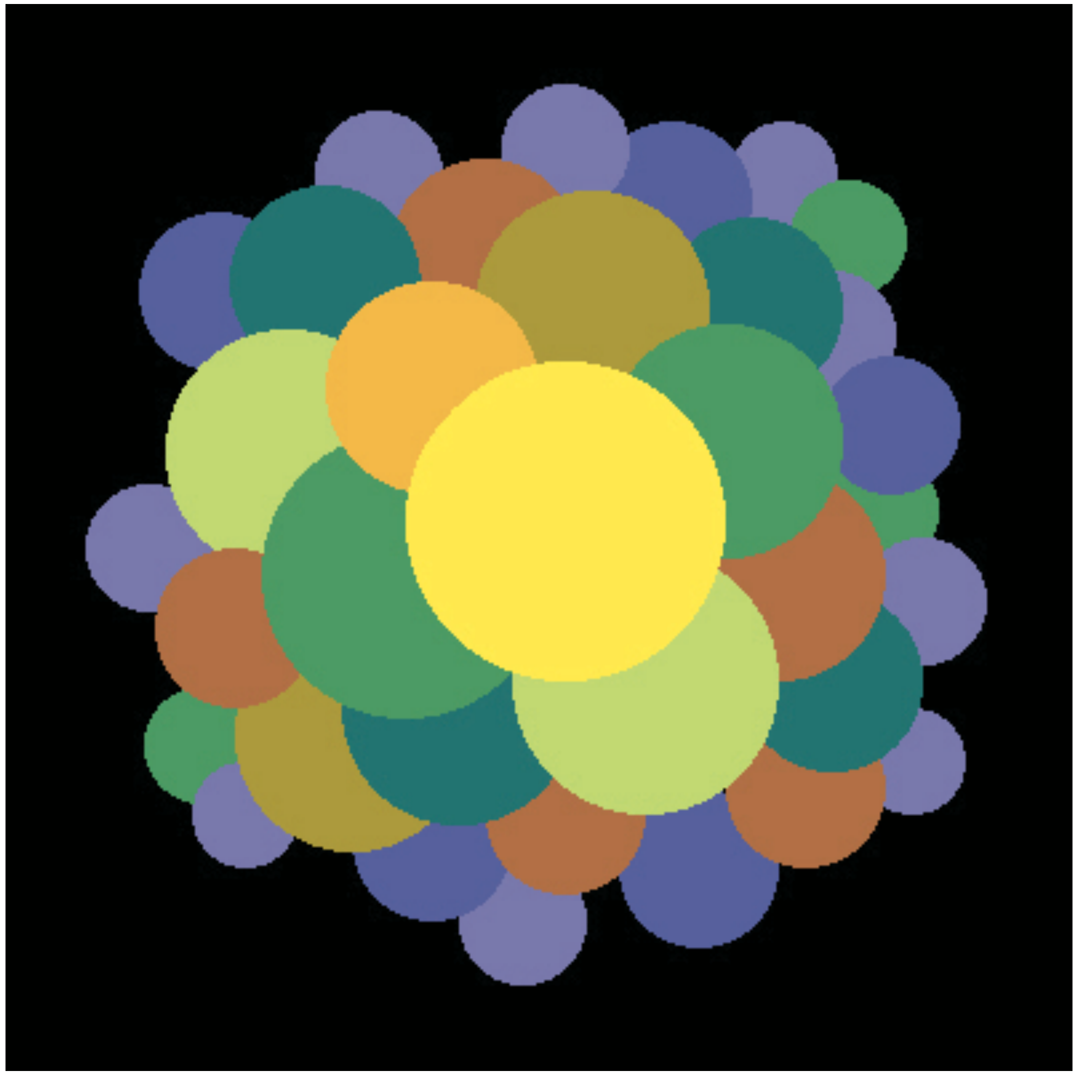
01418585
Rendering and Shading Techniques

Lecture 03

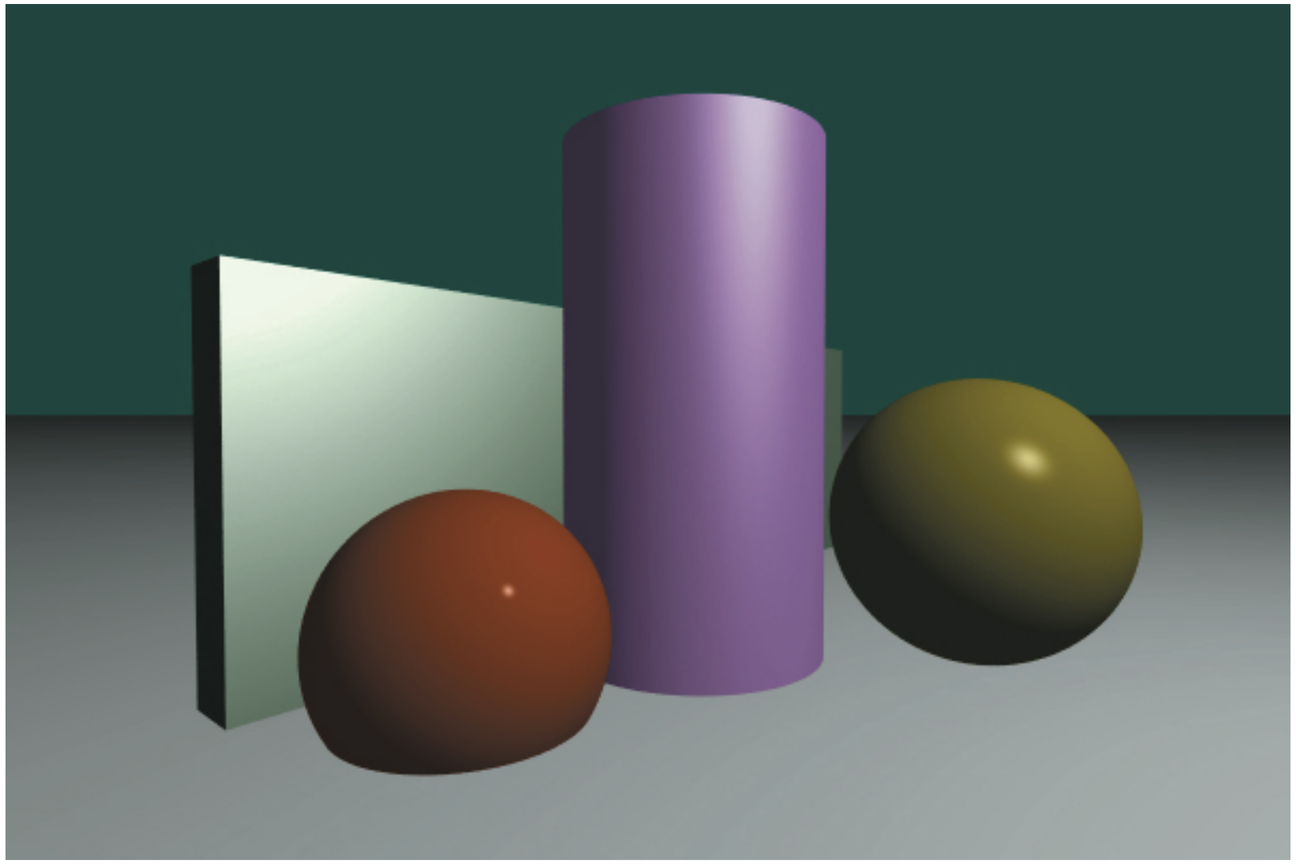
Ray Tracing with Local Illumination Model

At the end of lecture 01...



Last Lecture

- Perspective Camera
- Phong Lighting Model
  - Ambient
  - Diffuse
  - Specular



To shade with a Phong lighting model

- Need:
    - Light sources
    - Material properties
    - Surface normals
- Need to pass all this information to various parts of the system.

ShadeRec Structure

- Stores all information needed to shade a pixel.
  - Hit point (p)
  - Surface normal (n)
  - Outgoing light direction (wo, which stands for ω_o)
  - Surface parameterization (uv)
  - Whether the ray intersects the inside or the outside of the geometry
  - Time required for the ray to travel to the hit point.

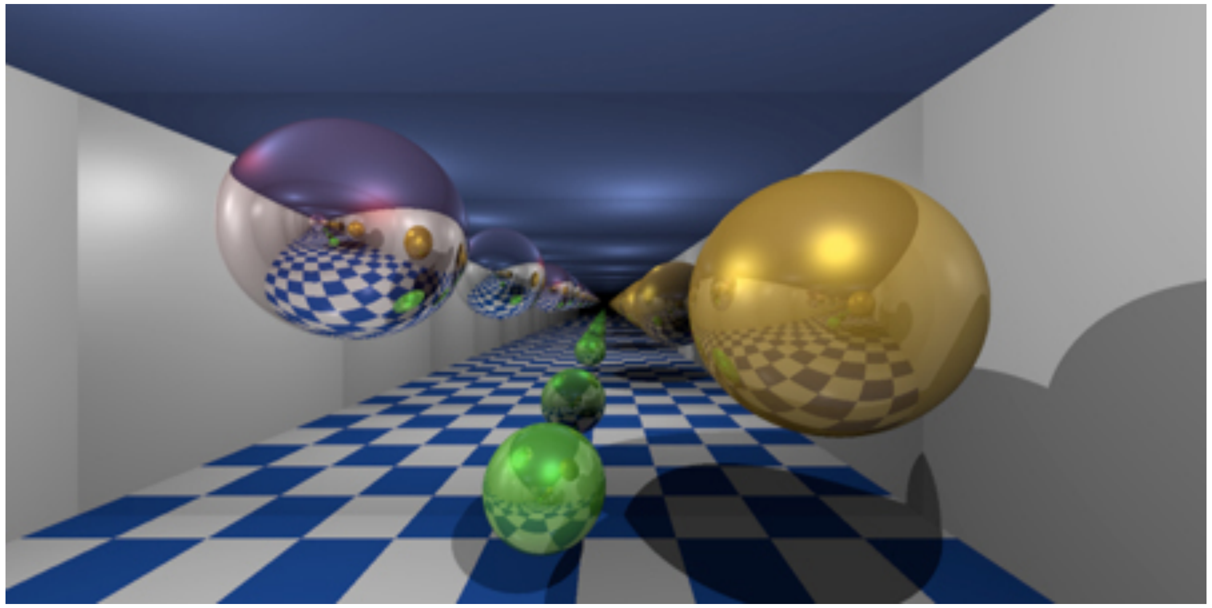
ShadeRec Structure

```
struct ShadeRec
{
    ShadeRec(
        Float3 _p = Float3(0,0,0),
        Float3 _n = Float3(0,0,1),
        Float3 _wo = Float3(0,0,1),
        Float2 _uv = Float2(0,0),
        bool _outside = true,
        float _time = 0);

public:
    Float3 point;
    Float3 normal;
    Float3 w_out;
    Float2 uv;
    bool outside;
    float time;
};
```

Surface Parameterization

- Assigning coordinates to each point on the surface.
  - Typically called “u” and “v”.
- Useful for texture mapping.

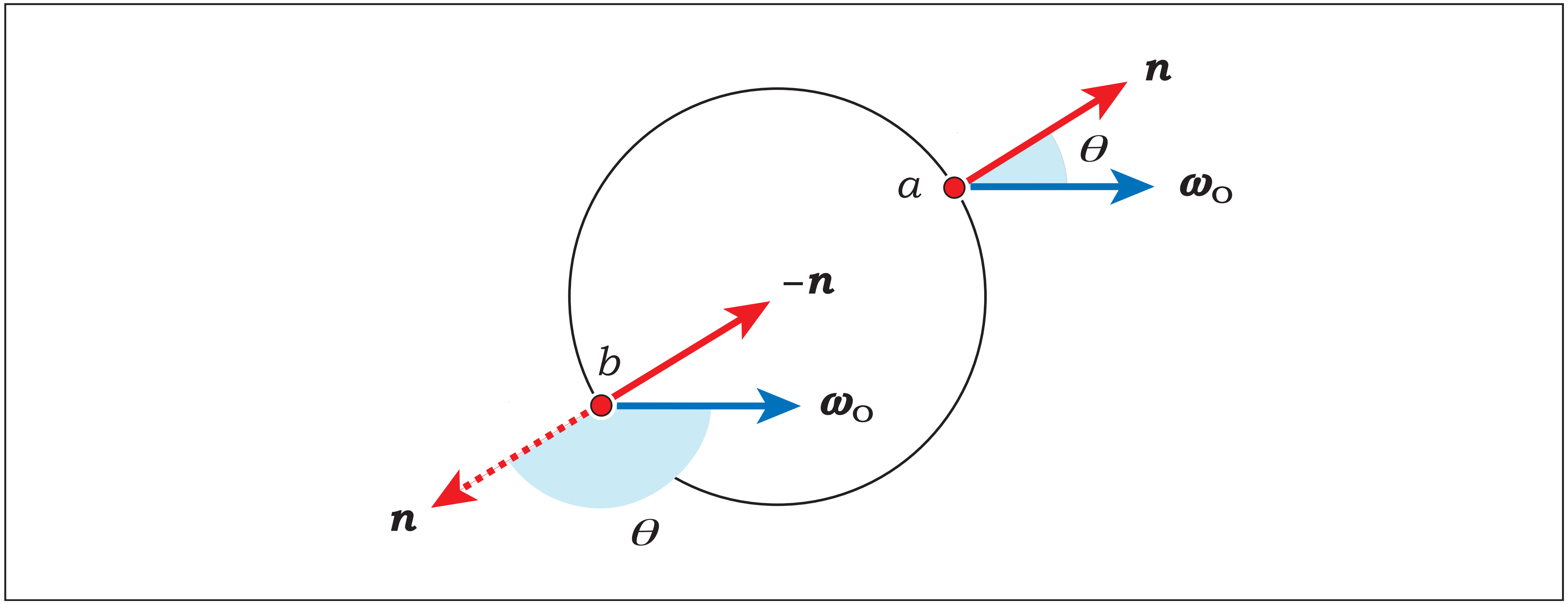


Inside or Outside?

- Useful when dealing with refraction.
- As a rule: **the normal always points outside of the surface**.

Reversing Normal

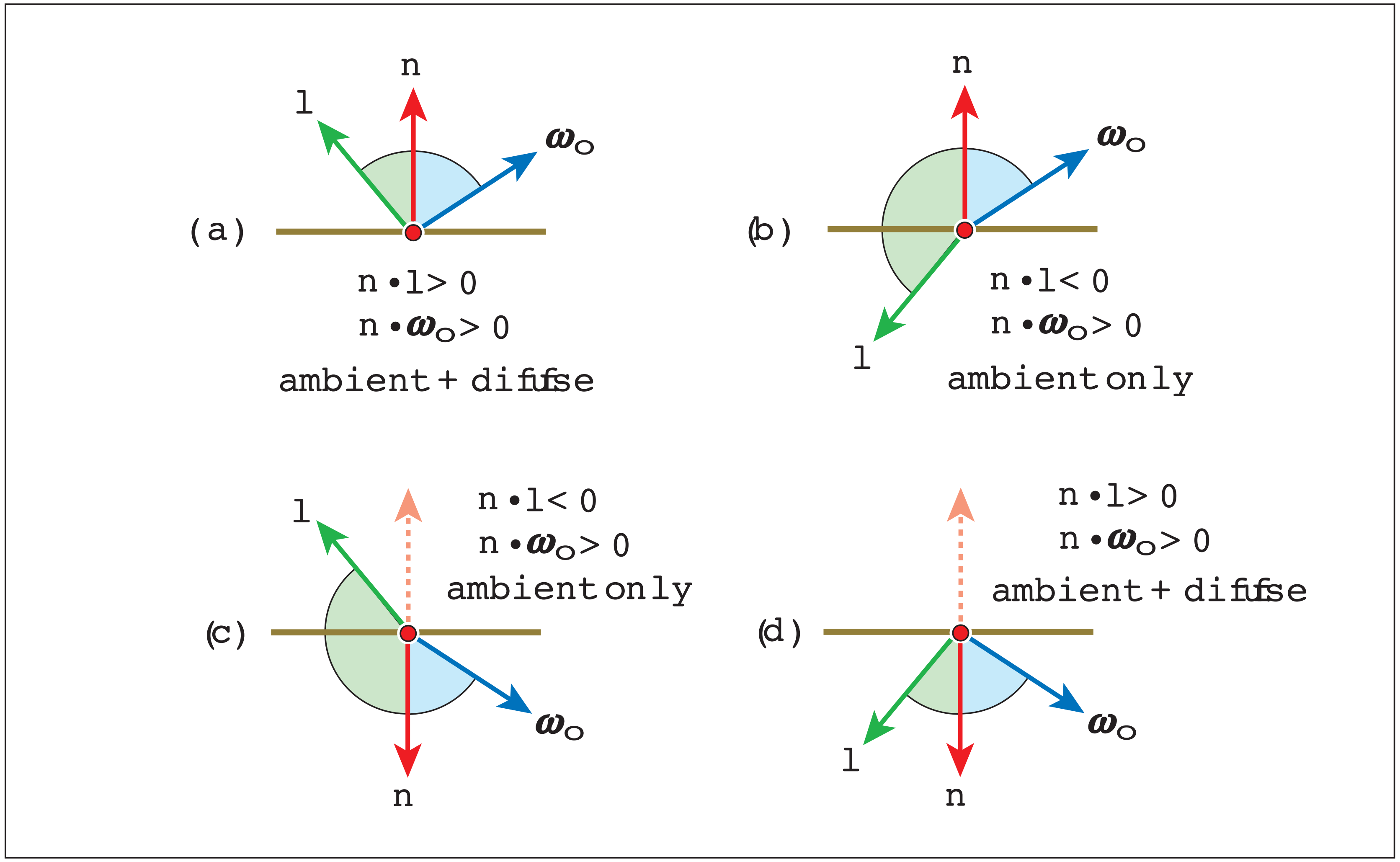
- Need to reverse normal direction when intersect is on the inside.
  - This happens when $\mathbf{n} \cdot \omega_o < 0$



Why Reversing Normal?

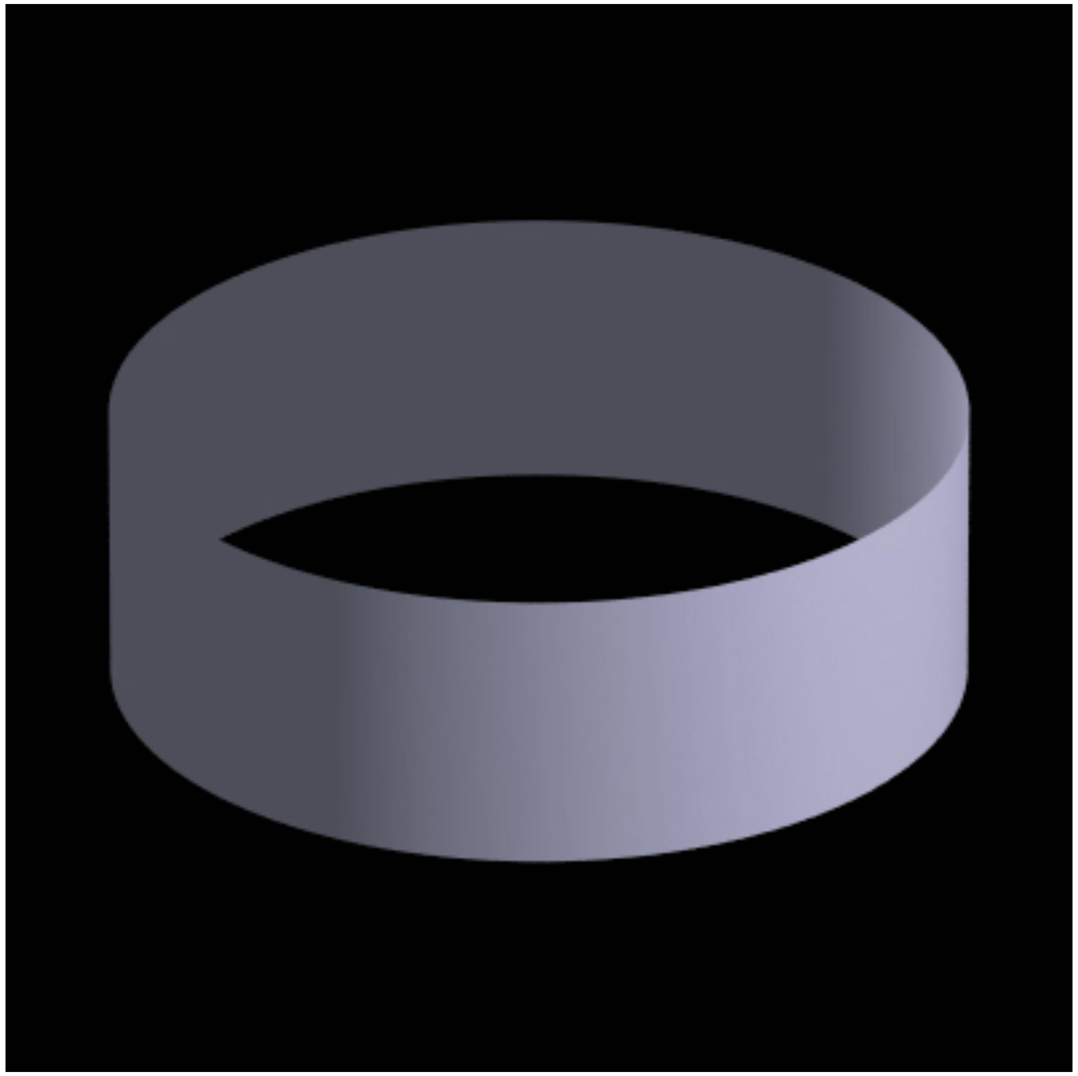
- Recall the ambient and diffuse term of the Phong lighting model:

$$I = k_a I_a + k_d(\mathbf{n} \cdot \omega_o) I_L$$

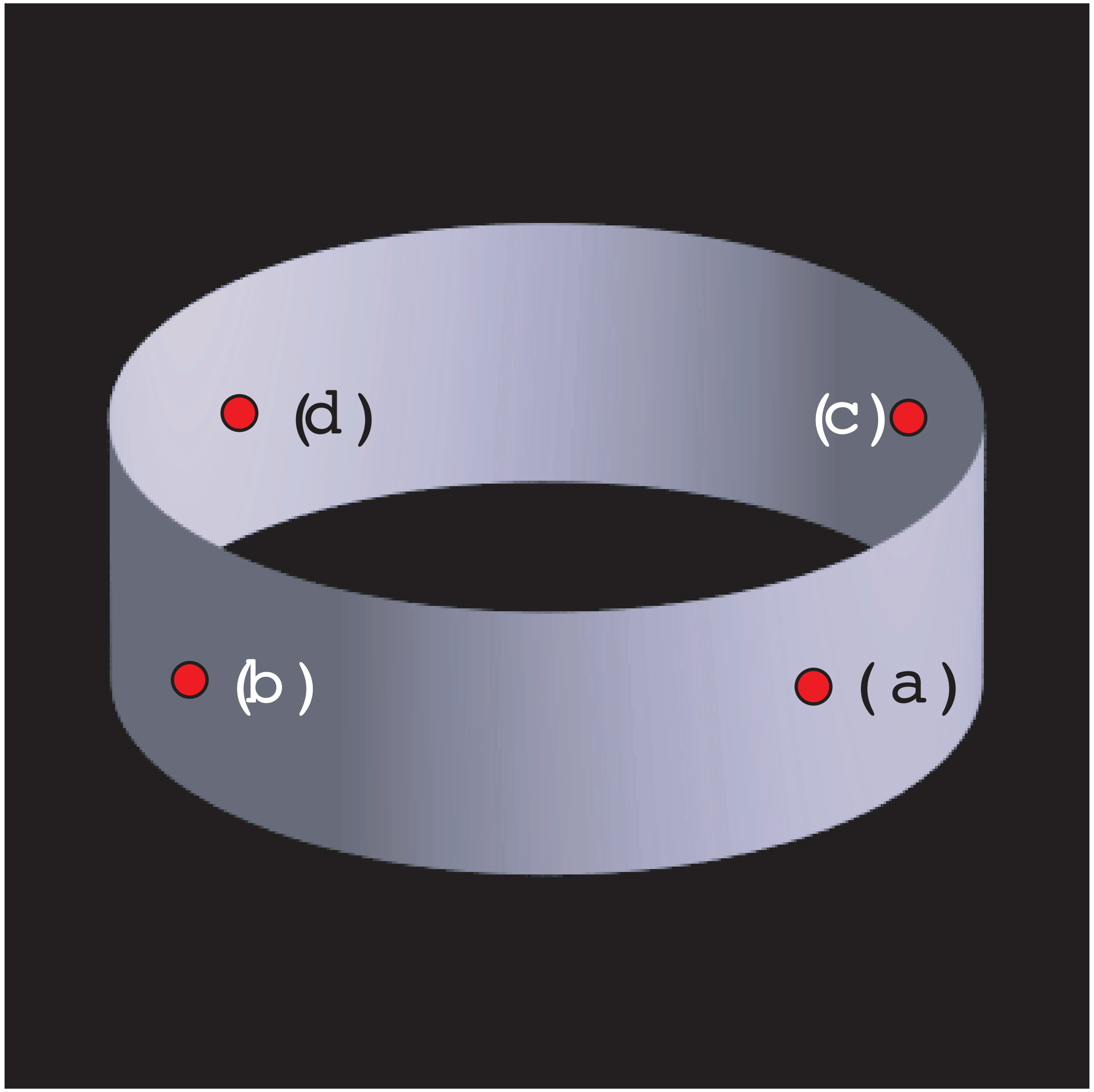


If you don’t reverse the normal...

- What’s wrong with this image?



What it should have been...



A Change to Shape Interface

- Two methods for intersection.
  - intersect -- compute ShadeRec for shading
  - intersect_p -- compute only whether the ray intersects the shape or not
- Use
  - intersect_p to find the closest hit
  - intersect to compute ShadeRec of the closest hit.
- Every Shape now has a pointer to Material. (More on this later.)

Modified Plane Class

- Lecture 01
  - Plane defined by a point (**a**), and a normal vector (**n**).
  - Information not enough to compute parameterization

- Define plane with three vectors
  - Normal (**n**), tangent (**t**), and binormal (**b**)
  - These vectors are orthogonal.
  - Relationship: $\mathbf{t} \times \mathbf{b} = \mathbf{n}$
  - All points in the plane can be written as:

$$\mathbf{p} = \mathbf{a} + u\mathbf{t} + v\mathbf{b}$$

  with (u,v) being its surface parameterization.

Modified Plane Class

- Notice that
  - Point a that defines the plane has UV coordinate (0,0)
  - u and v can be computed as follows:

$$u = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{t}$$
$$v = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{b}$$

Modified Plane Class

```
class Plane : public Shape
{
public:
    Plane(
        const Float3 &_point = Float3(0,0,0),
        const Float3 &_normal = Float3(0,1,0),
        const Float3 &_tangent = Float3(1,0,0),
        Material *_material = NULL);
    virtual ~Plane();
    virtual bool intersect_p(Ray &ray);
    virtual bool intersect(Ray &ray, ShadeRec &shade_rec);

public:
    Float3 point;
    Float3 normal;
    Float3 binormal;
    Float3 tangent;
};
```

Modified Plane Class

```
bool Plane::intersect( Ray &ray, ShadeRec &shade_rec )
{
    bool hit = intersect_p(ray);
    if (hit)
    {
        shade_rec.point = ray(ray.tmax);
        shade_rec.w_out = -ray.direction;

        if (dot(ray.direction, normal) < 0)
        {
            shade_rec.normal = normal;
            shade_rec.outside = true;
        }
        else
        {
            shade_rec.normal = -normal;
            shade_rec.outside = false;
        }
```

Modified Plane Class

```
        float tx = dot(shade_rec.point - point, tangent);
        float ty = dot(shade_rec.point - point, binormal);
        shade_rec.uv = Float2(tx, ty);

        shade_rec.time = ray.tmax;

        return true;
    }
    else
        return false;
}
```

Modified Sphere Class

- intersect_p is the same as in Lecture 01

- If **p** is the hit point, the surface normal at **p** is given by

$$\frac{\mathbf{p} - \mathbf{c}}{\|\mathbf{p} - \mathbf{c}\|}$$

  where **c** is the center of the sphere.

- It's possible to compute surface parameterization of a sphere.

  - But we won't do it now.

Modified Sphere Class

```
class Sphere : public Shape
{
public:
    Sphere(
        const Float3 &_center = Float3(0,0,0),
        float _radius = 1.0f,
        Material *_material = NULL);
    virtual ~Sphere();
    virtual bool intersect_p(Ray &ray);
    virtual bool intersect(Ray &ray, ShadeRec &shade_rec);

public:
    Float3 center;
    float radius;
};
```

Modified Sphere Class

```
bool Sphere::intersect( Ray &ray, ShadeRec &shade_rec )
{
    bool hit = intersect_p(ray);

    if (hit)
    {
        shade_rec.point = ray(ray.tmax);
        shade_rec.w_out = -ray.direction;
        shade_rec.time = ray.tmax;

        Float3 normal = normalize(shade_rec.point - center);
        if (dot(normal, ray.direction) <= 0)
        {
            shade_rec.outside = true;
            shade_rec.normal = normal;
        }
        else
        {
            shade_rec.outside = false;
            shade_rec.normal = -normal;
        }
```

Modified Sphere Class

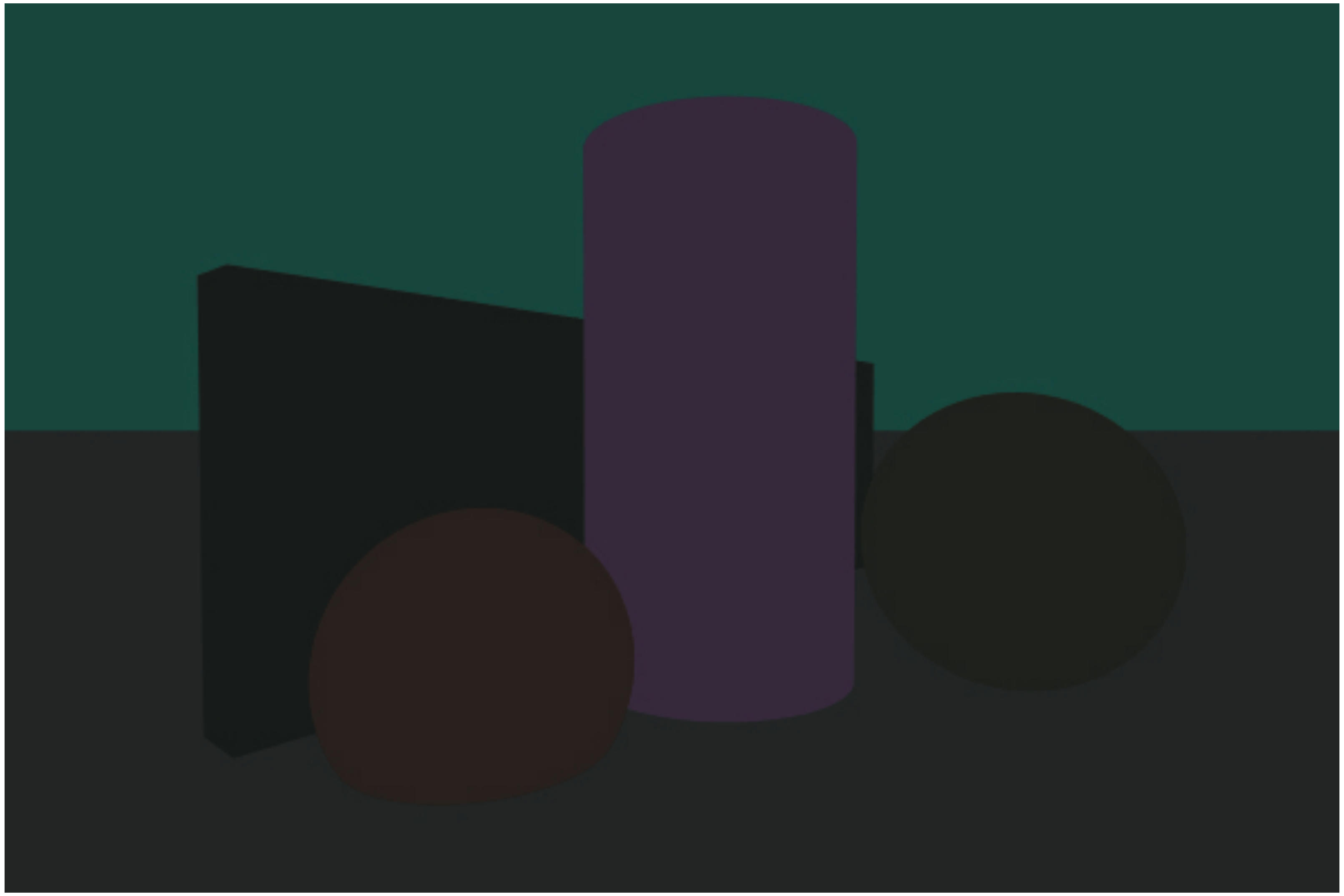
```
        // You cannot put texture on a Sphere yet.
        shade_rec.uv = Float2(0,0);

        return true;
    }
    else
        return false;
}
```

Light Sources

- Ambient light
  - Can store its color as a global variable.
  - Computed once for all pixels.
- Two types of light sources with no area:
  - Point lights
  - Directional lights
- The class two types are encapsulated by Light class.

Ambient Light



Light Class

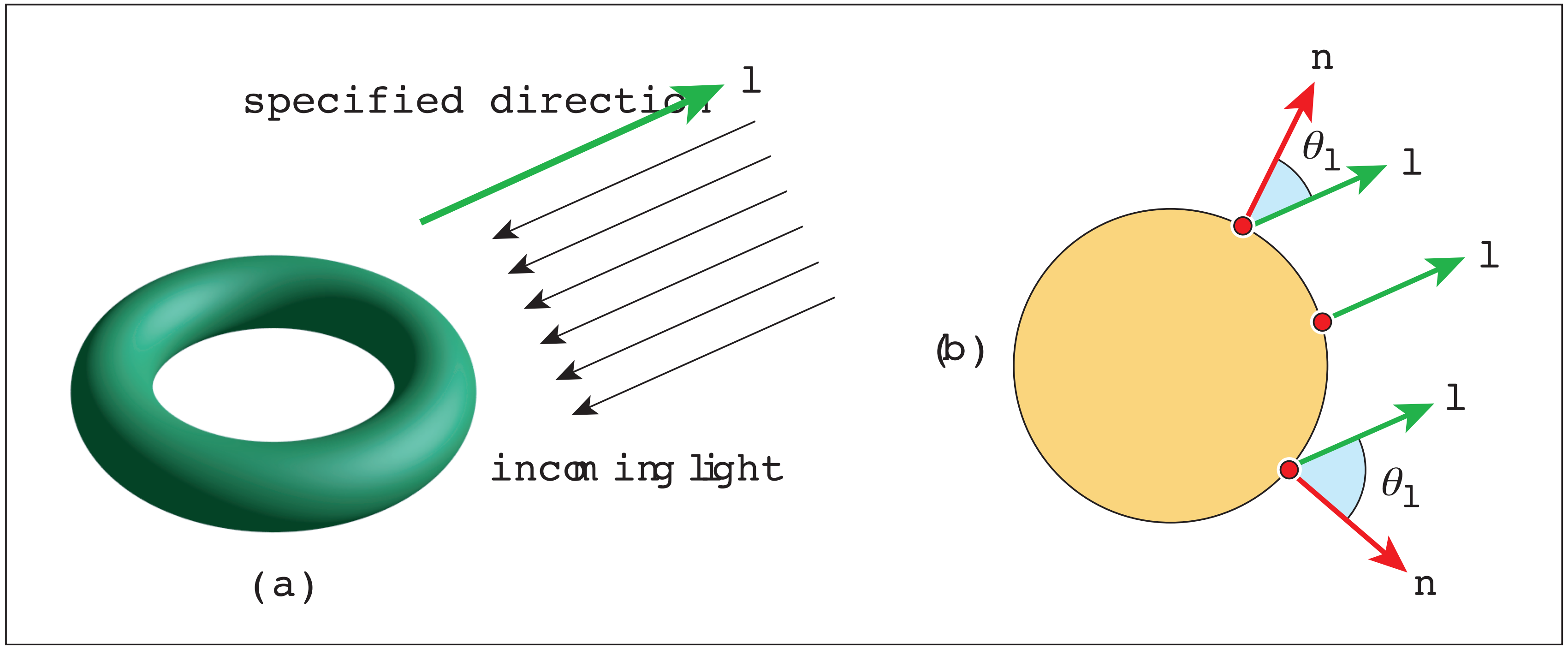
- ```
  class Light
  {
  public:
      Light();
      virtual ~Light();
      virtual void radiance(const Float3 &point, Float3 &wi, Float3 &Li) = 0;
  };
  ```

- void radiance(const Float3 &p, Float3 &wi, Float3 &Li)
  - Compute the light's intensity received by the given point p.
  - Outputs the vector from p to the light in wi
  - Outputs the light intensity in Li.
  ```

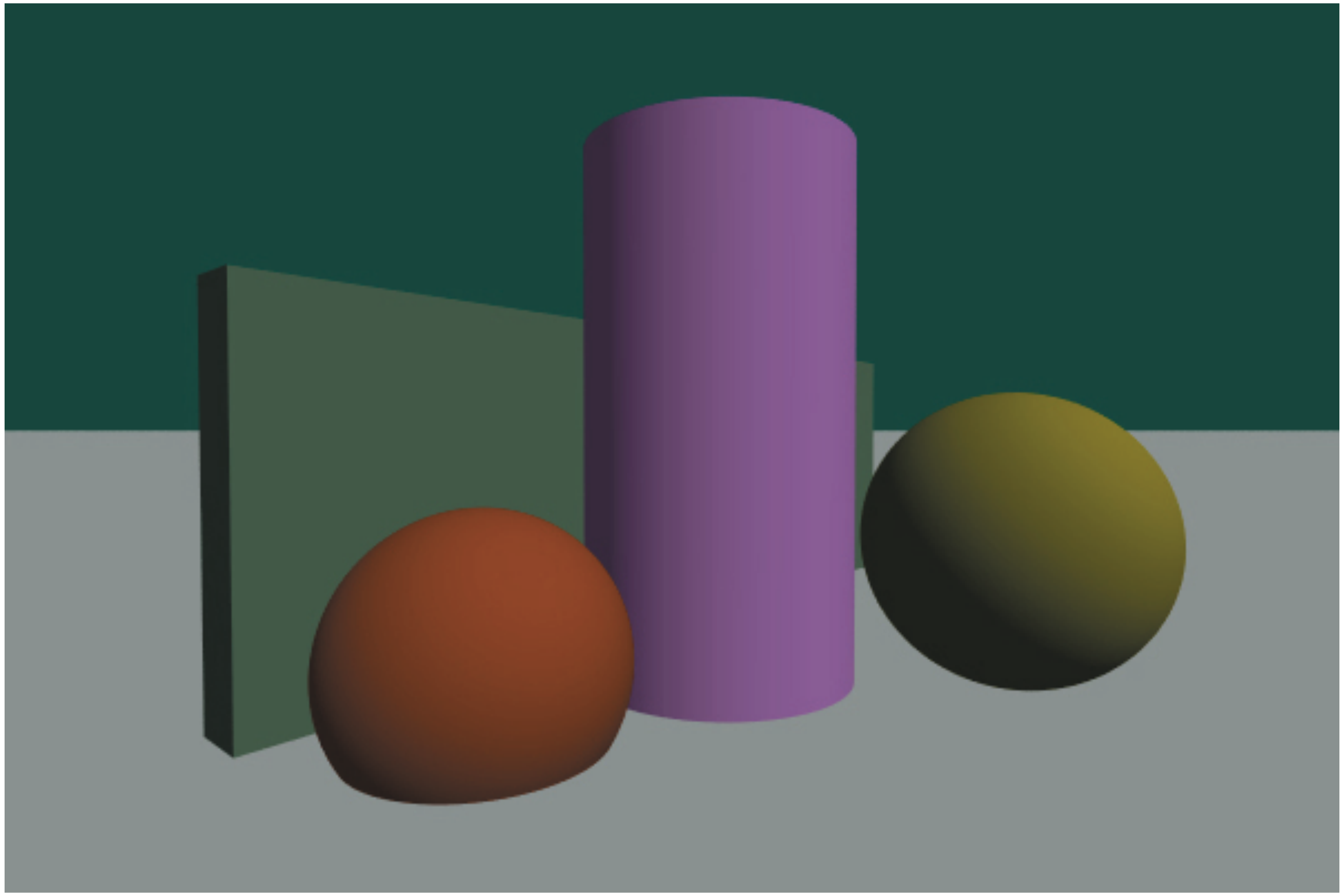
Directional Light

- Energy from a directional light source comes from a single direction.
- Energy to every point is the same.

- Thus, the object representing it needs to store:
  - Direction ω_i of the incoming light.
  - Intensity I_L of the light.



Directional Light



DirectionalLight Class

```
class DirectionalLight : public Light
{
public:
    DirectionalLight(const Float3 &_direction, const ScalableFloat3 &_intensity);
    virtual ~DirectionalLight();
    virtual void radiance(const Float3 &point, Float3 &wi, Float3 &Li);
    virtual Ray gen_shadow_ray(const ShadeRec &shade_rec) const;

public:
    Float3 direction;
    ScalableFloat3 intensity;
};


void DirectionalLight::radiance( const Float3 &point, Float3 &wi, Float3 &Li )
{
    wi = direction;
    Li = intensity;
}
```

Point Light

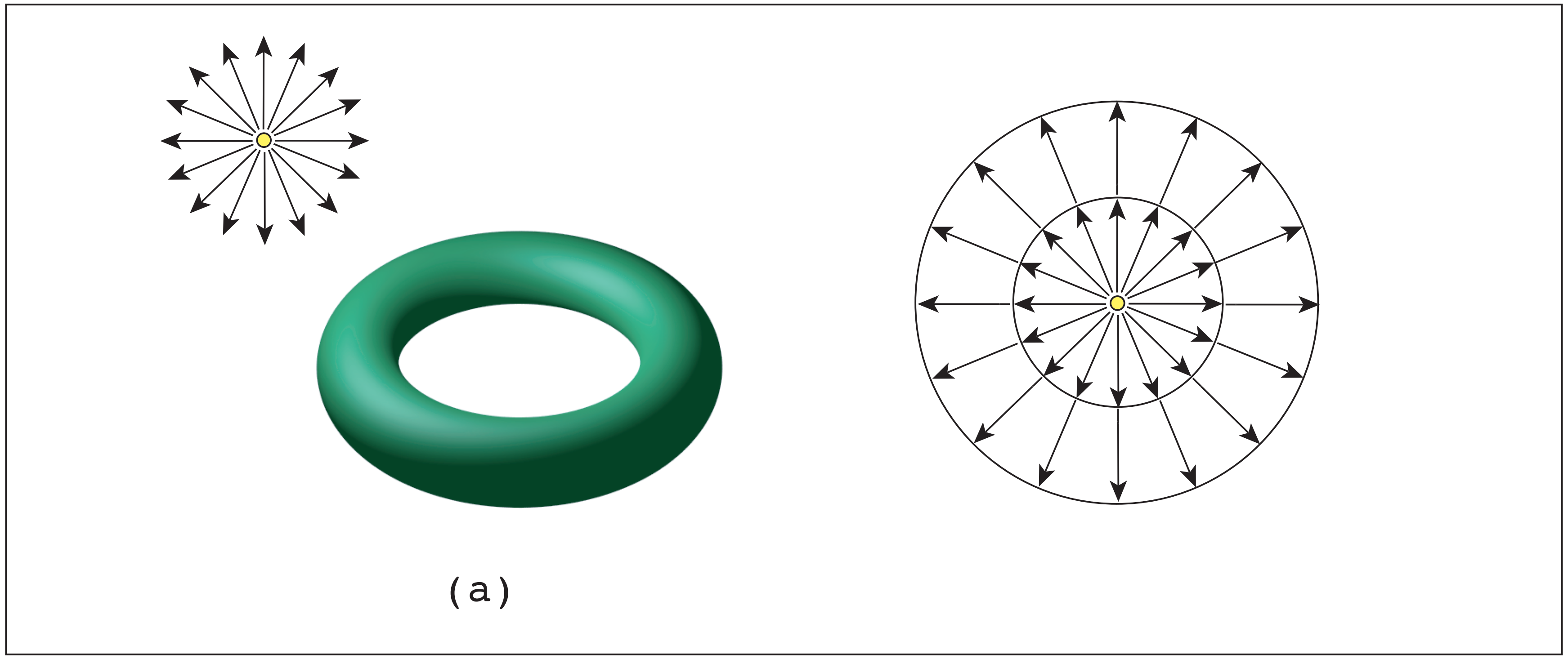
- A point light distribute all of its energy equally in all direction.
- Suppose the light has intensity I.
  The intensity at the point of distance r from the point's location is:

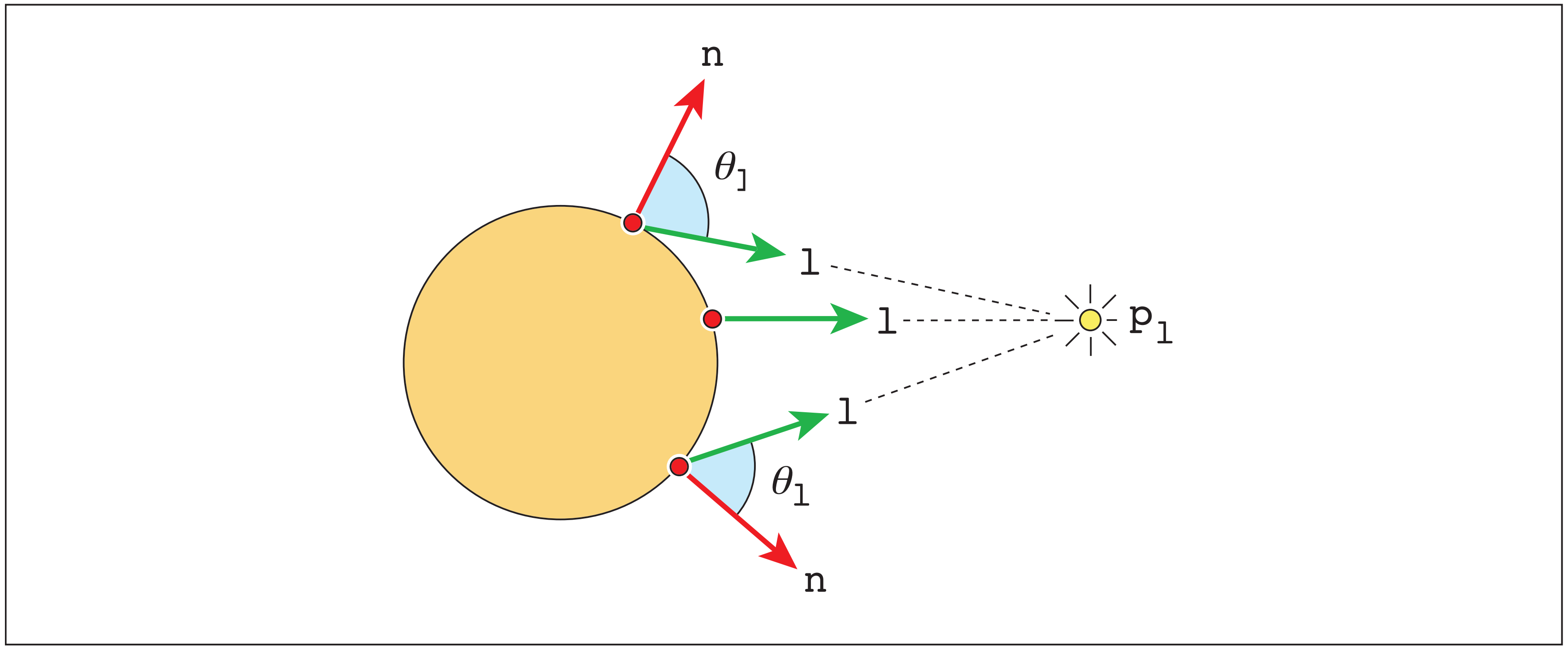
$$\frac{I}{4\pi r^2}$$

- In other words, if the point being shade is **p** and the light is at point **x**.
  Then the intensity the point receives is

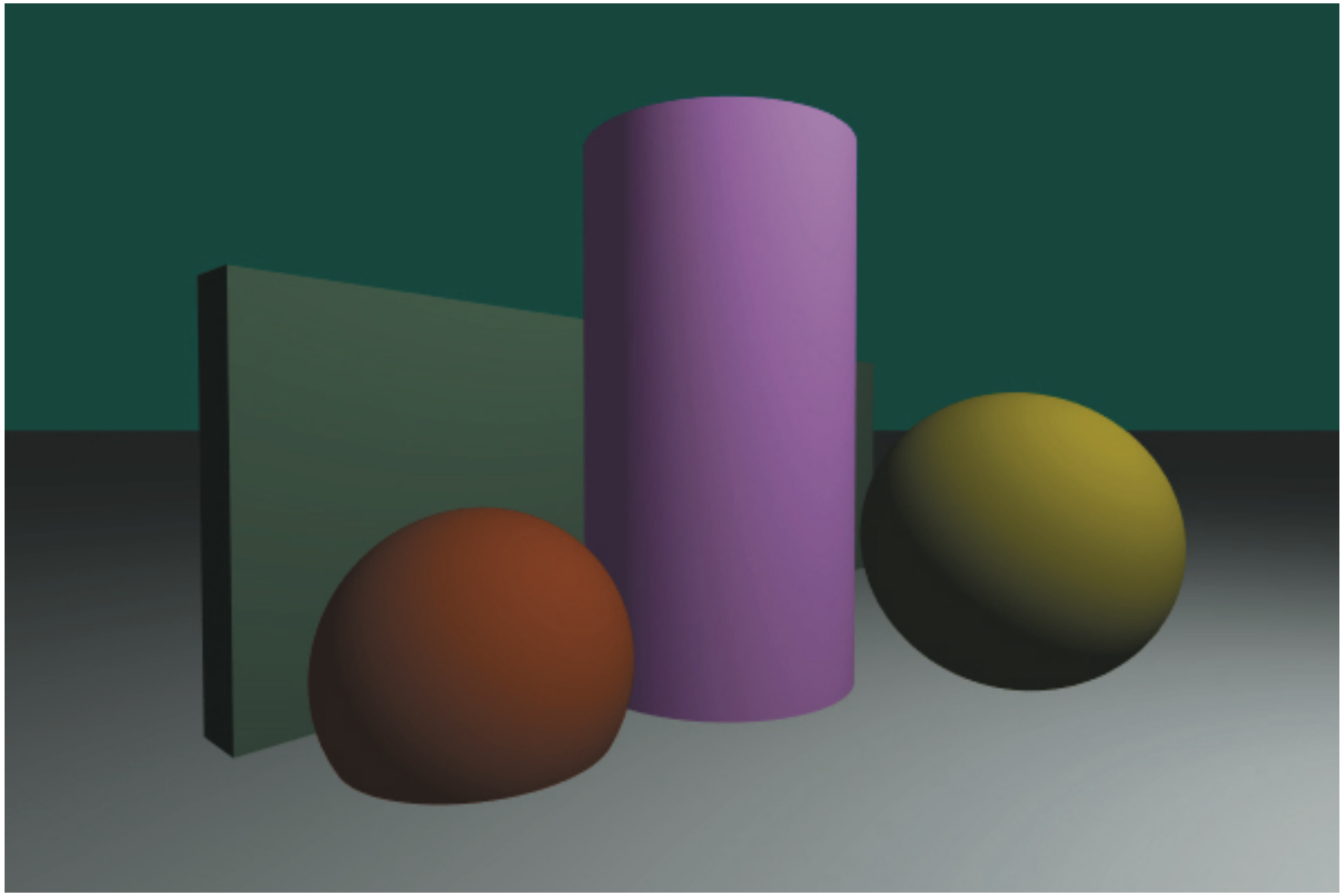
$$\frac{I}{4\pi \|\mathbf{p} - \mathbf{x}\|^2}$$

Point Light

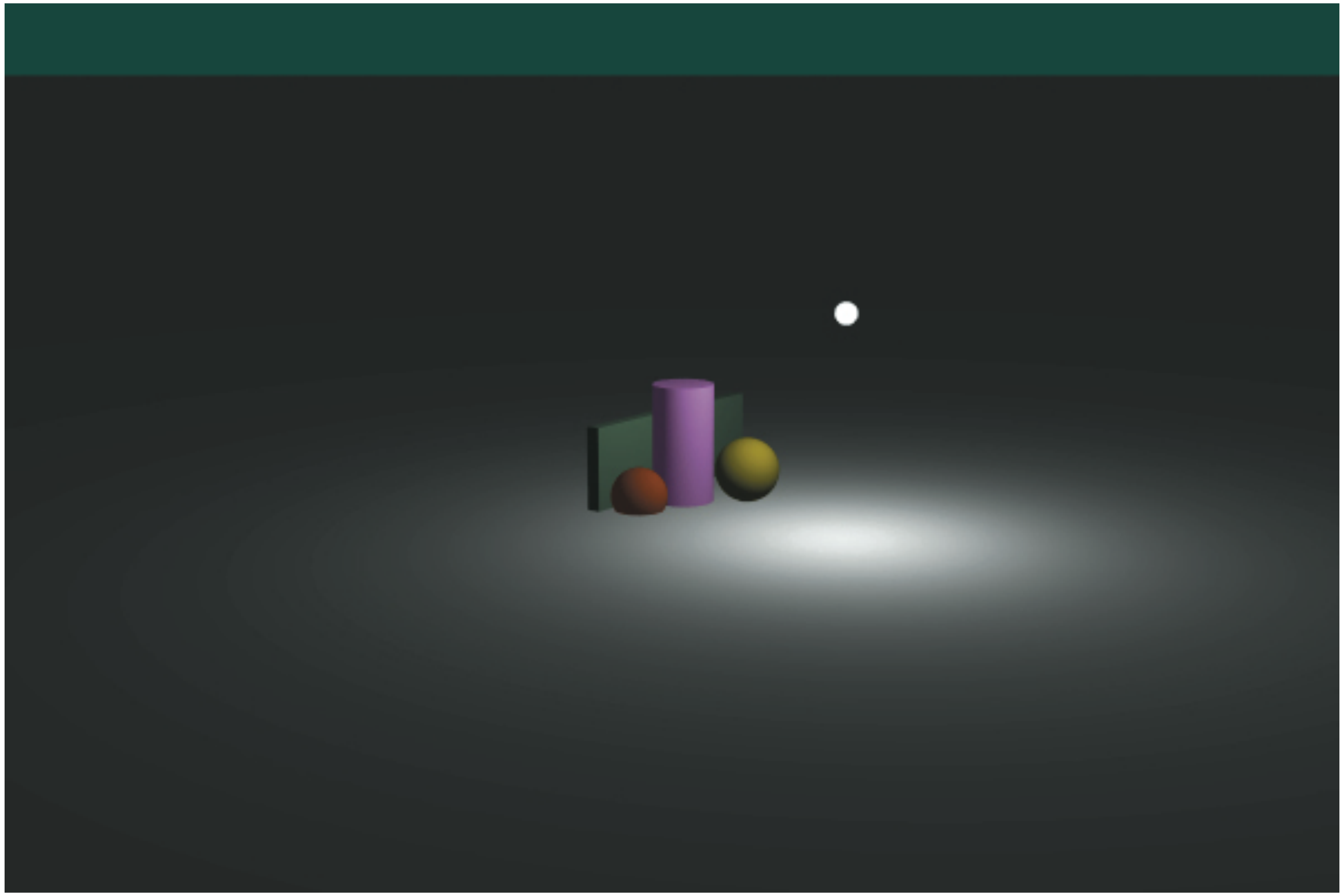




Point Light



Point Light



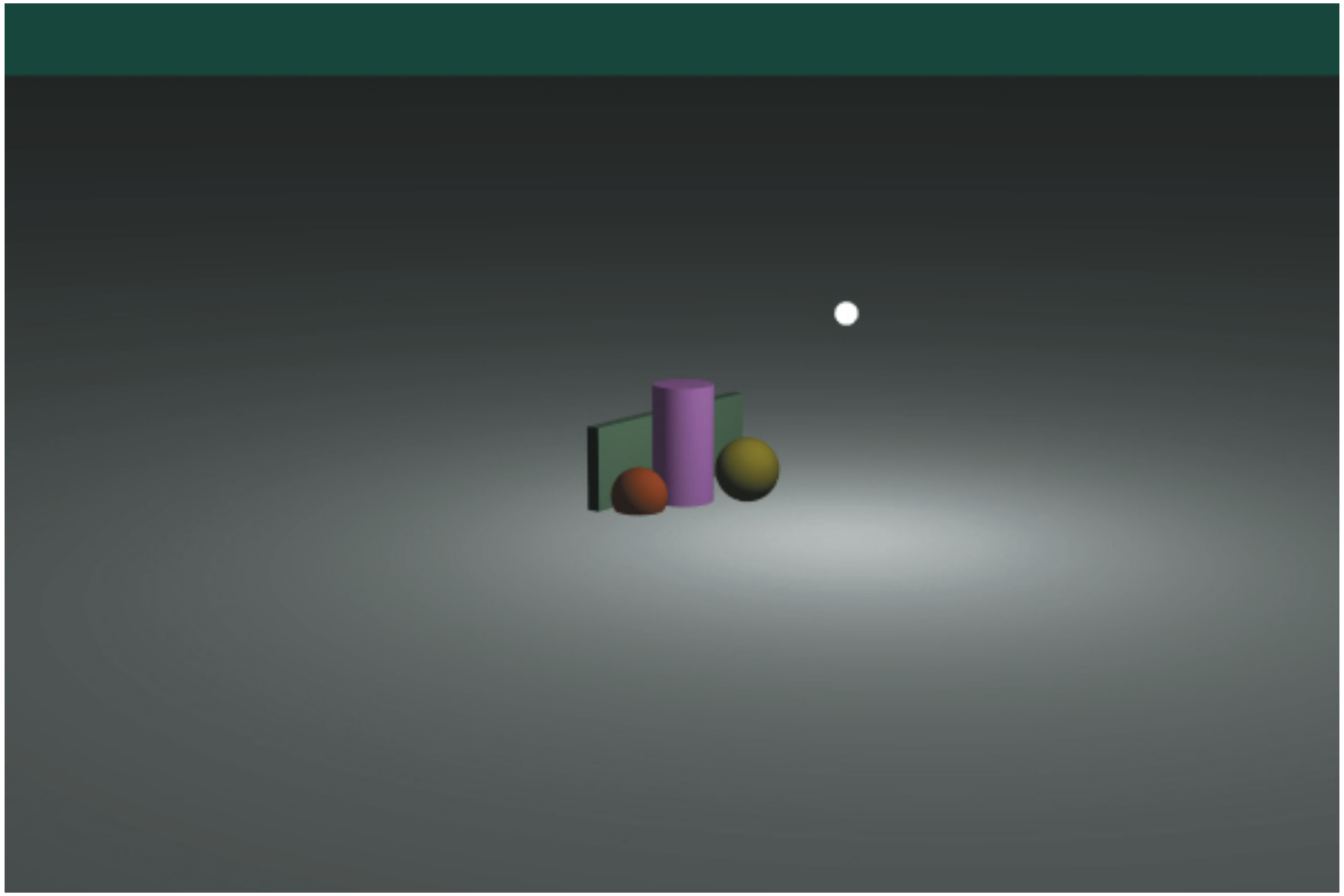
Point Light

- Notice that the light gets dimmer as the shaded point gets far away from it.
  - This is called **distance attenuation**.

- Problems with distance attenuation.
  - Light gets very dim very quickly.
  - Light overflows when shaded point is near the light’s position.

- For this lecture (and some in the future), we will avoid the above problems by saying that the intensity is constant on all receiving points.

Point Light without Distance Attenuation



PointLight Class

```
class PointLight : public Light
{
public:
    PointLight(const Float3 &_position, const ScalableFloat3 &_intensity);
    virtual ~PointLight();
    virtual void radiance(const Float3 &point, Float3 &wi, Float3 &Li);
    virtual Ray gen_shadow_ray(const ShadeRec &shade_rec) const;

public:
    Float3 position;
    ScalableFloat3 intensity;
};

void PointLight::radiance( const Float3 &point, Float3 &wi, Float3 &Li )
{
    Li = intensity.value();
    wi = normalize(position - point);
}
```

Material Class

- Used to model appearance of shapes.
  - That is, specifies how it interact with light.
- Three types of interaction with light.
  - Emitting light itself.
  - Reflecting ambient light.
  - Reflecting light from light sources (so called “direct illumination”).

Material Class

```
class Material
{
public:
    Material(const std::string &_name = "");
    virtual ~Material();

    virtual Float3 shade_emit(
        const ShadeRec &shade_rec) const;

    virtual Float3 shade_ambient(
        const Float3 &ambient_light,
        const ShadeRec &shade_rec) const;

    virtual Float3 shade_direct(
        const Float3 &wi, const Float3 &Li,
        const ShadeRec &shade_rec) const;

public:
    std::string name;
};
```

Types of Material We'll Implement

- Matte
  - Only ambient and diffuse components.
- Phong
  - Like Matte, but with specular component added.
- Checker
  - Used to represent surfaces with check pattern.
  - Specify three materials:
    - One for odd cells, one for even cells, and one for borders.

Matte Class

- Three attribute to store:
  - Color of light it emits (emission)
  - Ambient color (ambient)
  - Diffuse color (diffuse)

- ```
  class Matte : public Material
  {
  public:
      Matte( ... );
      virtual ~Matte ();

      /* shade_XXX functions go here */

  public:
      ScalableFloat3 ambient;
      ScalableFloat3 diffuse;
      ScalableFloat3 emission;
  };
  ```
  ```

Matte Class: shade_emit

- Just return the emission color.

- ```
  Float3 Matte::shade_emit(const ShadeRec &shade_rec) const
  {
      return emission.value();
  }
  ```
  ```

Matte Class: shade_ambient

- Multiply the ambient light with the ambient color.

-
```
Float3 Matte::shade_ambient(
    const Float3 &ambient_light,
    const ShadeRec &shade_rec) const
{
    return ambient_light * ambient.value();
}
```

Matte Class: shade_direct

- Take the normal and dot it with the light direction.
  - If the dot product is less than zero, make it zero. (VERY IMPORTANT)
- Then multiply the dot product with the diffuse color and the light intensity.
  - The reason why we divide by Pi will be given in the next two lectures.
- Formula

$$L_r = \frac{\rho}{\pi}(\mathbf{n} \cdot \omega_i) L_i$$

where

L_r is the intensity of the light reflected

ρ is the albedo (think of it as a diffuse color of some sort)

L_i is the incoming light intensity

Matte Class: shade_direct

- ```
  Float3 Matte::shade_direct(
      const Float3 &wi, const Float3 &Li,
      const ShadeRec &shade_rec) const
  {
      float cos_theta = MAX(dot(shade_rec.normal, wi), 0.0f);
      return diffuse.value() * Li * (cos_theta / M_PI);
  }
  ```
  ```

Phong Class

- Very similar to Matte.
- Store two more attributes.
  - Specular color (specular)
  - Shininess (shininess)
- ```
  class Phong : public Material
  {
  public:
      /* shade_XXX methods, constructer, destructor go here */

      ScalableFloat3 ambient;
      ScalableFloat3 diffuse;
      ScalableFloat3 emission;
      ScalableFloat3 specular;
      float shininess;
  };
  ```
  ```

Phong Class

- shade_emit and shade_ambient are the same as those of Matte

- shade_direct needs to compute specular term.

- Formula:

$$L_r = \left[\frac{\rho}{\pi} + k_s(\mathbf{r} \cdot \omega_o)\right](\mathbf{n} \cdot \omega_i)L_i$$

  where
  k_s is the specular color
  $\mathbf{r}$ is the incoming light direction reflected around the normal
  α is the shininess

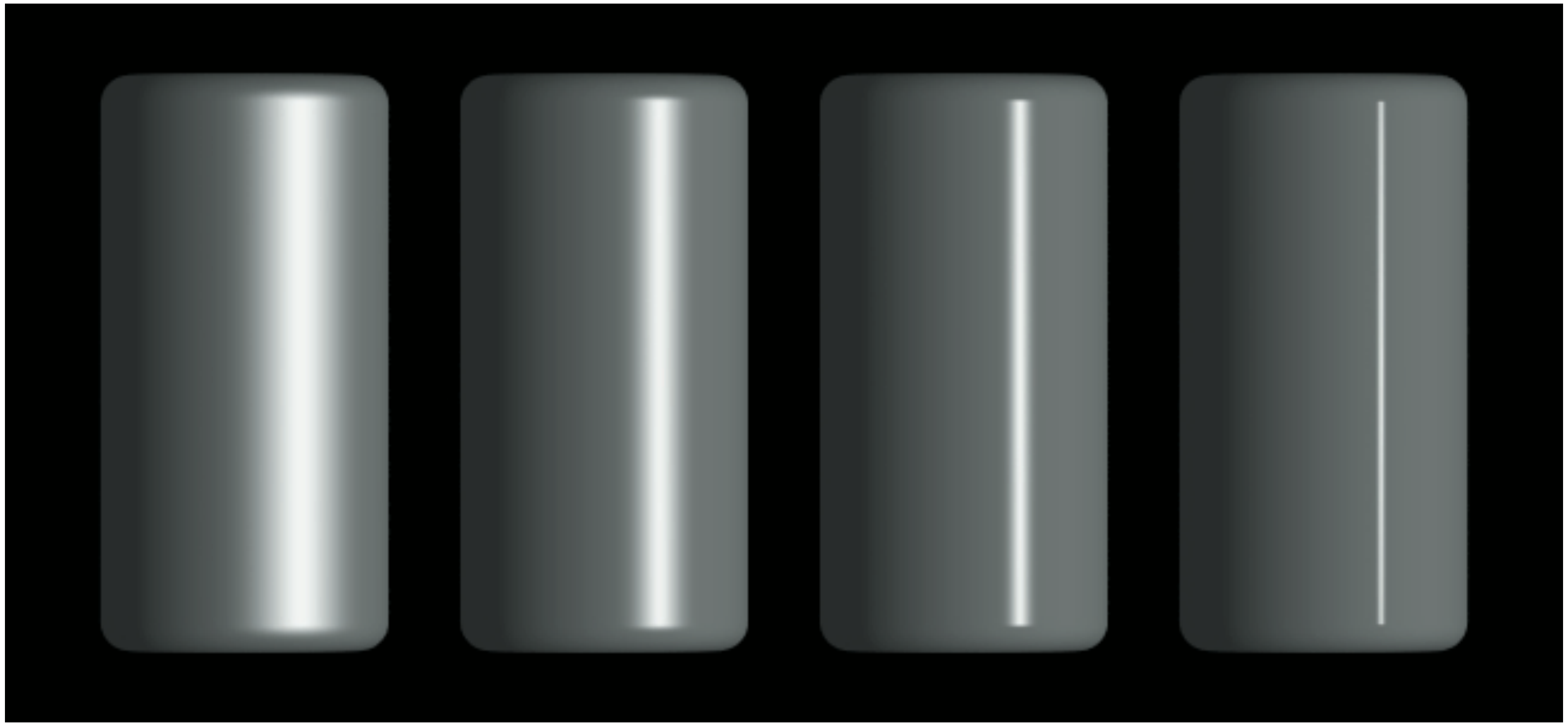
- How to compute **r**?

$$\mathbf{r} = 2(\mathbf{n} \cdot \omega_i)\mathbf{n} - \omega_i$$

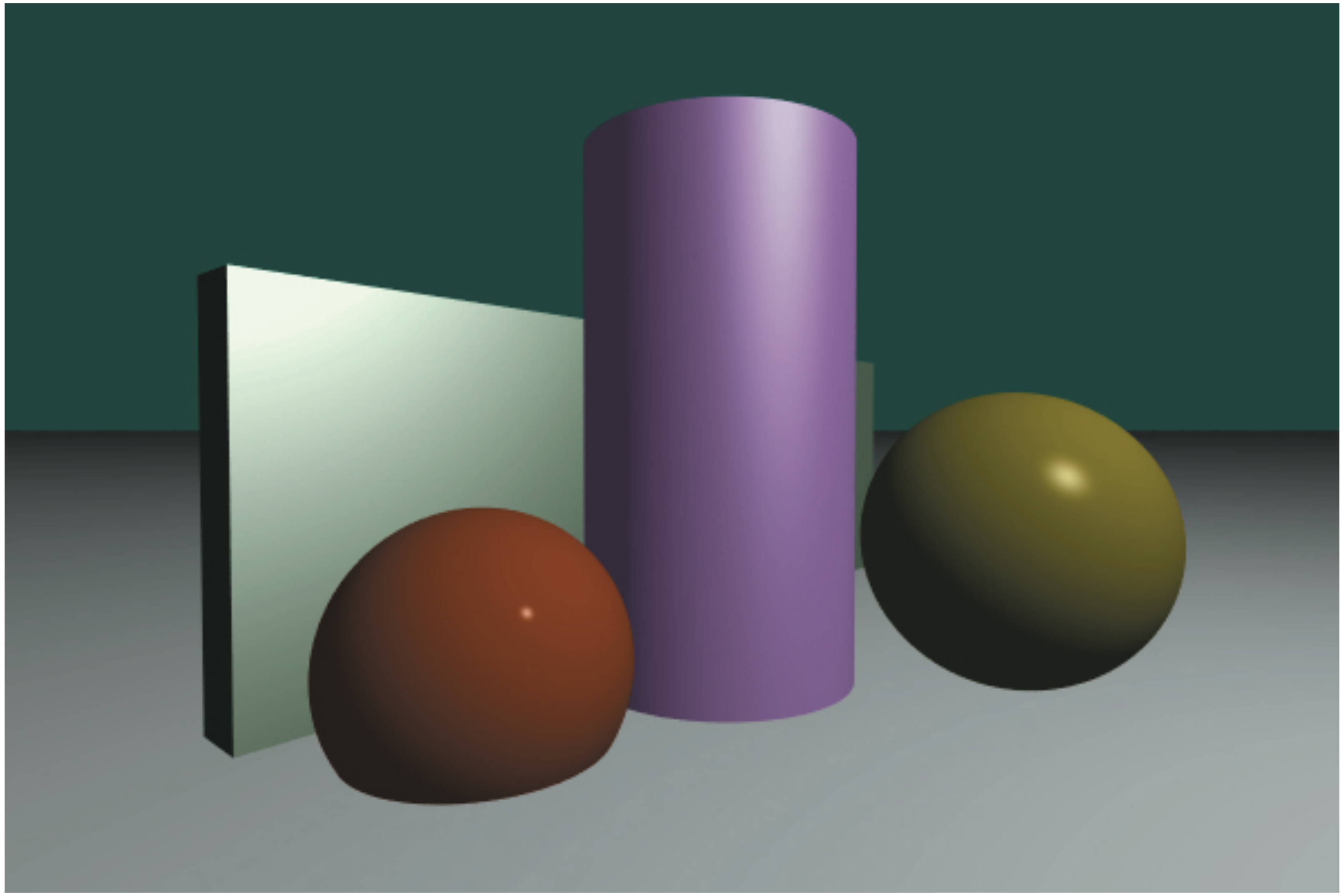
Phong Class: shade_direct

```
Float3 Phong::shade_direct( const Float3 &wi, const Float3 &Li,
    const ShadeRec &shade_rec ) const
{
    Float3 n = shade_rec.normal;
    Float3 wo = shade_rec.w_out;
    float cos_theta = MAX(0.0f, dot(n, wi));
    Float3 diffuse_color = diffuse.value() * Li * (cos_theta / M_PI);
    Float3 specular_color(0,0,0);
    if (dot(wi, n) > 0)
    {
        Float3 r = -wi + 2 * dot(wi,n) * n;
        float cos_alpha = fabsf(dot(wo, r));
        if (cos_alpha < 0) cos_alpha = 0;
        float c_specular = (float)pow(cos_alpha, shininess);
        specular_color = specular.value() * (c_specular * cos_theta) * Li;
    }
    return diffuse_color + specular_color;
}
```

Phong Class



Phong Class



Checker Class

- Contains:
  - Pointers to three materials
    - One for odd cells, one for even cells, and one for outlines
  - Cell size
  - Outline width
- Determine which material to used based on the UV parameterization in the ShadeRec.
- Delegate all the methods to the material at that shading point.

Checker Class

```
class Checker : public Material
{
public:
    Checker( ... );
    virtual ~Checker();

    /* shade_XXX methods go here */

public:
    float size;
    float outline_width;
    Material *even_material;
    Material *odd_material;
    Material *outline_material;
};
```

Checker Class: Determining Material to Use

```
Material * Checker::get_point_material( Float2 tex_coord ) const
{
    int qs = int(floorf(tex_coord.s / size));
    int qt = int(floorf(tex_coord.t / size));
    float rs = tex_coord.s - size * qs;
    float rt = tex_coord.t - size * qt;

    float w = outline_width / 2;
    if (outline_width > 0 &&
        (rs < w || rs > size - w ||
         rt < w || rt > size - w))
        return outline_material;
    else
    {
        if ((qs + qt) % 2 == 0)
            return even_material;
        else
            return odd_material;
    }
}
```

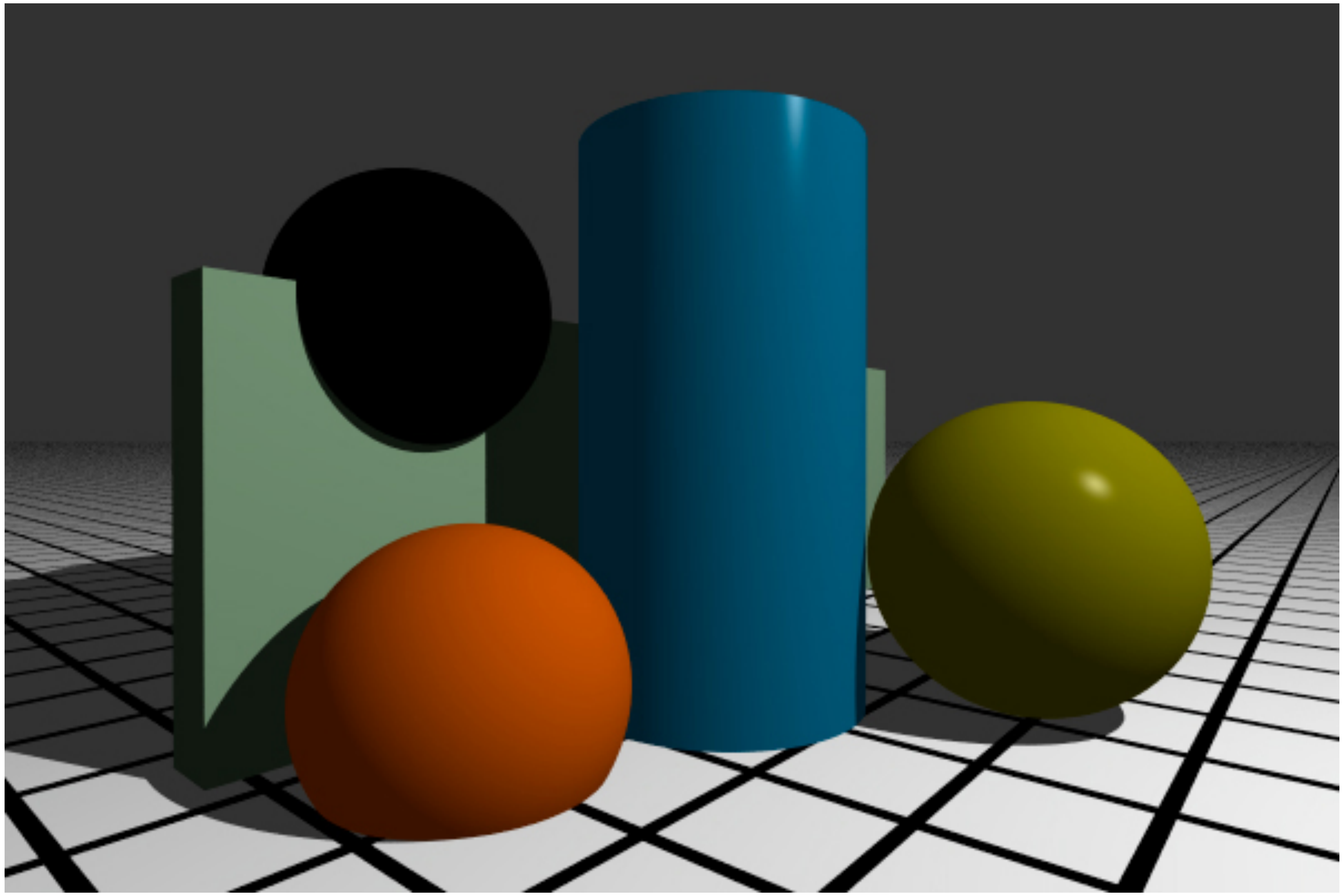
Check Class: shade_XXX methods

```
Float3 Checker::shade_emit( const ShadeRec &shade_rec ) const
{
    return get_point_material(shade_rec.uv)->shade_emit(shade_rec);
}

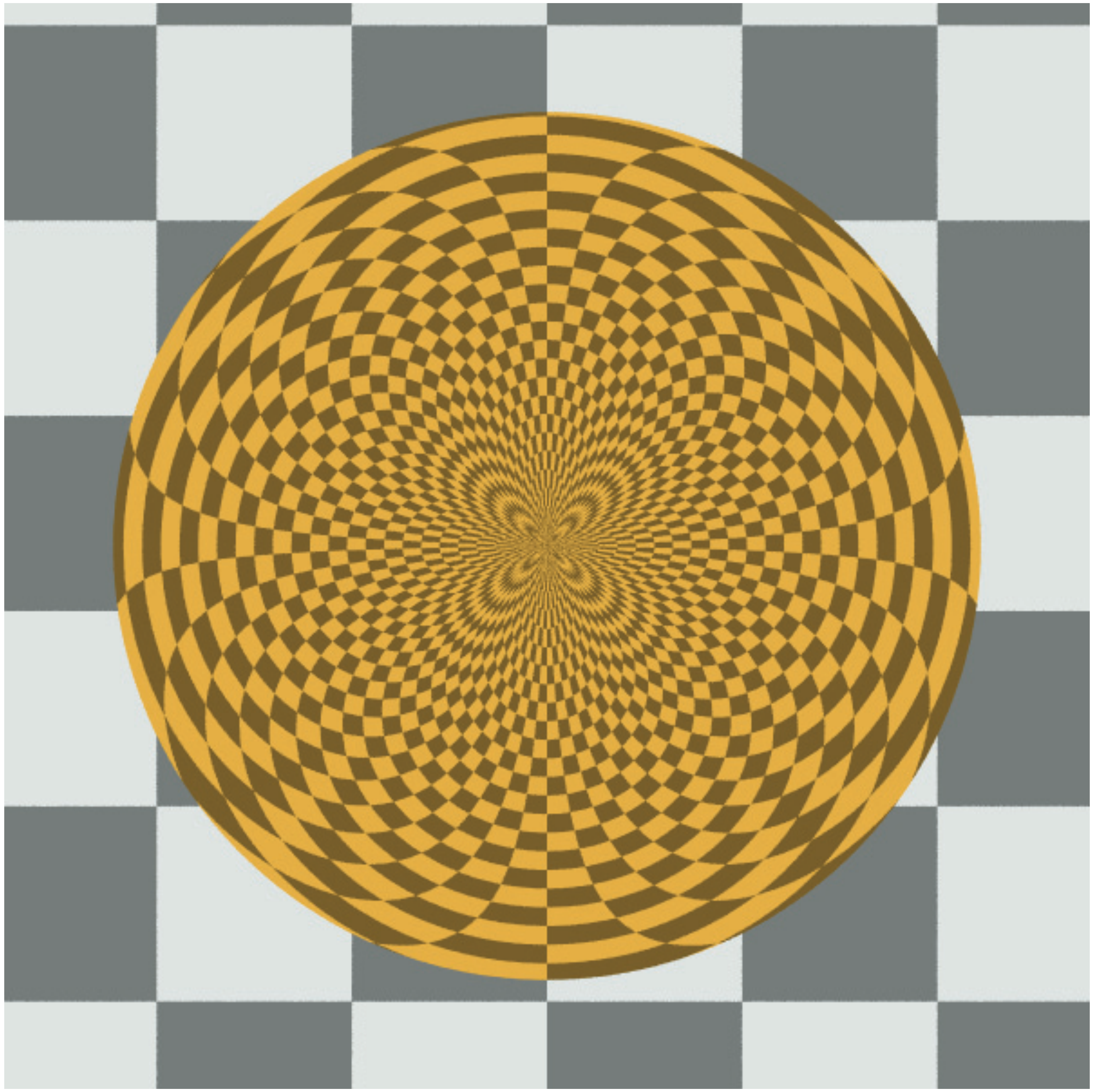
Float3 Checker::shade_ambient( const Float3 &ambient_light,
    const ShadeRec &shade_rec ) const
{
    return get_point_material(shade_rec.uv)->shade_ambient(
        ambient_light, shade_rec);
}

Float3 Checker::shade_direct( const Float3 &wi, const Float3 &Li,
    const ShadeRec &shade_rec ) const
{
    return get_point_material(shade_rec.uv)->shade_direct(wi, Li, shade_rec);
}
```

Checker Class



Checker Class



Putting It All Togther

- For each primary eye ray, we have to:
  - Locate the nearest hit point by.
  - Compute information at the hit point that is needed to shade it.
  - Retrieve the nearest shape’s material.
  - Compute the emissive and ambient component.
  - Iterate over all light sources.
    - Compute the incoming light.
    - Compute the direct illumination component.

Putting It All Together

- For each primary eye ray, we have to:
  - Call **intersect_p** of every shape.
  - Call the nearest shape's **intersect** to compute the ShadeRec.
  - Retrieve the nearest shape's material.
  - Call the material' **shade_emit** and **shade_ambient**.
  - Iterate over all light sources.
    - Call the light's **radiance** to compute incoming light.
    - Call the material's **shade_direct** for compute the light's contribution

Putting It All Together

```
FOR(iy, image_height)
FOR(ix, image_width)
{
    float sx = 2 * (ix + 0.5f) / rr->image_width - 1;
    float sy = 2 * (iy + 0.5f) / rr->image_height - 1;
    Ray ray = camera->gen_ray(sx, sy);

    Shape *hitted_shape = NULL;
    FOR(shape_index, shape_count)
    {
        Shape *shape = shapes[shape_index];
        if (shape->intersect_p(ray))
            hitted_shape = shape;
    }
```

Putting It All Together

```
    Float3 color(0,0,0);
    if (hitted_shape != NULL)
    {
        ray.tmax = INFINITY;
        ShadeRec shade_rec;
        hitted_shape->intersect(ray, shade_rec);
        Material *material = hitted_shape->material;

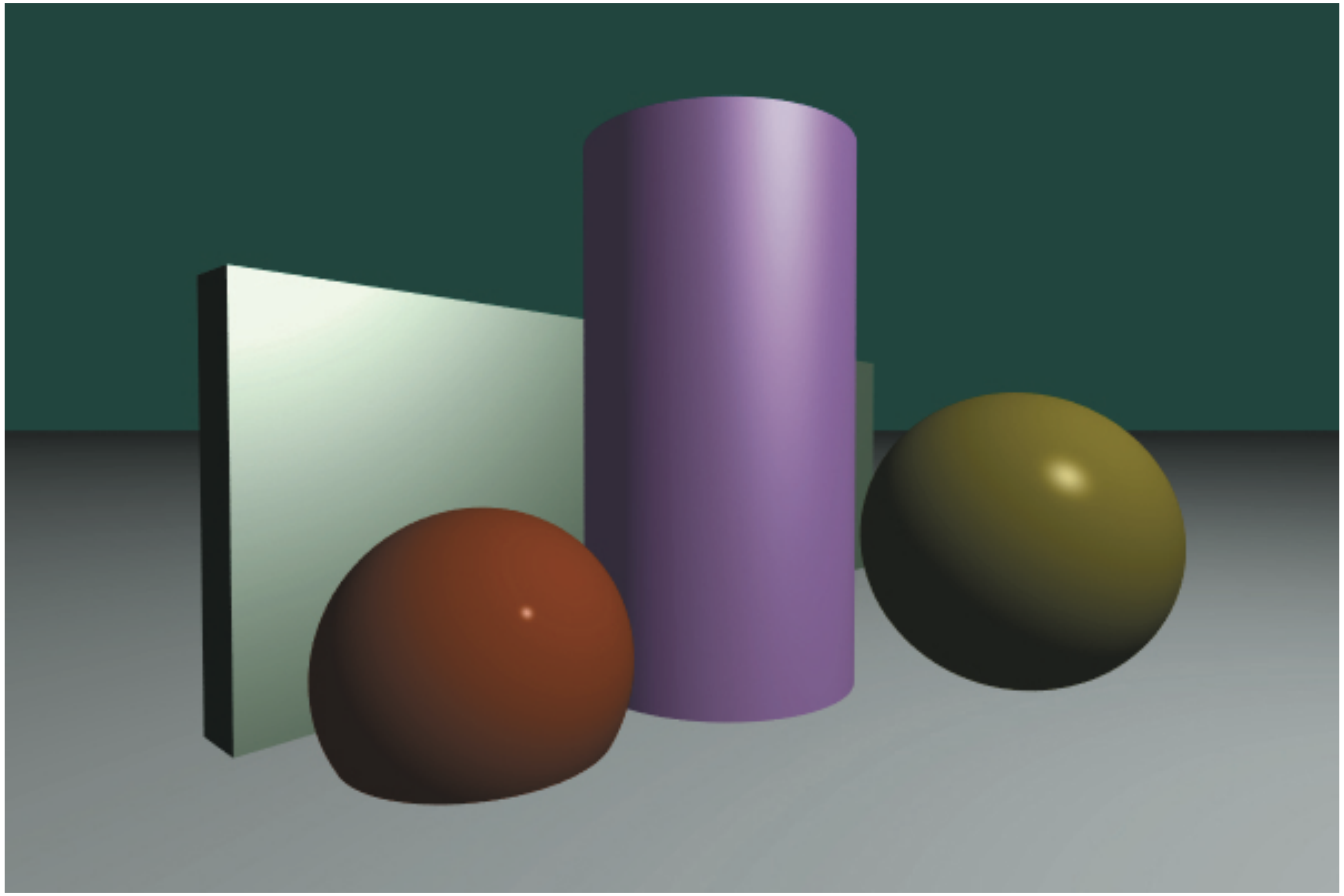
        color += material->shade_emit(shade_rec);
        color += material->shade_ambient(ambient_light, shade_rec);

        FOR(light_index, light_count)
        {
            Light *light = lights[light_index];
            Float3 Li, wi;
            light->radiance(shade_rec.point, wi, Li);
            color += material->shade_direct(wi, Li, shade_rec);
        }
    }
    else
        color = rr->background_color.value();

    texture->set_pixel(ix, iy, Float4(color,1));
}
```

Shadow

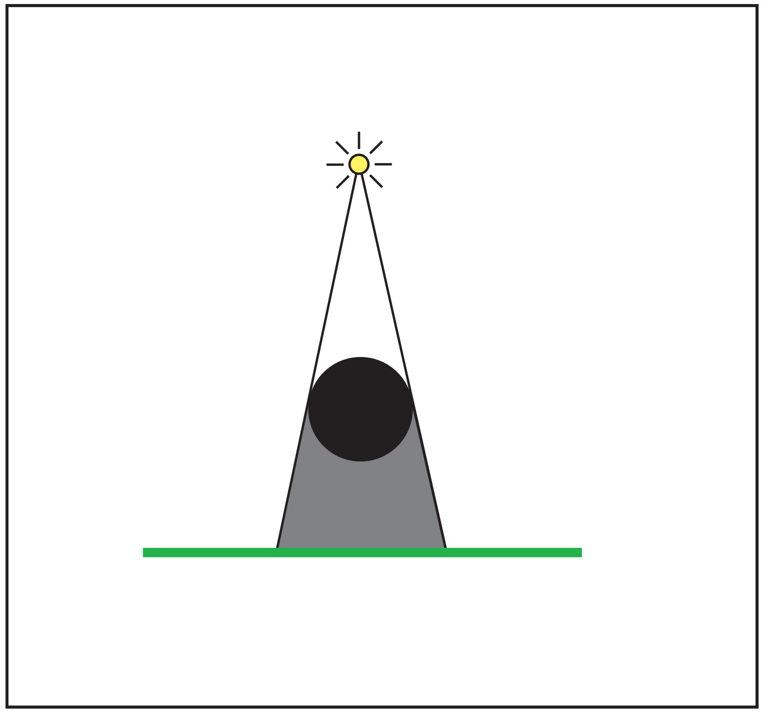
What we have done so far...

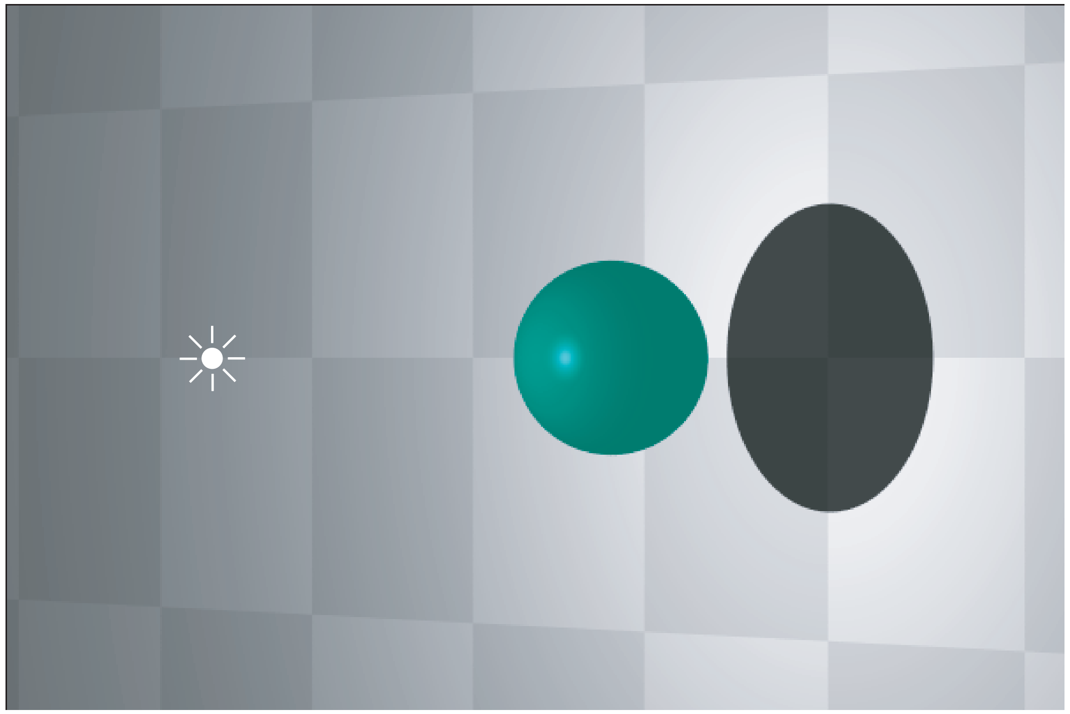


NO SHADOWS!!!

What is a shadow?

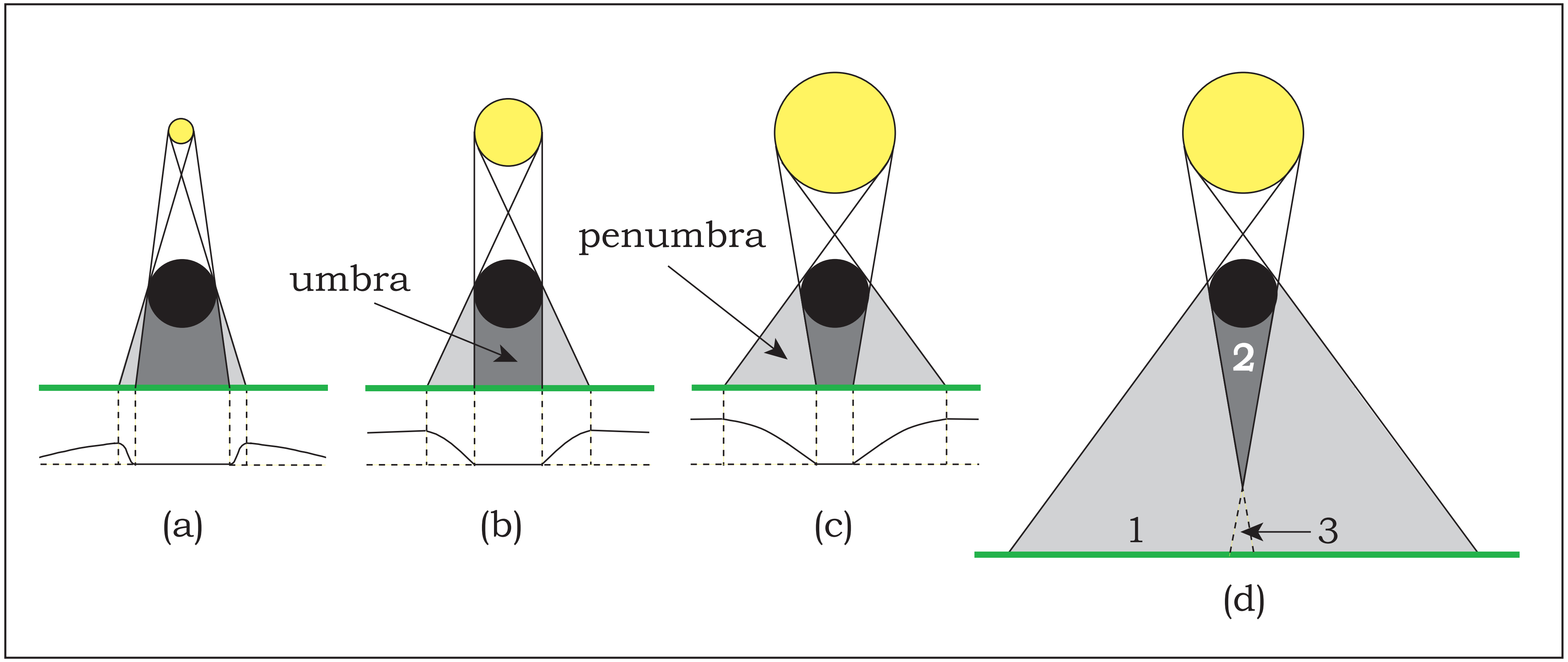
- A shadow is an area that doesn't receive light energy.
- If you stand in that area, you cannot see the light source.



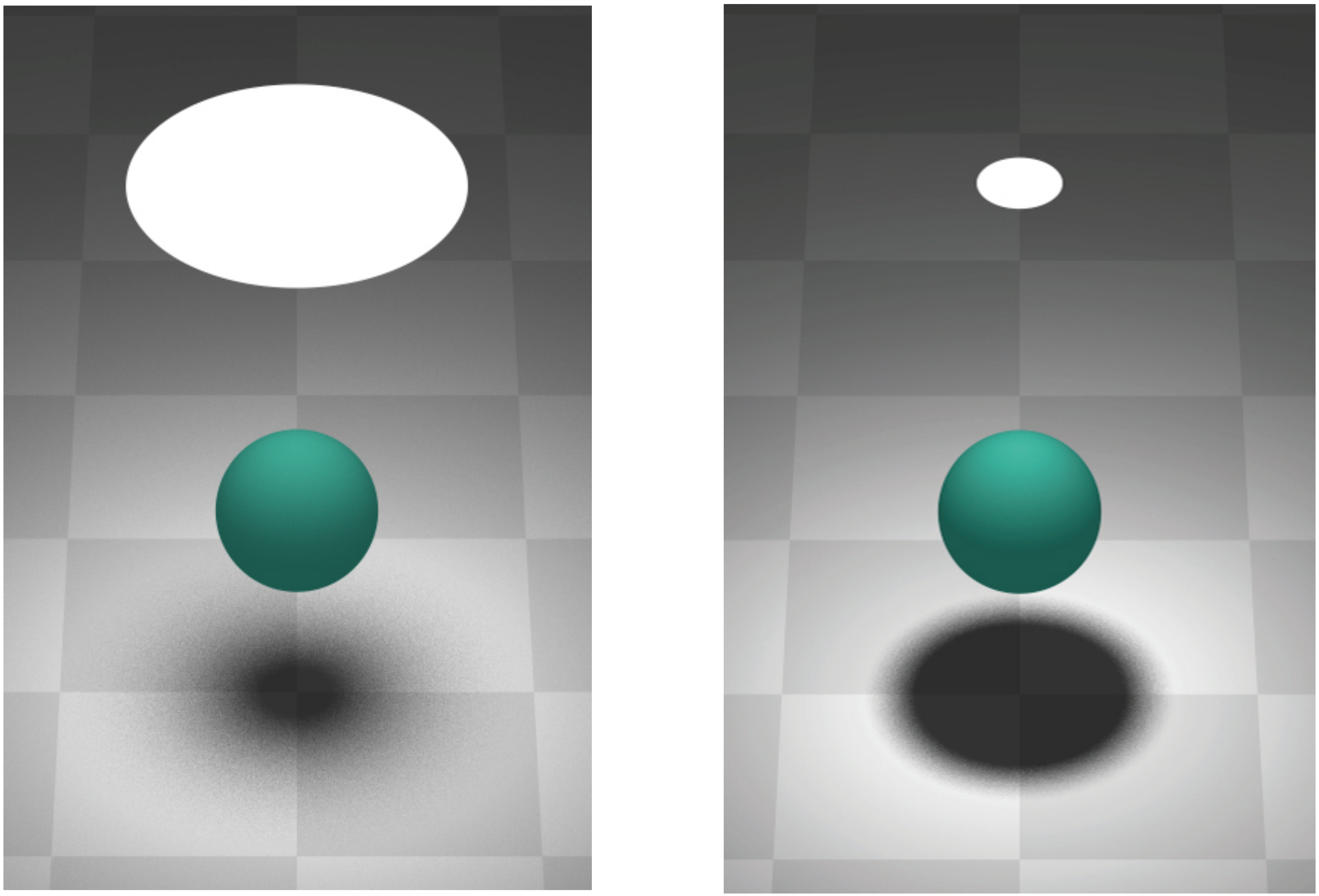


Umbra and Penumbra

- However, if the light has area, things work differently.
- Two types of shadows:
  - Umbra = Completely dark shadow. Light fully occluded.
  - Penumbra = Not so dark shadow. Light partially occluded.



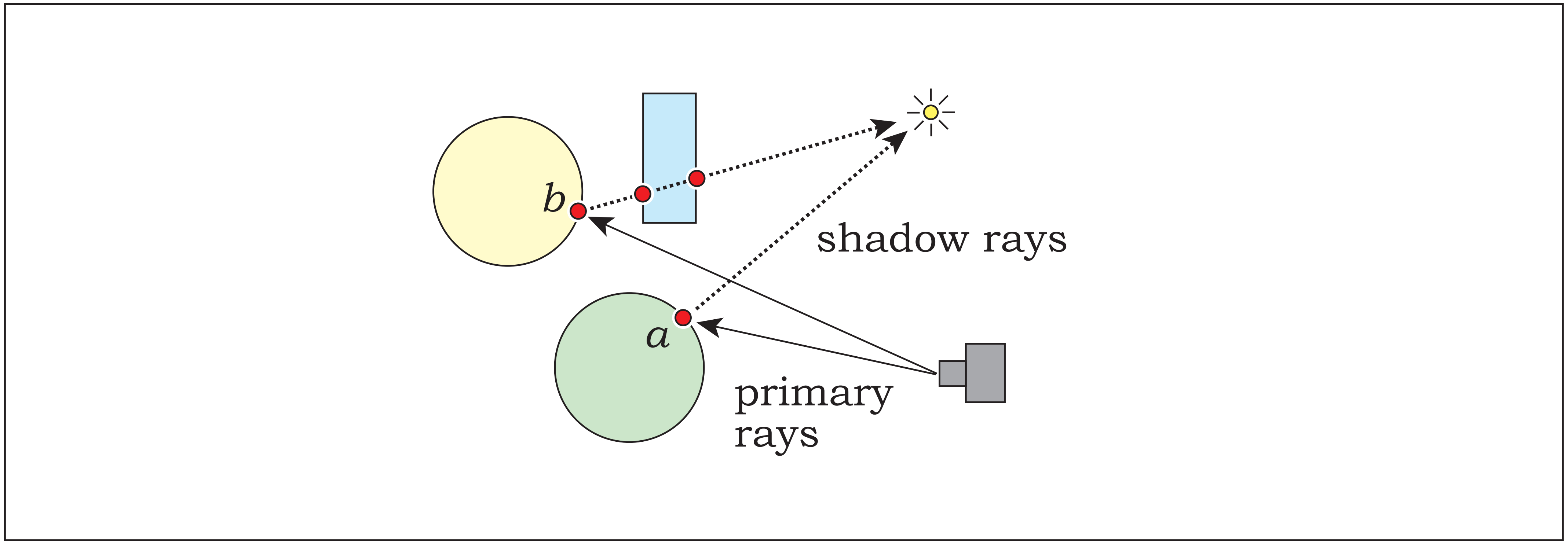
Umbra and Penumbra



Shadow Ray

- In this lecture, we only work with point or directional light.
  - Since they don't have area, they are either completely occluded or completely visible.
- The ray from any point in occluded area to the light source ***must hit something before it reaches the light***.
- For any pixel, we can check whether the pixel receives light energy by ***casting a ray from that point towards the light source*** and ***checking whether it hits something before the light source***.
- Such a ray is called a **shadow ray**.

Shadow Ray

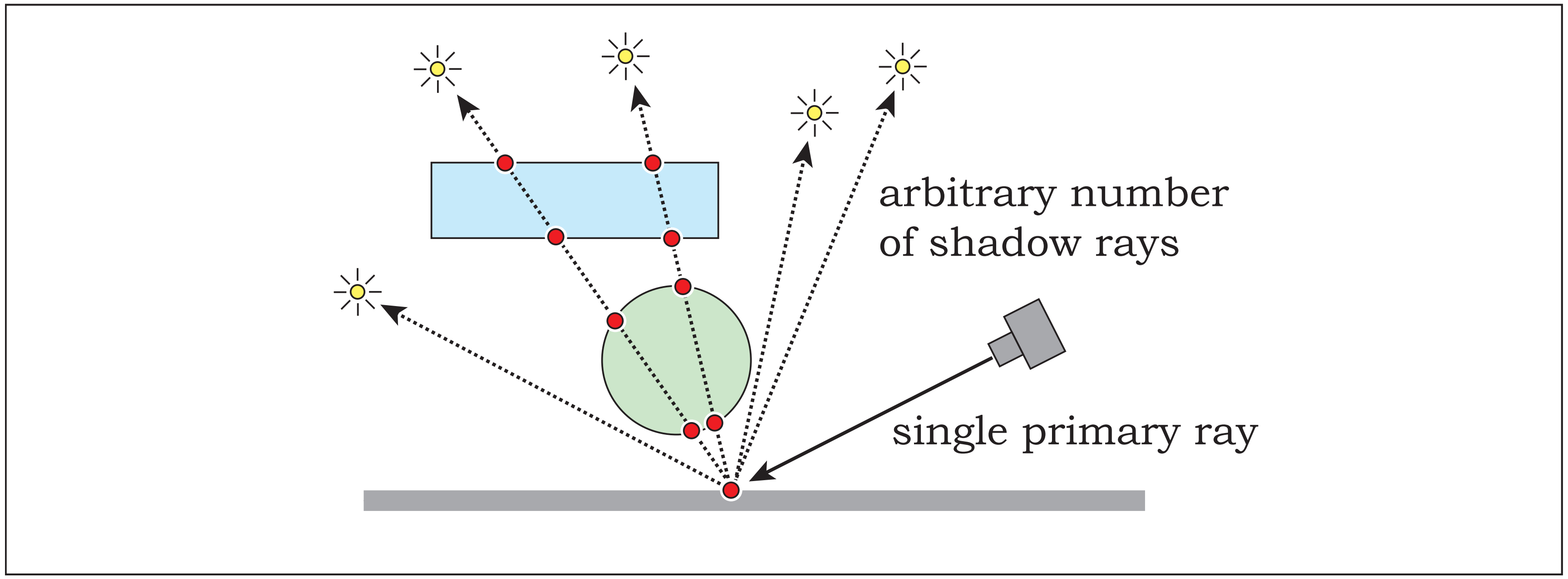


Shadow Ray

- For each pixel to shade, we shoot a primary ray.

- For each pixel to shade,
  for each light source,
  we cast one shadow ray to determine if the pixel sees the light,
  and add the light contribution if so.

- Difficulty: more light sources = more shadow rays



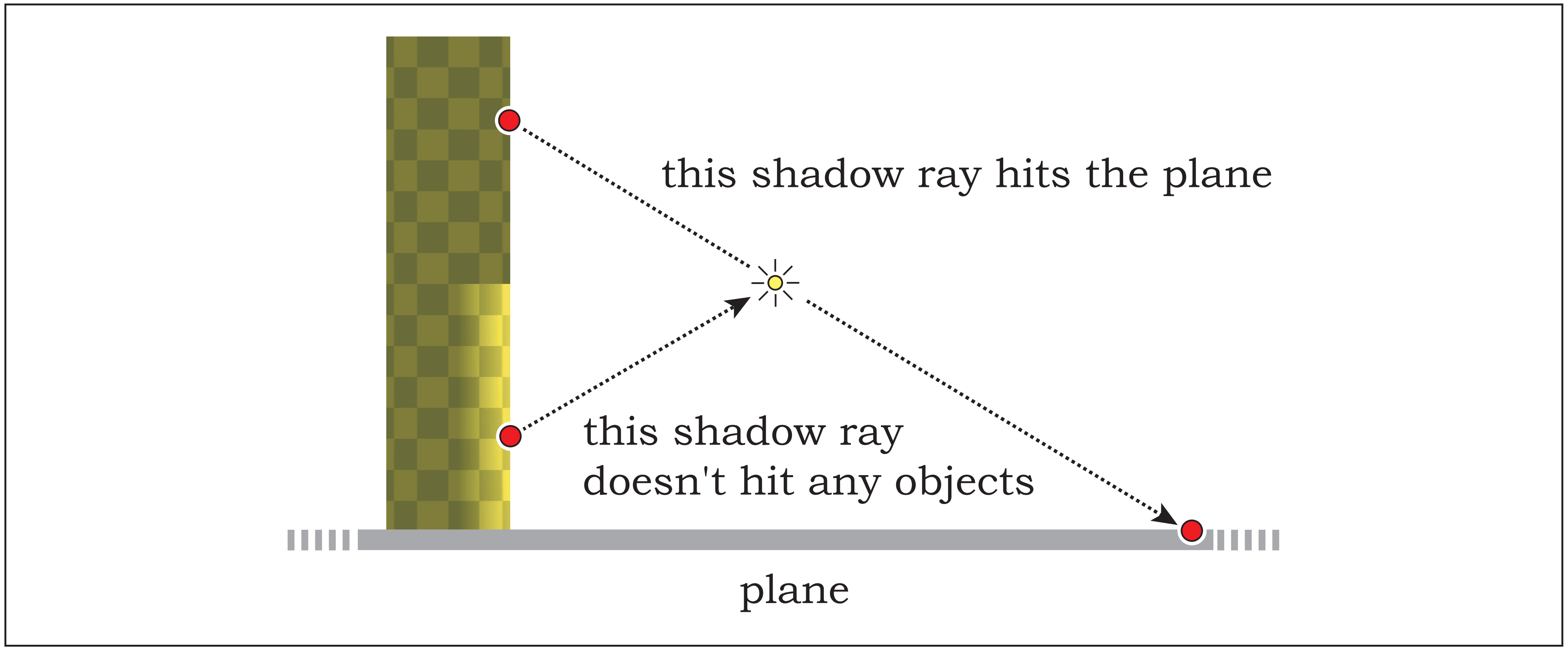
Shadow Ray for Directional Light

- Shadow Ray
  - Has the hit point as its origin.
  - Points towards the direction of the light.
  - Extends to infinity.
    - tmin = 0 (?)
    - tmax = ∞

Shadow Ray for Point Light Source

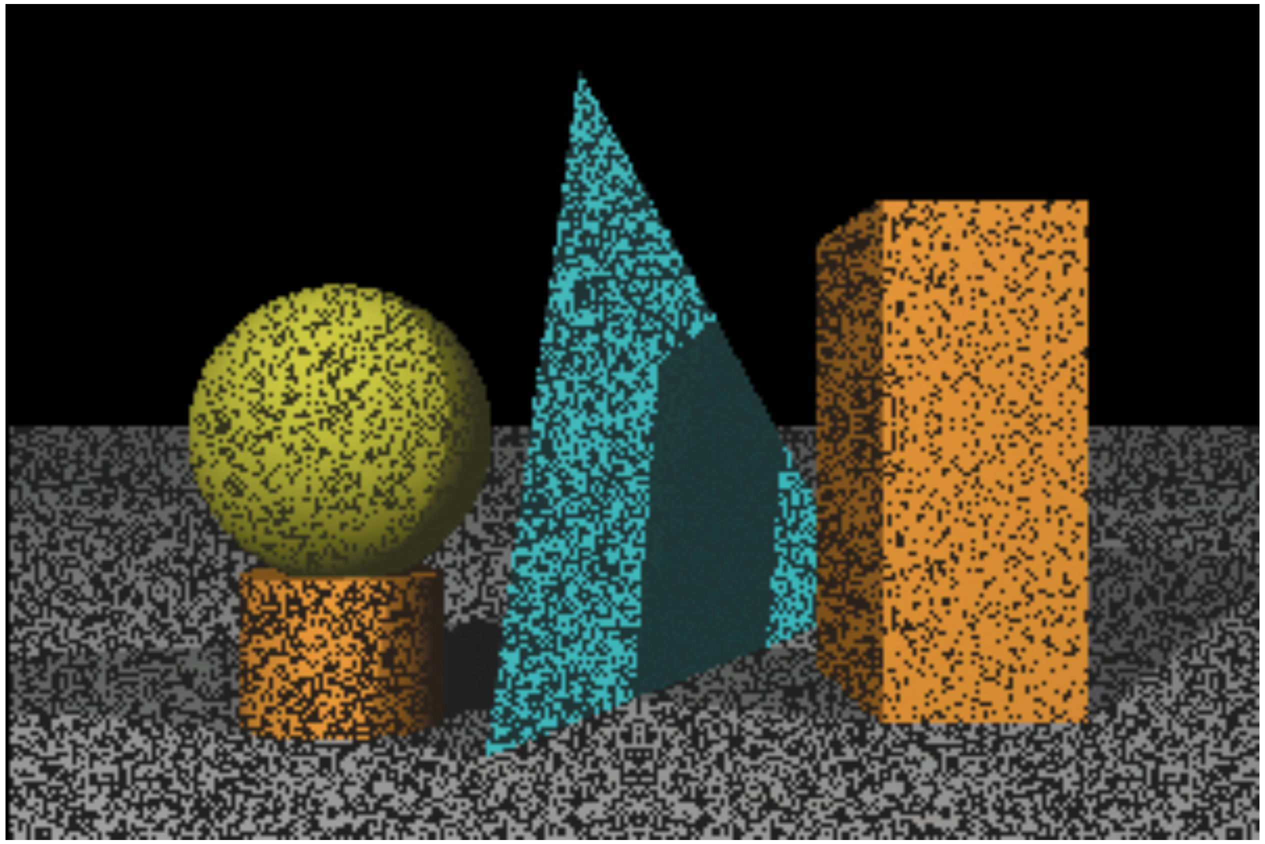
- Shadow Ray
  - Has the hit point as the origin.
  - Points towards the point light source's position.
  - Extends as far as the position of the point light source.
    - tmin = 0 (?)
    - tmax = $\|\mathbf{p} - \mathbf{p_L}\|$

Shadow Ray for Point Light Source



Self Shadowing

- If you set, tmin = 0 (the ray starts exactly at the hit point), you get this:



Self Shadowing

- Numerical computation on a computer is not precise.
- Due to numerical error, hit point might be slightly above or below the surface.
- If it is below, then it’s going to hit the surface its on and get blocked.

Introducing the Epsilon

- So, when creating shadow rays, don't set tmin = 0.
- Set it to a small constant, ε
- I typically use $\varepsilon = 0.0001$
- Moreover, I also displace starting point of the ray in the direction of the normal by the length of ε

$$\mathbf{o} = \mathbf{p} + \varepsilon\mathbf{n}$$

- These tricks eliminated most of the self-shadowing artifacts in my experience.

Change to Light Class

- Add method
  Ray gen_shadow_ray(const ShadeRec &shade_rec)
  to generate the shadow ray to the point in the ShadeRec

- ```
  class Light
  {
  public:
      Light();
      virtual ~Light();
      virtual void radiance(const Float3 &point, Float3 &wi, Float3 &Li) = 0;
      virtual Ray gen_shadow_ray(const ShadeRec &shade_rec) const = 0;
  };
  ```
  ```

DirectionalLight Class: gen_shadow_ray

```
Ray DirectionalLight::gen_shadow_ray( const ShadeRec &shade_rec ) const
{
    return Ray(shade_rec.point + RAY_EPSILON * shade_rec.normal,
               direction, RAY_EPSILON, INFINITY);
}
```

PointLight Class: gen_shadow_ray

```
Ray PointLight::gen_shadow_ray( const ShadeRec &shade_rec ) const
{
    Float3 point = shade_rec.point + RAY_EPSILON * shade_rec.normal;
    return Ray(point, normalize(position - point),
               RAY_EPSILON, (position-point).length() - RAY_EPSILON);
}
```

Putting It All Together

- For each primary eye ray, we have to:
  - Locate the nearest hit point by.
  - Compute information at the hit point that is needed to shade it.
  - Retrieve the nearest shape’s material.
  - Compute the emissive and ambient component.
  - Iterate over all light sources.
    - Compute the incoming light.
    - Compute the direct illumination component.

Putting It All Together

- For each primary eye ray, we have to:
  - Locate the nearest hit point by.
  - Compute information at the hit point that is needed to shade it.
  - Retrieve the nearest shape's material.
  - Compute the emissive and ambient component.
  - Iterate over all light sources.
    - Compute the incoming light.
    - Cast shadow ray to see if light is visible.
    - Compute the direct illumination component if the light is visible.

Putting It All Together

- For each primary eye ray, we have to:
  - Call **intersect_p** of every shape.
  - Call the nearest shape's **intersect** to compute the ShadeRec.
  - Retrieve the nearest shape's material.
  - Call the material' **shade_emit** and **shade_ambient**.
  - Iterate over all light sources.
    - Call the light's **radiance** to compute incoming light.
    - Call the material's **shade_direct** for compute the light's contribution

Putting It All Together

- For each primary eye ray, we have to:
  - Call **intersect_p** of every shape.
  - Call the nearest shape's **intersect** to compute the ShadeRec.
  - Retrieve the nearest shape's material.
  - Call the material' **shade_emit** and **shade_ambient**.
  - Iterate over all light sources.
    - Call the light's **radiance** to compute incoming light.
    - Call the light's **gen_shadow_ray**.
    - Iterate over all objects.
      - Call **intersect_p** of every object.
    - Call the material's **shade_direct** if none of the **intersect_p** return true.

hit_anything

- A convenient function that computes whether a ray hits any object.

- ```
  bool hit_anything(Ray &ray)
  {
      FOR(shape_index, shape_count)
      {
          Shape *shape = shapes[shape_index];
          if (shape->intersect_p(ray))
              return true;
      }
      return false;
  }
  ```
  ```

The Renderer (Only the Shading Part)

- ```
  Float3 color(0,0,0);
  if (hitted_shape != NULL)
  {
      ray.tmax = INFINITY;
      ShadeRec shade_rec;
      hitted_shape->intersect(ray, shade_rec);
      Material *material = hitted_shape->material;

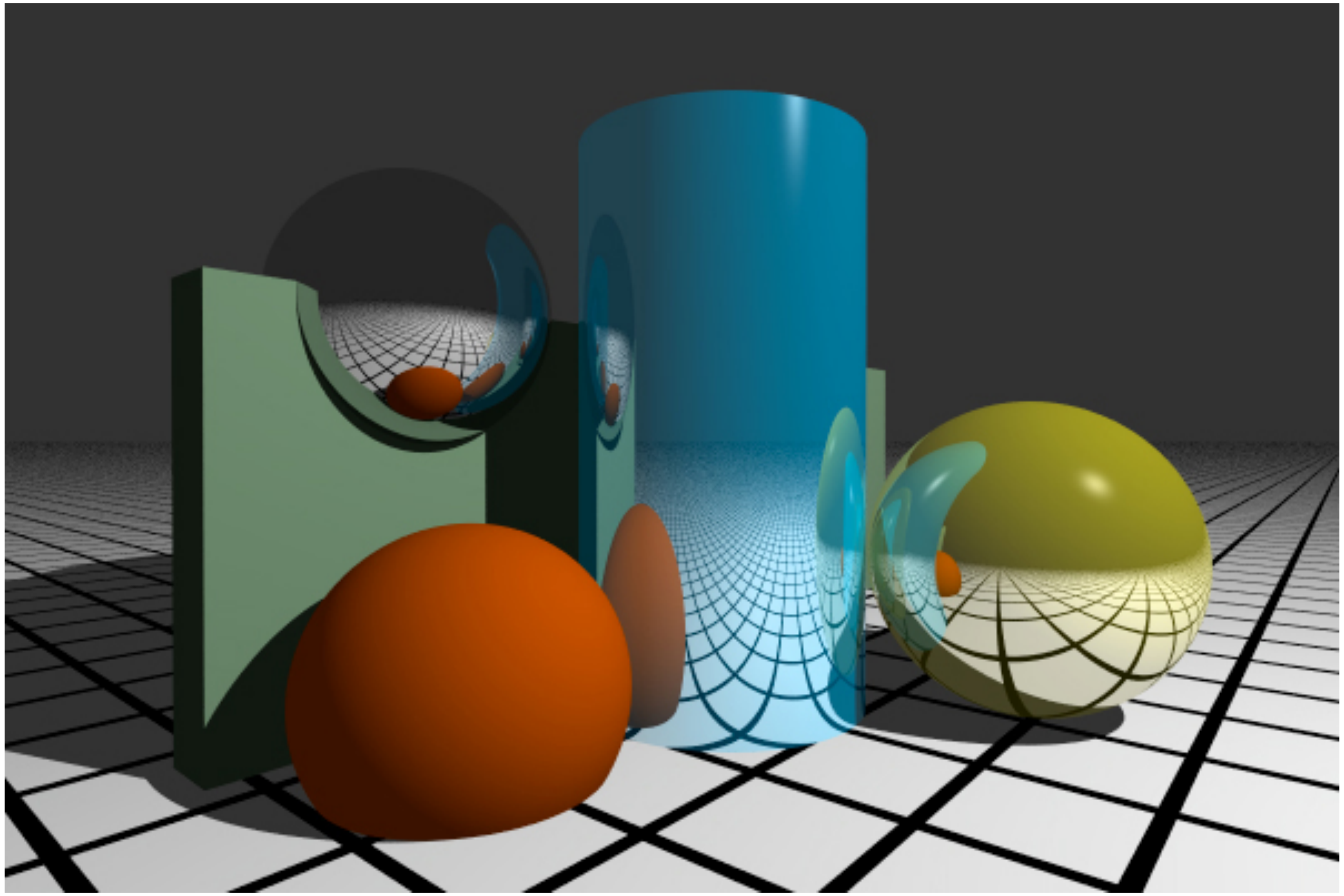
      color += material->shade_emit(shade_rec);
      color += material->shade_ambient(ambient_light, shade_rec);

      FOR(light_index, light_count)
      {
          Light *light = lights[light_index];
          Ray shadow_ray = light->gen_shadow_ray(shade_rec);
          if (!hit_anything(shadow_ray))
          {
              Float3 Li, wi;
              light->radiance(shade_rec.point, wi, Li);
              color += material->shade_direct(wi, Li, shade_rec);
          }
      }
  }
  ```
  ```

Perfect Reflection

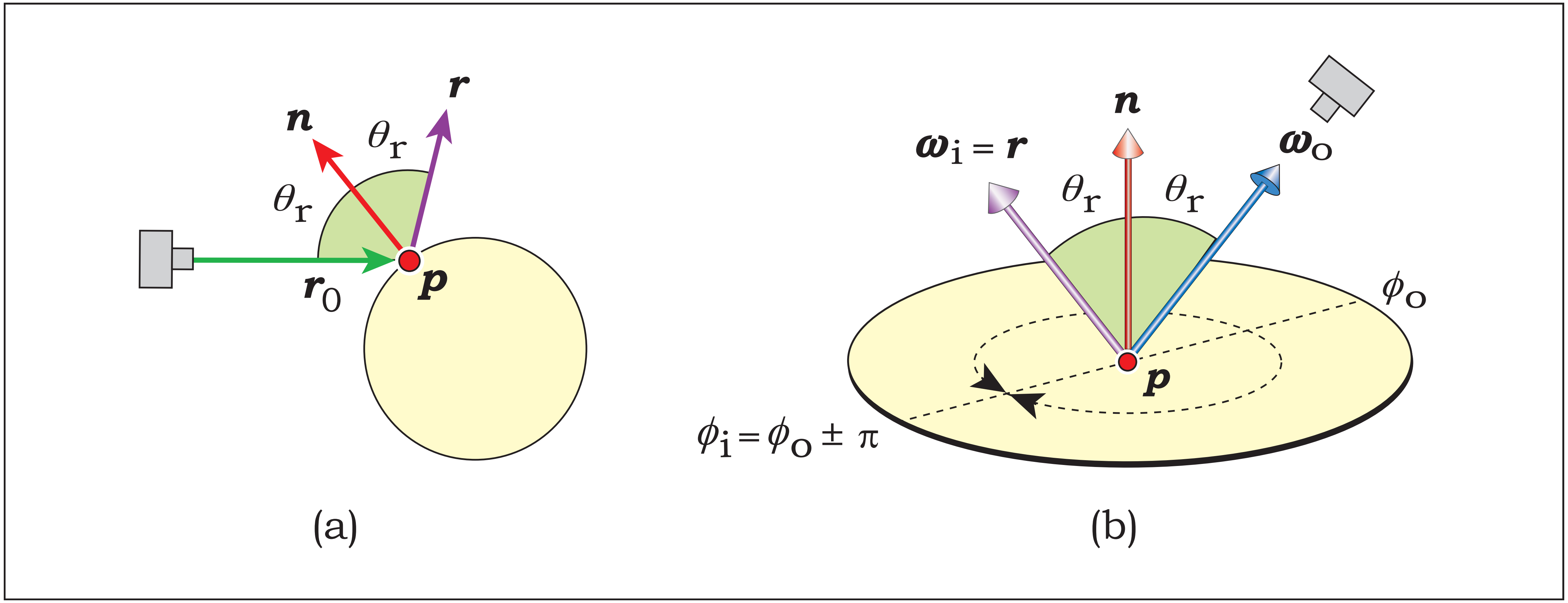
Goal of This Section

- To simulate mirrors.



Mirror Reflection

- A mirror-like appearance is caused by light reflecting off the surface in the mirror reflection direction to the eye.

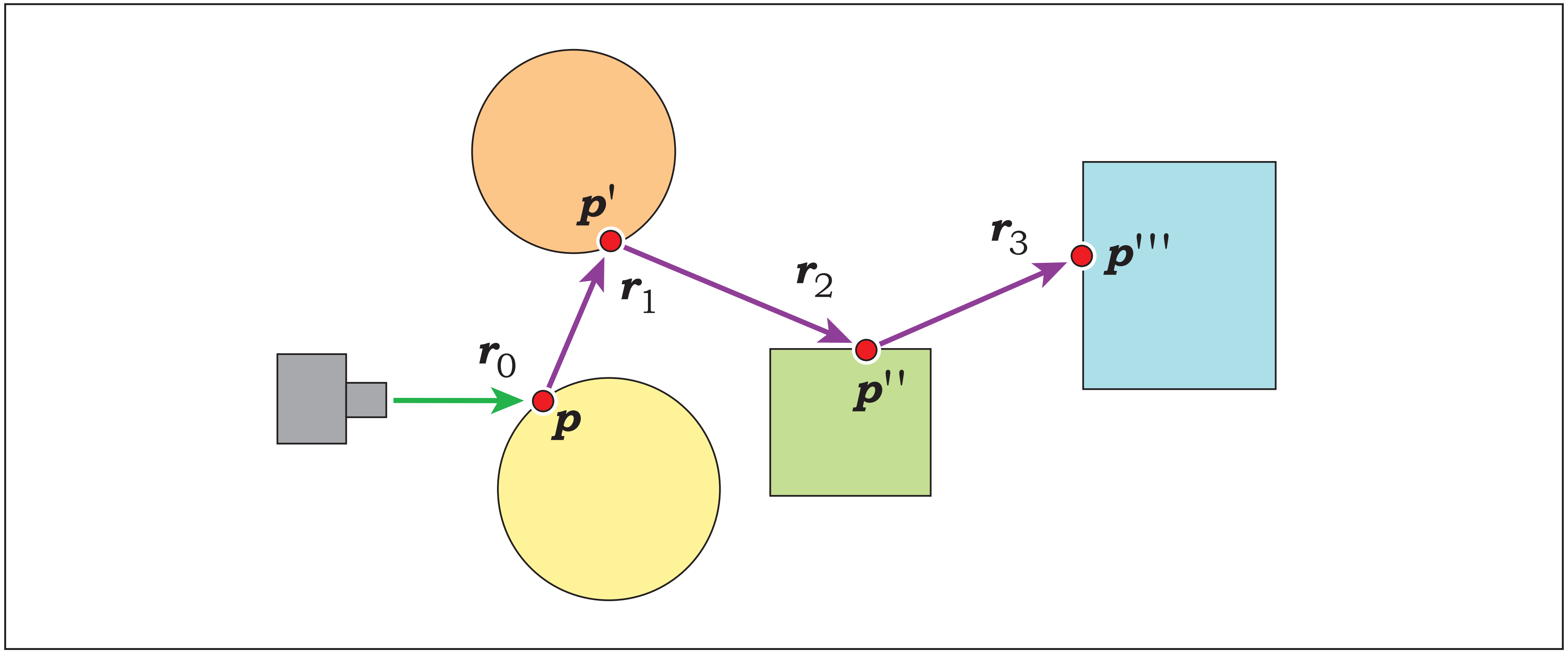


Simulating Mirror Reflection

- We can simulate mirror reflection by
  - Casting **reflected ray** in the direction of mirror reflection
  - Finding out what surface the ray hits.
  - Computing the color of that surface.
  - Combine the color with material property to produce the color of the mirror surface.

- The surface the reflected ray intersect can also be a mirror.
  - In this case, we generate another reflected ray and trace it.
  - We continue until we hit a non-mirror surface.
  - Or until we have done this, say, 10 times. (We can't continue forever.)

Simulating Mirror Reflection



Recursive Ray Tracing

- Finding the color of the surface a reflected ray intersects is the same as finding the color of the surface a primary ray intersects.
- We can encapsulate this process in a function.
  Let's call it **trace_ray**.
- If we determine if the material is perfectly specular (i.e., behaves like mirror), we can call **trace_ray** recursively to find the color along the reflected ray.

Pseudocode for trace_ray

```
trace_ray(ray, level)
{
  if level <= MAX_LEVEL
  {
    Find the first intersection point
    Shade direct illumination from light source

    if material is perfectly specular
    {
      Generate reflected_ray
      reflected_color = trace_ray(reflected_ray, level+1)
      Shade the point again taking into account reflected color
    }

    return the point's color
  }
  else
    return black
}
```

Pseudocode for Renderer

```
for each pixel:
    Generate ray to that pixel.
    color = trace_ray(ray)
    Save color to the appropriate pixel.
```

Change to Material Interface

- Add methods:
  - **is_perfectly_reflective**
    - Return true if and only if the material behaves like a mirror.
    - Most material will return false.
  - **gen_reflected_ray**
    - Generate reflected ray from ShadeRec.
  - **shade_reflect**
    - Combine the color along the reflected ray with material property to produce the contribution due to mirror reflection.

Change to Material Interface

```
class Material
{
public:
    Material(const std::string &_name = "");
    virtual ~Material();

    /* Other shade_XXX methods go here */

    virtual Float3 shade_reflect(
        const Float3 &wi, const Float3 &Li,
        const ShadeRec &shade_rec) const;

    virtual bool is_perfectly_reflective(const ShadeRec &shade_rec) const;

    virtual Ray gen_reflected_ray(const ShadeRec &shade_rec) const;

public:
    std::string name;
};
```

Material Class: gen_reflected_ray

- Use the formula for direction of mirror reflection.

$$\mathbf{r} = -\omega_o + 2(\mathbf{n} \cdot \omega_o)\mathbf{n}$$

- The ray's origin is the hit point plus epsilon times normal.

- tmin is also set to ε

- ```
Ray Material::gen_reflected_ray(const ShadeRec &shade_rec) const
{
    Float3 d = -shade_rec.w_out +
        2 * dot(shade_rec.normal, shade_rec.w_out) * shade_rec.normal;
    Float3 o = shade_rec.point;
    return Ray(o, d, RAY_EPSILON, INFINITY);
}
```
```

Reflective Class

- Represent a reflective material.

- Most attributes are like Phong.

  - ambient color,
    diffuse color,
    emissive color,
    specular color, and shininess

- Store one more attribute: **reflective color**

  - The color along the reflected ray gets multiplied by this to produce the contribution of mirror reflection.

- Return true in **is_perfectly_reflective**

Reflective Class

```
class Reflective : public Material
{
public:
    Reflective( ... );
    virtual ~Reflective();

    /* Methods go here */

public:
    ScalableFloat3 ambient;
    ScalableFloat3 diffuse;
    ScalableFloat3 emission;
    ScalableFloat3 specular;
    float shininess;
    ScalableFloat3 reflective;
};
```

Reflective Class: shade_reflect

```
Float3 Reflective::shade_reflect( const Float3 &wi, const Float3 &Li,
    const ShadeRec &shade_rec ) const
{
    return reflective.value() * Li;
}
```

The Renderer

```
Float3 trace_ray(Ray &ray, int depth)
{
    if (depth > MAX_DEPTH)
        return Float3(0,0,0);

    Shape *hitted_shape = NULL;
    /* Code to find first intersection goes here */

    Float3 color(0,0,0);
    if (hitted_shape != NULL)
    {
        ray.tmax = INFINITY;
        ShadeRec shade_rec;
        hitted_shape->intersect(ray, shade_rec);
        Material *material = hitted_shape->material;

        /* Code to shade emissive, ambient, and direct component goes here */
```

The Renderer

```
        if (material->is_perfectly_reflective(shade_rec))
        {
            Ray reflected_ray = material->gen_reflected_ray(shade_rec);
            Float3 Li = trace_ray(reflected_ray, depth+1, trans_coeff);
            Float3 reflected_color = material->shade_reflect(
                reflected_ray.direction, Li, shade_rec);
            color += reflected_color;
        }
    }
    else
        color = background_color;

    return color;
}
```

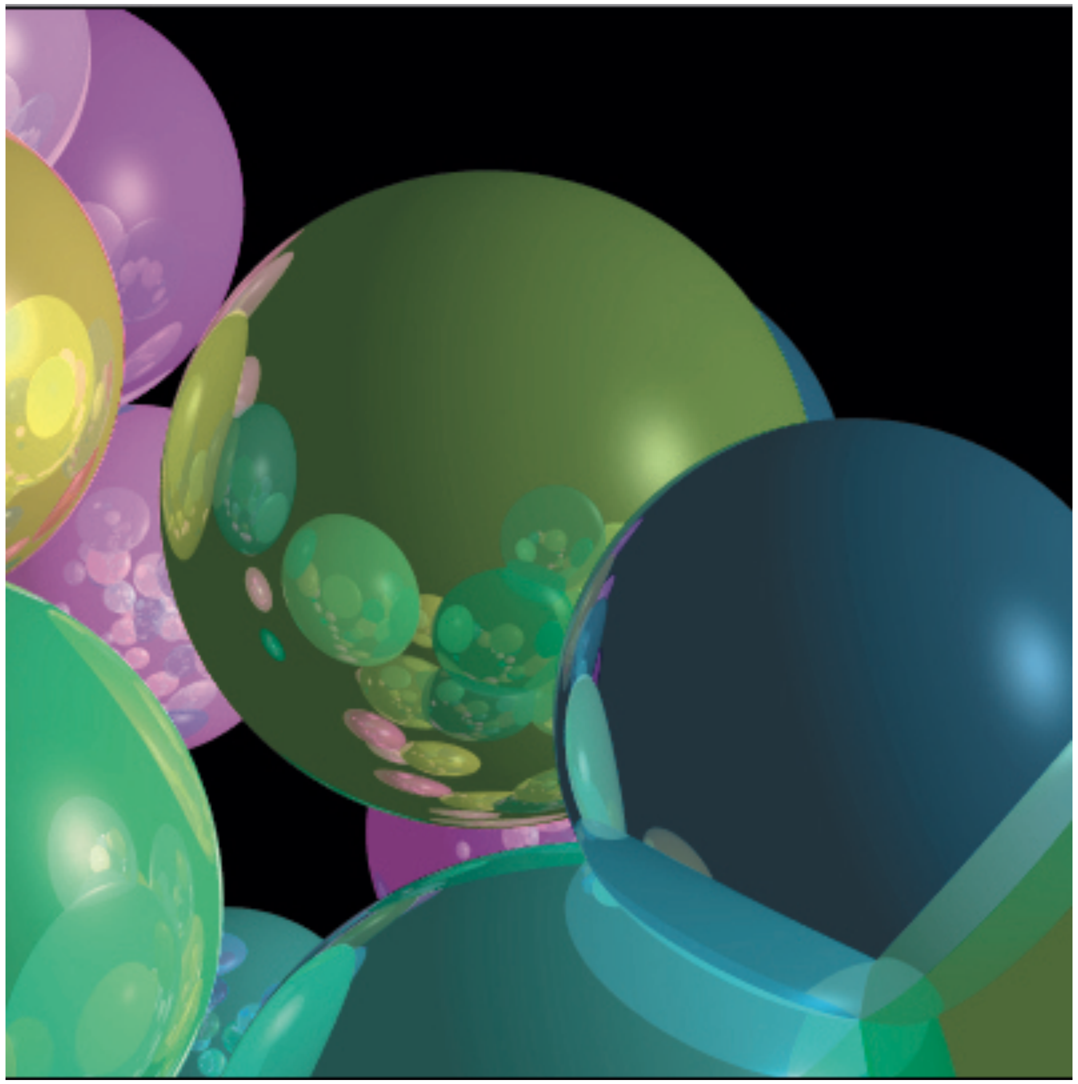
The Renderer

```
FOR(iy, image_height)
FOR(ix, image_width)
{
    float sx = 2 * (ix + 0.5f) / rr->image_width - 1;
    float sy = 2 * (iy + 0.5f) / rr->image_height - 1;
    Ray ray = rr->scene->camera->gen_ray(sx, sy);

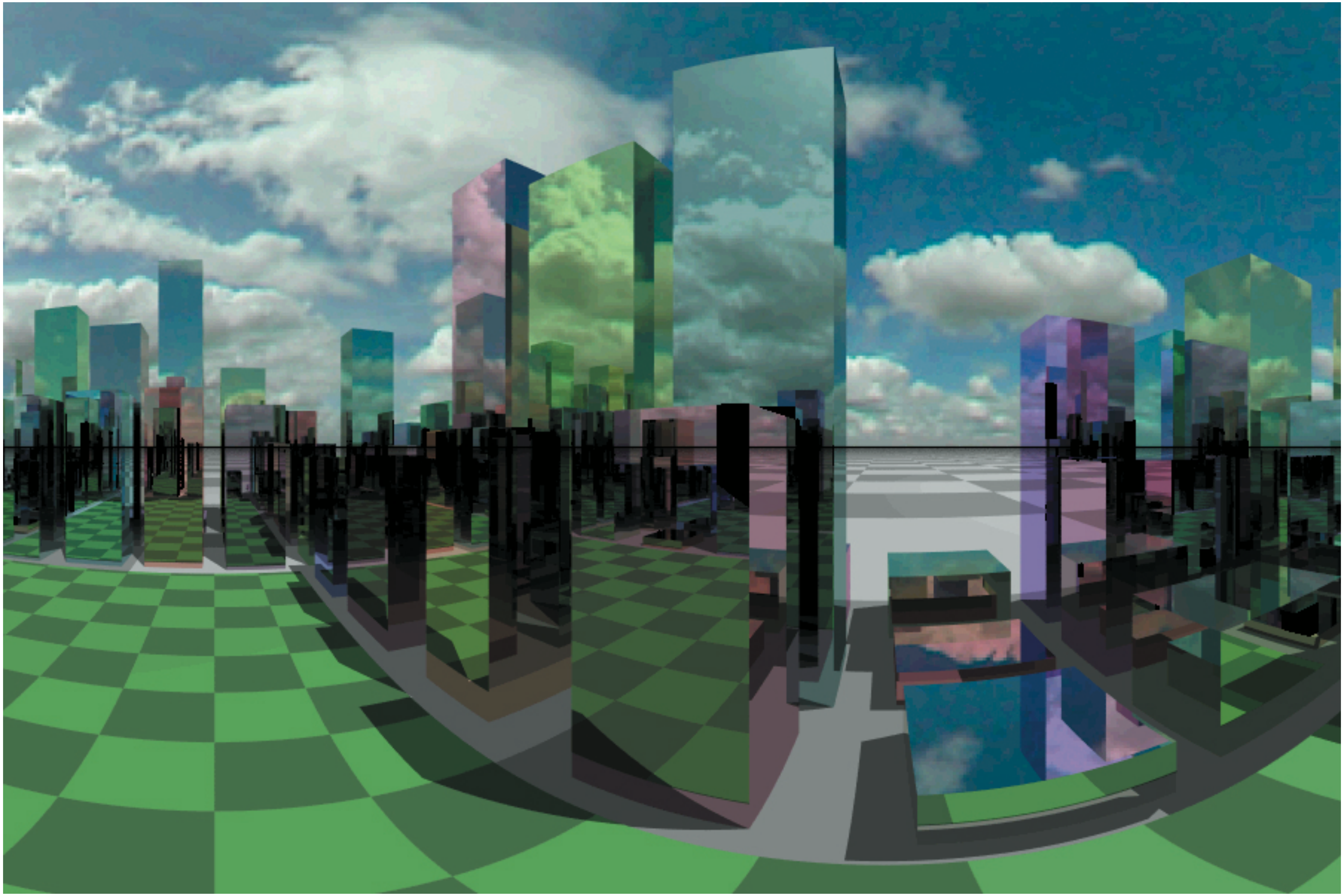
    Float3 color = trace_ray(ray, 1);

    texture->set_pixel(ix, iy, Float4(color,1));
}
```

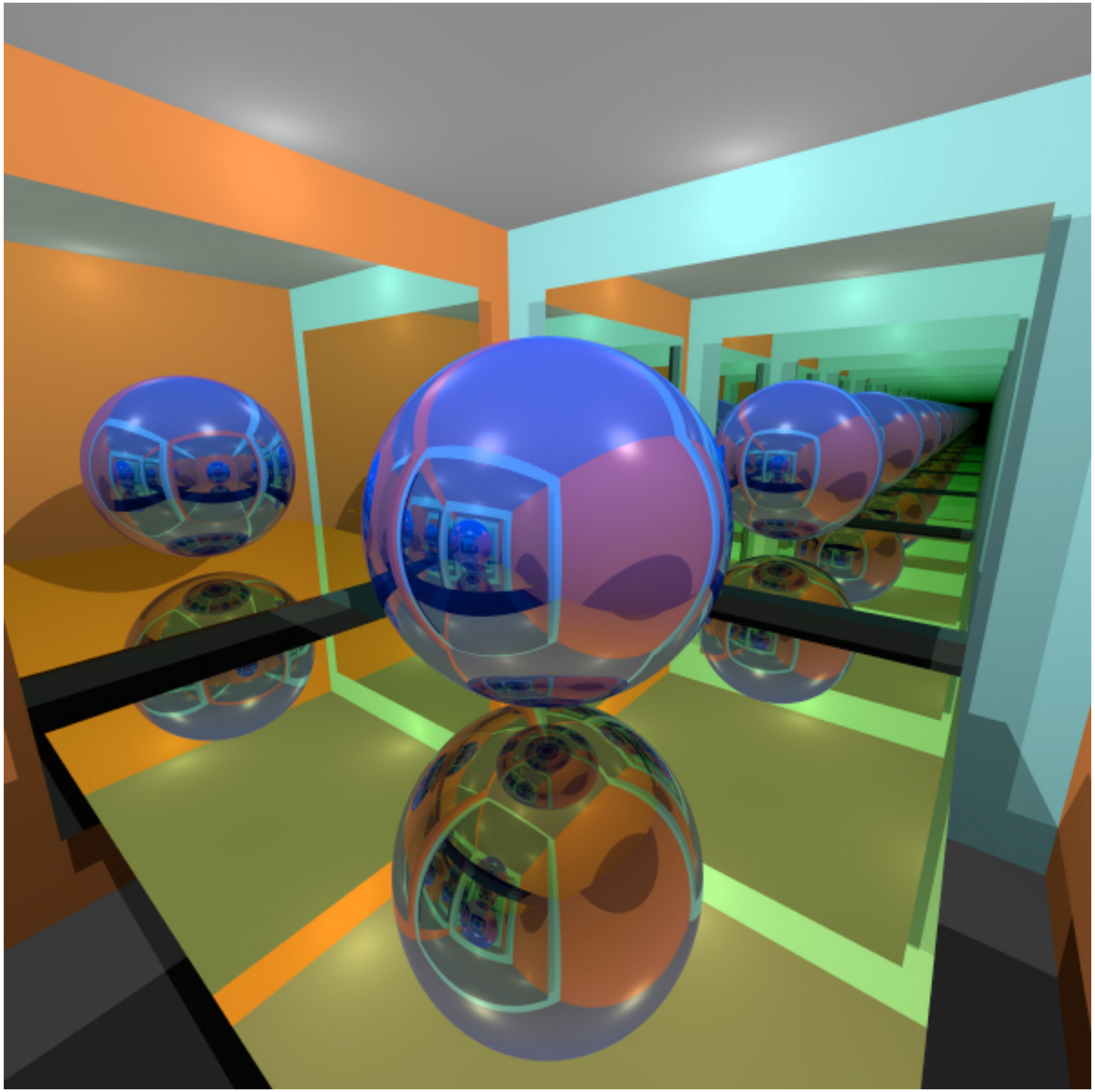
More Images Containing Mirrors



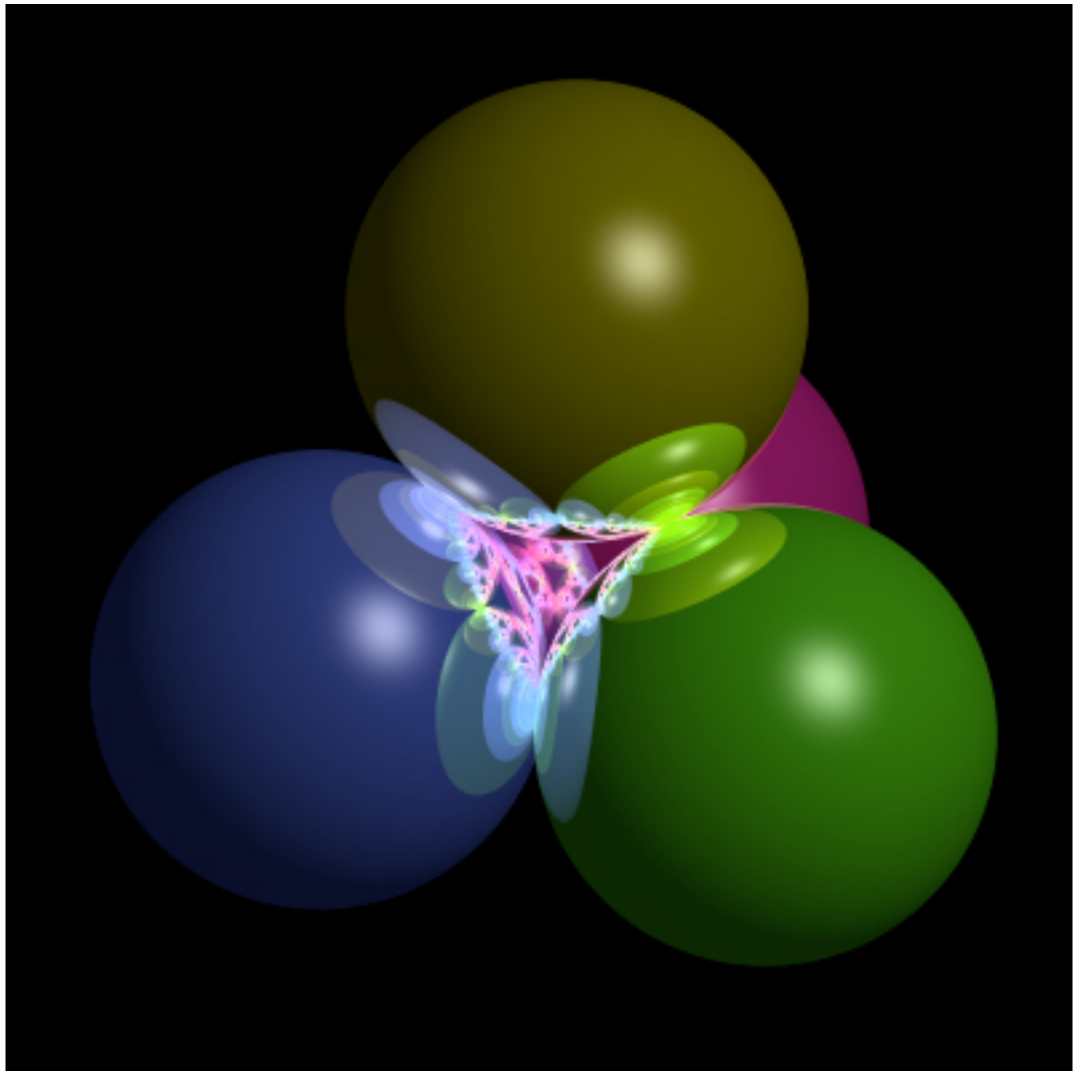
More Images Containing Mirrors



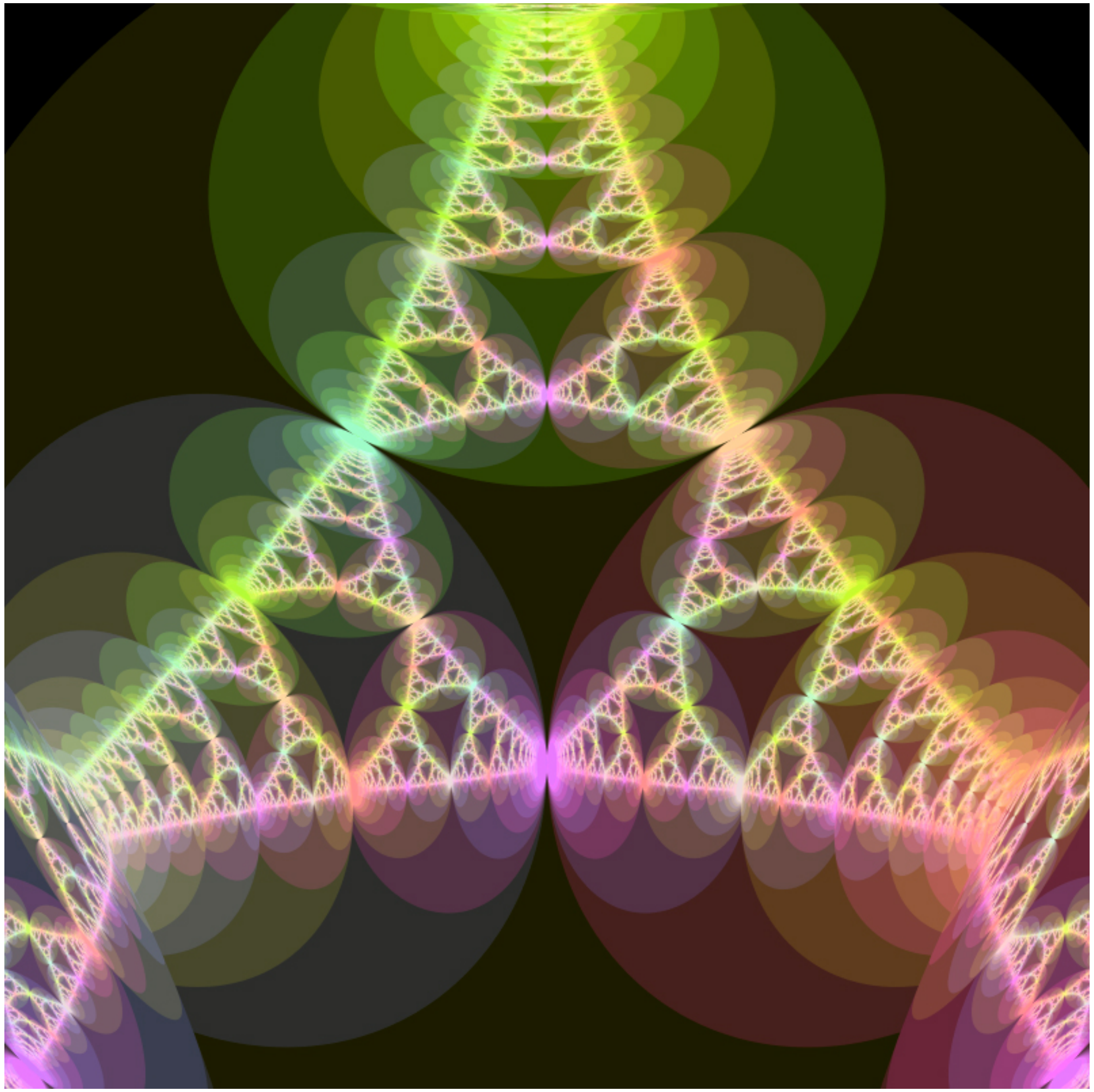
More Images Containing Mirrors



More Images Containing Mirrors



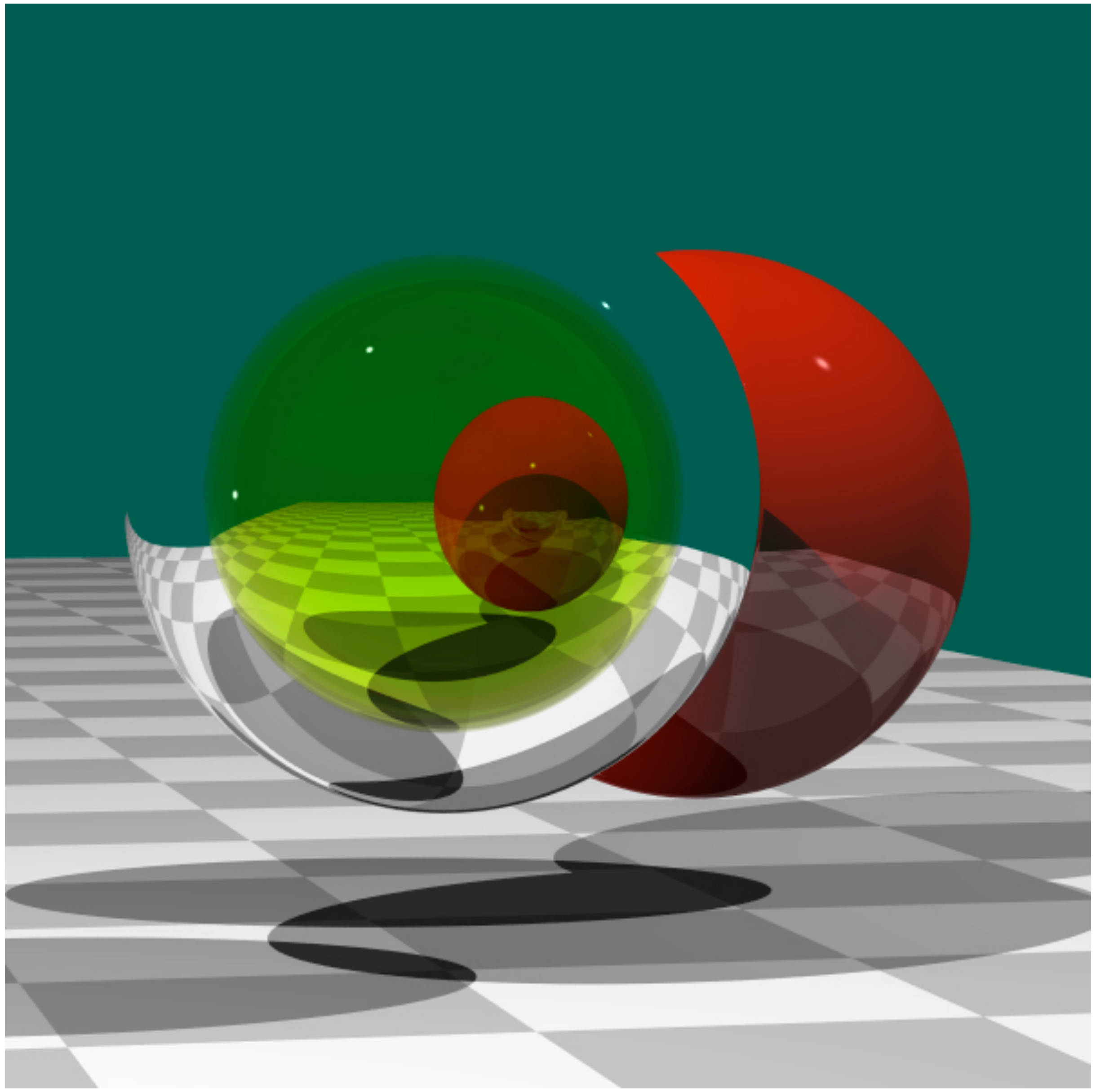
More Images Containing Mirrors



Perfect Transmission

Goal of This Section

- To simulate transparent objects.



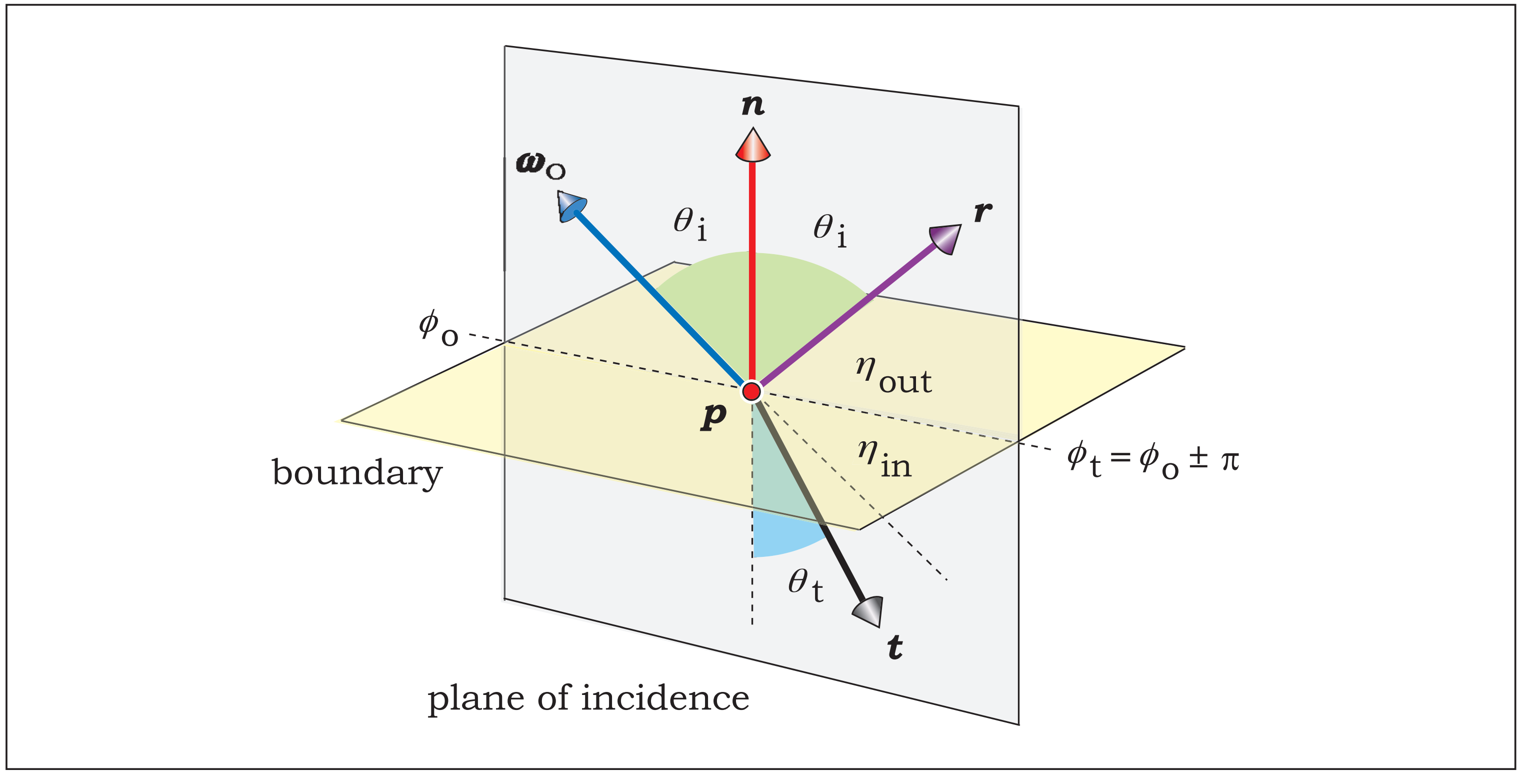
Physics of Refraction

- Light travels at speed $c = 2 \times 10^7 m/s$ in a perfect vacuum.
- In other media, light travels slower.
- The **absolute index of refraction** η is the ratio between c and the speed of light in that medium.

$$\eta = \frac{c}{v}$$

Physics of Refraction

- When a ray of light hits a surface of a transparent medium,
  - Light gets reflected along the reflected ray $\mathbf{r}$
  - Light might gets transmitted into the media along the **refracted ray** $\mathbf{t}$



Physics of Refraction

- The direction of **t** depends the **relative index of refraction**

$$\eta = \frac{\eta_{\text{in}}}{\eta_{\text{out}}}$$

where

η_{in} is the absolute index of refraction of the media the light goes into

η_{out} is the absolute index of refraction of the media the light comes from

- The direction can be determined by **Snell's law**:

$$\frac{\sin \theta_i}{\sin \theta_t} = \frac{\eta_{\text{in}}}{\eta_{\text{out}}} = \eta$$

Physics of Refraction

- The transmission direction **t** can be computed as follows:

$$\mathbf{t} = \frac{1}{\eta}\omega_o - \left(\cos\theta_i - \frac{1}{\eta}\cos\theta_t\right)\mathbf{n}$$

where

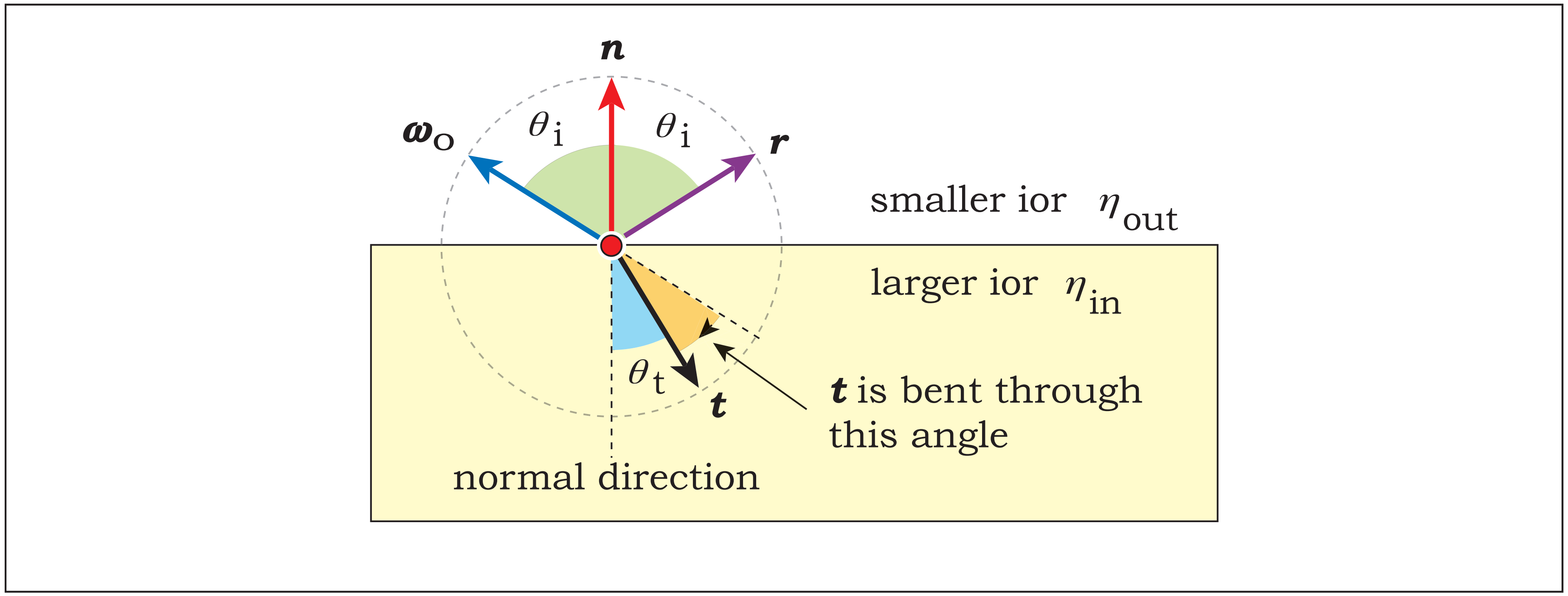
$$\cos\theta_i = \omega_i \cdot \mathbf{n}$$

and

$$\cos\theta_t = \sqrt{1 - \frac{1}{\eta^2}(1 - \cos^2\theta_i)}$$

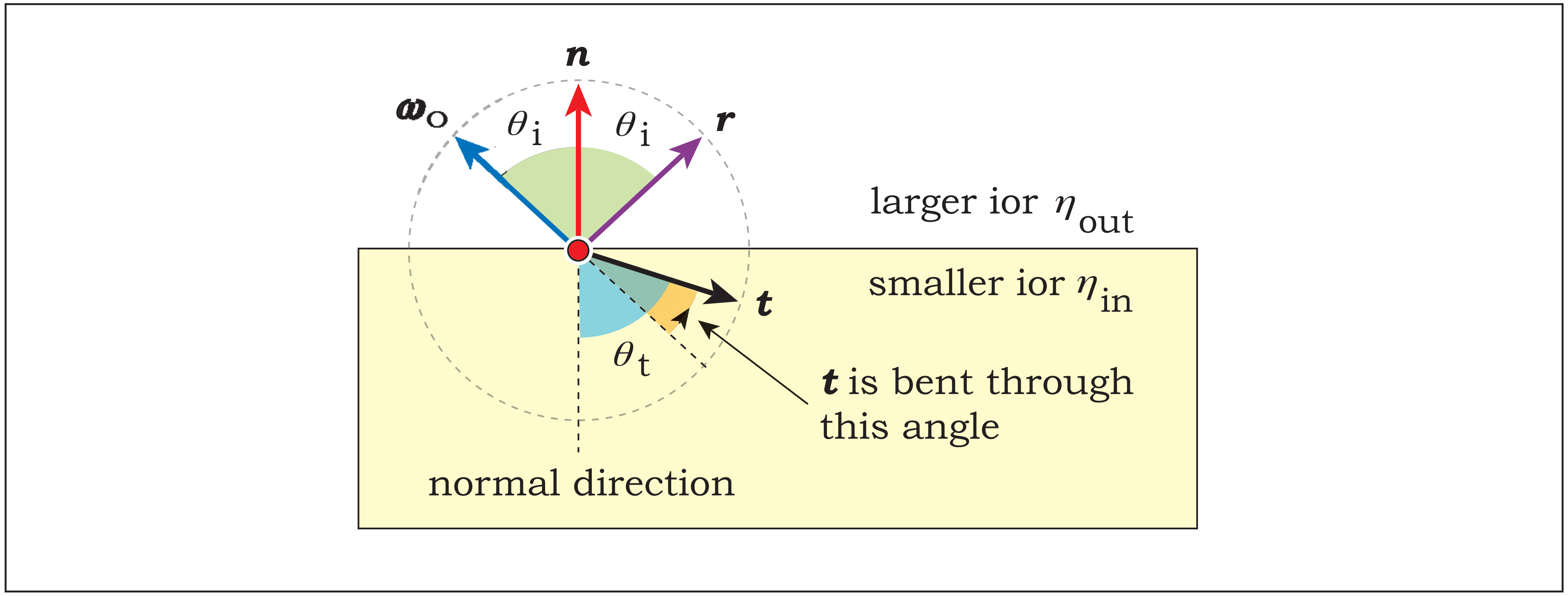
Physics of Refraction

- When light passes
  from a medium with a smaller index of refraction
  to a medium with a larger index of refraction,
  it bends **towards** the normal at the hit point.



Physics of Refraction

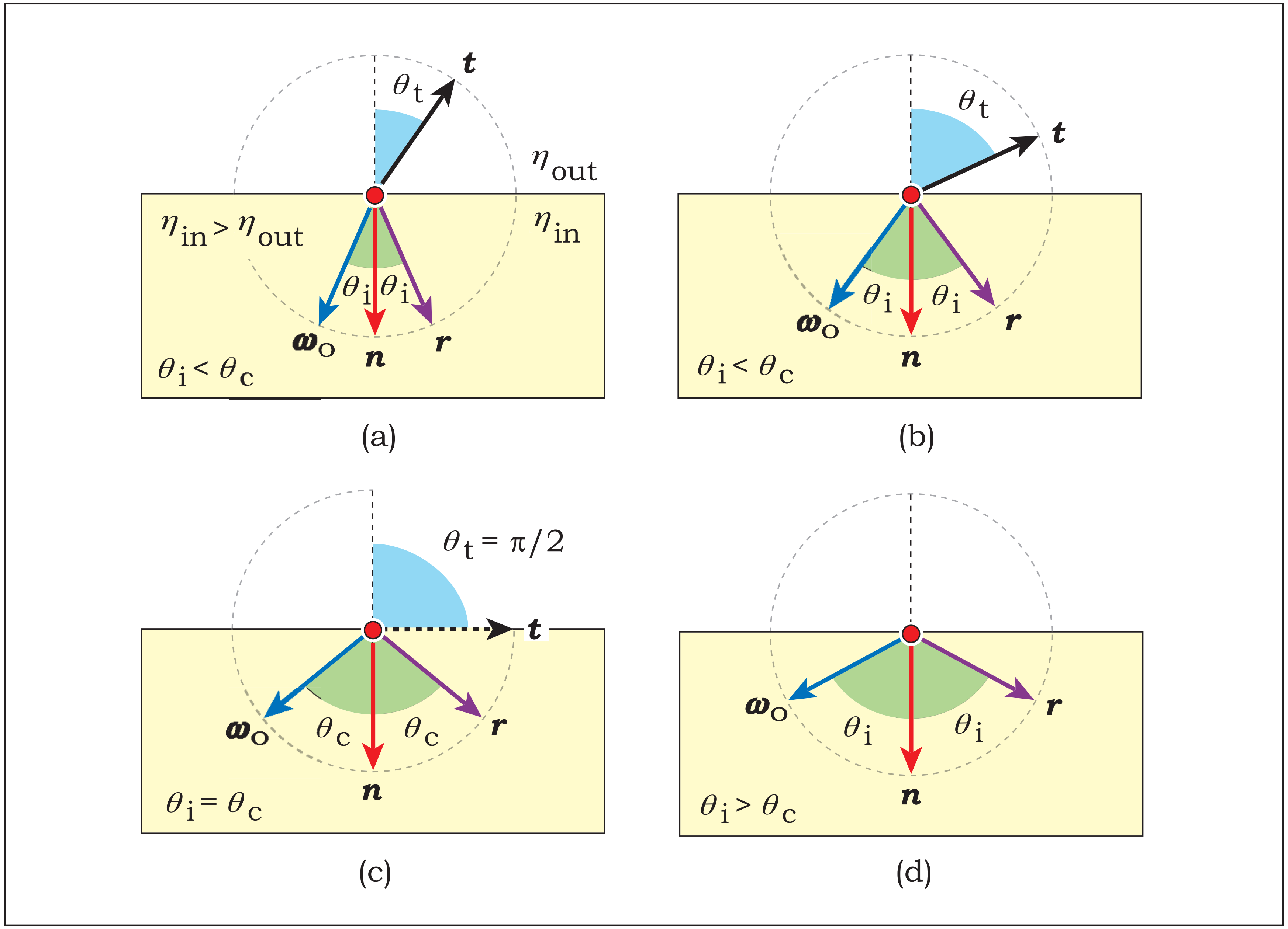
- When light passes
  from a medium with a larger index of refraction
  to a medium with a smaller index of refraction
  it bends **away** from the normal at the it point.



Total Internal Reflection

- In the last case, light might not get transmitted to the medium with smaller index of refraction.
- This happens when the incident angle θ_i is larger than the **critical angle** θ_c
- When this happens, light gets reflected off the surface as if the surface is a mirror.
- This phenomenon is called **total internal reflection**.

Total Internal Reflection



Total Internal Reflection

- We can check for total internal reflection by checking if

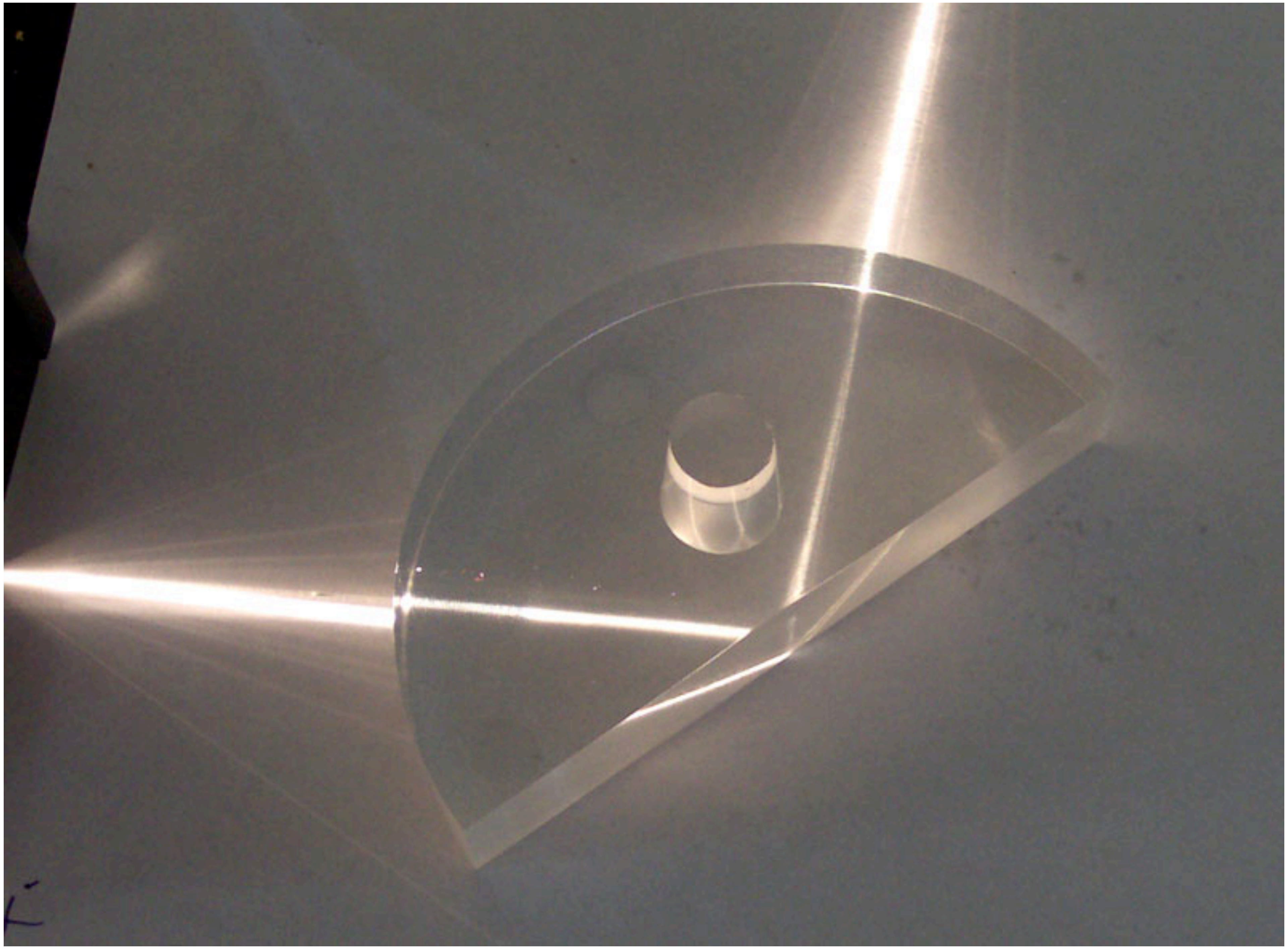
$$1 - \frac{1}{\eta^2}(1 - \cos^2\theta_i) < 0$$

- If so, we cannot compute

$$\cos\theta_t = \sqrt{1 - \frac{1}{\eta^2}(1 - \cos^2\theta_i)}$$

so there can be no transmission.

Total Internal Reflection

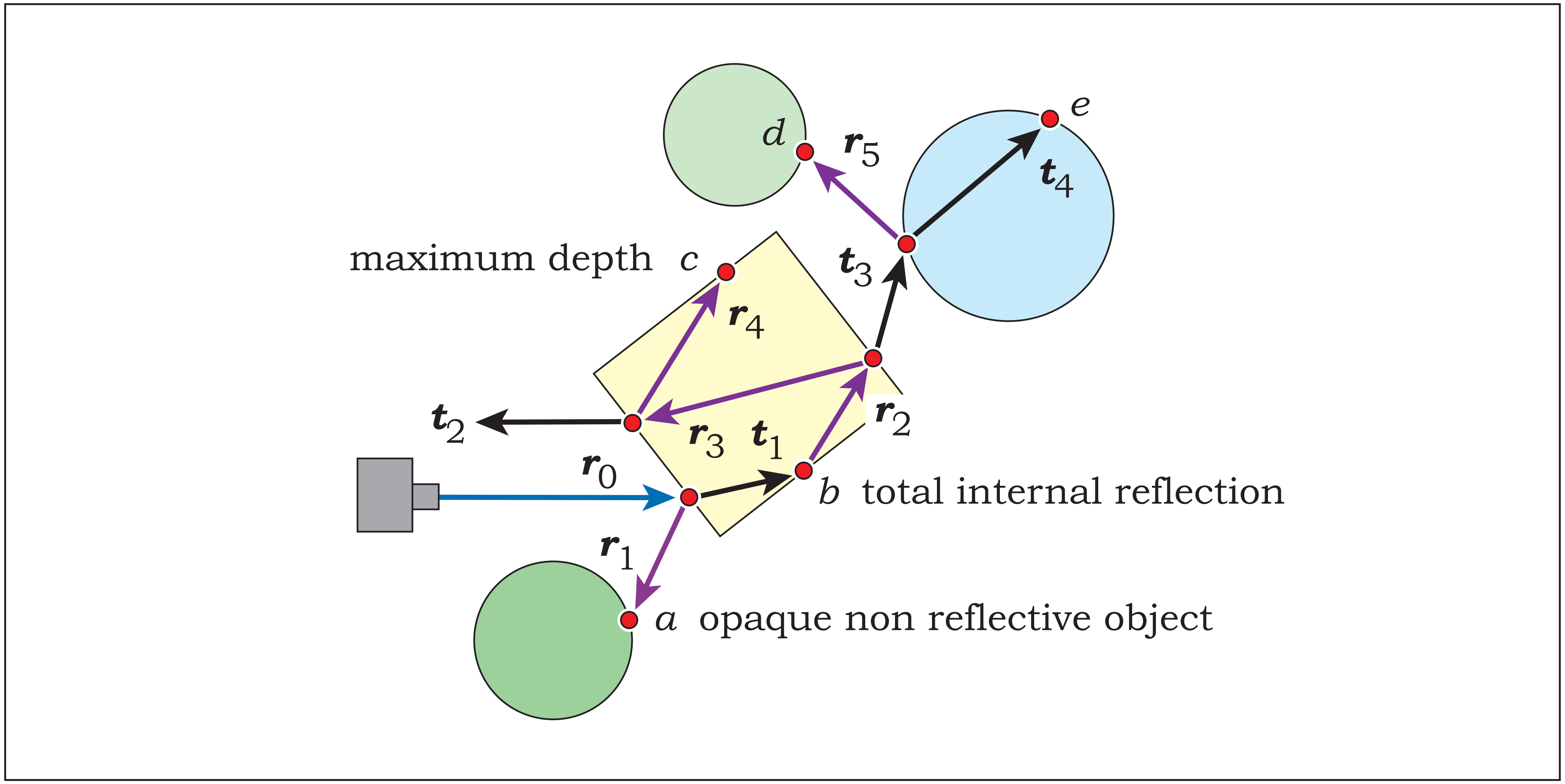


Total Internal Reflection

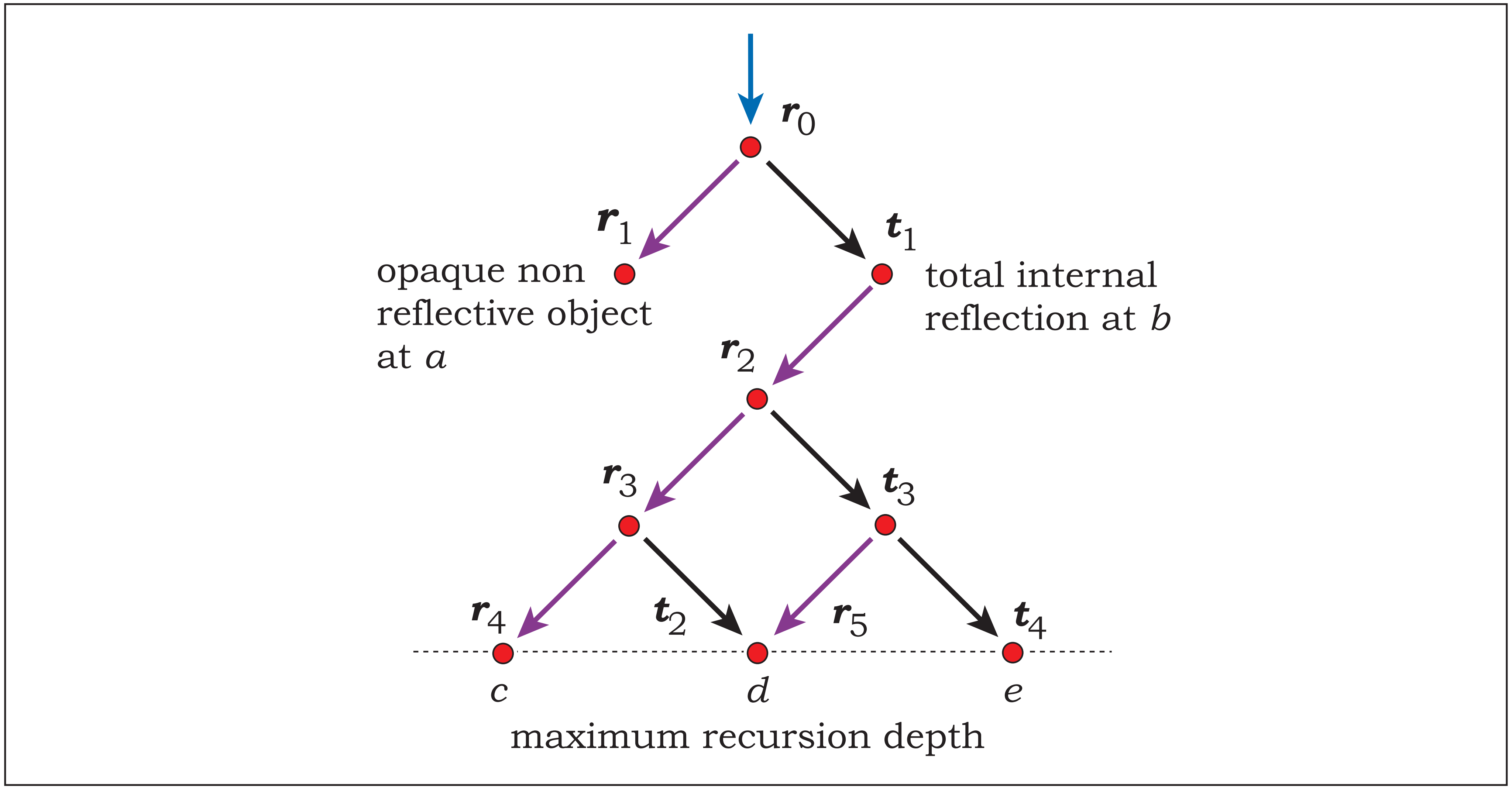
Simulating Refraction

- At each point of intersection, in addition to generating a reflected ray, we may generate a **refracted ray**.
- Not every ray-object intersection generates two rays.
- Total internal reflection cause only reflected ray to be generated.
- We can represent generated rays with a **ray tree**. (See the slide after the next.)

Simulating Refraction

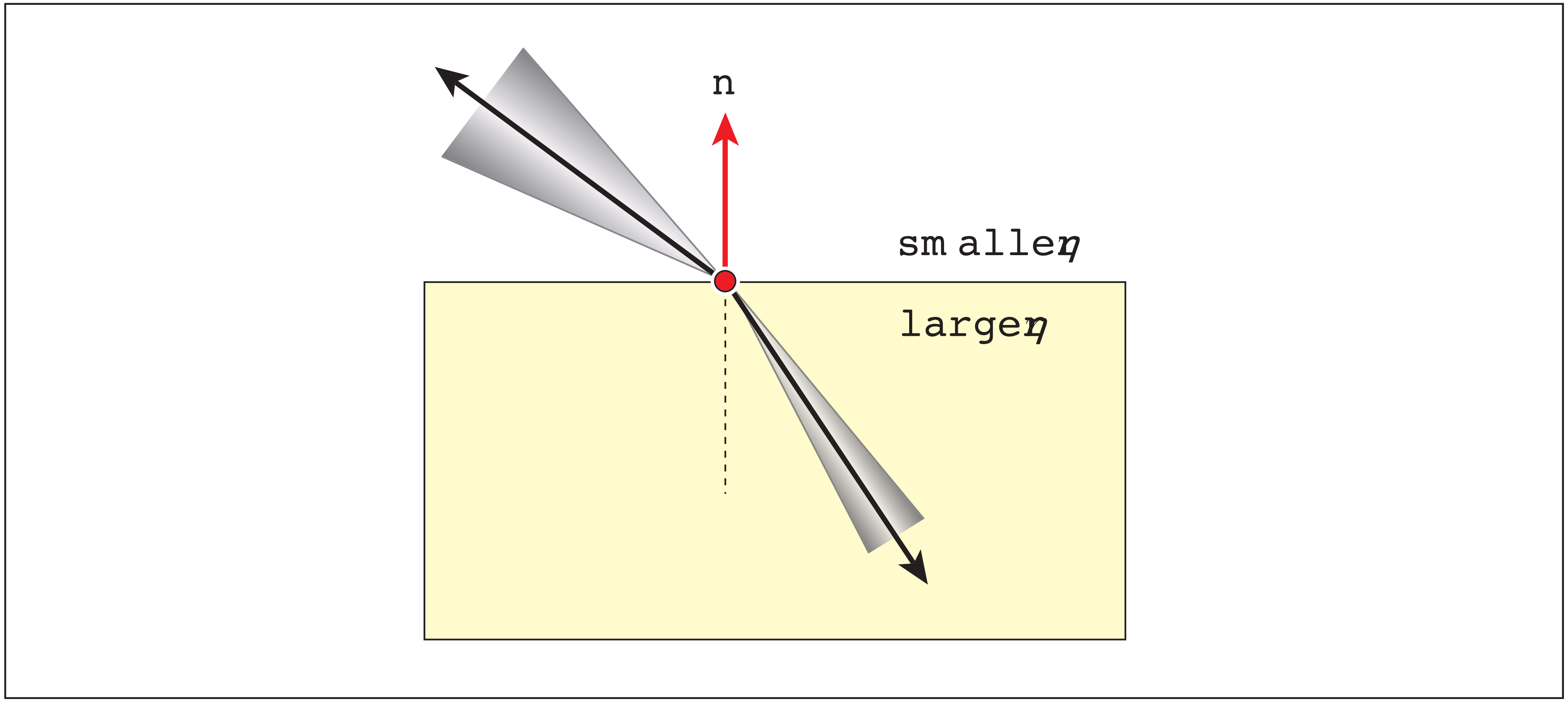


Simulating Refraction



Scaling Refracted Light

- When light passes between material with difference index of refraction, it either gets condensed or diluted



Scaling Refracted Light

- Suppose light with intensity L_i hits a surface of a transparent object. Then the light that gets transmitted is given by:

$$L_t = k_t \left(\frac{\eta_t^2}{\eta_i^2} \right) L_i$$

- k_t is a real number from 0 to 1 that indicates the fraction of light that gets refracted.

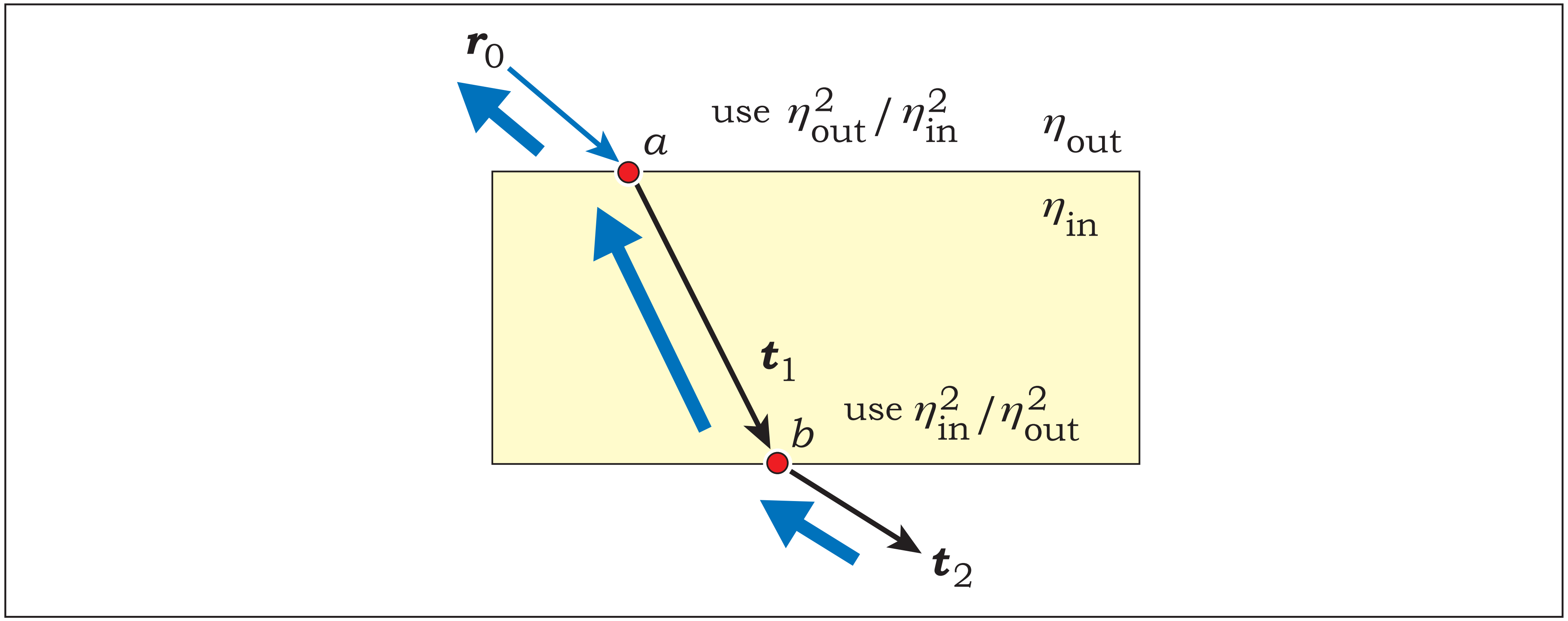
- Similarly, k_r is the fraction of light that gets reflected.

- Typically, $k_r + k_t = 1$

Compute the Correct Scaling Factor

- A ray can hit either inside the object or outside the object.
- ShadeRec has a field for inside/outside information.
- If intersection is outside, use $\eta_{\mathrm{out}}^2/\eta_{\mathrm{in}}^2$
- If intersection is inside, use $\eta_{\mathrm{in}}^2/\eta_{\mathrm{out}}^2$

Compute the Correct Scaling Factor



Fresnel Equations

- In some type of medium called **dielectrics** (e.g., glasses, clear plastics), k_t and k_r varies with the incident angle θ_i

- They can be computed as follows:

$$r_{\parallel} = \frac{\eta \cos\theta_i - \cos\theta_t}{\eta \cos\theta_i + \cos\theta_t}$$

$$r_{\perp} = \frac{\cos\theta_i - \eta \cos\theta_t}{\cos\theta_i + \eta \cos\theta_t}$$

$$k_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2)$$

$$k_t = 1 - k_r$$

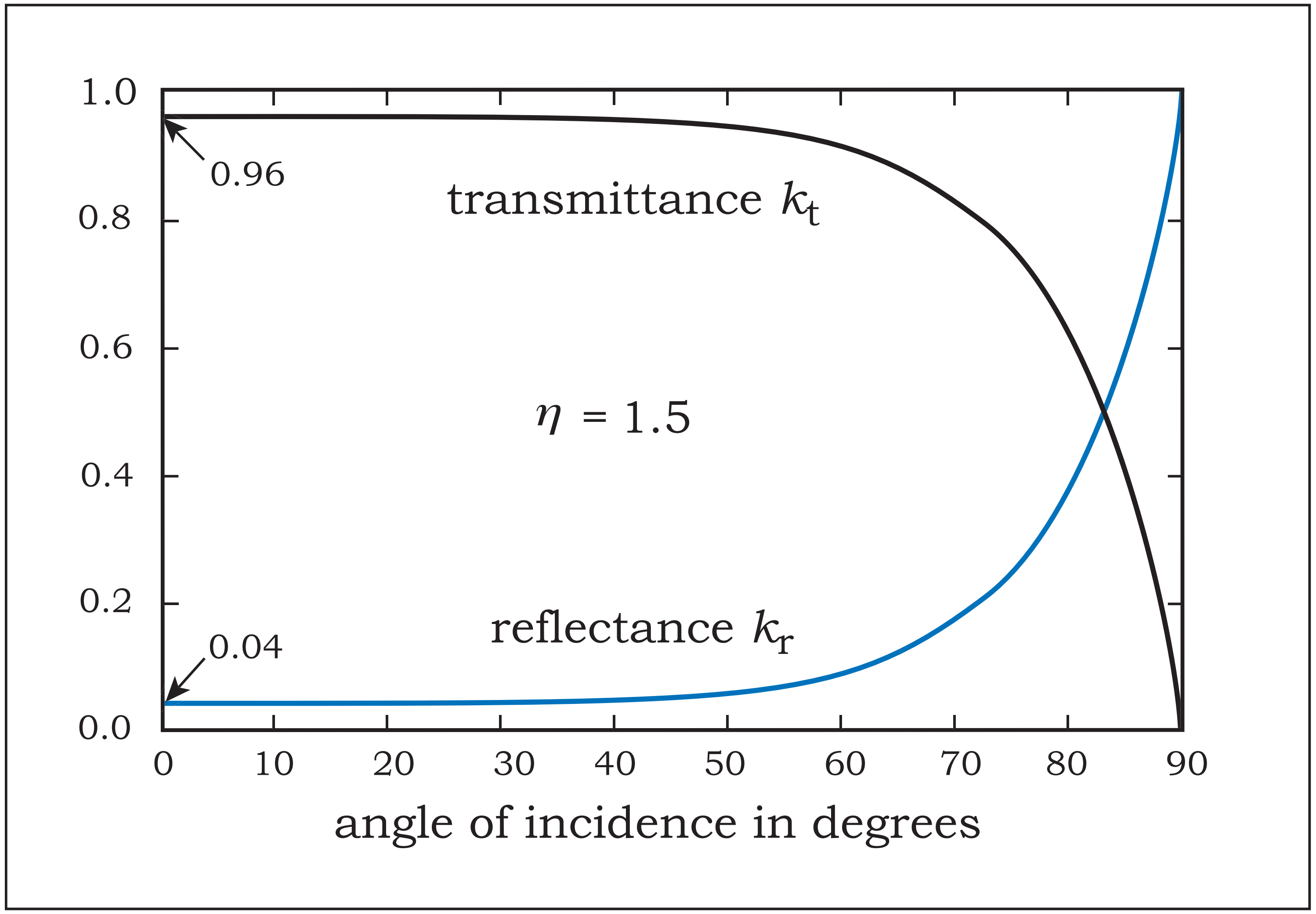
where

$\eta = \eta_t / \eta_i$ is the relative index of refraction

$r_{\parallel}$ is the reflected amplitude of light polarized parallel to the boundary

$r_{\perp}$ is the reflected amplitude of light polarized perpendicular to the boudary

Fresnel Equations



Fresnel Equations with Total Internal Reflection

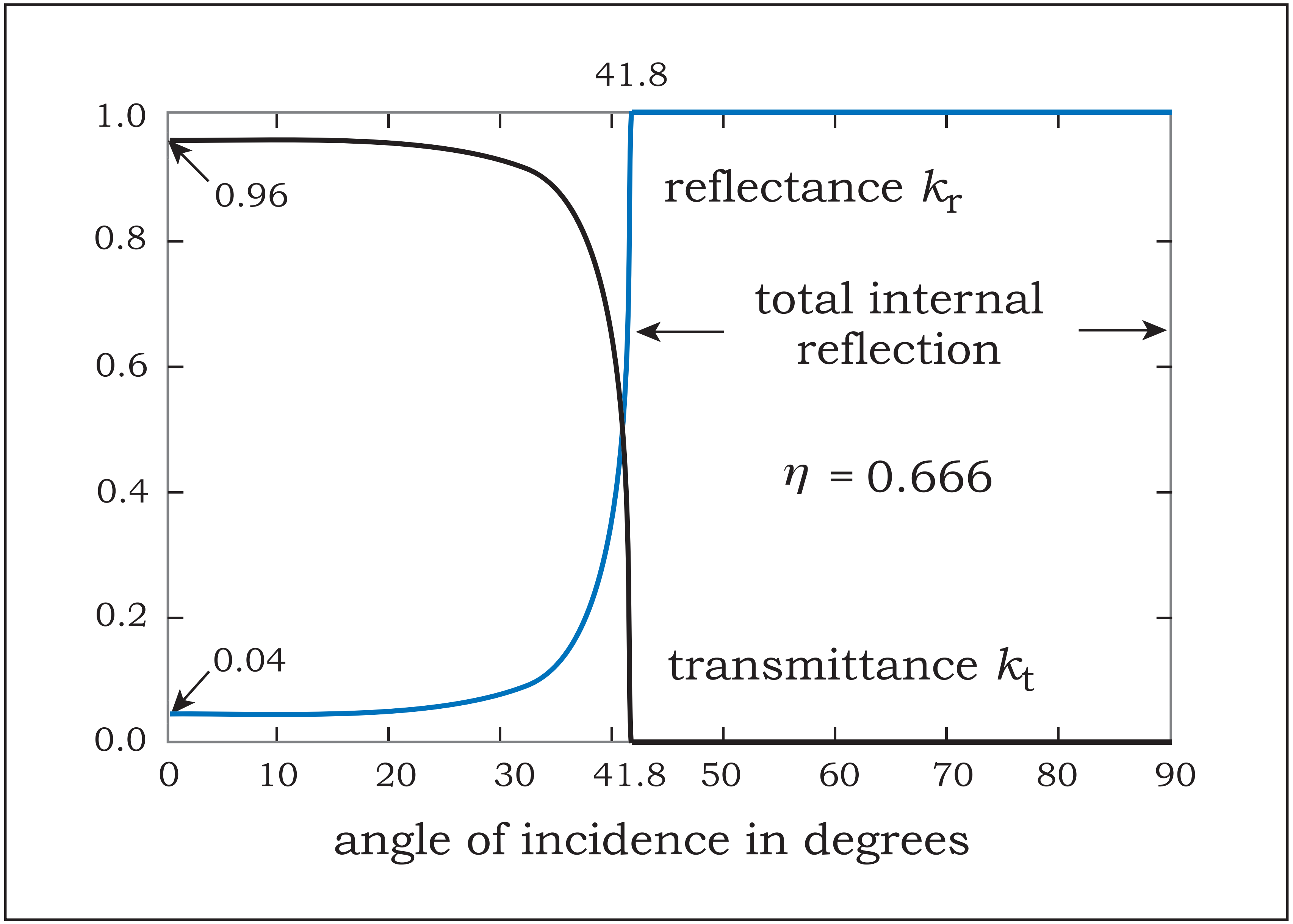


Image Generated Without Fresnel Equations

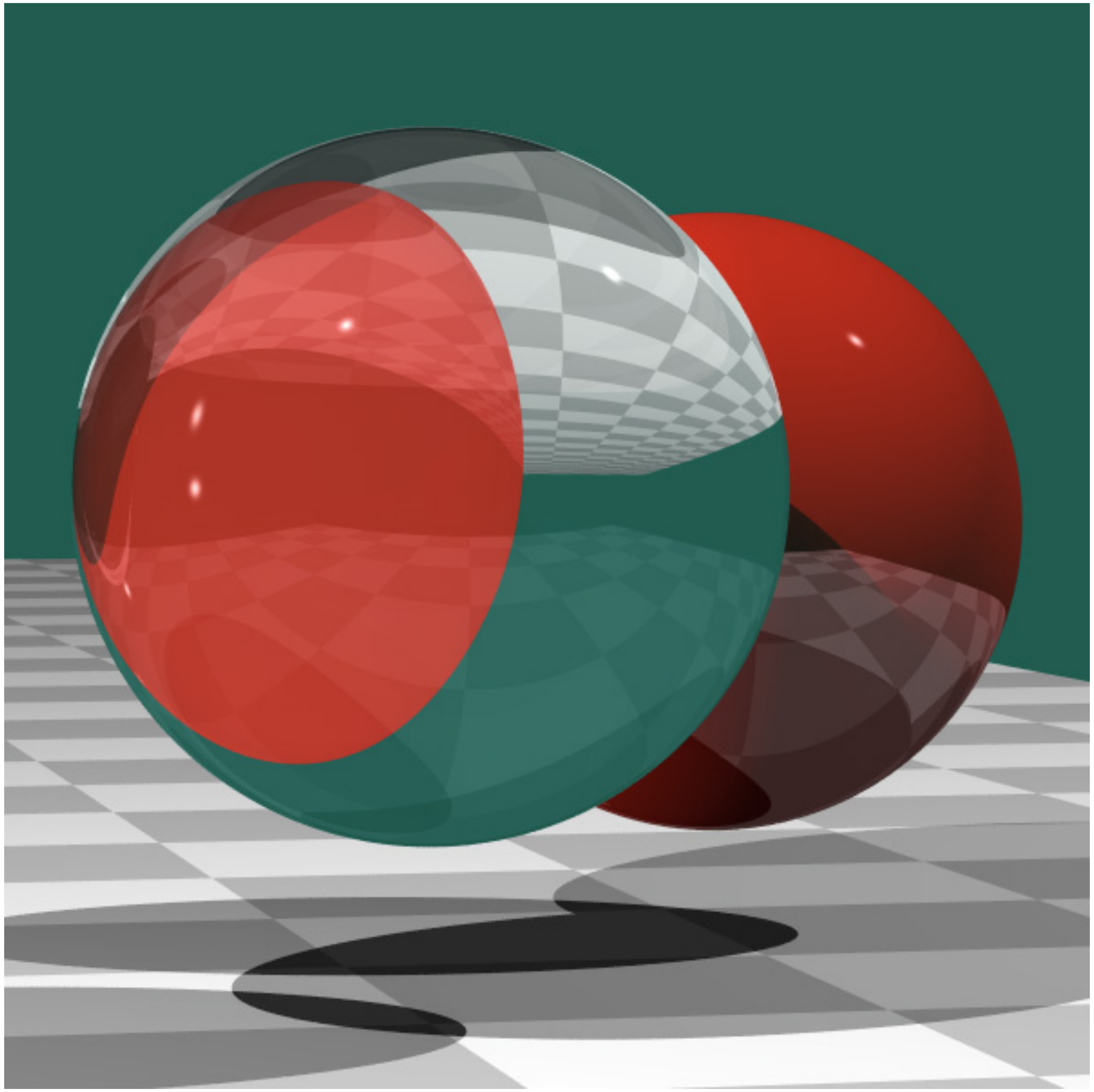
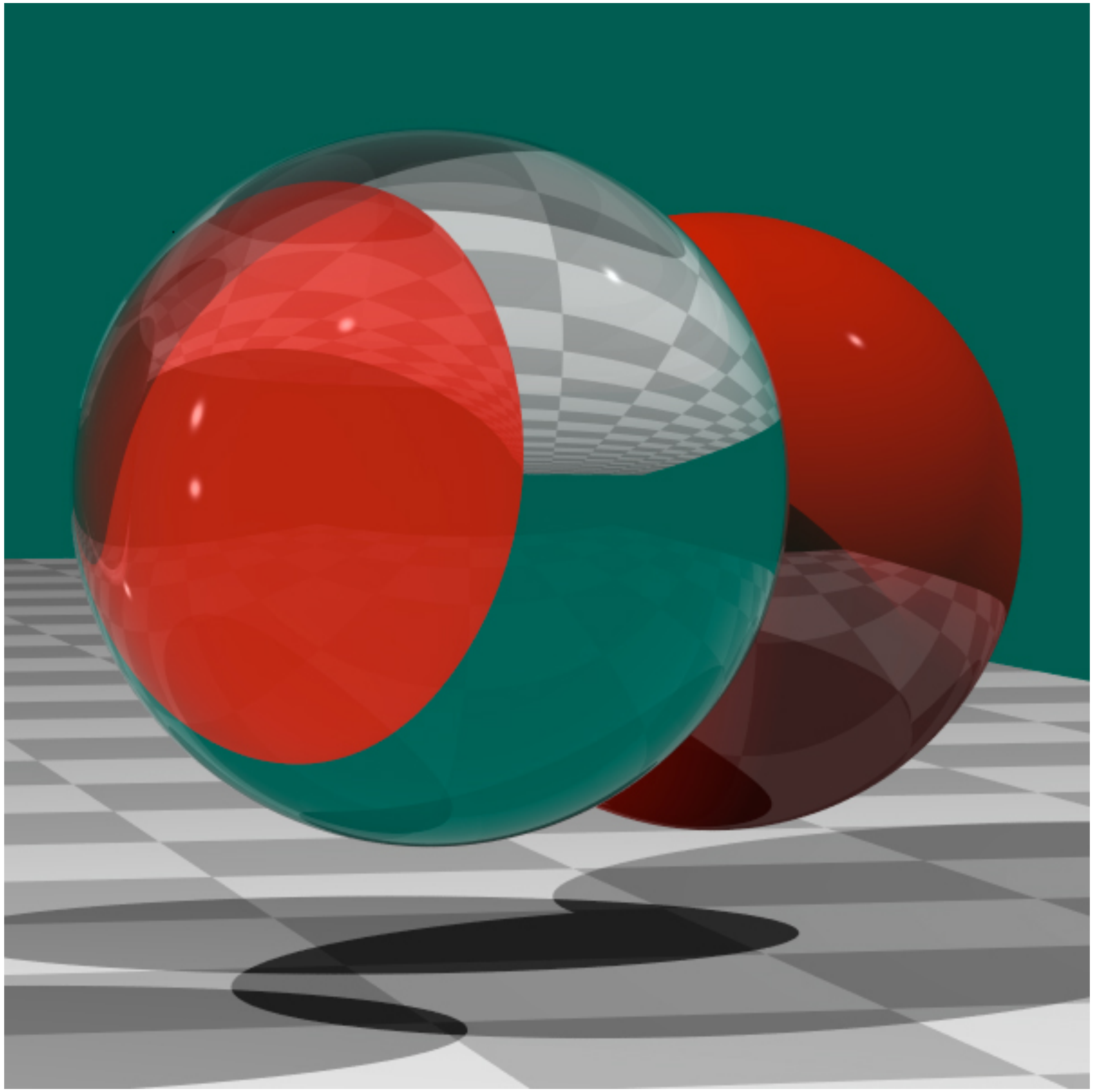


Image Generated With Fresnel Equation



Beer-Lambert Law

- When light travels though a dielectric, it gets attenuated.
(Light interacts with the dielectric's molecules and gets scattered.)

- The longer the distance traveled, the more attenuation.

- **Beer-Lambert Law** describes how much light gets attenuated with respected to distance:

$$\frac{dL}{L} = -\sigma \; dx$$

where σ is the attenuation coefficient, and x is the distance traveled.

- The solution to the above equation is:

$$L(x) = L_0 e^{-\sigma x}$$

where L_0 is the initial intensity of the light before going through the medium.

Beer-Lambert Law

- In a ray tracer, we rather specify the **filter color**

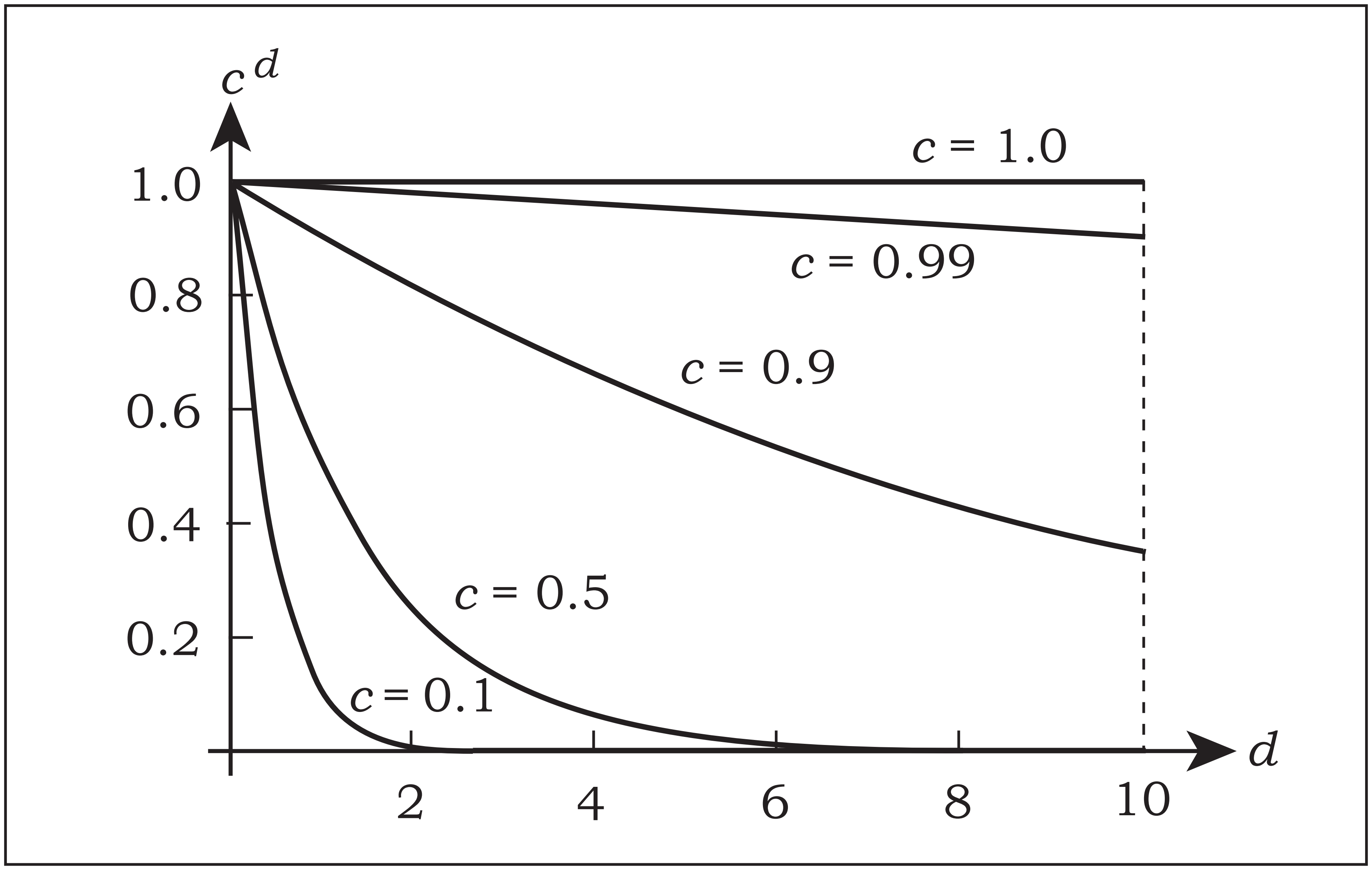
$$c_f = e^{-\sigma}$$

- The filter color tells what white light attenuates into if it travels through the medium by distance 1.

- The equation of the solution to Beer-Lambert law becomes

$$L(x) = (c_f)^x$$

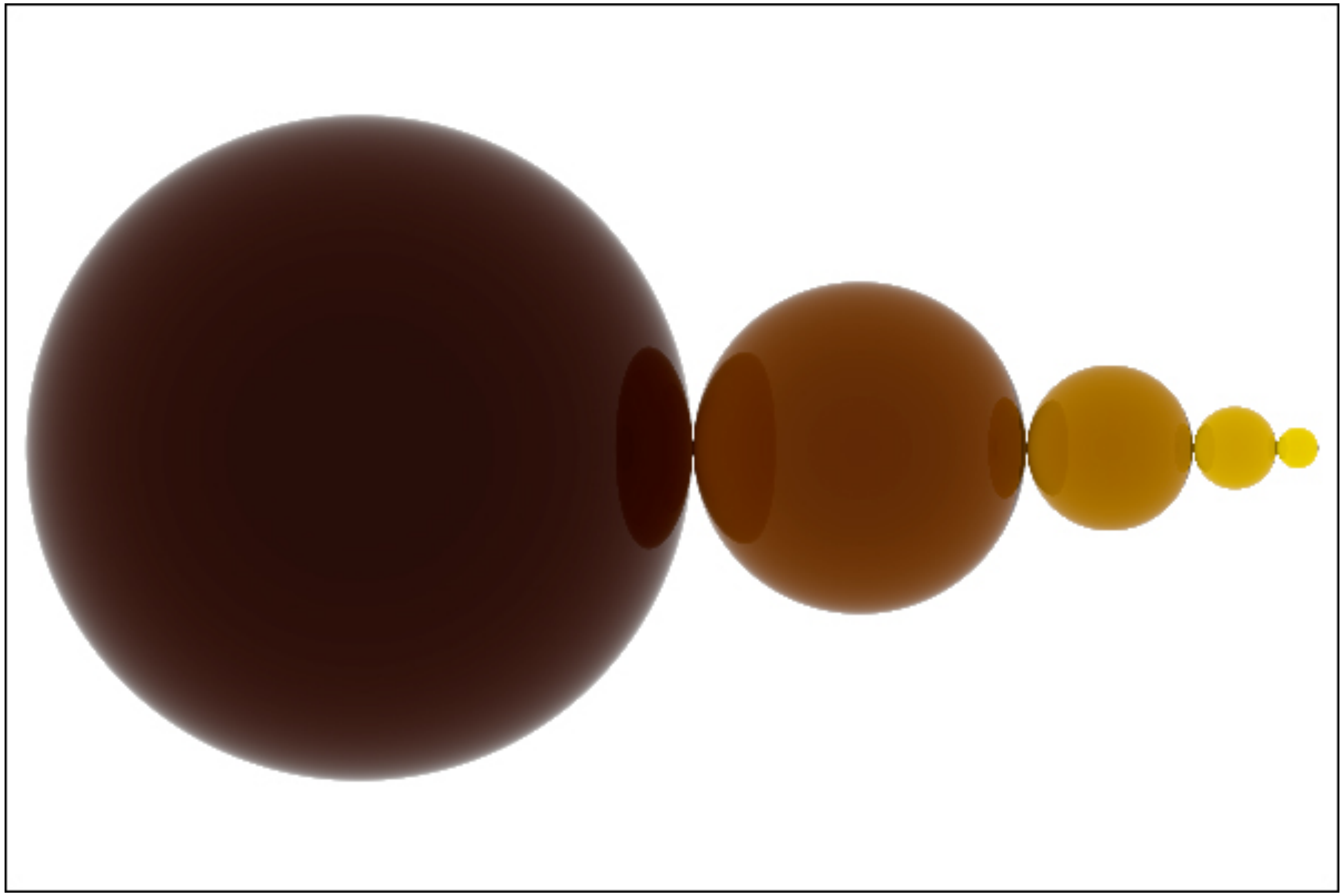
Beer-Lambert Law



Beer-Lambert Law

- When implementing the Beer-Lambert law
  - Need to keep track of the filter color of the medium the light is in.
  - Change the medium when casting refracted ray. (The medium changes.)
  - Use the time the ray travels as x because ray direction is always a unit vector.
  - Scale the color at the same time you scale it with $k_t \eta_t^2 / \eta_i^2$

Image Generated With Beer-Lambert Law



Pseudocode for trace_ray

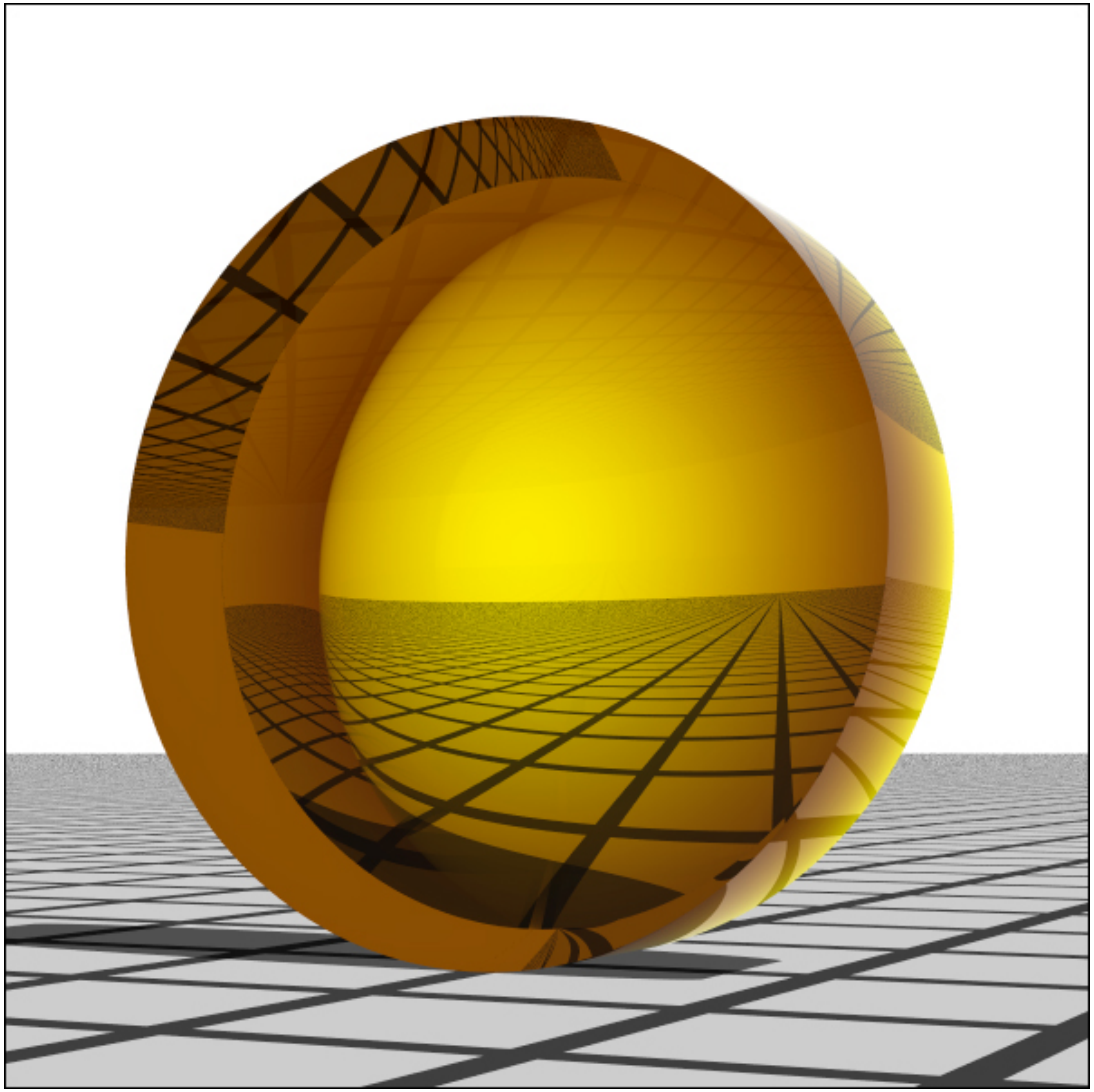
```
trace_ray(ray, level) {
  if level <= MAX_LEVEL {
    Find the first intersection point
    Shade direct illumination from light source

    if material is perfectly specular {
       ...
    }

    if material is perfectly transmittive {
      Generate refracted_ray
      refracted_color = trace_ray(reflected_ray, level+1)
      Scale refracted color by k_t(η_t^2/η_i^2) and filter color
      Shade the point again taking into account refracted color
    }

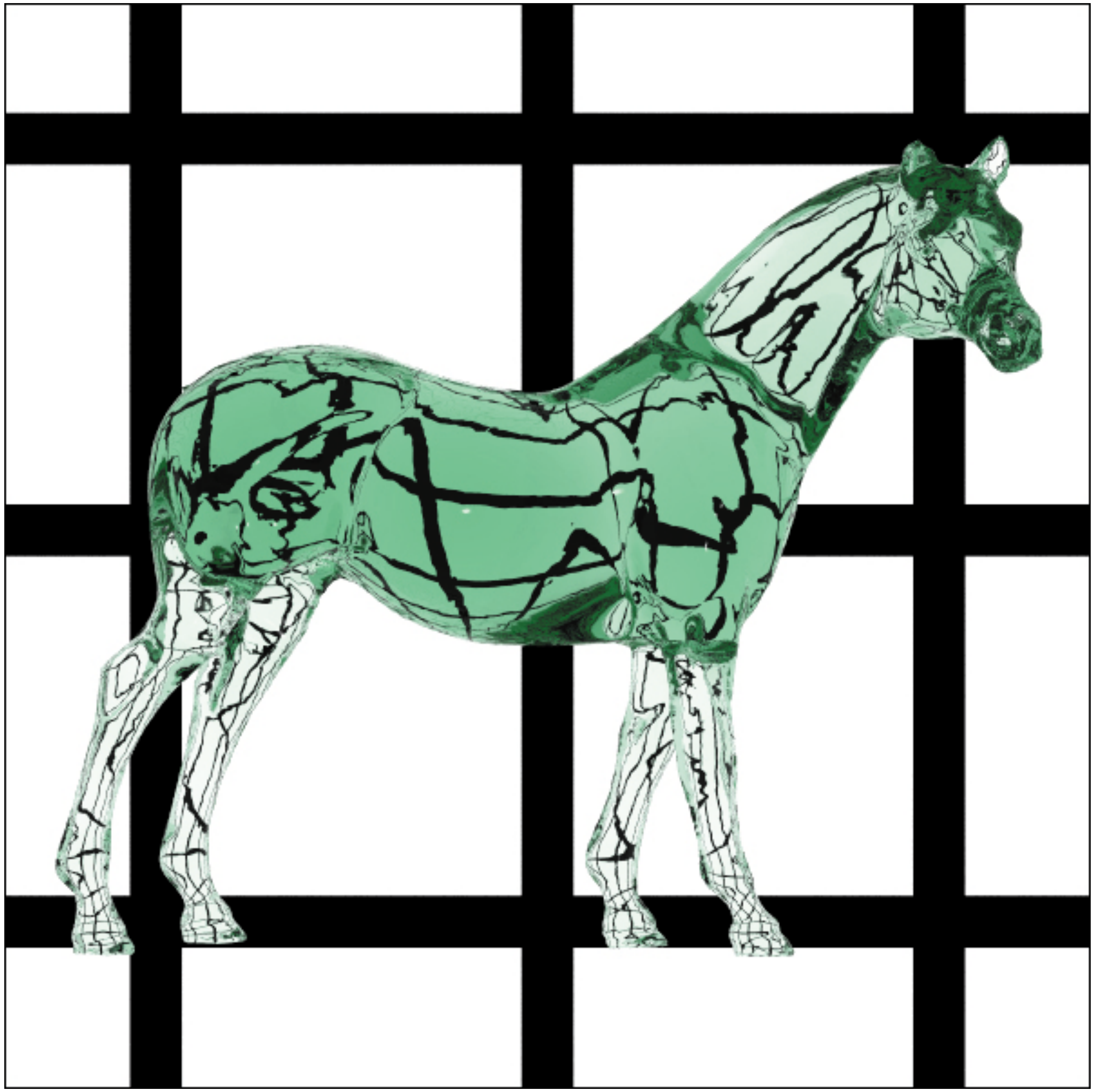
    return point's color
  }
  else
    return black
}
```

More Images with Refraction



More Images with Refraction

More Images with Refraction



More Images with Refraction