr, nerr);

  if (err < 1.) {
    ++nFound;
    float wt = 1.f - err;
    E += wt * sample->E;
    wAvg += wt * sample->wAvg;
    sumWt += wt;
  }
  return true;
}
```

$$\frac{d}{d_{\max}}$$

$$\sqrt{\frac{1-N\cdot N'}{1-\cos\theta_{\max}}}$$

$$w_i = 1-\max\left(\frac{d}{d_{\max}}, \sqrt{\frac{1-N\cdot N'}{1-\cos\theta_{\max}}}\right)$$

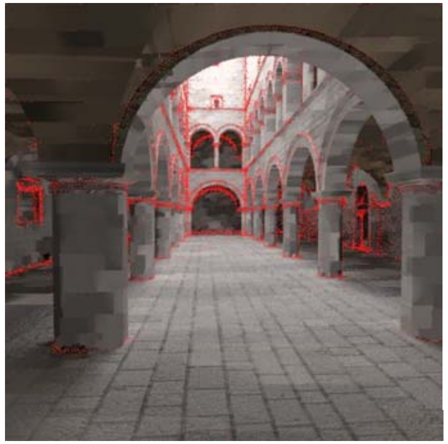
Comparison with same limited time

Irradiance caching
Blotch artifacts

Path tracing
High-frequency noises

Irradiance caching

Irradiance caching



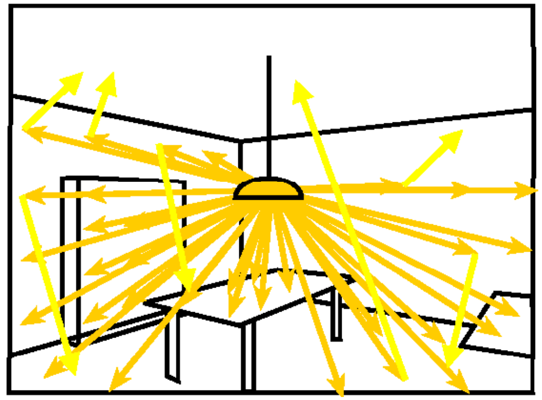
Irradiance sample positions

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

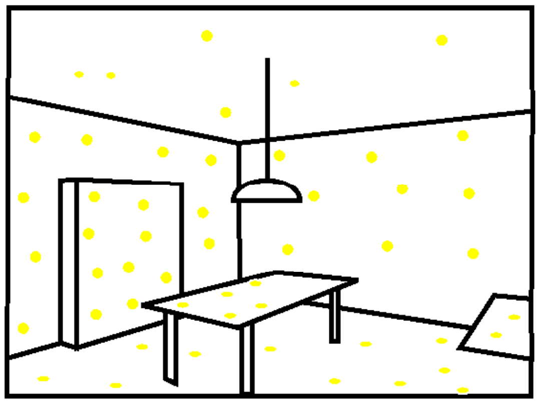
Photon tracing

- Preprocess: cast rays from light sources



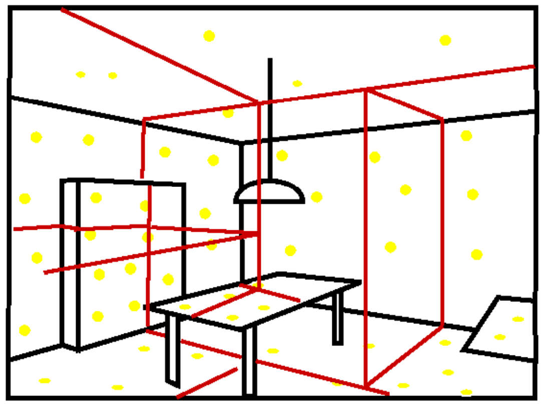
Photon tracing

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



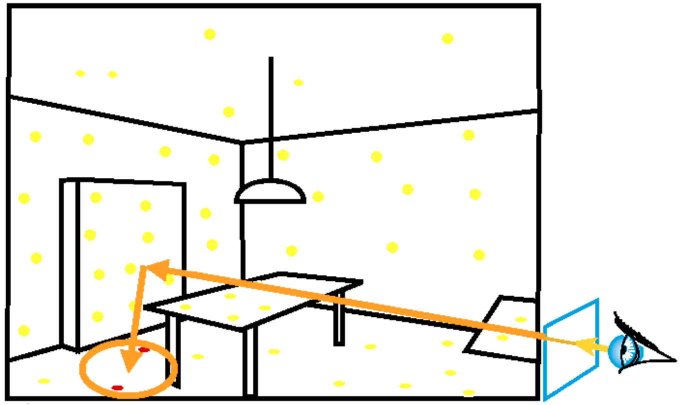
Photon map

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



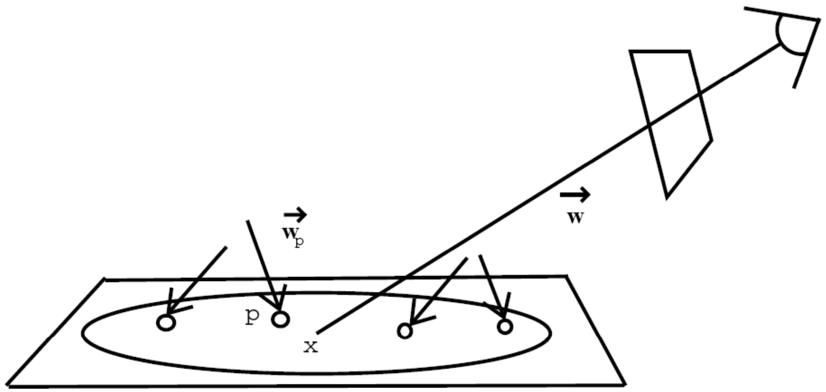
Rendering (final gathering)

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

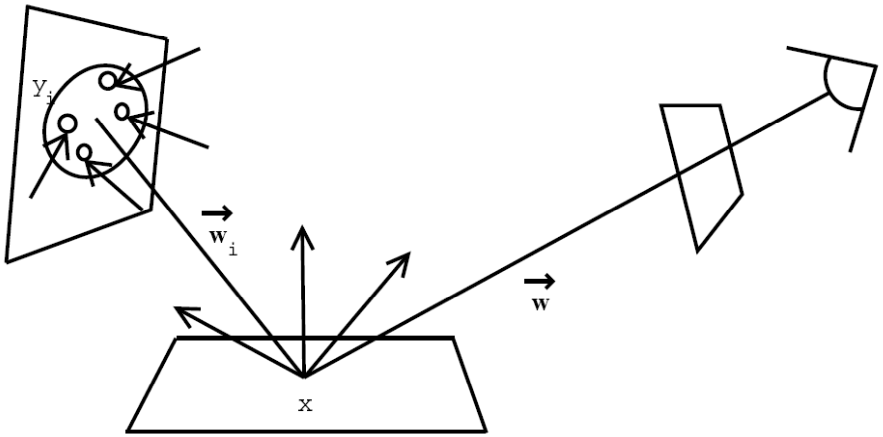


Rendering (without final gather)

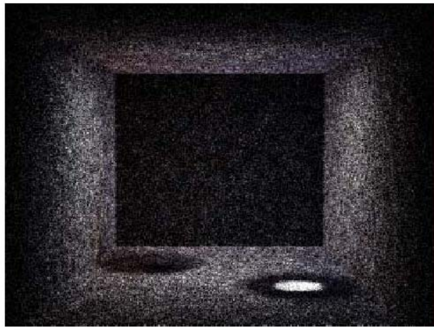
$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| \, d\omega_i$$



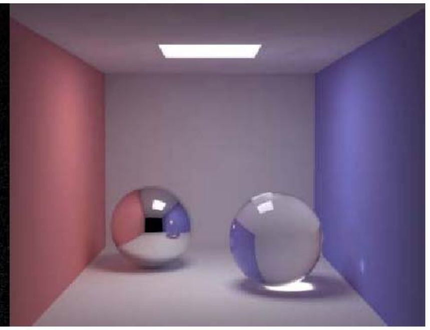
Rendering (with final gather)



Photon mapping results



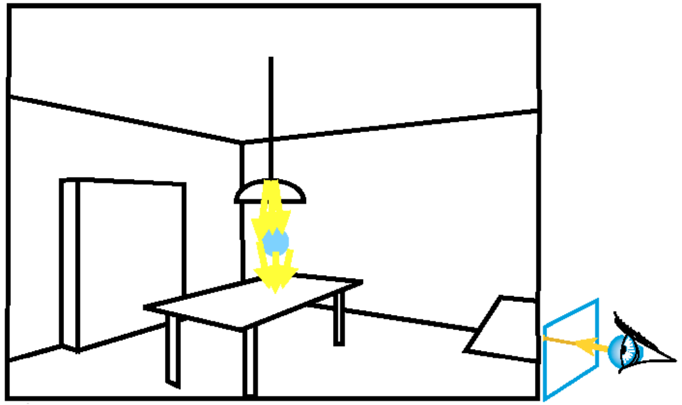
photon map



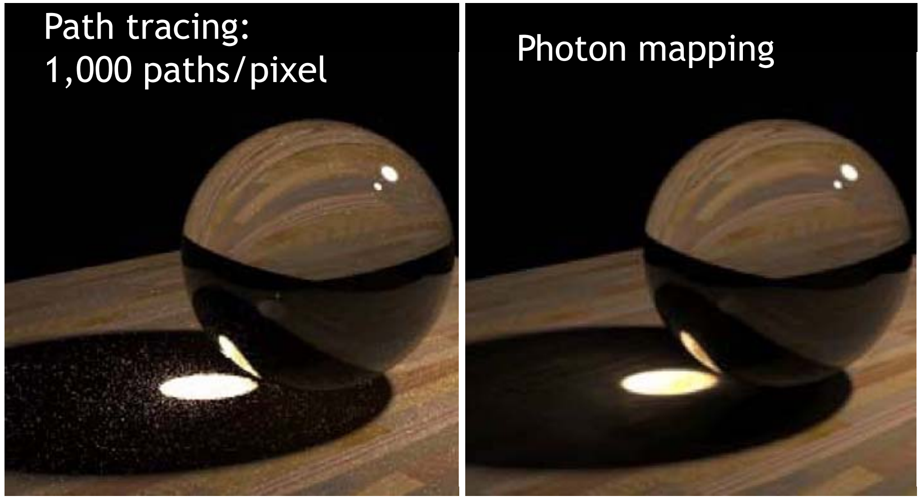
rendering

Photon mapping - caustics

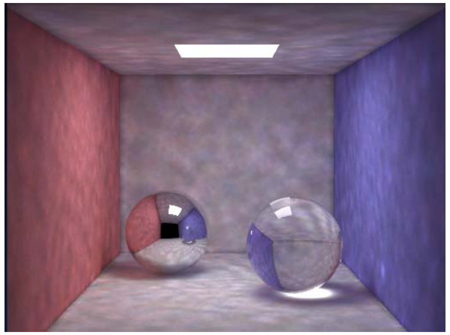
- Special photon map for specular reflection and refraction



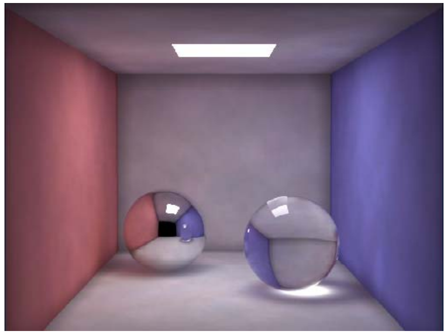
Caustics



Photon mapping



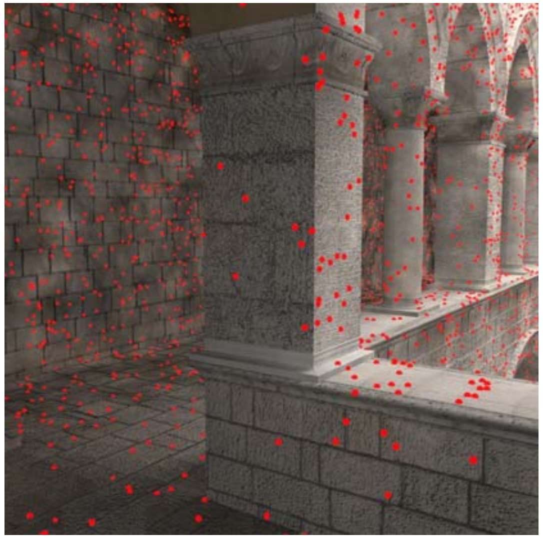
100K photons



500K photons

Photon map

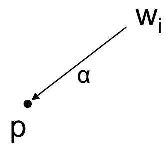
Kd-tree is used to store photons, decoupled from the scene geometry

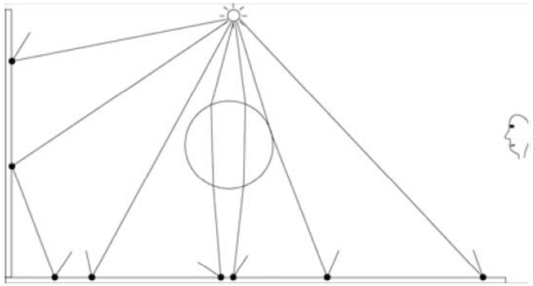


Photon shooting

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

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

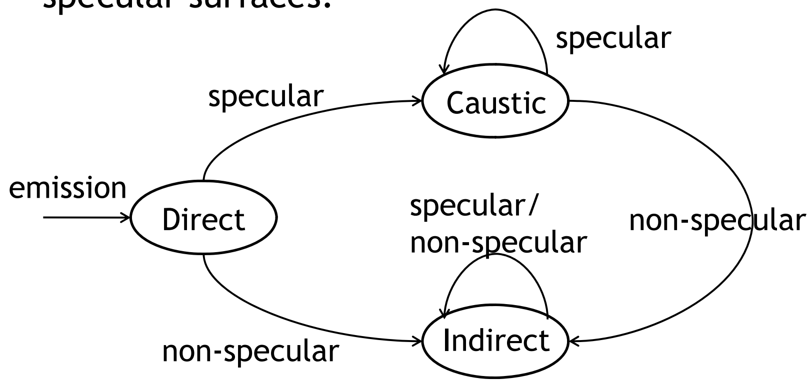




For 100 photons emitted from 100W source, each photon initially carries 1W.

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.



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

Results