Mestrado em
Engenharia Informática

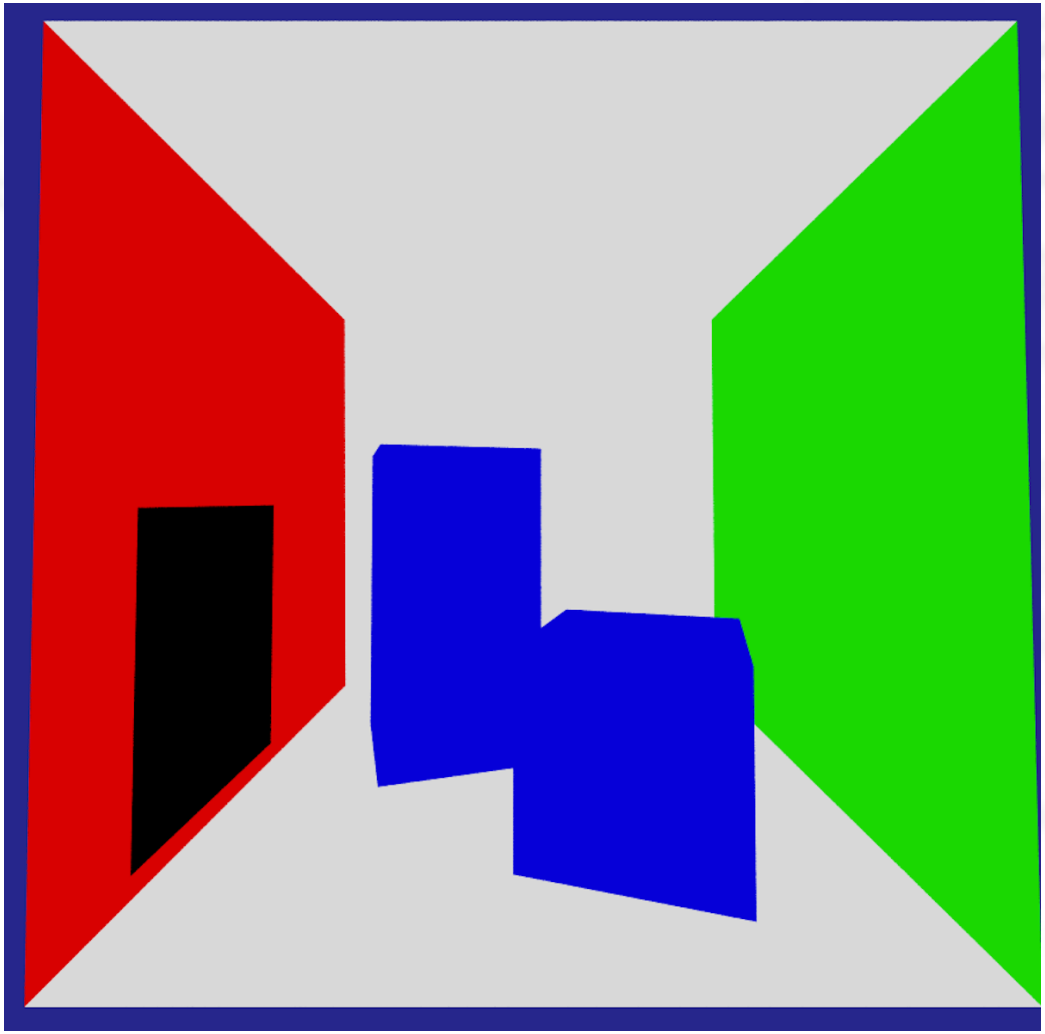
VI-RT

Whitted Ray Tracing

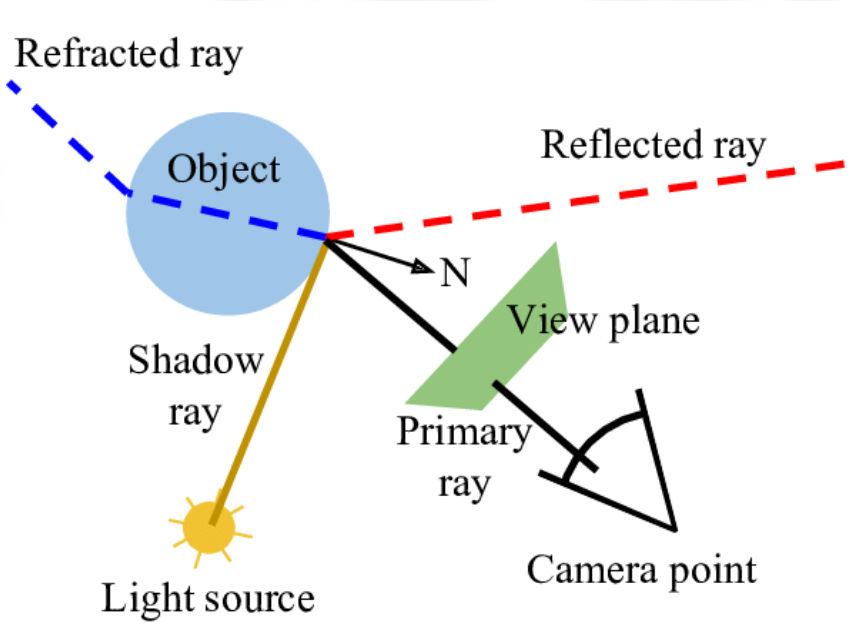
Visualização e Iluminação

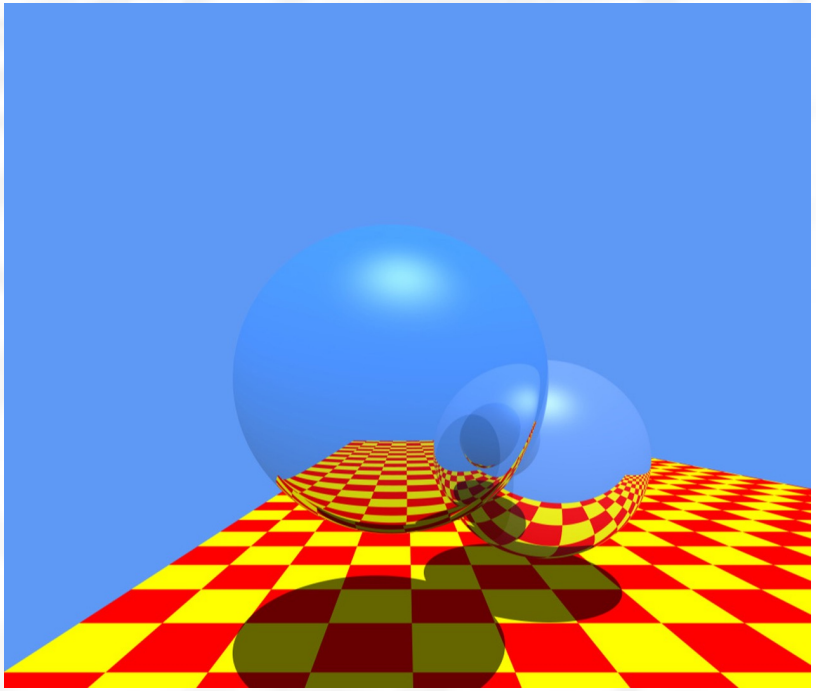
Luís Paulo Peixoto dos Santos

Ambient Lighting



use scene:
models/cornell_box_VI.obj


Refracted ray
Object
Reflected ray
N
View plane
Shadow
ray
Primary
ray
Camera point
Light source



ADD SUPPORT FOR POINT LIGHT SOURCES

PointLight.hpp

```
class PointLight: public Light {
public:
    RGB color;
    Point pos;
    PointLight (RGB _c, Point _p): color(_c), pos(_p) {
        type = POINT_LIGHT; }
    ~PointLight () {}
    // return the Light RGB radiance for a given point : p
    RGB L (Point p) {return color;}
    RGB L () {return color;}
    // return a point p and RGB radiance for a given probability
pair prob[2] (ignore probability for point light)
    RGB Sample_L (float *prob, Point *p) {
        *p = pos;
        return color;
    }
};
```

add light sources to the scene

```
    // add an ambient light to the scene
    AmbientLight *ambient = new AmbientLight(RGB(0.05,0.05,0.05));
    scene.lights.push_back(ambient);
    scene.numLights++;
    // add a point light to the scene
    PointLight *pl1 = new PointLight(RGB(0.65,0.65,0.65),
Point(288,508,282));
    scene.lights.push_back(pl1);
    scene.numLights++;
```

WHITTED STYLE SHADING

WhittedShader.hpp

```
#include "shader.hpp"
#include "Phong.hpp"

class WhittedShader: public Shader {
    RGB background;
    RGB directLighting (Intersection isect, Phong *f);
    RGB specularReflection (Intersection isect, Phong *f,
            int depth);
public:
    WhittedShader (Scene *scene, RGB bg): background(bg),
Shader(scene) {}
    RGB shade (bool intersected, Intersection isect,
            int depth);
};
```

WhittedShader.cpp

```
RGB WhittedShader::shade(bool intersected, Intersection isect, int
depth) {
    RGB color(0.,0.,0.);

    if (!intersected) {    return (background);    }
    // get the BRDF
   Phong *f = (Phong *)isect.f;

    // if there is a specular component sample it
    if (!f->Ks.isZero() && depth < 3)
        color += specularReflection (isect, f, depth+1);

    color += directLighting(isect, f);

    return color;
}
```

WhittedShader.cpp – direct

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

    // Loop over scene's light sources
    for (auto l = scene->lights.begin() ; l != scene->lights.end() ;
l++) {

        if ((*l)->type == AMBIENT_LIGHT) {
            if (!f->Ka.isZero()) color += f->Ka * (*l)->L();
            continue;
        }
        if ((*l)->type == POINT_LIGHT) {  // is it a point light ?
            ....
       }
     }
    return color; }
```

WhittedShader.cpp – direct

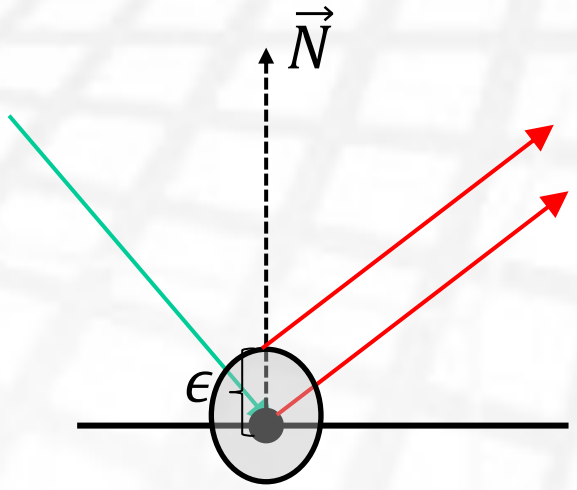
```
if ((*l)->type == POINT_LIGHT) {  // is it a point light ?
    if (!f->Kd.isZero()) {
     Point lpoint;
     // get the position and radiance of the light source
     RGB L = (*l)->Sample_L(NULL, &lpoint);

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

      // compute the cosine (Ldir , shading normal)
      float cosL = Ldir.dot(isect.sn);


       ...
    }
    continue; }
```

WhittedShader.cpp – direct

```
if ((*l)->type == POINT_LIGHT) {  // is it a point light ?
    if (!f->Kd.isZero()) {
      ...
      // compute the cosine (Ldir , shading normal)
      float cosL = Ldir.dot(isect.sn);
       if (cosL>0.)  { // the light is NOT behind the primitive
          // generate the shadow ray
          Ray shadow(isect.p, Ldir);
          // adjust origin EPSILON along the normal: avoid self occlusion
          shadow.adjustOrigin(isect.gn);

          if (scene->visibility(shadow, Ldistance-EPSILON))  // light source
not occluded
              color += f->Kd * L * cosL;
        } // end cosL > 0.
      ...
}
```

Ray.hpp – self-occlusion

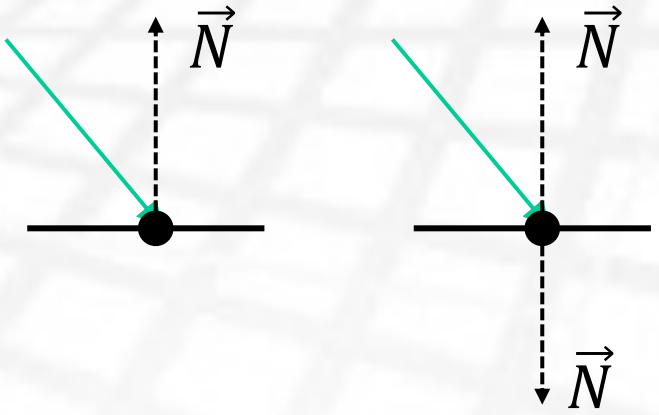
```
const float EPSILON=1e-3;

class Ray {
public:
...
void adjustOrigin (Vector normal) {
    Vector offset = EPSILON * normal;
    if (dir.dot(normal) < 0)
        offset = -1.f * offset;
    o = o + offset;
}
...}
```



- the secondary ray intersects the geometry where it has the origin
- the problem is made more probable due to limited accuracy
- adjust the origin along the normal

Normal Face Forward

```
// Fill Intersection data
Vector normal = f.geoNormal;
Vector wo = -1.f * r.dir;
// make sure the normal points to
the same side of the surface as wo
normal.Faceforward(wo);
isect->gn = normal;
isect->sn = normal;
```



When you compute an intersection make sure you return a Face Forwarding normal!

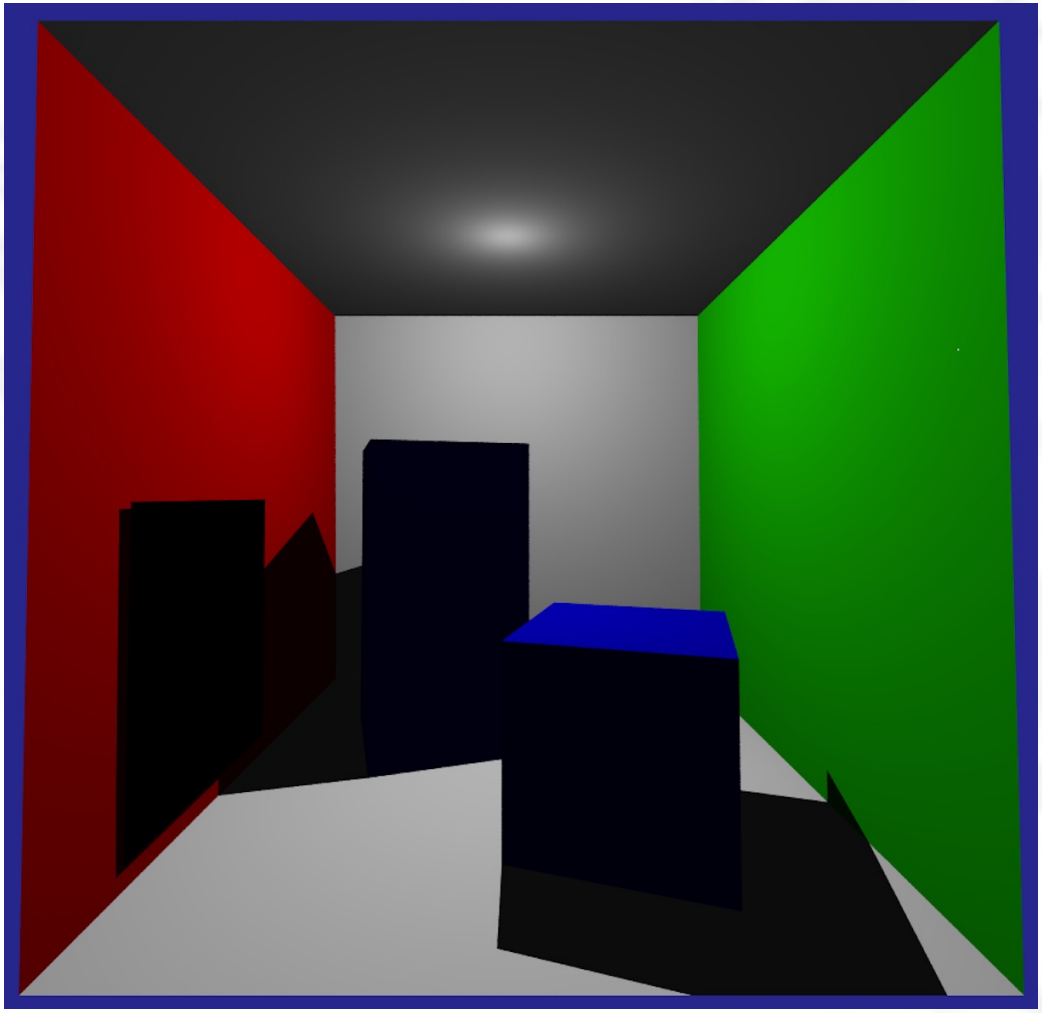
```
Vector Faceforward(const Vector &v) const {
  Vector vv = *this;
  return (vv.dot(v) < 0.f) ? -1.f * vv : vv; }
```

Scene.cpp – visibility

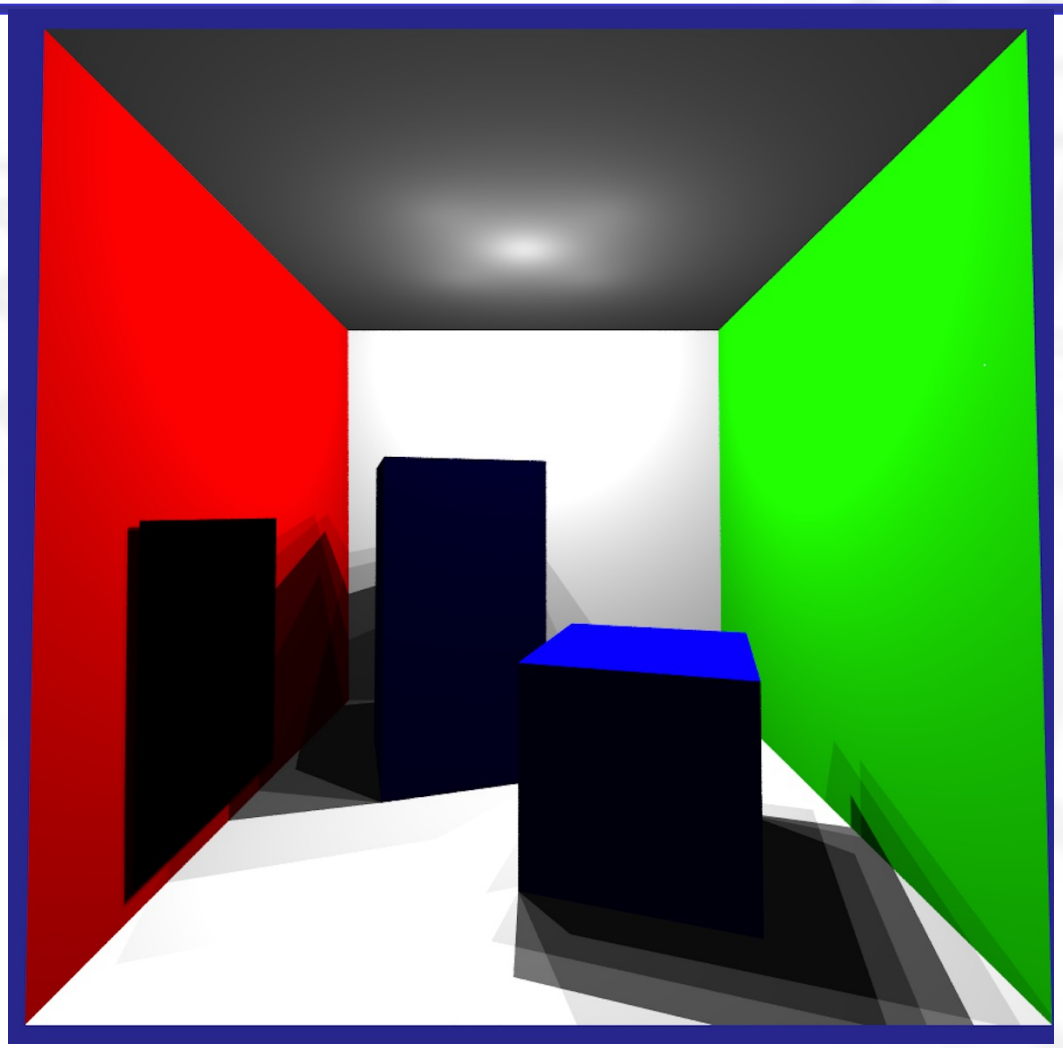
```
bool Scene::visibility (Ray s, const float maxL) {
    bool visible = true;
    Intersection curr_isect;

    // iterate over all primitives while visible
    for (auto p = prims.begin() ; p != prims.end() && visible ; p++) {
        if ((*p)->g->intersect(s, &curr_isect)) {
            if (curr_isect.depth < maxL) {
                visible = false;
            }
        }
    }
    return visible;  }
```

Similar to `Scene::trace()` but finishes immediately once the 1st intersection is found.

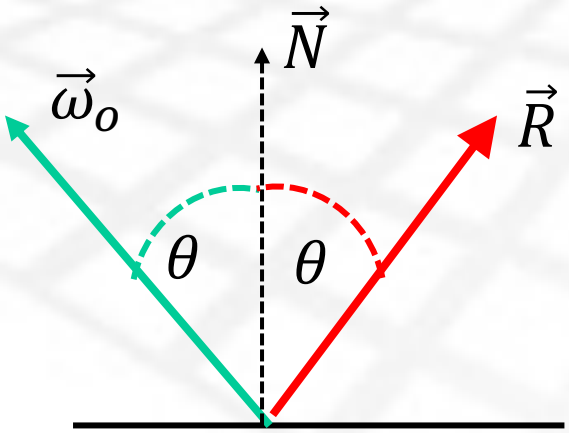
1 Point Light Source



5 Point Light Sources



Specular Reflection

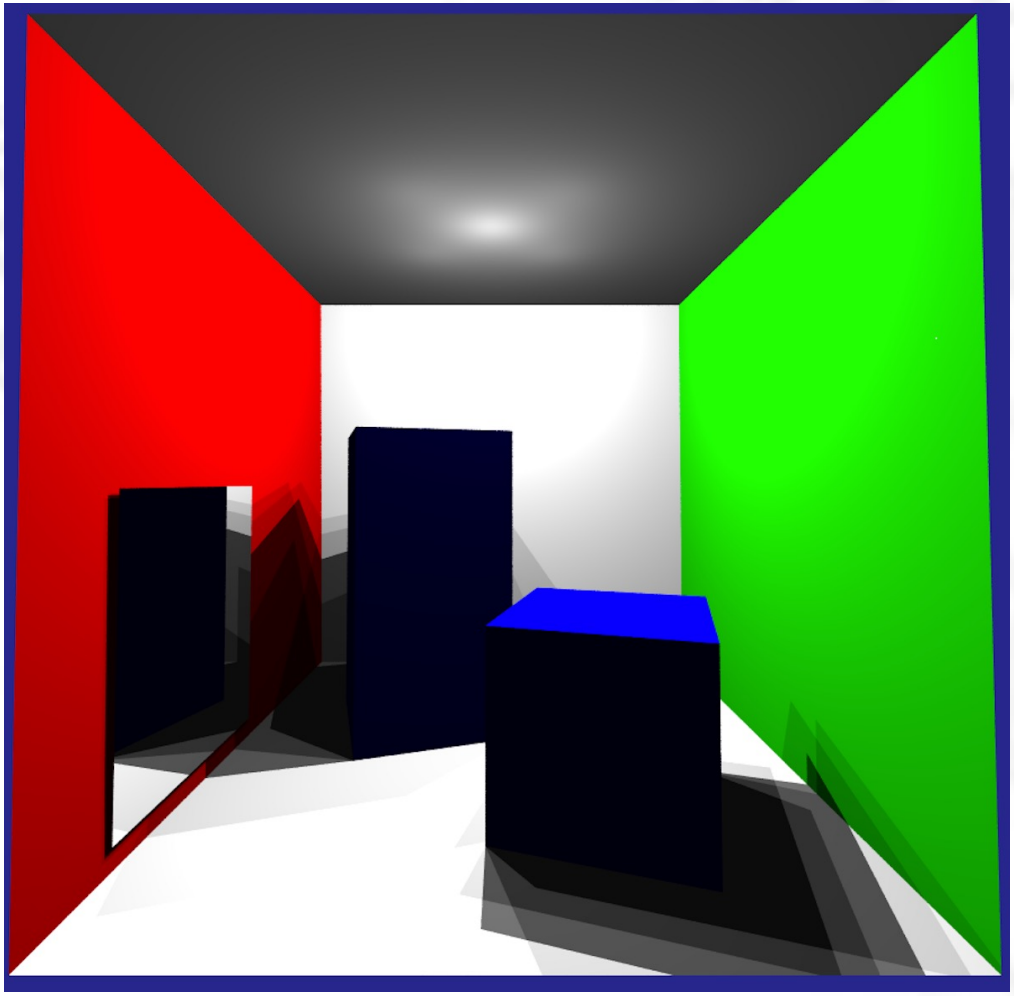


$$\vec{R} = 2(\vec{N} \cdot \vec{\omega}_o)\vec{N} - \vec{\omega}_o$$

WhittedShader.cpp – specular

```
RGB WhittedShader::specularReflection (Intersection isect, Phong *f,
int depth) {
    // generate the specular ray
    float cos = isect.gn.dot(isect.wo);
    Vector Rdir = 2.f * cos * isect.gn - isect.wo;
    Ray specular(isect.p, Rdir);
    specular.adjustOrigin(isect.gn);

    Intersection s_isect;
    // trace ray
    bool intersected = scene->trace(specular, &s_isect);
    // shade this intersection
    RGB color = shade (intersected, s_isect, depth+1);
    return color;
}
```

Specular Reflection



To think…

1. Does the image in slide 16 ha ve an Ambient component?

2. Why do the walls seem much more realistic on slide 16 than when using ambient lighting only (see slide 2)?

3. Does rendering time increase with the number of light sources?

4. Does the rendering time increase with the presence of a mirror?

5. If so, is there any relationship between the number of pixels onto which the mirror projects and this increase in rendering time?