Mestrado em
Engenharia Informática

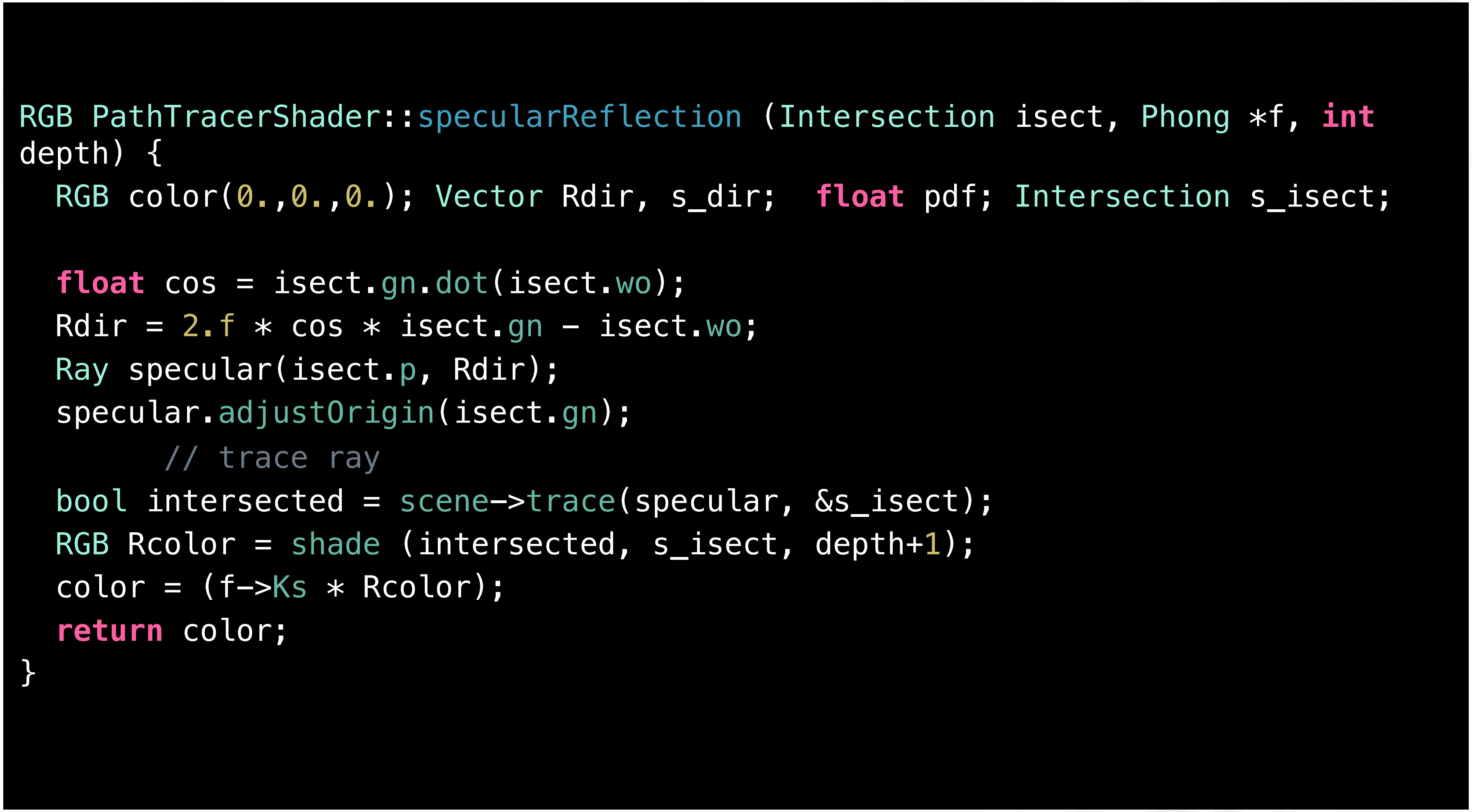
# VI-RT

# Monte Carlo and Distributed Rendering II: Path Tracing

Luís Paulo Peixoto dos Santos

Visualização e Iluminação

# Path Tracer Shader



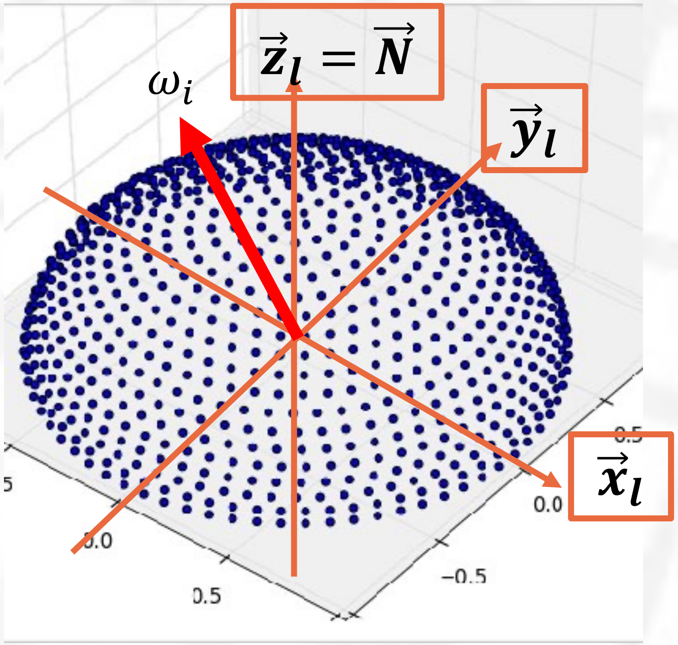
```
RGB PathTracerShader::specularReflection (Intersection isect, Phong *f, int
depth) {
  RGB color(0.,0.,0.); Vector Rdir, s_dir;  float pdf; Intersection s_isect;

  float cos = isect.gn.dot(isect.wo);
  Rdir = 2.f * cos * isect.gn - isect.wo;
  Ray specular(isect.p, Rdir);
  specular.adjustOrigin(isect.gn);
        // trace ray
  bool intersected = scene->trace(specular, &s_isect);
  RGB Rcolor = shade (intersected, s_isect, depth+1);
  color = (f->Ks * Rcolor);
  return color;
}
```

# Diffuse Reflections – the cosine

- Diffuse Interreflections

- Uniform sampling the hemisphere



$$\omega_{i,l} = \begin{cases} x_l = \cos(2\pi\xi_1)\sqrt{1-{\xi_2}^2} \\ y_l = \sin(2\pi\xi_1)\sqrt{1-{\xi_2}^2} \\ z_l = \xi_2 \end{cases}$$

$$p(\omega_{i,l}) = {}^{1}/_{2\pi}$$

$(x_l, y_l, z_l)$ are in the object local coordinate system

They must be rotated to the world coordinate system

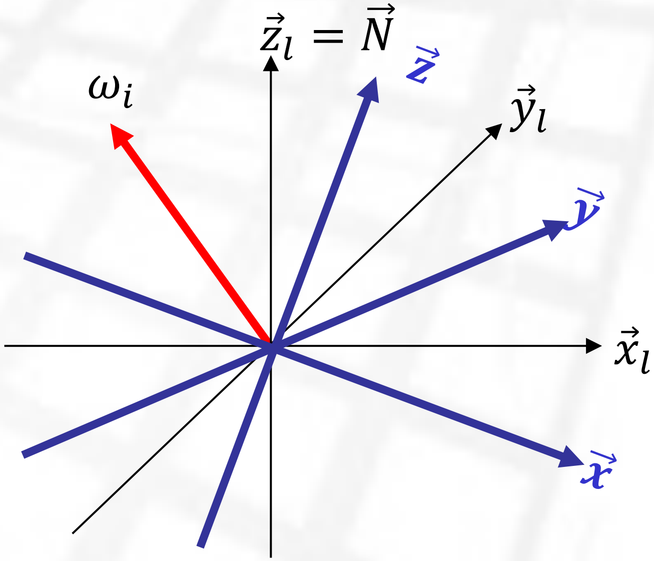
# Diffuse Reflections – the cosine

$$\omega_{i,l} = \begin{cases} x_l = \cos(2\pi\xi_1)\sqrt{1-{\xi_2}^2} \\ y_l = \sin(2\pi\xi_1)\sqrt{1-{\xi_2}^2} \\ z_l = \xi_2 \end{cases}$$

$$p(\omega_{i,l}) = {}^{1}/_{2\pi}$$

$$\vec{N}.CoordinateSystem(\vec{x}_l, \vec{y}_l)$$

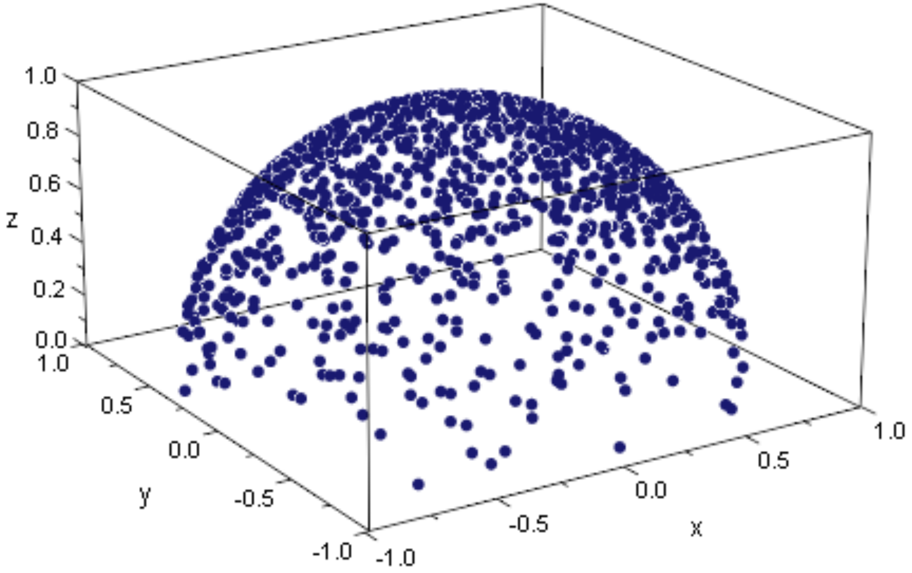
$$\omega_i = \omega_{i,l}.\mathrm{Rotate}(\vec{x}_l, \vec{y}_l, \vec{N})$$



# Diffuse Reflections – cosine sampled

- Diffuse Interreflections

- **Cosine sampling the hemisphere**
  - $x_l = \cos(2\pi\xi_1)\sqrt{1-\xi_2}$
  - $y_l = \sin(2\pi\xi_1)\sqrt{1-\xi_2}$
  - $z_l = \sqrt{\xi_2} = \cos\theta$

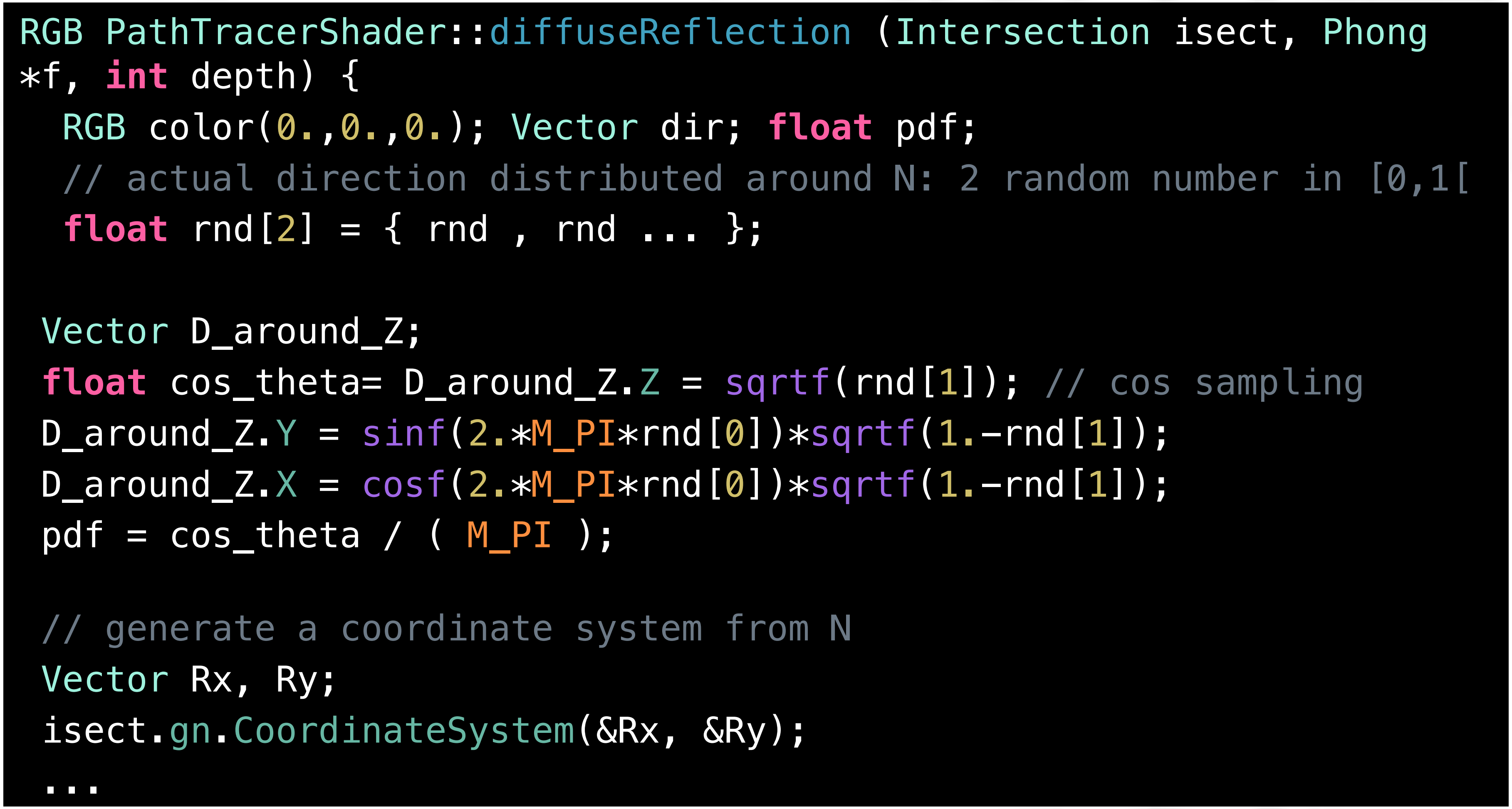


$$p(\omega_i) = \cos\theta / \pi$$

$$\vec{N}.CoordinateSystem(\vec{x}_l, \vec{y}_l)$$

$$\omega_i = \omega_{i,l}.\mathrm{Rotate}\left(\vec{x}_l, \vec{y}_l, \vec{N}\right)$$

# Path Tracer Shader

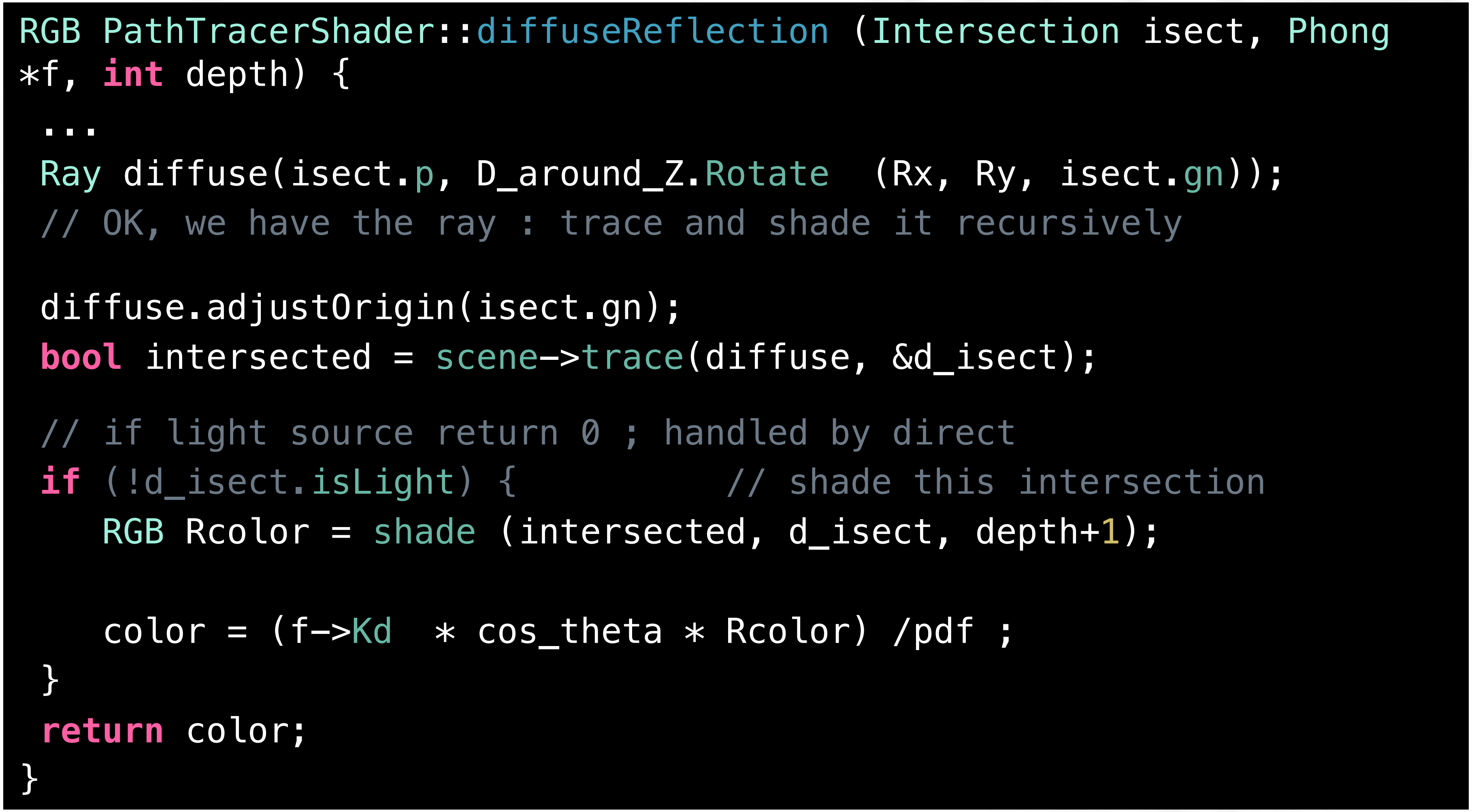


```
RGB PathTracerShader::diffuseReflection (Intersection isect, Phong
*f, int depth) {
  RGB color(0.,0.,0.); Vector dir; float pdf;
  // actual direction distributed around N: 2 random number in [0,1[
  float rnd[2] = { rnd , rnd ... };

 Vector D_around_Z;
 float cos_theta= D_around_Z.Z = sqrtf(rnd[1]); // cos sampling
 D_around_Z.Y = sinf(2.*M_PI*rnd[0])*sqrtf(1.-rnd[1]);
 D_around_Z.X = cosf(2.*M_PI*rnd[0])*sqrtf(1.-rnd[1]);
 pdf = cos_theta / ( M_PI );

 // generate a coordinate system from N
 Vector Rx, Ry;
 isect.gn.CoordinateSystem(&Rx, &Ry);
 ...
```

# Path Tracer Shader



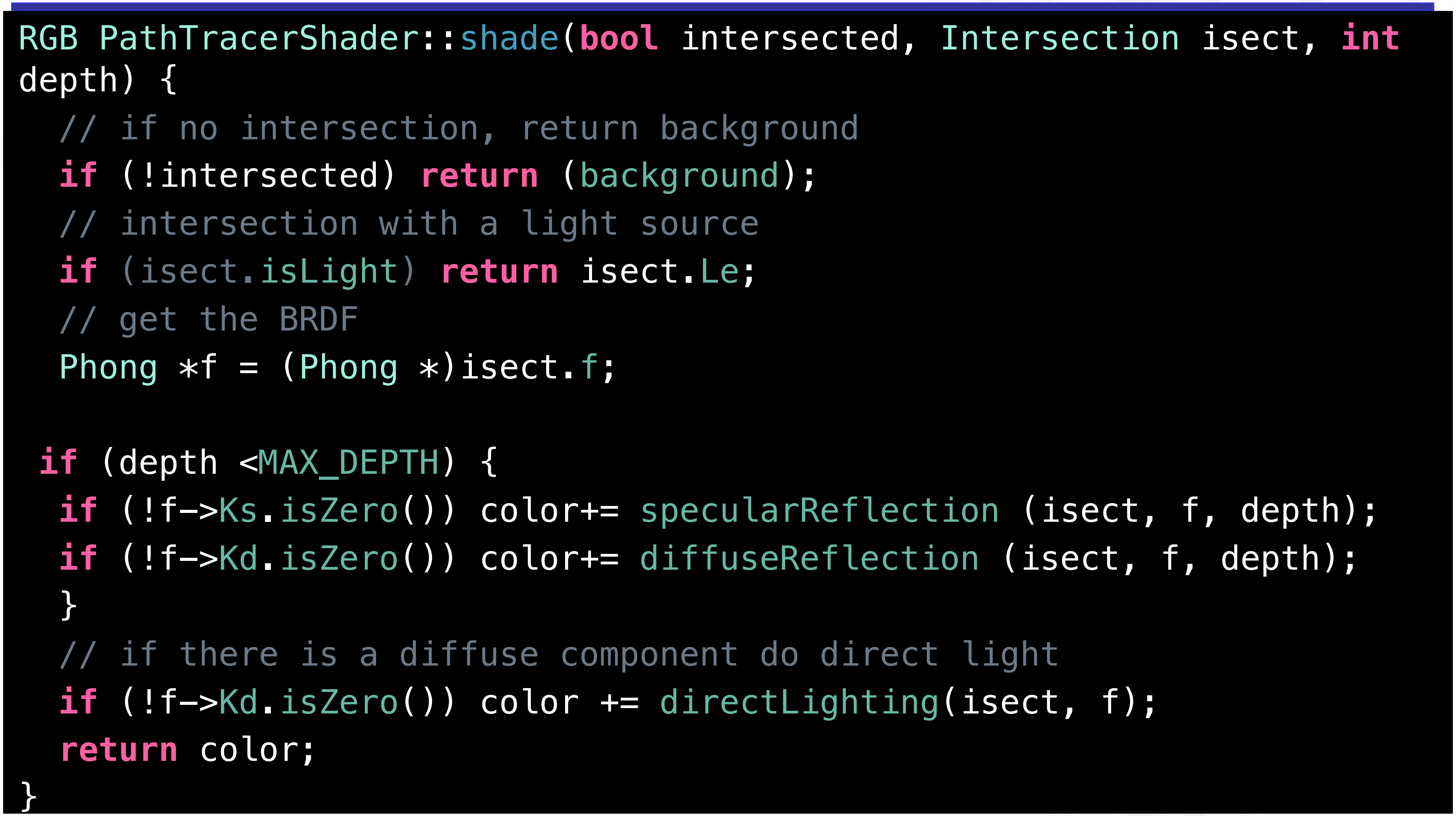
```
RGB PathTracerShader::diffuseReflection (Intersection isect, Phong
*f, int depth) {
 ...
 Ray diffuse(isect.p, D_around_Z.Rotate  (Rx, Ry, isect.gn));
 // OK, we have the ray : trace and shade it recursively

 diffuse.adjustOrigin(isect.gn);
 bool intersected = scene->trace(diffuse, &d_isect);

 // if light source return 0 ; handled by direct
 if (!d_isect.isLight) {          // shade this intersection
    RGB Rcolor = shade (intersected, d_isect, depth+1);

    color = (f->Kd  * cos_theta * Rcolor) /pdf ;
 }
 return color;
}
```

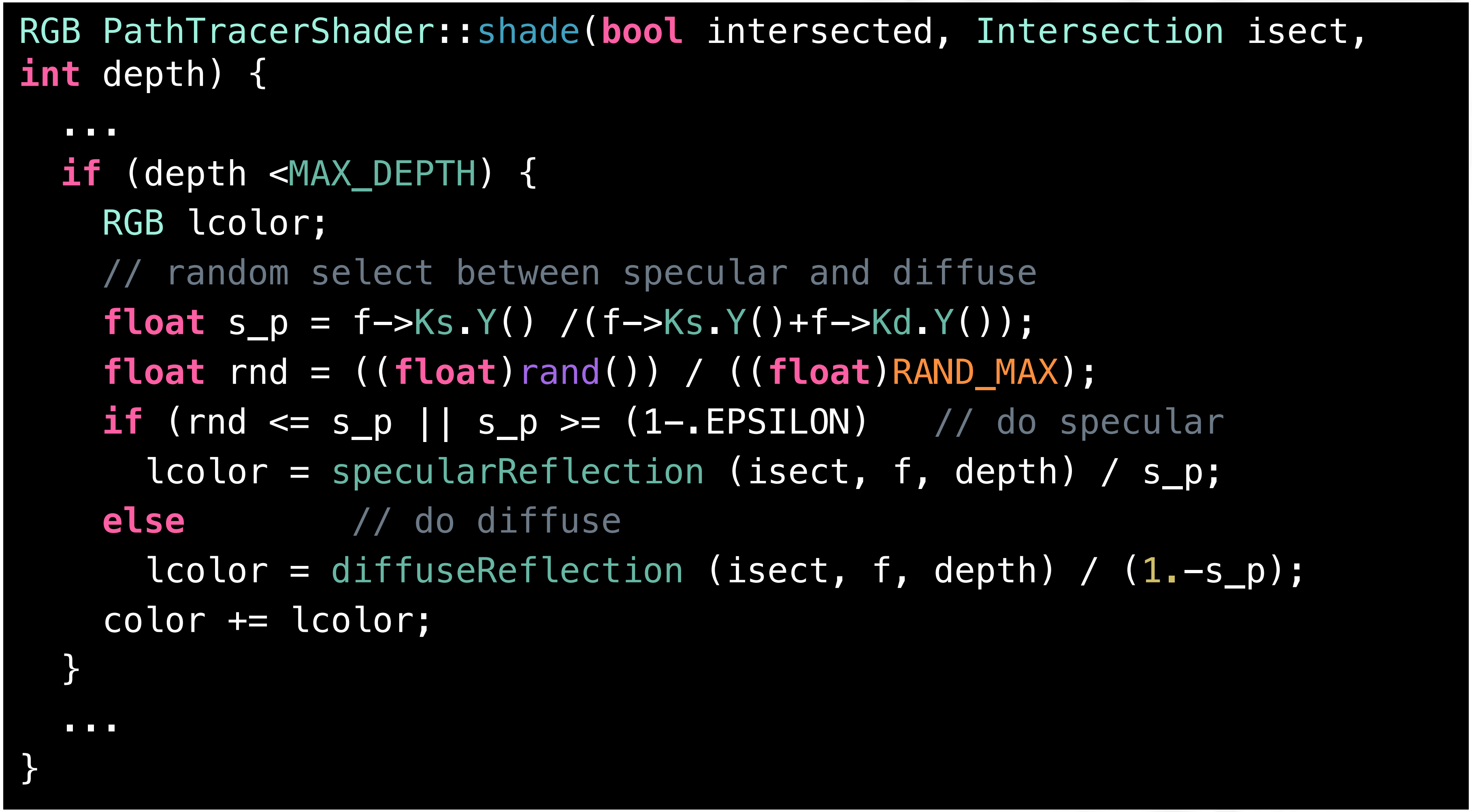
# The shade() method



```
RGB PathTracerShader::shade(bool intersected, Intersection isect, int
depth) {
  // if no intersection, return background
  if (!intersected) return (background);
  // intersection with a light source
  if (isect.isLight) return isect.Le;
  // get the BRDF
  Phong *f = (Phong *)isect.f;

 if (depth <MAX_DEPTH) {
  if (!f->Ks.isZero()) color+= specularReflection (isect, f, depth);
  if (!f->Kd.isZero()) color+= diffuseReflection (isect, f, depth);
  }
  // if there is a diffuse component do direct light
  if (!f->Kd.isZero()) color += directLighting(isect, f);
  return color;
}
```

# MC sampling: specular or diffuse ?



```
RGB PathTracerShader::shade(bool intersected, Intersection isect,
int depth) {
  ...
  if (depth <MAX_DEPTH) {
    RGB lcolor;
    // random select between specular and diffuse
    float s_p = f->Ks.Y() /(f->Ks.Y()+f->Kd.Y());
    float rnd = ((float)rand()) / ((float)RAND_MAX);
    if (rnd <= s_p || s_p >= (1-.EPSILON)   // do specular
      lcolor = specularReflection (isect, f, depth) / s_p;
    else        // do diffuse
      lcolor = diffuseReflection (isect, f, depth) / (1.-s_p);
    color += lcolor;
  }
  ...
}
```

# Monte Carlo: bias

- Quando parar a emissão de raios secundários *(path length)?*
  - Usar uma profundidade máxima fixa
  - Quando a contribuição esperada de um raio é inferior a um dado limite
- Estes são métodos determinísticos que afectam o valor do integral (*bias)*!

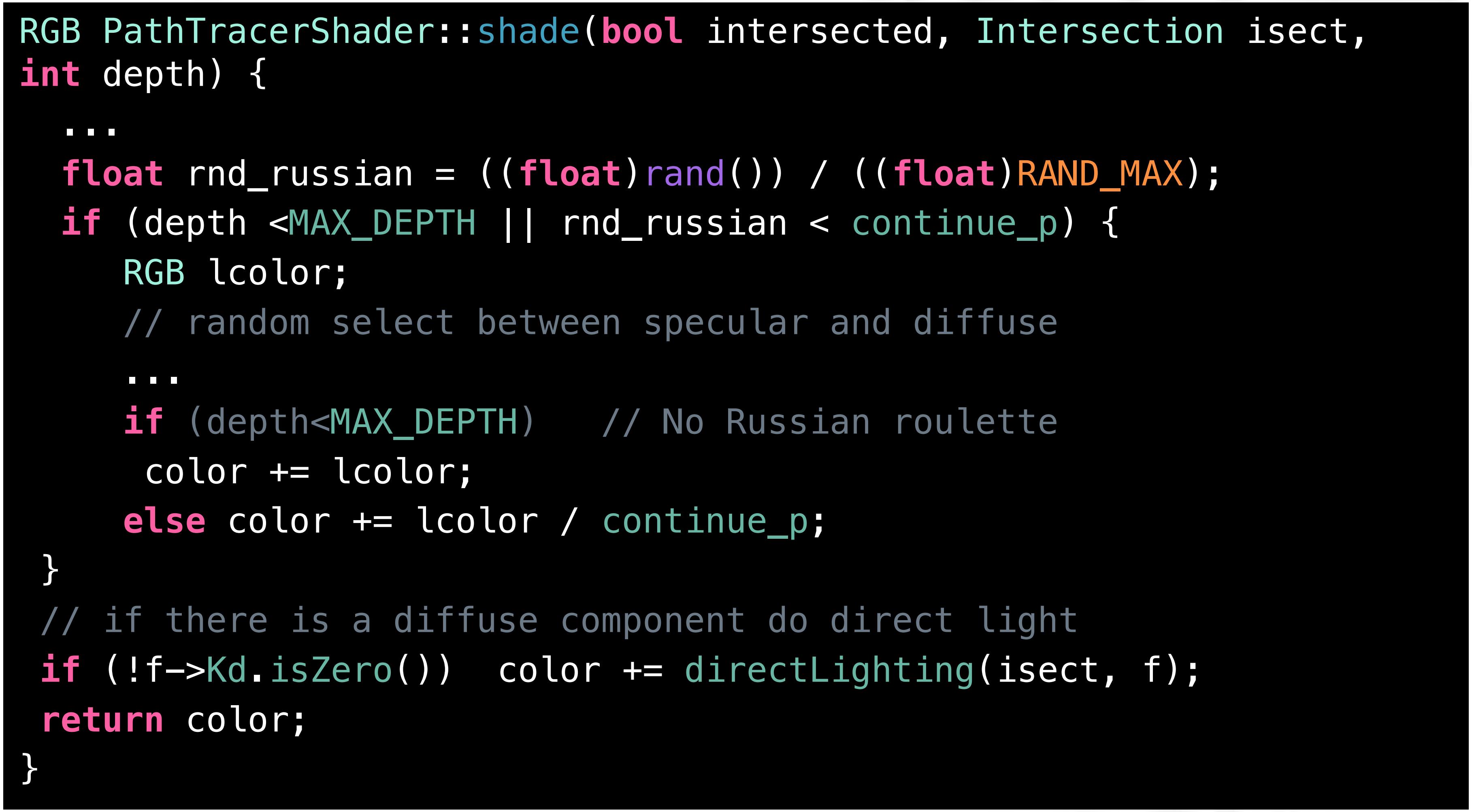
| *not biased* | *biased* |
|---|---|
| $\lim_{N\to\infty}\langle I\rangle = I$ | $\lim_{N\to\infty}\langle I\rangle = I + \varepsilon$ |

# Monte Carlo: roleta russa

- Definir a probabilidade $p_{cont}$ de continuar a travessia (disparar um raio secundário)
- Antes de disparar um raio gerar um número aleatório, ξ, uniformemente distribuído em [0, 1[
- Se ξ <= $p_{cont}$ então disparar o raio
- Se ξ > $p_{cont}$ então não disparar o raio
- Uma vez que há uma probabilidade de não disparar um raio, a contribuição dos raios disparados deve ser multiplicada por 1/ $p_{cont}$, para compensar aqueles que não são disparados

$$L(p \leftarrow \Psi) \approx \frac{1}{p_{cont}} (\xi \leq \alpha \ ? \ 0 : L(p \leftarrow \Psi_i))$$

# The shade( ) method: Russian roulette



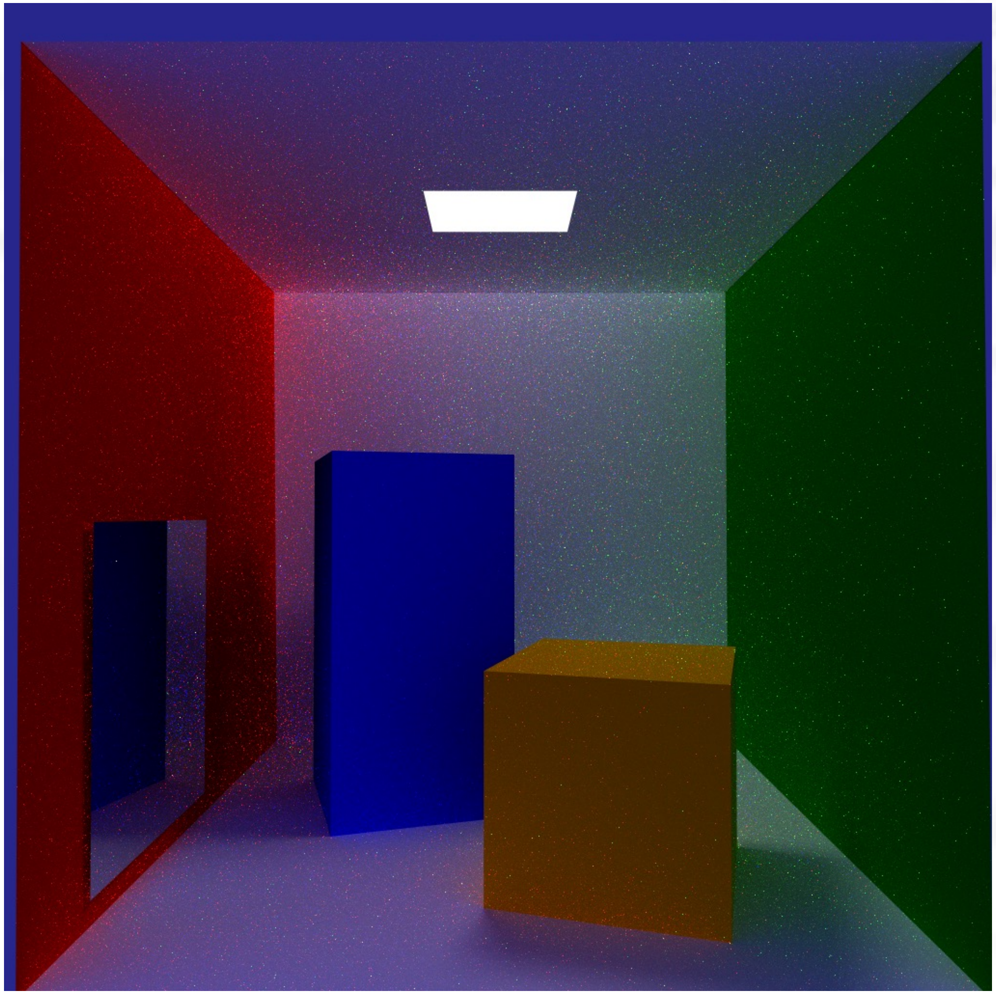
```
RGB PathTracerShader::shade(bool intersected, Intersection isect,
int depth) {
  ...
  float rnd_russian = ((float)rand()) / ((float)RAND_MAX);
  if (depth <MAX_DEPTH || rnd_russian < continue_p) {
     RGB lcolor;
     // random select between specular and diffuse
     ...
     if (depth<MAX_DEPTH)   // No Russian roulette
      color += lcolor;
     else color += lcolor / continue_p;
 }
 // if there is a diffuse component do direct light
 if (!f->Kd.isZero())  color += directLighting(isect, f);
 return color;
}
```

# PathTracerShader.hpp

```
class PathTracerShader: public Shader {
    RGB background;
    RGB directLighting (Intersection isect, Phong *f);
    RGB specularReflection (Intersection isect, Phong *f, int
depth);
    RGB diffuseReflection (Intersection isect, Phong *f, int depth);
    float continue_p;
    int MAX_DEPTH;

public:
    PathTracerShader (Scene *scene, RGB bg): background(bg),
Shader(scene) {continue_p = 0.5f; MAX_DEPTH=2;}
    RGB shade (bool intersected, Intersection isect, int depth);
};
```

# Path Traced Cornell Box



2048 spp
2815 secs