7 Basic Ray Tracing

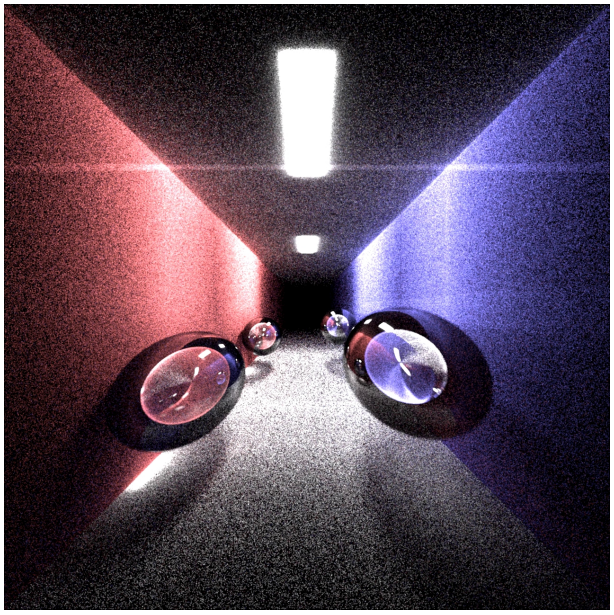
At this point we have a good understanding of the image formation process. We can set up a scene using geometric primitives and transformations, we know how to set up a camera at a specific location and orientation, and we can determine the projection from points on object surfaces to points on the image plane. We also know how to approximate object surface colours based on the lightsources in the scene and the material properties of the objects using the Phong illumination model.
This is sufficient to build a rendering engine that simulates the image formation process and that is able to realistically render a scene. The steps such a rendering engine would carry out are straightforward:

```
Forward Ray Tracer

1.0     Cast a ray from the lightsource (in a valid direction
        w.r.t. lightsource type)

2.0     Trace the ray - determine if it hits an object or the
        camera's aperture

3.0     If the ray hits an object

                3.1a  Determine whether the ray is bounced
                      or refracted
                3.2a  Determine the direction of the bounced
                      or refracted ray
                3.3a  Update the ray's colour depending on
                      the surface material
                3.4a  Trace the new ray

        If the ray hits the aperture

                3.1b  Determine the coordinates of the point
                      where the ray hits the image plane
                3.2b  Update the colour at that location by
                      adding the ray's colour to whatever value
                      is currently at that location

        else
                Stop tracing this ray, go back to 1.0
```

The above process describes a *Forward Ray Tracer*. It is a ray tracer because the core of the method

involves tracing the path of light rays from their origin point (which is either a light source or an object surface from where it has been bounced or refracted) checking whether the ray hits any object(s) in the scene, or the camera's lens. It is called a *forward* ray tracer because the tracing starts at the lightsource and follows rays as they make their way through the scene and into the camera.

The only tricky part of the above method is what happens when a ray hits an object. At this point, depending on how accurately we model the material of the object's surface, there could be a large number of possible outcomes for the ray. But leaving that aside for now, we can still render visually rich images using approximations such as the Phong model. The scene below is the output of a forward ray tracer, running on a simple scene with five walls and two spheres (the back wall is a mirror).



Render created with a forward ray tracer

Even with such a simple geometry, the forward ray tracer can render complex lighting effects such as transparency, reflections, caustics (light that is concentrated due to refracting objects), colour bleeding (notice the blue and red hues where the gray walls meet the colour ones), area light sources, and soft shadows. All of these effects are difficult to render with simple local models such as the Phong illumination model.

The downside of the forward ray tracer is its inordinate computational cost. The scene below required 50 billion light-source rays to be cast. This figure does not include all the rays generated from bouncing, refracting, and reflecting these 50 billion rays as they travel through the scene. The computational cost is this high because most of the rays being traced will not end at the camera. The likelihood that a ray that has been randomly bounced off some object in the scene will happen to have such a direction that it will end up at the camera's lense is tiny. In order to capture enough light rays to produce an image, we have to trace huge quantities of rays. Most of

the work done in tracing these rays is wasted. Despite the huge computational burden, the final scene still looks noisy and black speckles are visible in what should be smooth-coloured surfaces. These are locations for which no light rays were captured.

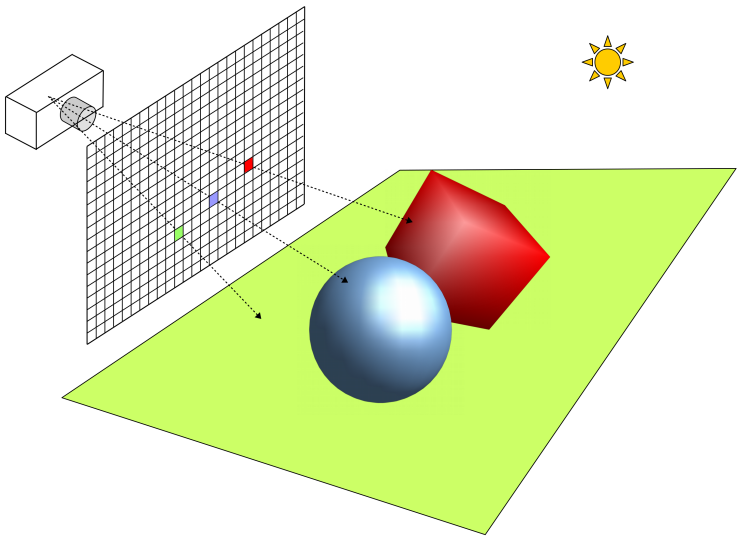
Due to their computational expense, forward raytracers are not used in practice for rendering complex scenes (they have some applications in real-time rendering that we will study later on). Instead, state-of-the-art rendering uses reverse ray tracing - which we will simply call *ray tracing* to produce very high quality images at a reasonable computational expense.

A scene rendered with a modern ray tracer. [Source: Wikipedia, author: Gilles Tran]

In this section we will examine in detail the process of ray tracing a scene. The ray tracer will use the Phong illumination model, and incorporate global illumination effects such as reflections, refraction, and shadows, that are not considered by local illumination models such as Phong.

7.1 The Ray Tracing Process

Assuming a pinhole camera, the colour at a pixel corresponds to a unique ray of light, arriving at the camera in a direction given by the line through the pixel and the center of projection as shown below.



Basic ray-tracing process. The colour at each pixel is the colour of the ray of light through that pixel and the center of projection

Therefore, to render a scene the ray tracer must figure out for each pixel in the image what the direction of the corresponding ray is, and the colour of that ray. The colour of the ray depends on where it originates. If the ray comes directly from the light source, it will have the colour of that lightsource. If it has been bounced off, reflected, or refracted by an object in the scene, then the colour will depend on the object's surface properties as well as the scene's lightsources, and may be affected by secondary illumination (light that has been reflected off other surfaces in the scene). The basic ray tracing algorithm, also known as Whitted ray tracing after its inventor, Turner Whitted, consists of the following:

```
For each pixel:
  1.1) Cast a ray from the eye of the camera through the pixel,
       and find the first surface hit by the ray.
  1.2) Determine the surface radiance at the surface intersection
       with a combination of local and global models.

       1.2.1) The local component is given by the Phong illimination
              model, evaluated at the surface in question.
       1.2.2) To estimate the global component, trace rays from
              the surface point to possible incident directions
              to determine how much light comes from each direction.
```

The above formulation leads to a recursive algorithm that follows rays of light from the camera back to a lightsource. In Whitted ray tracing, the global component involves only two additional

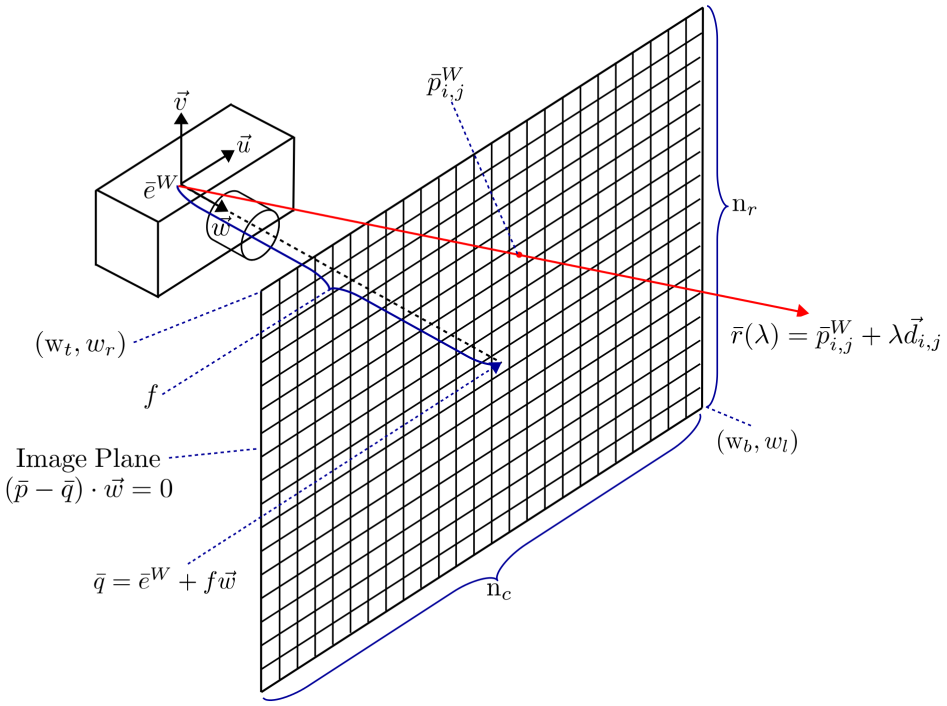
rays, one for handling perfect reflections, and one for handling refraction. We will now examine each of the steps involved in the raytracing process.

Computational Issues

- Forming rays
- Finding ray intersections with objects
- Finding the closest object intersection
- Determining surface normals at the intersection
- Evaluating reflectance models at the intersection

7.2 Ray Casting

We want to find the ray from the eye through pixel (i, j). First we must determine the loction of this pixel in *world coordinates*. All further computations will take place in the world coordinate frame in which objects and light sources are defined.



Geometry of ray casting. We need to determine the origin in world coordinates $\bar{p}_{i,j}$, and direction $\vec{d}_{i,j}$ of a ray through the pixel at (i, j)

- Camera Model
  $\bar{e}^W$ is the origin of the camera, in world space.
  $\vec{u}$, $\vec{v}$, and $\vec{w}$ are the world space directions corresponding to the $\vec{x}$, $\vec{y}$, and $\vec{z}$ axes in eye space.
  The image plane is defined by $(\bar{p} - \bar{q}) \cdot \vec{w} = 0$, or $\bar{q} + a\vec{u} + b\vec{v}$, where $\bar{q} = \bar{e}^W + f\vec{w}$.

- Window
  A window in the view-plane is defined by its boundaries in camera coordinates: w_l, w_r, w_t, and w_b. (In other words, the left-most edge is the line (w_l, λ, f).)

- Viewport
  Let the viewport (i.e., output image) have columns $0...n_c - 1$ and rows $0...n_r - 1$ (i.e. the image size is $n_r \times n_c$ pixels). $(0, 0)$ is the upper left entry.
  The camera coordinates of pixel (i, j) are as follows:

$$\bar{p}^C_{i,j} = (w_l + i\Delta u, w_t + j\Delta v, f)$$

$$\Delta u = \frac{w_r - w_l}{n_c - 1}$$

$$\Delta v = \frac{w_b - w_t}{n_r - 1}$$

  In world coordinates, this is:

$$\bar{p}^W_{i,j} = \begin{pmatrix} | & | & | \\ \vec{u} & \vec{v} & \vec{w} \\ | & | & | \end{pmatrix} \bar{p}^C_{i,j} + \bar{e}^W$$

- Ray: Finally, the ray is then defined in world coordinates as follows:

$$\bar{r}(\lambda) = \bar{p}^W_{i,j} + \lambda\vec{d}_{i,j}$$

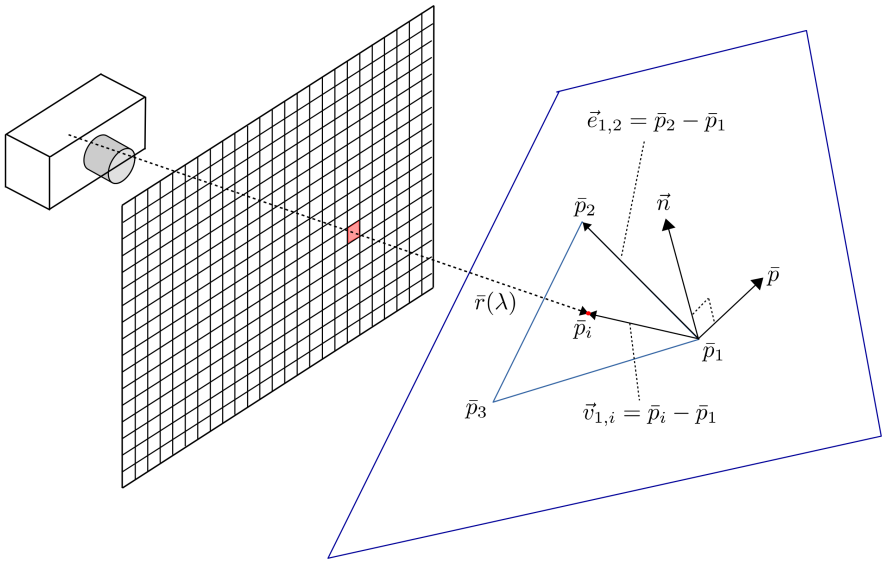
  where $\vec{d}_{i,j} = \bar{p}^W_{i,j} - \bar{e}^W$. For $\lambda > 0$, all points on the ray lie in front of the viewplane along a single line of sight.

7.3 Intersections

We now have a ray through some pixel in our image, the next step is to figure out whether that ray hits any of the objects or light sources in the scene. This means that we need to check for intersection between the ray, denoted by $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d}$, $\lambda > 0$, and each object or area light source. The intersection test will depend on the type of object. Here we develop intersection tests for the most common primitive shapes. We will then show how to apply these simple tests to affinely deformed shapes so we can handle generalized versions of graphics primitives and thus render a scene of arbitrary complexity.

7.3.1 Triangles

Complex shapes in computer graphics are very often modelled as triangle meshes. Therefore, solving for the intersection between a ray and a triangle should be our first task.
Define a triangle with three points, $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$.



Testing for intersection between a ray and a triangle

Here are two ways to solve for the ray-triangle intersection.

- Intersect $\bar{r}(\lambda)$ with the plane $(\bar{p} - \bar{p}_1) \cdot \vec{n} = 0$ for $\vec{n} = (\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)$ by substituting $\bar{r}(\lambda)$ for $\bar{p}$ and solving for λ. That is:

$$(\bar{a} + \lambda\vec{d} - \bar{p}_1) \cdot \vec{n} = 0$$

$$\lambda^* = \frac{(\bar{p}_1 - \bar{a}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$$

  The intersection point $\bar{p}_i = \bar{r}(\lambda^*)$.

  > *Aside:*
  > What does it mean when $\vec{d} \cdot \vec{n} = 0$?
  > What does it mean when $\vec{d} \cdot \vec{n} = 0$ and $(\bar{p}_1 - \bar{a}) \cdot \vec{n} = 0$?

  Once the intersection is found, test whether it happens inside the triangle. To do this, we check against each edge as follows: Take the edge $\vec{e}_{1,2} = \bar{p}_2 - \bar{p}_1$, and the vector $\vec{v}_{1,i} = \bar{p}_i - \bar{p}_1$. The intersection point is inside with respect to edge $\vec{e}_{1,2}$ if $(\vec{e}_{1,2} \times \vec{v}_{i,1}) \cdot \vec{n} \geq 0$. Similarly for the remaining two edges.

- An alternate method is to solve for α and β where $\bar{p}(\alpha,\beta)\bar{p}_1 + \alpha(\bar{p}_2 - \bar{p}_1) + \beta(\bar{p}_3 - \bar{p}_1)$, i.e. $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d} = \bar{p}_1 + \alpha(\bar{p}_2 - \bar{p}_1) + \beta(\bar{p}_3 - \bar{p}_1)$. This leads to the 3x3 system

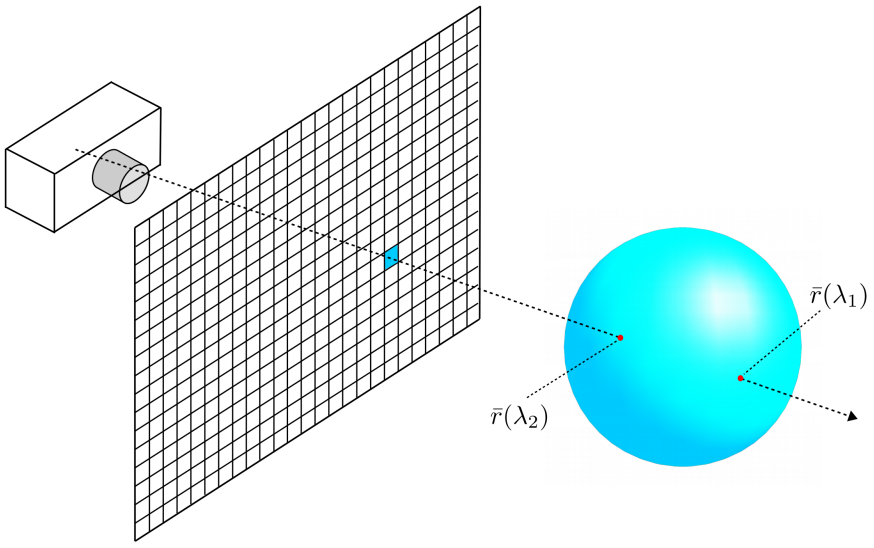
$$\begin{pmatrix} | & | & | \\ -(\bar{p}_2 - \bar{p}_1) & -(\bar{p}_3 - \bar{p}_1) & \vec{d} \\ | & | & | \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \lambda \end{pmatrix} = (\bar{p}_1 - \bar{a})$$

  Invert the matrix and solve for α, β, and λ. The intersection is in the triangle when the following conditions are all true:

$$\alpha \geq 0$$
$$\beta \geq 0$$
$$\alpha + \beta \leq 1$$

7.3.2 Spheres

Define the unit sphere centered at $\bar{c}$ by $||\bar{p} - \bar{c}||^2 = 1$.



A ray can intersect a sphere at two different points

Substitute a point on the ray $\bar{r}(\lambda)$ into this equation:

$$(\bar{a} + \lambda\vec{d} - \bar{c}) \cdot (\bar{a} + \lambda\vec{d} - \bar{c}) - 1 = 0$$

Expand this equation and write it in terms of the quadratic form:

$$A\lambda^2 + 2B\lambda + C = 0$$
$$A = \vec{d} \cdot \vec{d}$$
$$B = (\bar{a} - \bar{c}) \cdot \vec{d}$$
$$C = (\bar{a} - \bar{c}) \cdot (\bar{a} - \bar{c}) - 1$$

The solution is then:

$$\lambda = \frac{-2B \pm \sqrt{4B^2 - 4AC}}{2A} = -\frac{B}{A} \pm \frac{\sqrt{D}}{A}, D = B^2 - AC$$

If $D < 0$, there are no intersections. If $D = 0$, there is one intersection; the ray grazes the sphere. If $D > 0$, there are two intersections with two values for λ, λ_1 and λ_2.

When $D > 0$, three cases of interest exist:

- $\lambda_1 < 0$ and $\lambda_2 < 0$. Both intersections are behind the view-plane, and are not visible.
- $\lambda_1 > 0$ and $\lambda_2 < 0$. The $\bar{p}(\lambda_1)$ is a visible intersection, but $\bar{p}(\lambda_1)$ is not.
- $\lambda_1 > \lambda_2$ and $\lambda_2 > 0$. Both intersections are in front of the view-plane. $\bar{p}(\lambda_2)$ is the closest intersection.

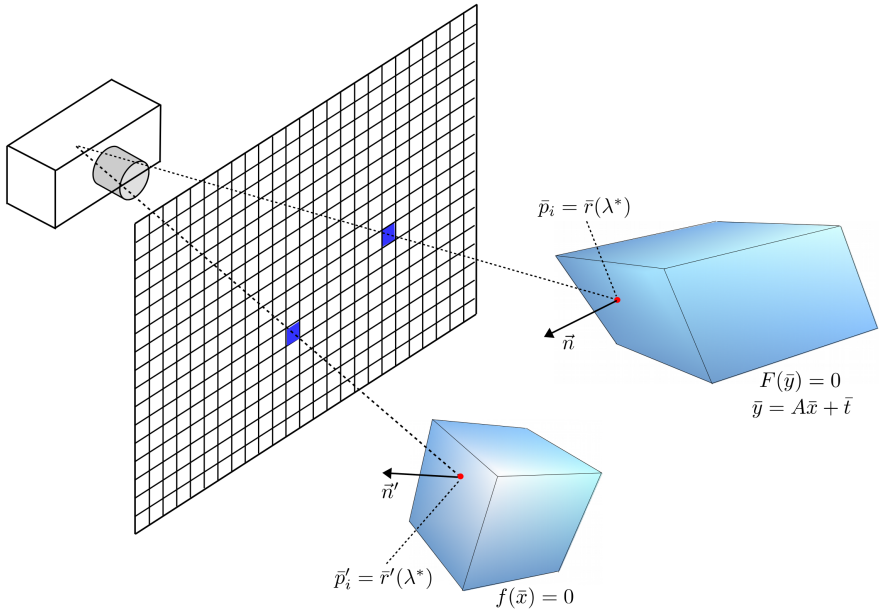
7.3.3 Cylinders and Cones

A right-circular cylinder may be defined by $x^2 + y^2 = 1$ for $|z| \leq 1$. A cone may be defined by $x^2 + y^2 - \frac{1}{4}(1 - z)^2 = 0$ for $0 \leq z \leq 1$.

- Find intersection with "quadratic wall," ignoring constraints on z, e.g. using $x^2 + y^2 = 1$ or $x^2 + y^2 - \frac{1}{4}(1 - z^2) = 0$. Then test the z component of $\bar{p}(\lambda^*)$ against the constraint on z, e.g. $z \leq 1$ or $z < 1$.
- Intersect the ray with the planes containing the base or cap (e.g. $z = 1$ for the cylinder). Then test the x and y components of $\bar{p}(\lambda^*)$ to see if they satisfy interior constraints (e.g. $x^2 + y^2 < 1$ for the cylinder).
- If there are multiple intersections, then take the intersection with the smallest positive λ (i.e., closest to the start of the ray).

7.3.4 Affinely Deformed Objects

Given our set of intersection tests for simple 3D primitive shapes. We will now see how we can use these same tests to handle any affinely transformed version of our basic shapes, therefore enabling us to render any scenes that can bee created by combining suitably transformed shapes with ease.
Proposition: Given an intersection method for an object, it is easy to intersect rays with affinely deformed versions of the object. We assume here that the affine transformation is invertible.



A canonical objects defined by the equation $f(\bar{x}) = 0$ and its affinely transformed version $F(\bar{y}) = 0$. We can compute the intersection point between a ray and the deformed object by using the intersection test for the canonical shape with a suitably transformed ray.

- Let $F(\bar{y}) = 0$ be the deformed version of $f(\bar{x}) = 0$, where $\bar{y} = \mathbf{A}\bar{x} + \vec{t}$.
  i.e. $F(\bar{y}) = f(\mathbf{A}^{-1}(\bar{y} - \vec{t})) = 0$, so $F(\bar{y}) = 0$ iff $f(\bar{x}) = 0$.

- Given an intersection method for $f(\bar{x}) = 0$, find the intersection of $\bar{r}(\lambda) = \bar{a} + \lambda\vec{d}$ and $F(\bar{y}) = 0$, where $\lambda > 0$.

- **Solution:** Substitute $\bar{r}(\lambda)$ into the implicit equation $f = F(\bar{y})$:

$$\begin{aligned} F\left(\bar{r}(\lambda)\right) &= f\left(\mathbf{A}^{-1}\left(\bar{r}\left(\lambda\right) - \vec{t}\right)\right) \\ &= f(\mathbf{A}^{-1}(\bar{a} + \lambda\vec{d} - \vec{t})) \\ &= f(\bar{a}' + \lambda\vec{d}') \\ &= f(\bar{r}'(\lambda)) \end{aligned}$$

  where

$$\begin{aligned} \bar{a}' &= \mathbf{A}^{-1}(\bar{a} - \vec{t}) \\ \vec{d}' &= \mathbf{A}^{-1}\vec{d} \end{aligned}$$

  i.e. intersecting $F(\bar{y})$ with $\bar{r}(\lambda)$ is equivalent to intersecting $f(\mathbf{x})$ with $\bar{r}'(\lambda) = \bar{a}' + \lambda\vec{d}'$ where $\lambda > 0$. The value of λ at the intersection point is the same in both cases.

- **Exercise:** Verify that, at the solution λ^*, with an affine deformation $\bar{y} = \mathbf{A}\bar{x} + \vec{t}$, that $\bar{r}(\lambda^*) = \mathbf{A}\bar{r}'(\lambda^*) + \vec{t}$.

7.4 Surface Normals at Intersection Points

Once we have found intersection between a ray and the closest object in the scene along its direction, we need to know the surface orientation at the intersection point. This will be needed to perform the shading computations that will determine the surface colour. In short: We cast a ray out from a pixel and find the first surface hit, and then we want to know how much light leave the surface along the same ray but in the reverse direction, back to the camera.
Toward this end, one critical property of the surface geometry that we need to compute is the surface normal at the intersection point.

- For mesh surfaces, we can interpolate smoothly from face normals. This assumes the underlying surface is smooth.
- Otherwise we can just use the face normal.
- For smooth surfaces (e.g. with implicit forms $f(\bar{x}) = 0$ or parametric forms $\mathbf{s}(\alpha, \beta)$), take

$$\vec{n} = \frac{\nabla f(\bar{x})}{||\nabla f(\bar{x})||}$$

or

$$\vec{n} = \frac{\frac{\partial \mathbf{s}}{\partial \alpha} \times \frac{\partial \mathbf{s}}{\partial \beta}}{||\frac{\partial \mathbf{s}}{\partial \alpha} \times \frac{\partial \mathbf{s}}{\partial \beta}||}.$$

as the case requires.

7.4.1 Normals for affinely-deformed surfaces

Let $f(\bar{x}) = 0$ be an implicit surface, and let $F(\bar{y}) = \mathbf{A}\bar{x} + \vec{t}$ be an affine transformation, where $\mathbf{A}$ is invertible. The affinely-deformed surface is

$$F(\bar{y}) = f(Q^{-1}(\bar{x})) = f(\mathbf{A}^{-1}(\bar{x} - \vec{t})) = 0 \tag{1}$$

A normal of F at a point $\bar{q}$ is given by

$$\vec{n} = \frac{\mathbf{A}^{-T}\vec{n}'}{||\mathbf{A}^{-T}\vec{n}'||} \tag{2}$$

where $\mathbf{A}^{-T} = (\mathbf{A}^{-1})^T$ and $\vec{n}'$ is the normal of f at $\bar{p} = Q^{-1}(\bar{q})$.

1*Derivation:*
Let $\bar{s} = \bar{r}(\lambda^*)$ be the intersection point, and let $(\bar{p} - \bar{s}) \cdot \vec{n}' = 0$ be the tangent plane at the intersection point. We can also write this as:

$$(\bar{p} - \bar{s})^T \vec{n}' = 0 \tag{3}$$

Substituting in $\bar{q} = \mathbf{A}\bar{p} + \vec{t}$ and solving gives:

$$(\bar{p} - \bar{s})^T \vec{n}' = (\mathbf{A}^{-1}(\bar{q} - \vec{t}) - \bar{s})^T \vec{n}' \quad (4)$$

$$= (\bar{q} - (\mathbf{A}\bar{s} + \vec{t}))^T \mathbf{A}^{-T} \vec{n}' \quad (5)$$

In other words, the tangent plane at the transformed point has normal $\mathbf{A}^{-T}\vec{n}'$ and passes through point $(\mathbf{A}\bar{s} + \vec{t})$. After normalization to ensure the normal has unit length, we thus obtain $\vec{n} = \frac{\mathbf{A}^{-T}\vec{n}'}{||\mathbf{A}^{-T}\vec{n}'||}$.

```
Ray casting and intersection procedure: finds the closest
intersection between a ray and any objects in the scene.

  1.1) Form ray through the pixel at (i,j).

  1.2) Initialize closest_lambda=Inf, closest_normal=[0 0 0],
       closest_intersection=[-Inf -Inf -Inf]

  1.3) For each object in the scene:

     1.3.1) Compute transformed ray using the inverse object's
            transformation
     1.3.2) Compute the intersection between the transformed
            ray and the corresponding canonical object. Obtain
            the smallest value of lambda > 0, and the corresponding
            intersection, get the normal at the intersection point

     1.3.3) If (lambda < closest_lambda)

        1.3.3.1) Substitute the lambda obtained in 1.2.2 in the
                 equation for the original ray to find the intersection
                 with the deformed object, update closest_intersection
        1.3.3.2) Compute the normal for the deformed object and update
                 closest_normal
        1.3.3.3) Update closest_lambda=lambda
```

7.5 The Scene Signature

Ray casting and intersection testing are the fundamental ray tracing operations. In order to verify they are working correctly, we generate what is called a *scene signature*.

Scene signature for a scene composed of a fractal arrangement of spheres, a ground plane, and a back plane. The colour for each object was chosen randomly. If the ray casting and intersection code is correct, all objects will appear at the right location, and have the correct size and shape. There should be no unexpected breaks, holes, or corrupted image regions.

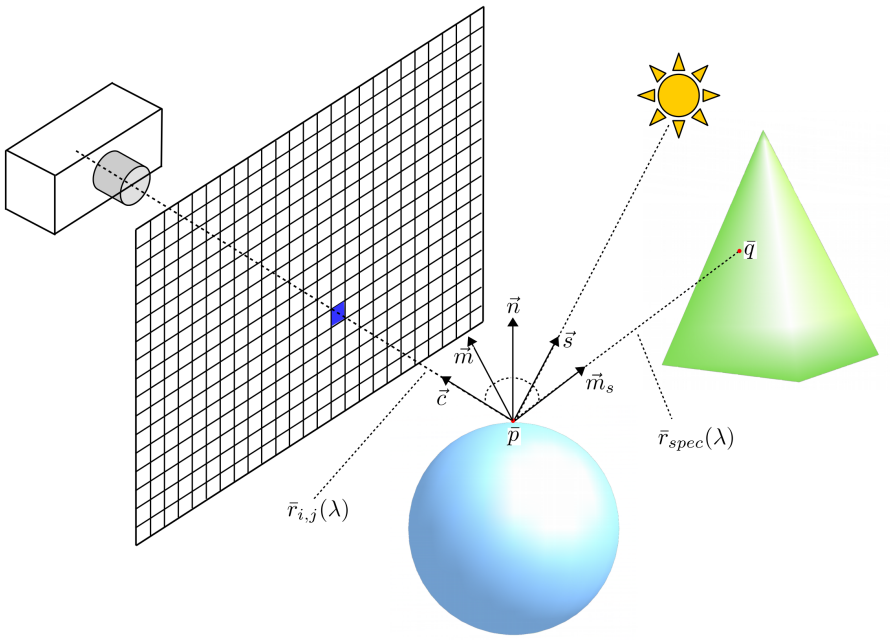
We first design a scene for which the expected layout of the objects in the corresponding image is easy to verify. We then assign each object a fixed, constant colour (preferably unique). At this point we use the ray casting and intersection procedure described above to render an image in which the colour at each pixel is the constant colour assigned to the closest object intersected by the corresponding ray (or a background colour if the ray does not intersect any object in the scene). If the scene signature is correct, we can proceed to implement the shading component of the ray-tracer which will assign an actual colour to each object based on its material properties and the scene's illumination.

7.6 Shading

Once we have deetermined the scene surface that is closest to the camera in the direction of the ray, we need to deyermine the colour and intensity of the light that leaves that surface in the direction of the pixel.
This is a complex task. The colour and intensity of light along the ray is determined by the properties of the material, and the light arriving at that specific surface point from any other location in the scene. This includes direct illumination from lightsources, as well as indirect illumination resulting from light reflected off other scene surfaces. Additionally, if the surface has any transparency, we must consider light refracted by the object as well.
As we saw before, we can approximate the local appearance of a surface via the Phong illumination model. We will now develop the simplest model of shading for ray tracing, also known as *Whitted* ray tracing. It uses the Phong illumination model, and adds to it global illumination effects that include shadows, secondary specular reflection, and refraction.

7.6.1 Basic (Whitted) Ray Tracing

In Whitted ray tracing we assume that that the light reflected from the surface is a combination of the intensity computed by the Phong model, along with one component due to perfect specular reflection. That is, we know that the only incoming light at $\bar{p}$ that will be reflected in the direction $-\vec{d}_{i,j}$ will be that coming from the corresponding mirror direction (i.e., $\vec{m}_s = -2(\vec{d}_{i,j} \cdot \vec{n}))\vec{n} + \vec{d}_{i,j}$). We can find out how much light is incoming from direction $\vec{m}_s$ be casting another ray into that direction from $\bar{p}$ and calculating the light reflected from the first surface hit.
Note that this secondary specular reflection is different from the *specular component* of the Phong model which gives you specular highlights only and does not require ray casting but depends exclusively on local properties of the surface.
Because of the need to cast additional rays in the specular direction (and later on we will need to do a similar thing also for refraction) the ray tracing process becomes recursive: In order to compute light colour and intensity at a point, we must first determine the same quantities for other scene points that reflect light onto it.



Geometry of Whitted ray tracing. All vectors are unit length, and can be determined from the geometry of the surface at the intersection point. The colour and intensity of light traveling back to the camera along ray $\bar{r}_{i,j}(\lambda)$ is computed using the Phong illumination model, plus the intensity of light arriving at point $\bar{p}$ from the direction of $\bar{r}_{spec}(\lambda)$ which is obtained via a recursive call to the raytracing procedure.

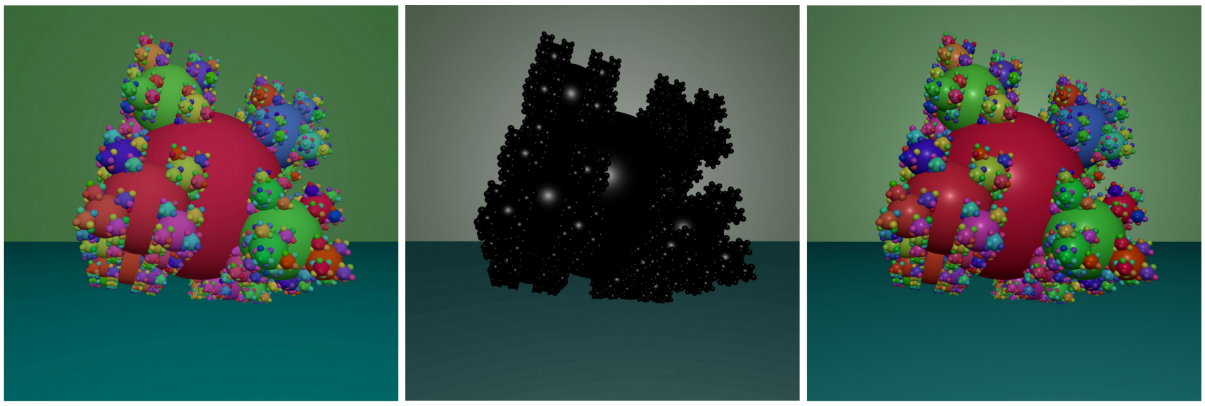
In summary, for basic (Whitted) ray tracing, the reflectance model calculation comprises:

- A local model (e.g., Phong) to account for diffuse and off-axis specular reflection (highlights) due to light sources.
- An ambient term to approximate the global diffuse components.
- Cast rays from $\bar{p}$ into direction $\vec{m}_s = -2(\vec{d}_{i,j} \cdot \vec{n})\vec{n} + \vec{d}_{i,j}$ to estimate ideal mirror reflections due to light coming from other ojects (i.e., secondary reflection).

For a ray $r(\lambda) = \bar{a} + \lambda\vec{d}$ which hits a surface point point $\bar{p}$ with normal $\vec{n}$, the reflectance is given by

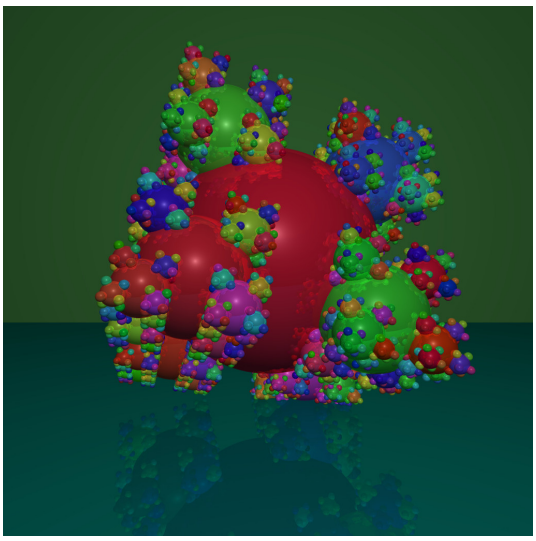
$$E = r_a I_a + r_d I_d \max(0, \vec{n} \cdot \vec{s}) + r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha + r_g I_{spec}$$

where r_a, r_d, and r_s are the reflection coefficients of the Phong model, i.e. they define the surface's reflectance properties as far as the ray tracer is concerned; I_a, I_d, and I_s are the light source intensities for the ambient, diffuse and specular terms of the Phong model (typically I_d and I_s are the same, and correspond the intensity of a light source in the scene); $\vec{s}$ is the light source direction from $\bar{p}$, $\vec{c} = -\vec{d}_{i,j}$ is the direction from $\bar{p}$ to the camera, and $\vec{m} = 2(\vec{s} \cdot \vec{n}))\vec{n} - \vec{s}$ is the perfect mirror direction with respect to the lightsource.



Local components of the shading model. From left to right: Diffuse component, specular component, and full Phong illumination.

Finally, I_{spec} is the light obtained from the recursive ray trace call for a ray from $\bar{p}$ travelling in direction $\vec{m}_s$ and gives the model some ability to account for secondary illumination. The amount of secondary reflection is controlled by r_g - an additional material property.



Local Phong model plus secondary reflections

Note:

As was previously discussed, we handle colour by representing both light sources and materials in terms of their RGB colour values. Therefor, in practice, we need

to evaluate the shading equation above 3 times, once for each colour component. The light intensities and albedos will be defined as RGB triplets. For example, $I_d = [I_{dR} \;\; I_{dG} \;\; I_{dB}]$, $r_d = [r_{dR} \;\; r_{dG} \;\; r_{dB}]$, and the remaining quantities are defined in an analogous way.

For multiple light sources, the final value for radiance returning along the ray from $\bar{p}$ to the camera is computed as the sum of the contributions from each light source to the total light intensity in direction $\vec{c}$. That is, for a scene with L lightsources:

$E = r_a Ia + \sum_{k=1}^{L} (r_d I_{d,k} \max(0, \vec{n} \cdot \vec{s}_k) + r_s I_{s,k} \max(0, \vec{c} \cdot \vec{m}_k)^\alpha) + r_g I_{spec}$

where $\vec{s}_k$ is the unit vector pointing from $\bar{p}$ to lightsource k, $\vec{m}_k$ is the corresponding direction of mirror reflection, $I_{d,k}$ is the diffuse albedo for this lightsource, and $I_{s,k}$ is the corresponding specular albedo.

7.6.2 Shadows

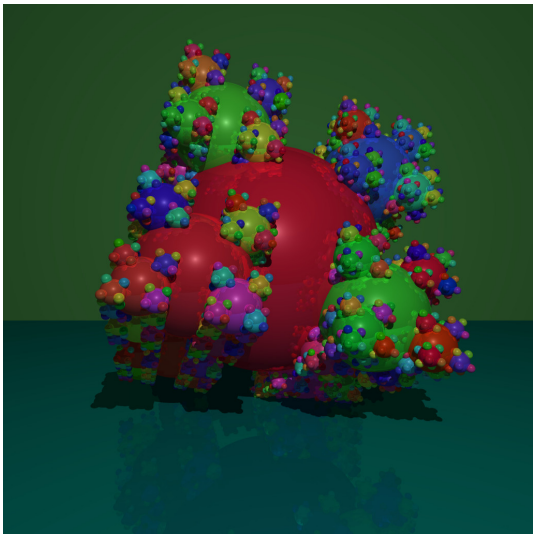
The next step in building the ray tracer is to correctly account for shadows. Shadows occur wherever lightsources are blocked from illuminating a surface by other objects in the scene. Therefore, shadows can easily be implemented by casting a ray from $\bar{p}$ in the direction of the lightsource:

$\bar{r}_{shade}(\lambda) = \bar{p} + \lambda(\bar{l} - \bar{p})$

where $\bar{l}$ is the lightsource's position. Note that the direction vector for this ray $\vec{d}_{shade} = (\bar{l} - \bar{p})$ is **not** unit length. We test for intersection between this *shadow ray* and any objects in the scene **other than** the object that contains point $\bar{p}$. If the closest object intersection for the shadow ray occurs for $0 < \lambda < 1$ then there is an object between the surface point $\bar{p}$ and the lightsource. If that is the case, we turn off the diffuse and specular components of the Phong model, which reduces the shading equation for point $\bar{p}$ to

$E = r_a I_a + r_g I_{spec}$

For a scene with multiple light sources, we must cast L shadow rays and zero-out the diffuse and specular contributions of any lightsources that are blocked from the point of view of $\bar{p}$.



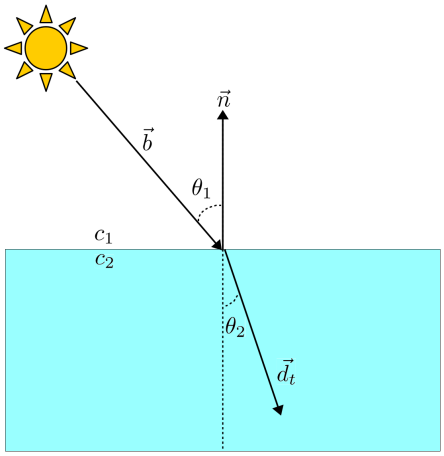
Shadows implemented via ray-casting of shadow rays.

7.6.3 Transmission/Refraction

The final component of our basic raytracer handles transparent or partially transparent objects. When light encouters a transmissive medium, it is refracted, meaning it changes direction of travel.

From left to right: Light refraction through a plastic block, in a glass of water, and simulated refraction for transparent objects in the Tantalum renderer. [Source: Wikipedia, authors: ajizai, Meganbecket27, Benedikt Bitterli

In order to simulate refraction, let's briefly review how light changes direction when it encounters a different, transmissive medium. The constant c which we normally think of when we speak of the speed of light, is actually the speed of light in vacuum. In any other medium (including air), light travels at a slower speed. The change in direction when light transitions from one medium to a different one is the result of this change in velocity.

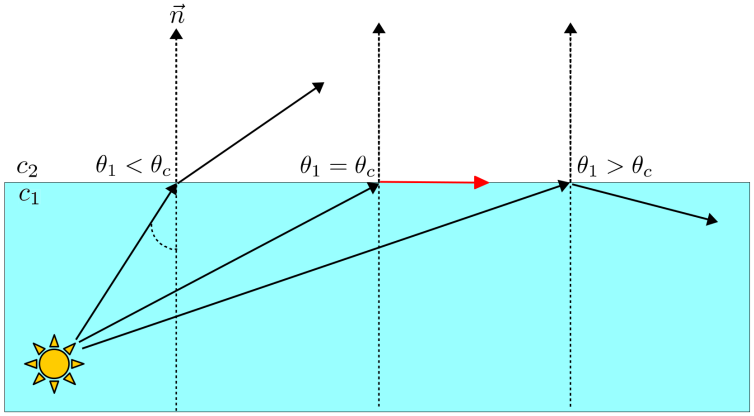


Refraction of light at the boundary between two media with different refraction indexes

The precise amount of bending is described by *Snell's law*:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{c_1}{c_2}$$

Here c_1 and c_2 represent the speed of light in the corresponding medium. If $c_2 < c_1$, light bends towards the normal (e.g. from air to water or glass). If $c_2 > c_1$, light bends away from the normal (e.g. from water or glass to air). This has some interesting consequences. In particular, for light travelling from a slower to a faster medium Snell's law dictates that for $c_2 > c_1$ there exists a critical angle θ_c such that any light arriving at the boundary at an angle greater than θ_c is reflected in full back into the first medium (no transmission occurs). This phenomenon is called total internal reflection, and is what allows materials such as fiber optics to conduct light despite bends and turns in the material.



Total internal reflection when $c_2 > c_1$. Beyond θ_c light is reflected back into the first medium

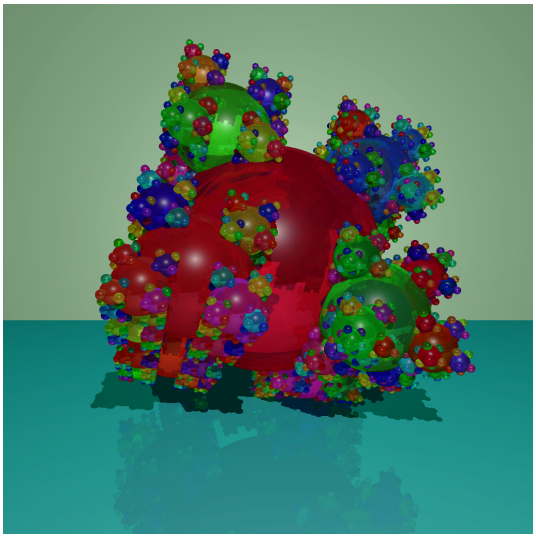
In order to account for refraction effects in our basic ray tracer, we extend the shading model to include one component from light refracted through the object. The colour and intensity of light for this transmissive component is obtained via a recursive raytrace call for a ray along the direction of the transmitted light. There are several ways to incorporate the transmission component to our shading model, one such model is given by:

$$E = \gamma(r_a I_a + r_d I_d \max(0, \vec{n} \cdot \vec{s}) + r_s I_s \max(0, \vec{c} \cdot \vec{m})^\alpha) + (1-\gamma) r_t I_t + r_g I_{spec}$$

Where now we have an additional component $r_t I_t$ for the transmitted light, and a constant γ that controls how transparent the object. For $\gamma = 1$ the object is completely opaque and the equation above reduces to the familiar Whitted ray tracing shading model, for $\gamma = 0$ the object is completely transparent and its appearance is completely determined by light transmitted through it (though notice that the material can change the colour of transmitted light via r_t). For other values of γ, appearance becomes a mixture of the Phong local illumination with light from the transmitted component. Note that the global specular component is not affected by γ in this model.
The direction for the transmitted ray is given by:

$$\vec{d_t} = r\vec{b} + \left(rc - \sqrt{1 - r^2(1-c^2)} \right) \vec{n}$$

Where $\vec{b}$ is the direction of the incident ray, $c = -\vec{n} \cdot \vec{b}$, and $r = c_2/c_1 = n_1/n_2$ and relates the speed of light or index of refraction of the two materials. With the above, our ray tracer can handle fairly complex scenes consisting of objects with visually interesting properties.



Complete shading model for basic ray tracing, including local Phong illumination, global specular reflections, and transparency

Note:
Normally, materials are described not in terms of the speed of light within the material, but in terms of their *index of refraction*. The index of refraction for a material is usually represented by the letter n (be careful not to confuse this with the surface normal), and corresponds to the ratio between the speed of light in vacuum and that of the medium. For example, in the diagram above the index of refraction for the first material is $n_1 = c/c_1$. Indices of refraction for specific materials can be found easily online.

A consequence of this is that the terms c_2/c_1 in the equation for the direction of the transmitted ray become n_1/n_2. The ray tracing code must keep track of the speed of light or the index of refraction for the medium a ray is travelling through.

Note:

Pseudo-Code: Recursive Ray Tracer

for each pixel (i,j)
  $<$ compute ray $\vec{\mathbf{r}}_{ij}(\lambda) = \bar{\mathbf{p}}_{ij} + \lambda\vec{\mathbf{d}}_{ij}$ where $\vec{\mathbf{d}}_{ij} = \bar{\mathbf{p}}_{ij} - \vec{\mathbf{e}}$ $>$
  I = rayTrace($\bar{\mathbf{p}}_{ij}$, $\vec{d}_{ij}$, 1);
  setpixel(i, j, I)
end **for**

rayTrace($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, depth)
  findFirstHit($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, *output var* obj, λ, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$)
  if $\lambda > 0$ then
    I = rtShade(obj, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $-\vec{\mathbf{b}}$, depth)
  else
    I = background;
  end **if**
  return(I)

findFirstHit($\bar{\mathbf{a}}$, $\vec{\mathbf{b}}$, *output var* OBJ, λ_h, $\bar{\mathbf{p}}_h$, $\vec{\mathbf{n}}_h$)
  $\lambda_h = -1$;
  loop over all objects in scene, with object identifiers objID_k
    $<$ find $\lambda*$ for the closest legitimate intersection of ray $\vec{\mathbf{r}}_{ij}(\lambda)$ and object $>$
    if ($\lambda_h < 0$ or $\lambda^* < \lambda_h$) and $\lambda^* > 0$ then
      $\lambda_h = \lambda^*$
      $\bar{\mathbf{p}}_h = \bar{\mathbf{a}} + \lambda^*\vec{\mathbf{b}}$;
      $<$ determine normal at hit point $\vec{\mathbf{n}}_h$ $>$
      OBJ = objID_k
    end **if**
  end **loop**

rtShade(OBJ, $\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, depth)
  /* Local Component */
  findFirstHit($\bar{\mathbf{p}}$, $\vec{\mathbf{l}}^w - \bar{\mathbf{p}}$, *output var* temp, λ_h);
  if $0 < \lambda_h < 1$ then
    I_l = ambientTerm;
  else
    I_l = phongModel($\bar{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, OBJ.localparams)
  end **if**
  /* Global Component */
  if depth $<$ maxDepth then

if OBJ has specular reflection **then**
  $<$ calculate mirror direction $\vec{\mathbf{m}}_s = -\vec{\mathbf{d}}_e + 2\vec{\mathbf{n}} \cdot \vec{\mathbf{d}}_e\vec{\mathbf{n}}$ $>$
  I_{spec} = rayTrace($\bar{\mathbf{p}}$, $\vec{\mathbf{m}}_s$, depth+1)
  $<$ scale I_{spec} by OBJ.specularReflCoef $>$
end if
if OBJ is refractive **then**
  $<$ calculate refractive direction $\vec{\mathbf{t}}$ $>$
  if not total internal reflection **then**
    I_{refr} = rayTrace($\bar{\mathbf{p}}$, $\vec{\mathbf{t}}$, depth+1)
    $<$ scale I_{refr} by OBJ.refractiveReflCoef $>$
  end if
end if
$I_g = I_{spec} + I_{refr}$
else
$I_g = 0$
end if
return($I_l + I_g$)