Mestrado em
Engenharia Informática

VI-RT
Monte Carlo and Distributed Rendering

Visualização e Iluminação

Luís Paulo Peixoto dos Santos

Quadratura numérica determinística –
Exemplo: método dos rectângulos

- O domínio é **uniformemente** subdividido em N subdomínios de extensão h, [a_i, a_i+h[
- Para cada subdomínio é calculado um **estimador primário** dado pelo produto da extensão do domínio e o valor da função no centro do subdomínio
- O **estimador final** é a **média aritmética** dos estimadores primários

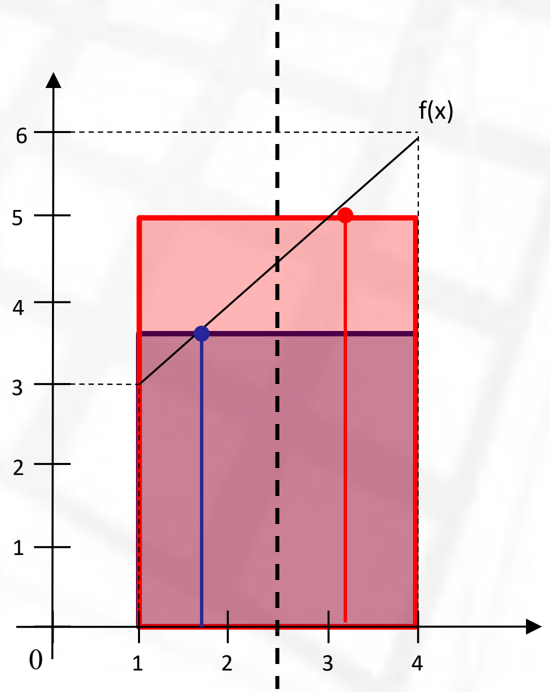
$$I = \int_1^4 (x+2)dx = \left.\frac{x^2}{2}\right|_1^4 + 2x\Big|_1^4 = 13,5$$

$$N = 2; h = 1.5; a_o = 1; a_1 = 2.5$$

$$f(x_0) = f(1.75) = 3.75 \quad \langle I_{x0} \rangle = 3 * f(1.75) = 11.25$$

$$f(x_1) = f(3.25) = 5.25 \quad \langle I_{x1} \rangle = 3 * f(3.25) = 15.75$$

$$< I >= \frac{1}{2} * (11.25 + 15.75) = 13.5$$



Integração de Monte Carlo: quadratura

$$\int_D f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} = \frac{\|D\|}{N}\sum_{i=1}^{N} f(x_i) \qquad p(x) = \frac{1}{\|D\|}$$

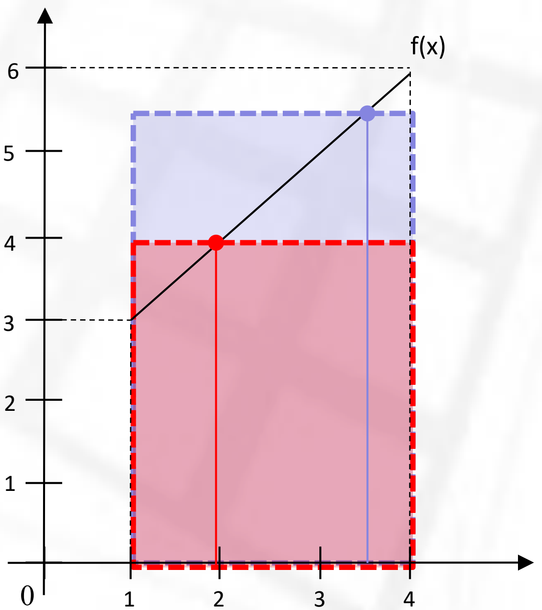
O valor do integral é aproximado, para cada x_i, por um volume cuja altura é o valor de $f(x_i)$ e cuja largura é dada pelo tamanho do domínio de integração *(b-a)*

$$I = \int_1^4 (x+2)dx = \left.\frac{x^2}{2}\right|_1^4 + 2x\Big|_1^4 = 13{,}5$$

$x_0 = 2$ $\quad f(x_0) = f(2) = 4$ $\quad \langle I_0 \rangle = (4-1)\, f(x_0) = 12$

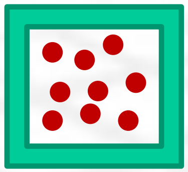
$x_1 = 3{,}5$ $\quad f(x_1) = f(3{,}5) = 5{,}5$ $\quad \langle I_1 \rangle = (4-1)\, f(x_1) = 16{,}5$

$\langle I \rangle = ½ * (\langle I_0 \rangle + \langle I_1 \rangle) = (12 + 16{,}5) / 2 = 14.25$



Pixel Color and Monte Carlo

Pixel



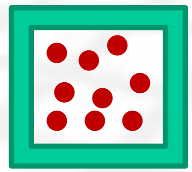
$$I_p = \int_{A_p} f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}$$

If the positions x within the area of the pixel are selected with constant probability (uniform sampling) then the probability of selecting any x is $p(x) = \frac{1}{A_p}$.

The uniform probability of uniformly sampling a point x on a set S is always the reciprocal of S's extent ('size') : $p(x) = \frac{1}{\|S\|}$.

Pixel Color and Monte Carlo

Pixel



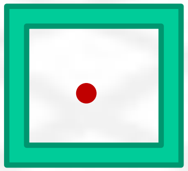
$$I_p = \int_{A_p} f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{1/A_p} = \frac{A_p}{N}\sum_{i=1}^{N} f(x_i)$$

A pixel is a virtual entity so we assume $A_p = 1$

$$I_p = \int_{A_p} f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

Jittering the primary ray

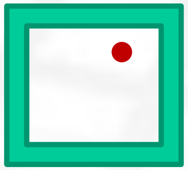
Pixel



$$x_s = x + 0.5$$

$$y_s = y + 0.5$$

Pixel



$$x_s = x + jitter()$$

$$y_s = y + jitter()$$

$$jitter(\quad) = \mathrm{rand}\ [0,1[$$

Jittering the primary ray

- If the screen space is regularly sampled then aliasing artifacts occur

- If the screen space is stochastically sampled then noise appears

StandardRenderer.cpp

```
const bool jitter = true;

for (y=0 ; y< H ; y++) {  // loop over rows
  for (x=0 ; x< W ; x++) { // loop over columns
      RGB color(0.,0.,0.);
      for (ss=0 ; ss < spp ; ss++) { // multiple samples per pixel

         // each pixel sample block code
         ...

      }
      color = color / spp;
      img->set(x,y,color);
  }
}
```

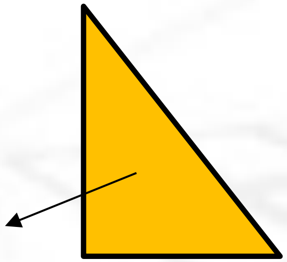
StandardRenderer.cpp

```
// each pixel sample block code

if (jitter) {
  float jitterV[2];
  jitterV[0] = ((float)rand()) / ((float)RAND_MAX);
  jitterV[1] = ((float)rand()) / ((float)RAND_MAX);
  cam->GenerateRay(x, y, &primary, jitterV);
} else {
  cam->GenerateRay(x, y, &primary);
}

intersected = scene->trace(primary, &isect);
this_color = shd->shade (intersected, isect);
color += this_color;
```

perspective.cpp

```
bool Perspective::GenerateRay(const int x, const int y,
             Ray *r, const float *cam_jitter) {
    float xc, yc;

    if (cam_jitter==NULL) {
        xc = 2.f * ((float)x + .5f)/W - 1.f;
        yc = 2.f * ((float)(H-y-1) + .5f)/H - 1.f;
    } else {
        xc = 2.f * ((float)x + cam_jitter[0])/W - 1.f;
        yc = 2.f * ((float)(H-y-1) + cam_jitter[1])/H - 1.f;
    }
    ... Generate primary ray
}
```

AreaLight.hpp



`normal` – identifies the light emissive side of the triangle

`power` – the total power of the light source, independently of its sizes

`intensity` – power per unit area

```
class AreaLight: public Light {
public:
    RGB intensity, power;
    Triangle *gem;
    float pdf;
    AreaLight (RGB _power, Point _v1, Point
_v2, Point _v3, Vector _n): power(_power) {
        type = AREA_LIGHT;
        gem = new Triangle (_v1, _v2, _v3,
_n);
        pdf = 1.f/gem->area();  // for
uniform sampling the area
        intensity = _power * pdf;
    }
    ~AreaLight () {delete gem;}
    ...
```

Triangle: new Geometry class

```
class Triangle: public Geometry {
public:
    Point v1, v2, v3;
    Vector normal,edge1, edge2;
    BB bb;
    bool intersect (Ray r, Intersection *isect);

    Triangle(Point _v1, Point _v2, Point _v3, Vector _normal):
v1(_v1), v2(_v2), v3(_v3), normal(_normal) {
        edge1 = v1.vec2point(v2); edge2 = v1.vec2point(v3);
        bb.min.set(v1.X, v1.Y, v1.Z); bb.max.set(v1.X, v1.Y, v1.Z);
        bb.update(v2);  bb.update(v3);
    }
    // https://www.mathopenref.com/heronsformula.html
    float area () {...}
}
```

AreaLight.hpp

```
class AreaLight: public Light {
...
// return a point p, RGB radiance and pdf given a rand pair (0, 1(
// sample point: "Globbbl Illumination Compendium", pp. 12, item 18
RGB Sample_L (float *r, Point *p, float &_pdf) {
        const float sqrt_r0 = sqrtf(r[0]);
        const float alpha = 1.f - sqrt_r0;
        const float beta = (1.f-r[1]) * sqrt_r0;
        const float gamma = r[1] * sqrt_r0;
        p->X = alpha*gem->v1.X + beta*gem->v2.X + gamma*gem->v3.X;
        p->Y = alpha*gem->v1.Y + beta*gem->v2.Y + gamma*gem->v3.Y;
        p->Z = alpha*gem->v1.Z + beta*gem->v2.Z + gamma*gem->v3.Z;
        _pdf = pdf;
        return intensity;
    }
}
```

DistributedShader.hpp

```
class DistributedShader: public Shader {
    RGB background;
    RGB directLighting (Intersection isect, Phong *f);
    RGB specularReflection (Intersection isect, Phong *f,
                            int depth);

public:
    DistributedShader (Scene *scene, RGB bg):
              background(bg), Shader(scene) {}
    RGB shade (bool intersected, Intersection isect, int depth);
};
```

DistributedShader.cpp

```
RGB DistributedShader::shade(bool intersected, Intersection isect, int depth) {
    RGB color(0.,0.,0.);
    if (!intersected)        return (background);

    // intersection with a light source
    if (isect.isLight)      return isect.Le;

    Phong *f = (Phong *)isect.f;

    // if there is a specular component sample it
    if (!f->Ks.isZero() && depth <4)
       color += specularReflection (isect, f, depth+1);

    // if there is a diffuse component do direct light
    if (!f->Kd.isZero()) color += directLighting(isect, f);

    return color;
};
```

DistributedShader.cpp

```
RGB DistributedShader::directLighting (Intersection isect, Phong *f) {
  RGB color(0.,0.,0.);

  for (auto l = scene->lights.begin() ; l != scene->lights.end() ; l++) {

    if (l->type == AMBIENT_LIGHT) {  // is it an ambient light ?
        ...
    }
    if (l->type == POINT_LIGHT) {  // is it a point light ?
        ...
    }
    if (l->type == AREA_LIGHT) {  // is it an area light ?
        ...
    }  // end area light
  }
  return color;
}
```

DistributedShader.cpp

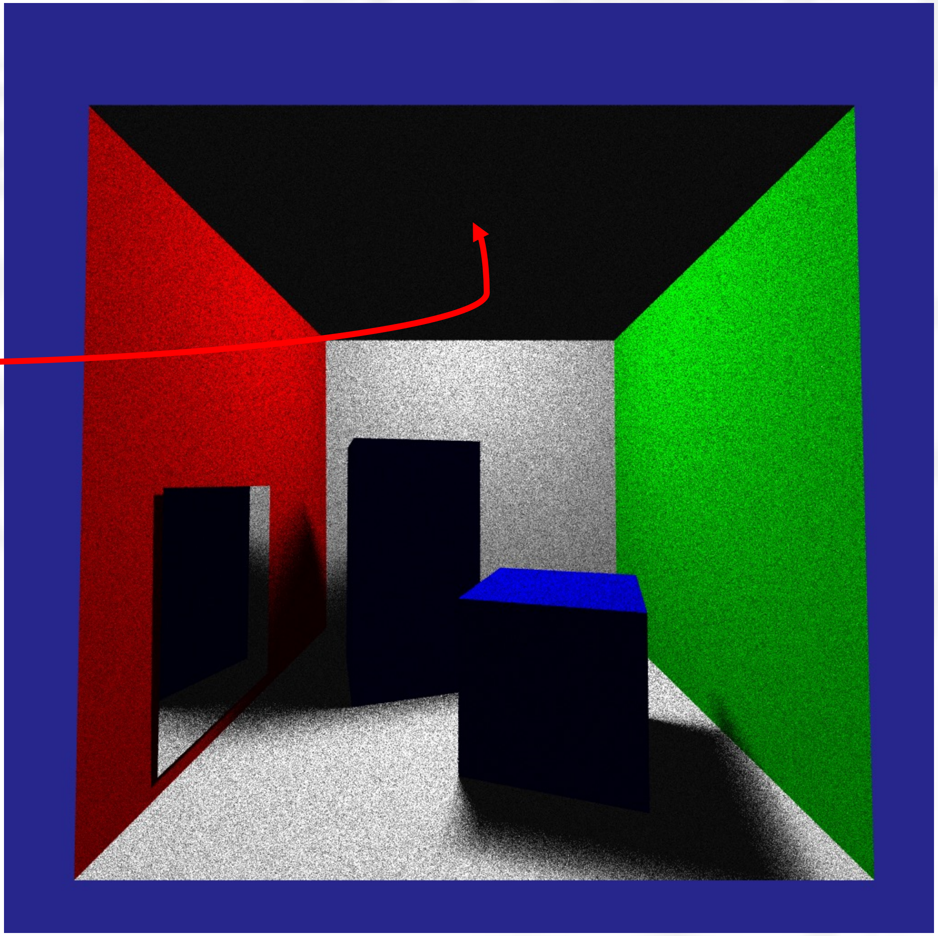
```
    if (l->type == AREA_LIGHT) {  // is it an area light ?
        if (!f->Kd.isZero()) {
            RGB L, Kd = f->Kd;
            Point lpoint;
            float l_pdf;
            AreaLight *al = (AreaLight *)l;

            float rnd[2];
            rnd[0] = ((float)rand()) / ((float)RAND_MAX);
            rnd[1] = ((float)rand()) / ((float)RAND_MAX);
            L = al->Sample_L(rnd, &lpoint, l_pdf);

            // compute the direction from the intersection point to the light source
            Vector Ldir = isect.p.vec2point(lpoint);
            const float Ldistance = Ldir.norm();

            // now normalize Ldir
            Ldir.normalize();

            ...
```

DistributedShader.cpp

```
if (l->type == AREA_LIGHT) {  // is it an area light ?
   ...
   // cosine between Ldir  and the shading normal at the intersection point
   float cosL = Ldir.dot(isect.sn);
   // cosine between Ldir  and the area light source normal
   float cosL_LA = Ldir.dot(al->gem->normal);
   // shade
   if (cosL>0. and cosL_LA<=0.)  { // light NOT behind primitive AND light normal points to
the ray o
       // generate the shadow ray
       ...
       shadow.adjustOrigin(isect.gn);
       if (scene->visibility(shadow, Ldistance-EPSILON)) {  // light source not occluded
          color += (Kd * L * cosL) / l_pdf;
       }
    } // end cosL > 0.
  }
}  // end area light
return color;
}
```

Stochastically sampled area lights

The area light source is not visible!



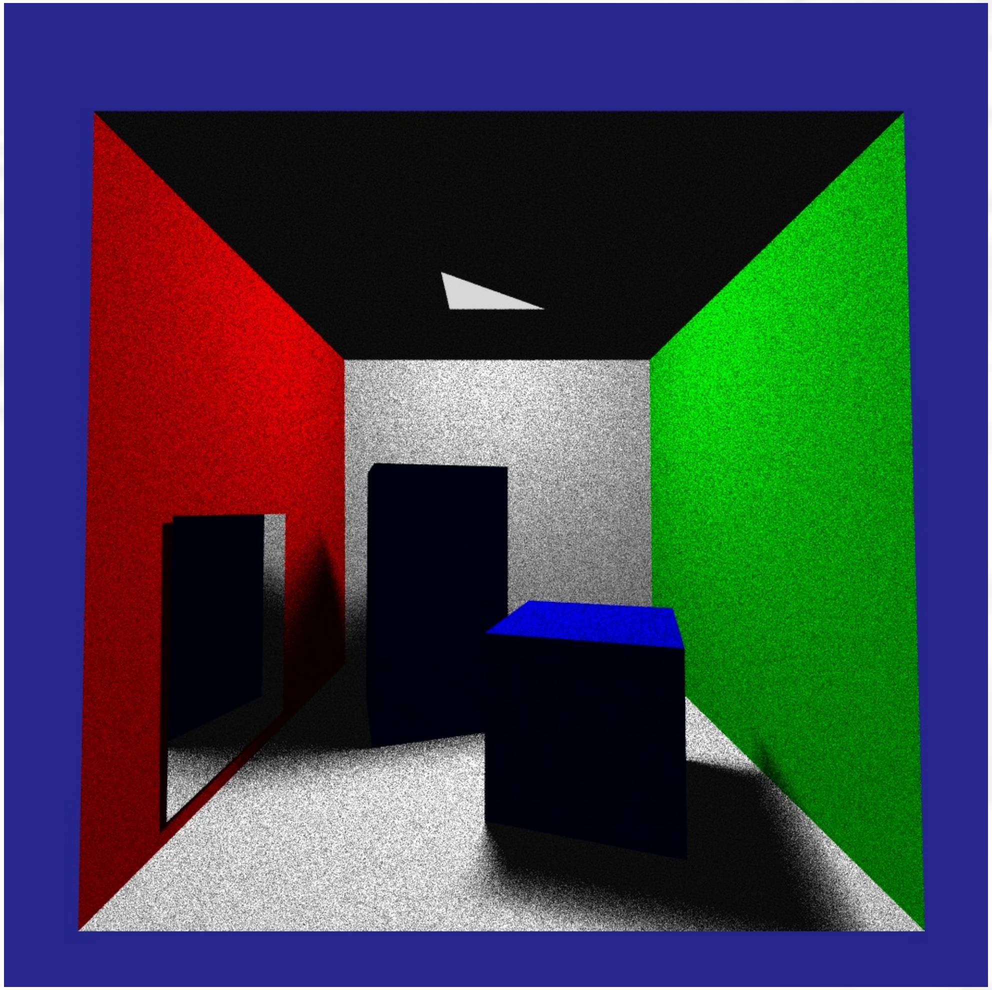
Scene.cpp

```
bool Scene::trace (Ray r, Intersection *isect) {
    Intersection curr_isect;
    bool intersection = false;

    // iterate over all primitives
    for (auto p = prims.begin() ; p != prims.end() ; p++) {
      ...
    }

    isect->isLight = false;
    // now iterate over light sources and intersect with those that have
geometry
    for (auto l = lights.begin() ; l != lights.end() ; l++) {
        ... see next slide
    }
    return intersection;
}
```

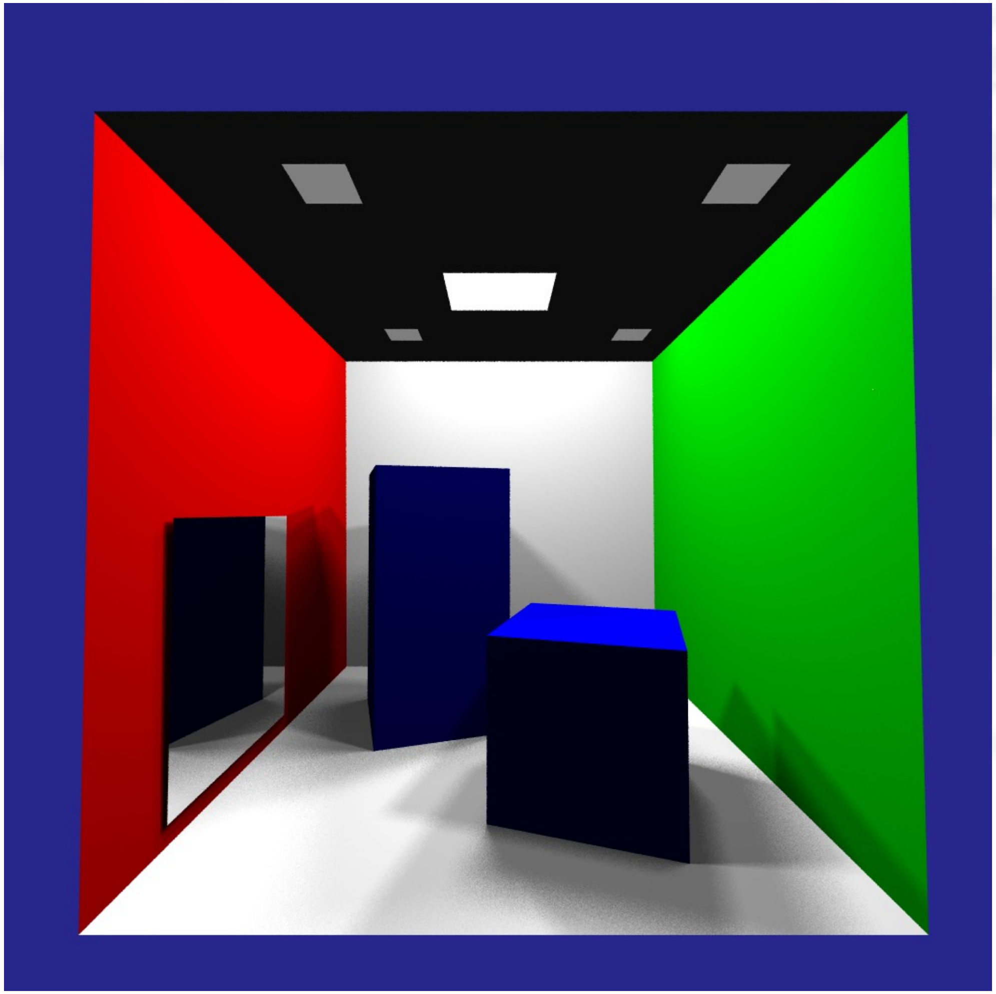
Scene.cpp

```
    isect->isLight = false;
    for (auto l = lights.begin() ; l != lights.end() ; l++) {
        if ((*l)->type == AREA_LIGHT) {
            AreaLight *al = (AreaLight *)*l;
            if (al->gem->intersect(r, &curr_isect)) {
                if (!intersection) { // first intersection
                    intersection = true;
                    *isect = curr_isect;
                    isect->isLight = true;
                    isect->Le = al->L();
                }
                else if (curr_isect.depth < isect->depth) {
                    *isect = curr_isect;
                    isect->isLight = true;
                    isect->Le = al->L();
                }
            }
        }
```

Stochastically sampled area lights



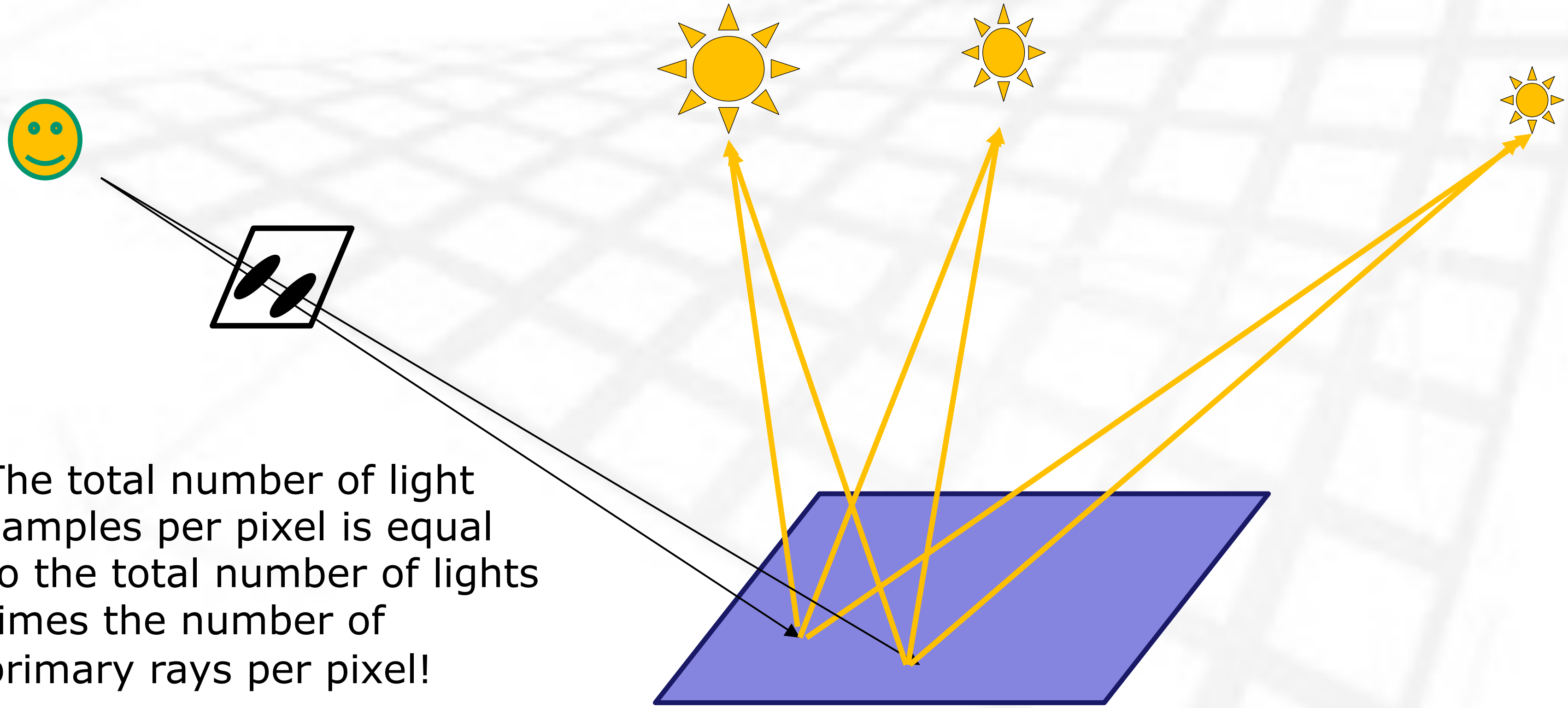
Multiple light sources



10 area light sources
spp = 16
T = 34.8 secs

At each intersection one shadow ray is shot towards each light source.
Is this the best choice?
Can we do any better?

Multiple light sources

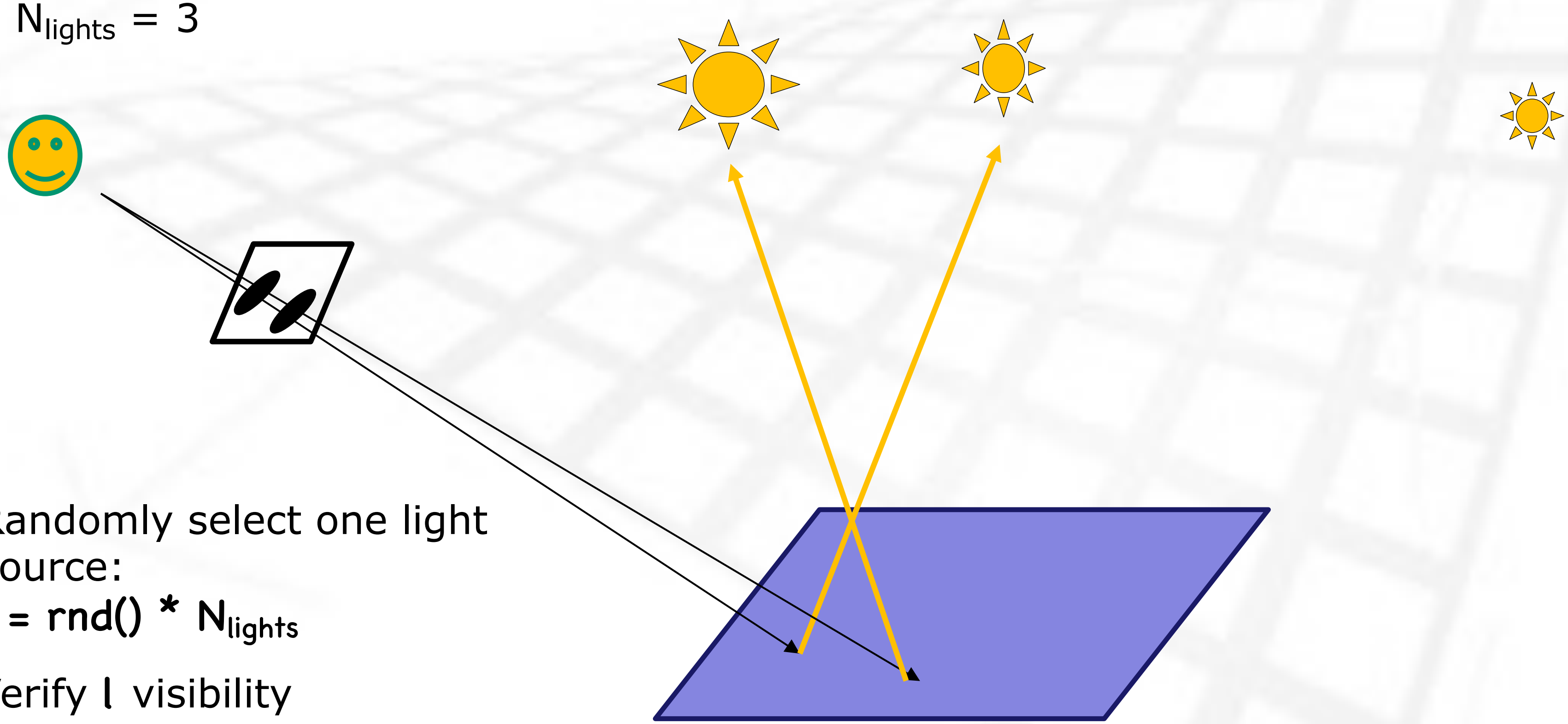


The total number of light samples per pixel is equal to the total number of lights times the number of primary rays per pixel!

Do we need all this? Can we do better?
Can we exploit the fact that multiple primary rays are shot per pixel?

Multiple light sources

$N_{lights} = 3$



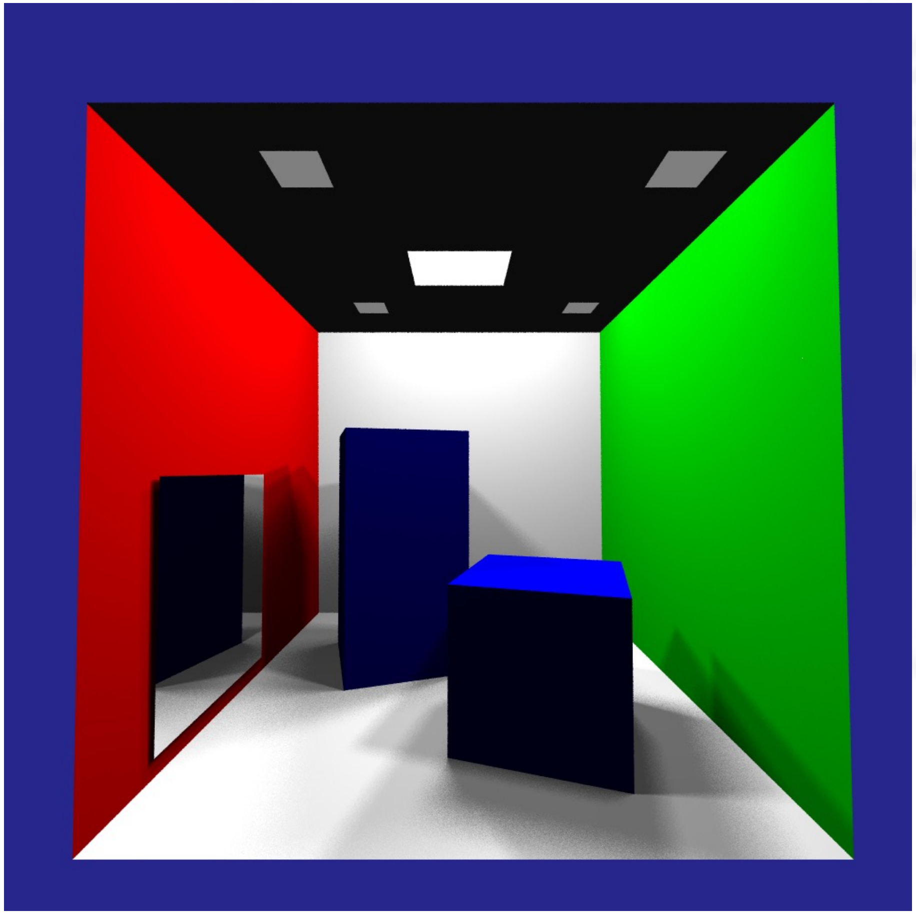
Randomly select one light source:
$l = rnd() * N_{lights}$

Verify l visibility

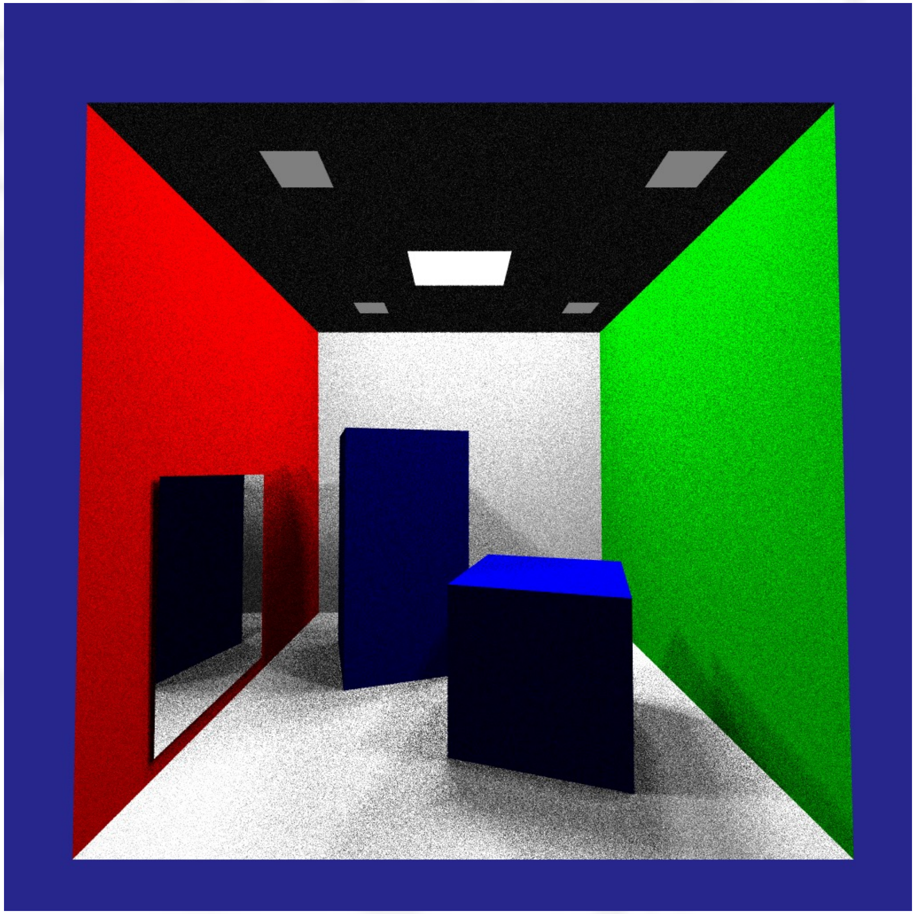
Weigh l contribution by the probability with which l was selected: $p = 1/ N_{lights} = 1/3$

$color = color\_l / p = color\_l / (1/ N_{lights}) = color\_l * N_{lights}$

Monte Carlo sampling the light sources



spp=16; T = 34.8 secs



spp=16; T = 6.5 secs
stochast. select 1 light source