Ray Tracing Notes
CS445 Computer Graphics, Fall 2014 (last modified 8/4/14)

Jenny Orr, Willamette University

1 The Ray Trace Algorithm

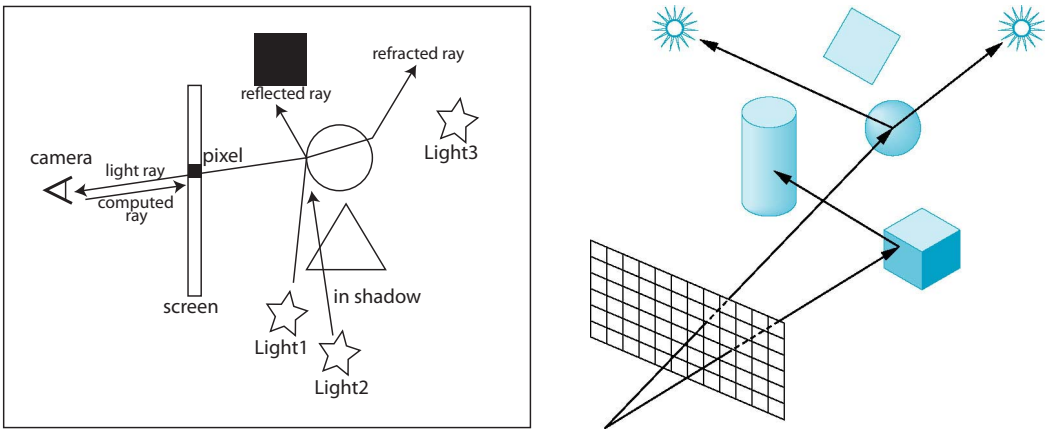


Figure 1

```
1   For each pixel in image {
2       compute ray
3       pixel_color = Trace(ray)
4   }
5   color Trace(ray) {
6       For each object
7           find intersection (if any)
8           check if intersection is closest
9       If no intersection exists
10          color = background color
11      else for the closest intersection
12          for each light  // local color
13              color += ambient
14              if (! inShadow(shadowRay)) color += diffuse + specular
15          if reflective
16              color += k_r Trace(reflected ray)
17          if refractive
18              color += k_t Trace(transmitted ray)
19      return color
20  }
21  boolean inShadow(shadowRay) {
22      for each object
23          if object intersects shadowRay return true
24      return false
25  }
```

1.1 Complexity

w = width of image
h = height of image
n = number of objects
l = number of lights
d = levels of recursion for reflection/refraction

Assuming no recursion or shadows $O(w * h * (n + l * n))$. How does this change if shadows and recursion are added? What about anti-aliasing?

2 Computing the Ray

In general, points P on a ray can be expressed *parametrically* as

$$P = P_0 + t\ dir$$

where P_0 is the starting point, *dir* is a unit vector pointing in the ray's direction, and $t \geq 0$ is the parameter. When $t = 0$, P corresponds to P_0 and as t increases, P moves along the ray. In line 2 of the code, we calculate a ray which starts at the camera P_0 and points from P_0 to the given pixel located at P_1 on a virtual screen (view plane), as shown in Figure 2.

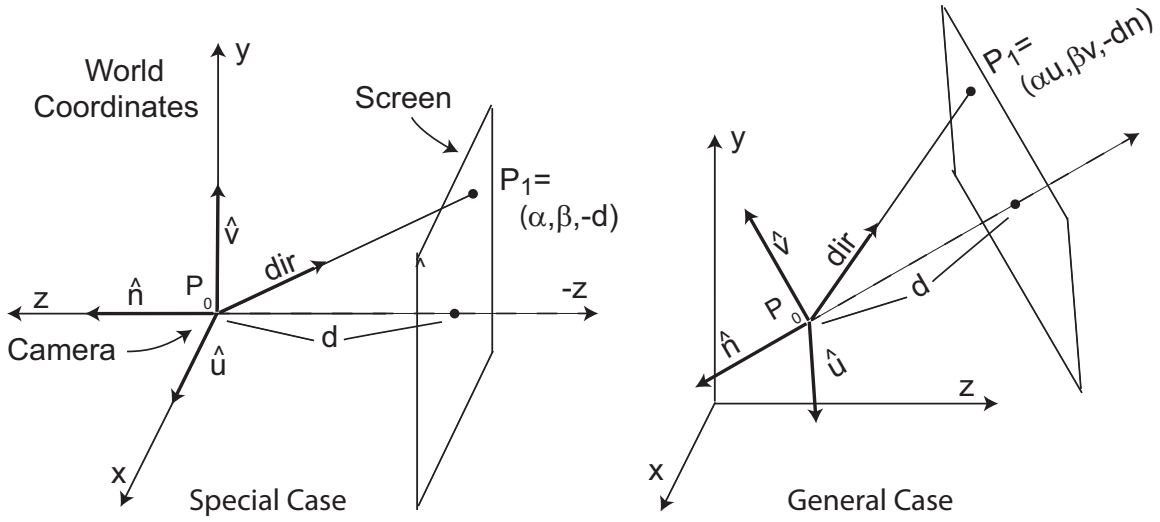


Figure 2

If we know P_1, we can calculate dir by the equation

$$\begin{aligned} dir &= \text{unit vector in direction from } P_0 \text{ to } P_1 \\ &= \frac{P_1 - P_0}{||P_1 - P_0||}. \end{aligned}$$

Once the ray is computed, the method Trace is called (line 3) on this ray. In lines 6-8, we loop over all the objects in the scene, compute the t value of the intersection point between our ray and each object, keeping track along the way of which intersection is the closest. The closest intersection is simply the one with the smallest positive t, as shown in Figure 3.

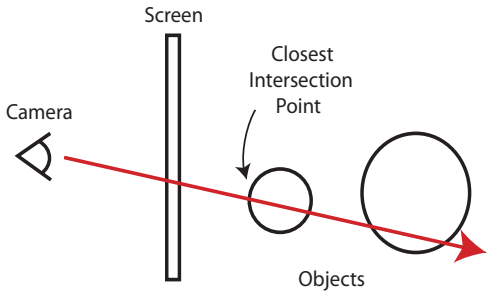


Figure 3

Note, intersections with $t = 0$ are problematic because they are right on the camera. We, therefore, only consider values $t > \epsilon$ for some small value ϵ. Negative t values indicate that the object is behind the camera and, thus, are not in view.

Throughout, we assume we are working in a right-handed coordinate system.

2.1 Computing the Pixel Location in World Space (P_1): Special Case

To compute the ray, we need to compute the location of the pixel P_1 as shown in Figure 4. We consider here the special case where the camera sits at the origin of our World Coordinate (WC) system, looking along the *negative* z axis. The image we are calculating is the projection of the scene onto a virtual screen (view plane) that sits a distance d from the camera. Its normal is aligned with the z axis. The screen has some width W and height H as measured in WC. The screen is divided up into pixels corresponding the desired resolution of our image.

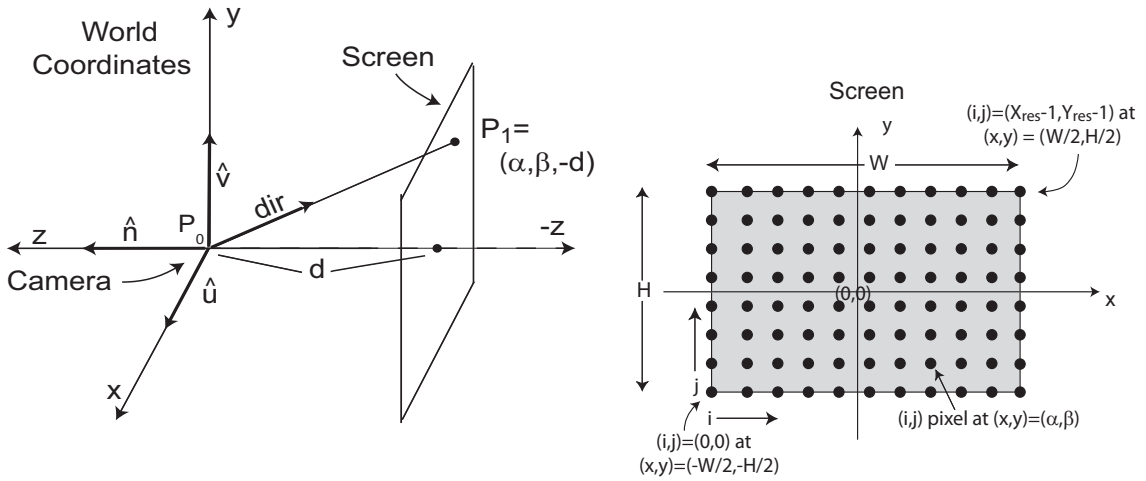


Figure 4

The orientation of the camera is specified by unit vectors $\hat{u}$, $\hat{v}$, and $\hat{n}$, where $\hat{u}$ points to the camera's right, $\hat{v}$ points up, and $\hat{n}$ points along the (negative) direction the camera is looking. (Note, if we were using a left-handed coordinate system, $\hat{n}$ would point in the positive direction of the camera.) In this special case, $\hat{u}$, $\hat{v}$, and $\hat{n}$ are aligned with the coordinate axes.

Thus, to determine the WC coordinates P_1 of a given pixel on the screen, we need to specify the following information:

$$
\begin{aligned}
P_0 &= \text{location of camera} = (0,0,0). \text{ Looks along negative } \hat{n}. \\
\hat{n} &= \text{unit vector normal to view plane, in picture } = (0,0,1) \\
\hat{v} &= \text{unit vector in up direction, in picture } = (0,1,0) \\
\hat{u} &= \text{unit vector in right direction, in picture } = (1,0,0) \\
d &= \text{distance of camera from view screen in WC.} \\
H &= \text{height of screen in WC.} \\
W &= \text{width of screen in WC} \\
X_{res} &= \text{number of pixels in a row} \\
Y_{res} &= \text{number of pixels in a column}
\end{aligned}
$$

For the graphics system we are considering, the pixel indices $(i = 0, j = 0)$ correspond to the bottom left of the image, which has world coordinates $(-W/2, -H/2)$. Pixel indices $(X_{res} - 1, Y_{res} - 1)$ correspond to the upper right, or world coordinates $(W/2, H/2)$. Note, index i corresponds to the *column* index and j to the *row* index. In the actual implementation, one may need to store the image as (j, i).

Based on the above inputs, the location of the i, j^{th} pixel can be written as:

$$P_1 = (x, y, z) = (x, y, -d) = \left(-\frac{W}{2} + \frac{W \cdot i}{X_{res} - 1}, -\frac{H}{2} + \frac{H \cdot j}{Y_{res} - 1}, -d\right)$$

We can plug this into the direction of the ray:

$$dir = \frac{P_1 - P_0}{||P_1 - P_0||} = \frac{P_1}{||P_1||}$$

and then into equation of the ray:

$$P = P_0 + t\ dir = t\ dir$$

2.2 Computing View Coordinate Axes: General Case

Suppose $\hat{u}$, $\hat{v}$, and $\hat{n}$ are not aligned with the coordinate axes x, y, and z but instead are shown as on the right side of Figure 2? In this case, we calculate $\hat{u}$, $\hat{v}$, and $\hat{n}$ assuming we are given the following information:

$$\begin{aligned} \text{VPN} &= \text{normal to view plane (camera points in -VPN direction)} \\ \text{VUP} &= \text{up direction of camera} \end{aligned}$$

We can then calculate $\hat{u}$, $\hat{v}$, and $\hat{n}$ as follows:

$$\begin{aligned} \hat{n} &= \frac{\text{VPN}}{||\text{VPN}||} \\ \hat{u} &= \frac{\text{VUP} \times \text{VPN}}{||\text{VUP} \times \text{VPN}||} \\ \hat{v} &= \hat{n} \times \hat{u} \end{aligned}$$

Given $\hat{u}$, $\hat{v}$, and $\hat{n}$, we can write the difference $P_1 - P_0$ (in WC) as

$$P_1 - P_0 = \alpha\hat{u} + \beta\hat{v} - d\hat{n}$$

where α and β can be calculated similarly as before:

$$(\alpha, \beta) = \left(-\frac{W}{2} + \frac{W \cdot i}{X_{res} - 1}, -\frac{H}{2} + \frac{H \cdot j}{Y_{res} - 1}\right)$$

3 Computing Intersections

Given P_1, we can calculate the ray. Given the ray, we next need to calculate the intersection of the ray with the objects in the scene. Below we show how this is done for spheres and planes.

3.1 Spheres

A sphere can be represented by its center $P_c = (a, b, c)$ and radius r. Given these, the equation for the sphere can be written as

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2.$$

In vector notation, this becomes

$$(P - P_c) \cdot (P - P_c) = r^2.$$

where $P = (x, y, z)$. That is, any point P satisfying the above equation, must be on the surface of the sphere.

To find the ray-sphere intersection we need to find the points that simultaneously satisfy both the equation of the sphere and the equation of the ray. To find these points, we insert the equation of the ray in for P to give

$$(P_0 + t\ dir - P_c) \cdot (P_0 + t\ dir - P_c) = r^2$$

and solve for t. Multiplying out we obtain

$$(P_0 - P_c) \cdot (P_0 - P_c) + 2\ dir \cdot (P_0 - P_c)t + \ dir \cdot dir\ t^2 = r^2$$

which is a quadratic equation in t. Defining the scalars B and C as

$$\begin{aligned} B &= 2\ dir \cdot (P_0 - P_c) \\ C &= (P_0 - P_c) \cdot (P_0 - P_c) - r^2 \end{aligned}$$

and noting that $dir \cdot\ dir = 1$, we obtain the equation $t^2 + Bt + C = 0$. The solution is

$$t = \frac{-B \pm \sqrt{B^2 - 4C}}{2}.$$

There are 0, 1, or 2 real solutions to this depending on the value of the discriminant $D = B^2 - 4C$. In particular,

$D < 0$	no intersections.
$D = 0$	1 intersection, ray grazes sphere.
$D > 0$	2 intersections, take smaller t value.

If $t < 0$ then the object is behind viewer and is thrown out.

Given a point P on the surface of a sphere, it is simple to determine the surface normal at P:

$$normal = \frac{P - P_c}{||P - P_c||}$$

In general, one can easily compute the normal to a surface if one has the equation of the surface written as a function $f(x, y, z) = 0$. The normal is then just the gradient of f. For example, in the case of a sphere, we have

$$f(x, y, z) = (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - r^2$$

The gradient is then

$$\begin{aligned} \text{normal} &= \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right) \\ &= (2(x - x_c), 2(y - y_c), 2(z - z_c)) \end{aligned}$$

which, when normalized, gives the same result as $(P - P_c)/||P - P_c||$

3.2 Planes

A plane is defined by its normal N and a point on the plane Q_0. The equation of the plane is then given by

$$(P - Q_0) \cdot N = 0$$

where $P = (x, y, z)$. To find the intersection of the plane and the ray we insert the expression for the ray into the above equation to obtain

$$(P_0 + t\ dir - Q_0) \cdot N = 0$$

and solving for t gives

$$t = \frac{(Q_0 - P_0) \cdot N}{dir \cdot N}$$

Note, a zero denominator means that the line and plane are parallel. In such a case there are either no intersections if the ray does not lie in the plane, or there are an infinite number of intersections.

Rectangles aligned with the axes are also easy to represent because they are just planes with the range constrained.

4 Computing Pixel Color: Phong Lighting Model

At this point, we have found the intersection point of our ray with the closest object in the scene. Recall, that this ray points along the line of sight from the camera (viewer) to a particular pixel in the screen. If the viewer looks along the ray, they will look through this pixel to see the closest object at the intersection point. Thus, the color at this intersection point is exactly the color we want to set this pixel. Therefore, our next task is to determine the color at the intersection point (lines 12-14 in code). We ignore shadows and reflections for the moment. We also assume that light sources are simple *point* lights, i.e. they have a point location, have no physical extent and they shine light equally in all directions.

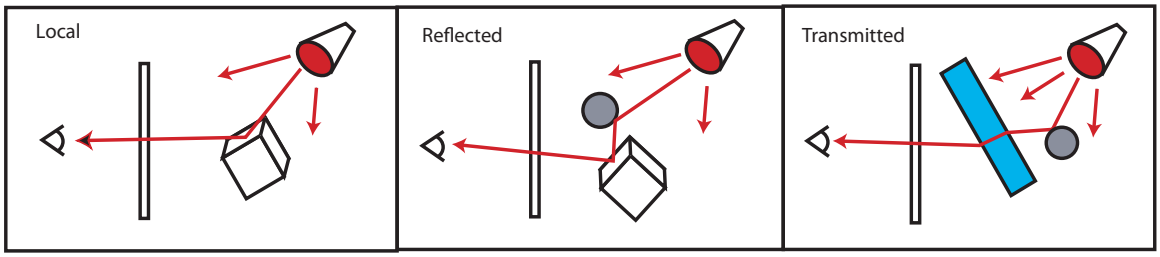


Figure 5

The color at the intersection point is composed of 1) local color based on the direct interaction of surface with the light sources, 2) the color resulting from reflected light, and 3) color from refracted light:

$$\text{pixel color} = \text{local color} + \alpha_1\text{*reflected} + \alpha_2\text{*refracted}$$

where α_1 and α_2 are material dependent weights (less than 1) that control how much light is reflected or refracted.

Here, we will discuss only the local color as modeled by the Phong Lighting Model. Note that the reflected and refracted light can be computed by recursively applying the Phong Model.

The Phong Lighting Model computes the local color by dividing the light-surface interaction into three components: ambient, diffuse, and specular. The local color is the sum of each contribution:

$$\text{local color} = \text{ambient} + \text{diffuse} + \text{specular}$$

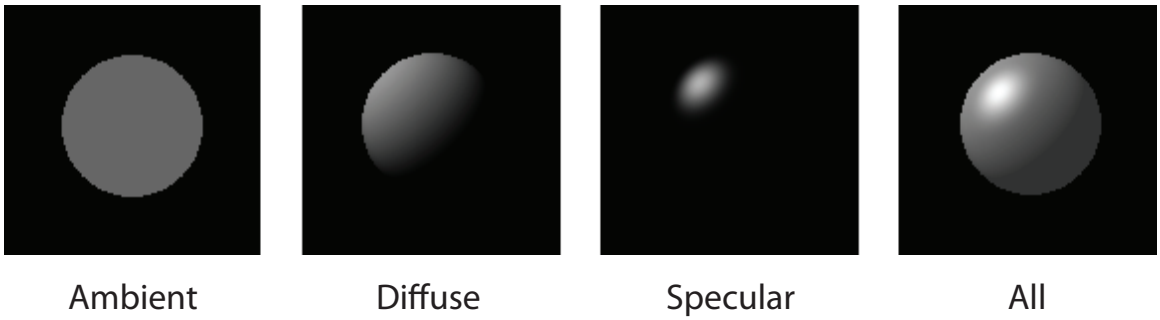


Figure 6

The needed parameters are listed here and will be discussed in more detail next.

Intersection parameters:

N = unit vector along surface normal at the intersection point.
V = unit vector pointing from intersection point to camera
L_j = unit vector pointing from intersection point to j^{th} light source

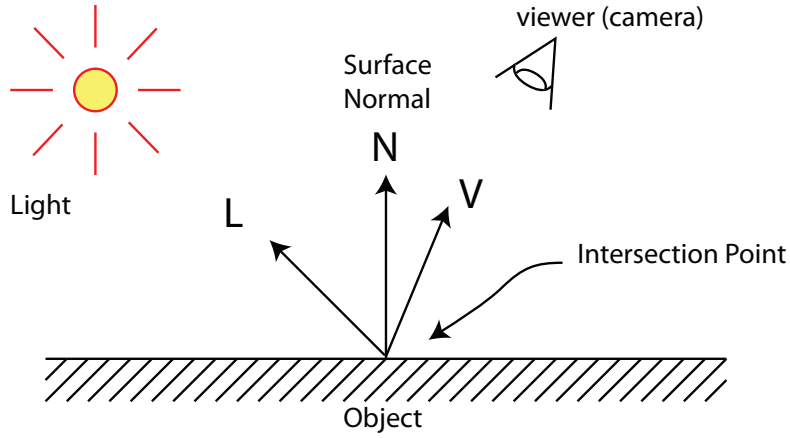


Figure 7

Surface Material Parameters:

c_a = $(c_{a,r}, c_{a,g}, c_{a,b})$ = ambient rgb color components of surface
c_d = $(c_{d,r}, c_{d,g}, c_{d,b})$ = diffuse rgb color components of surface
c_s = $(c_{s,r}, c_{s,g}, c_{s,b})$ = specular rgb color components of surface
k_a = coefficient of ambient reflection of surface
k_a = coefficient of diffuse reflection of surface
k_s = coefficient of specular reflection of surface
k_r = reflection coefficient (used only for *recursive* reflection - see "Reflections" section)
n = specularity (measure of the sharpness of the specular highlight)

The coefficients of reflection (k_a, k_d, k_s, and k_r) are in the range (0,1) and are a measure of how much the surface reflects the light. The surface colors (c_a, c_d and c_s) represent the base color of the surface and are also in the range (0,1).

Light Parameters:

$I_{a,global}$ = $(I_{a,r,global}, I_{a,g,global}, I_{a,b,global})$
= ambient rgb color/intensity components of a global ambient light source.
$I_{a,j}$ = $(I_{a,r,j}, I_{a,g,j}, I_{a,b,j})$ = ambient rgb color/intensity components of j^{th} light source.
$I_{d,j}$ = $(I_{d,r,j}, I_{d,g,j}, I_{d,b,j})$ = diffuse rgb color/intensity components of j^{th} light source.
$I_{s,j}$ = $(I_{s,r,j}, I_{s,g,j}, I_{s,b,j})$ = specular rgb color/intensity components of j^{th} light source.

The light color/intensity (I_a, I_d and I_s) model both the rgb color and the brightness of the light sources. One could separate the color from the intensity for more control.

Note, in the equations that follow, we define the product $*$ of two vectors $A = (a_0, a_1, a_2)$ and $B = (b_0, b_1, b_2)$ as

$$A * B = (a_0 * b_0, a_1 * b_1, a_2 * b_2)$$

4.1 Ambient Color Contribution

Ambient light is the "background" light that results from many rays bouncing off of objects multiple times. The ambient light allows us to see all surfaces in a room even if there is no direct light on the surfaces. Since it would be too complex to model every single ray, we instead model the average behavior. The modeling is very simplistic in that we assume that ambient light is constant over the entire scene. Each light source contributes to the ambient light. We also often add a generic ambient light independent of any light source and only dependent on the scene. This allows us to increase the brightness of the scene as a whole.

Because ambient light is constant, it illuminates all surfaces with an equal brightness. A scene with only ambient light is very flat.

The contribution of ambient light to the pixel color is:

$$C_a = k_a(I_{a,global} * c_a) + k_a \sum_j (I_{a,j} * c_a)$$

For simplicity of notation, we can treat the global ambient light as the 0^{th} light source, with $I_{a,0} = I_{a,global}$, $I_{d,0} = 0$, and $I_{s,0} = 0$ so that the ambient contribution can be written simply as

$$C_a = k_a \sum_j (I_{a,j} * c_a)$$

4.2 Diffuse and Specular

Diffuse and specular components are a result of light coming directly from a light source to the intersection point (of ray and object) and then being reflected from the object back to the observer's eyes.

Surfaces (i.e. materials) that are rough tend to scatter the light in all directions and have a rather dull finish. This type of illumination is referred to as diffuse.

Surfaces that are smooth and highly reflective (i.e. shiny) do not scatter the light but rather reflect light mostly in one direction. Such objects have what are called specular highlights. While the difference between diffuse and specular is rather artificial, it is still useful to model them separately.

To compute the intensity of the diffuse and specular components, we calculate how much light from the light source bounces from the intersection point back to the eye. This is described more below.

4.2.1 Diffuse Light Component

Objects that have a dull, matte finish are said to have a large diffuse component. The surface at the macroscopic level may appear smooth but at the microscopic level, there are many tiny bumps that scatter light in all directions.

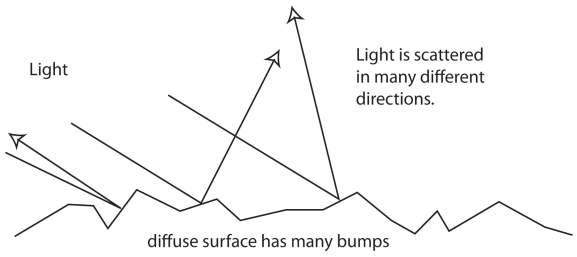


Figure 8

The microscopic surface normal at any given point, is generally not aligned with the macroscopic surface normal. As a result, light appears to scatter in many directions. We make the simplifying assumption that the light is scattered *equally* in all directions so that the *brightness is independent of viewing angle*. However, *the brightness is dependent on the angle that the light ray makes with the surface normal*.

The diffuse light contribution can be calculated as

$$C_d = k_d \sum_j (c_d * I_{d,j})(L_j \cdot N)$$

which is *independent* of V but changes as the angle of L_j changes. Note, $(L_j \cdot N) < 0$ implies that the light is below the surface and thus does not contribute to the color, assuming the surface is opaque.

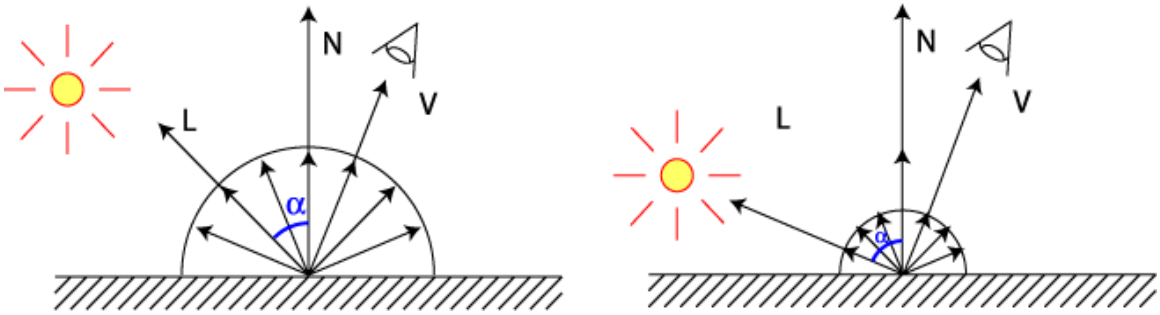


Figure 9: small α gives large diffuse (left), large α gives small diffuse (right)

4.2.2 An Aside on the Diffuse Light Formula

We attempt here to explain in more detail why the diffuse light contribution is calculated as it is. Imagine a cylinder of light coming from the light source and falling on the object (here we are ignoring light attenuation). The cross sectional area of this cylinder is A. The energy per unit area hitting the surface when the light is right overhead is E/A where E is the energy emitted from the light source. If the light is at an angle then the surface area hit by the light is larger and is given by $A_\theta = A/\cos\theta$ so the energy per unit area is $E\cos\theta/A$, i.e. it is smaller by an amount $\cos\theta$. Thus the brightness is a function of $L \cdot N = \cos\theta$ where L is the unit vector pointing from the intersection point to the light source and N is the unit normal to the surface.

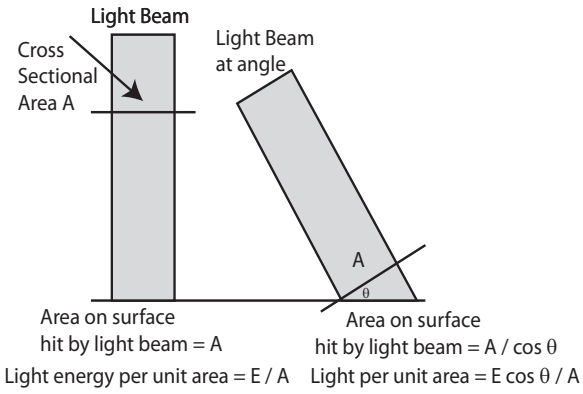


Figure 10

Why does the brightness not depend on the viewing angle? Once the light is fixed, we have a fixed energy per unit area hitting the surface. As the viewing angle increases the area viewed increases. However, the amount of this energy that gets reflected decreases so as to offset the increase in viewing area. Thus the viewer sees the same brightness from any angle.

4.2.3 Specular Light Component

Objects that are shiny have what we call specular highlights. Unlike diffuse surfaces, shiny surfaces are very smooth even at the microscopic level. When light hits the surface, the light is reflected mostly in one direction, the direction of reflection R. A small amount of spreading is allowed around R.

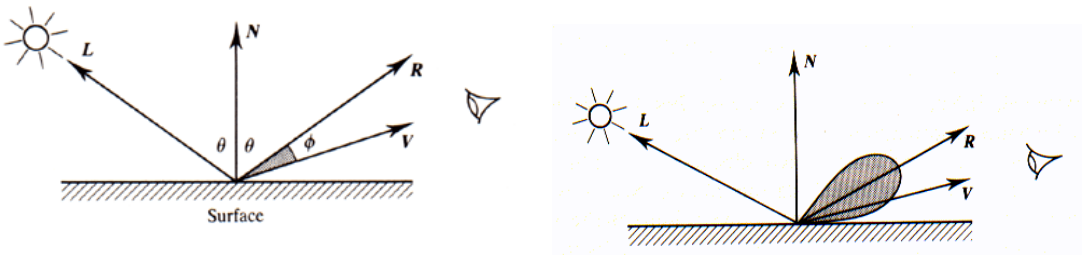


Figure 11

The specular light contribution can be calculated as

$$C_s = k_s \sum_j (c_s * I_{s,j})(V \cdot R_j)^n$$

where R_j is a unit vector in the direction of the reflected ray of the j^{th} light source. It can be calculated with the vector equation

$$R_j = 2(L_j \cdot N)N - L_j$$

As in the diffuse case, $(L_j \cdot N) < 0$ implies that the light is below the surface and thus does not contribute to the color, assuming the surface is opaque. One should

also check to make sure that $(V \cdot N) > 0$ to make sure the viewer is above the surface and $(V \cdot R) > 0$ to make sure that the viewer is within 90 degrees of R.

The specular light is very bright if the viewer's angle is close to R (i.e. small ϕ) and drops off very quickly as ϕ increases. The exponent n is called the specularity and is a measure of the sharpness of the specular highlight. If n is small, the highlight is spread out. The larger n becomes, the sharper the highlight becomes. Typically n ranges from 1 up to several hundred.

The color of the specular highlight is often set to the color of the light source and not the object, i.e. c_s is set to 1. This is because we expect that a highly reflective surface will not alter the light. A mirror is the extreme case of this.

4.3 Final Pixel Color

Combining the ambient, diffuse, and specular contributions (see Figure 6), we have that the pixel's local color at the intersection point is computed by the equation:

$$C = \sum_j \{k_a(I_{a,j} * c_a) + k_d(c_d * I_{d,j})(L_j \cdot N) + k_s(c_s * I_{s,j})(V \cdot R_j)^n\}$$

4.4 Shadows

The diffuse and specular components do not contribute to the color of the intersection point if there is an opaque object that sits between the intersection point and the light position. One can check for this as follows. Compute the **shadow ray** which is a ray with origin (P_0) set to the intersection point and direction (dir) set to the unit vector pointing from the intersection point to the light source. One then loops over all objects in the scene to check to see if the shadow ray intersects any object (see lines 22-24 in code). One must be sure that the object is *between* the light source and intersection point. If so, then that light does not contribute to the diffuse or specular light intensity at the intersection point (see "if" condition in line 14 of code).

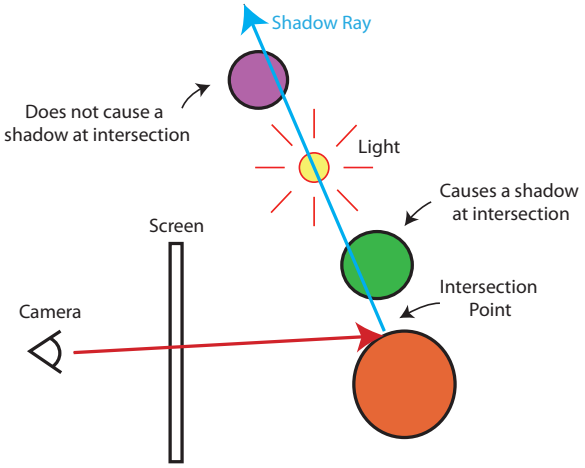


Figure 12

4.5 Reflections

Reflections can be computed recursively as follows. Compute the ray with origin (P_0) set to the intersection point and direction (dir) equal to the reflected direction of V:

$$R_v = 2(V \cdot N)N - V$$

This direction is chosen because it is most likely to contribute the largest reflected value. We can then recursively call the Trace method with this ray to compute the reflected color which is then added to the local color as shown in lines 15-16 of the code. A depth variable can be added to control the number of recursions. Generally, 1 or 2 recursions are sufficient.

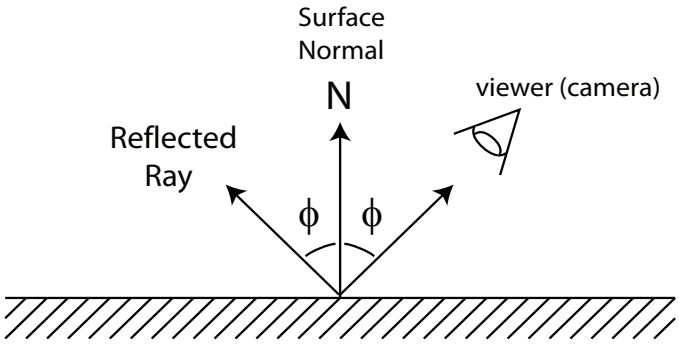


Figure 13

This approach is a crude approximation that works best with fairly reflective surfaces. For diffuse surfaces, the approximation is poor, which is why global illumination methods such as radiosity were developed.

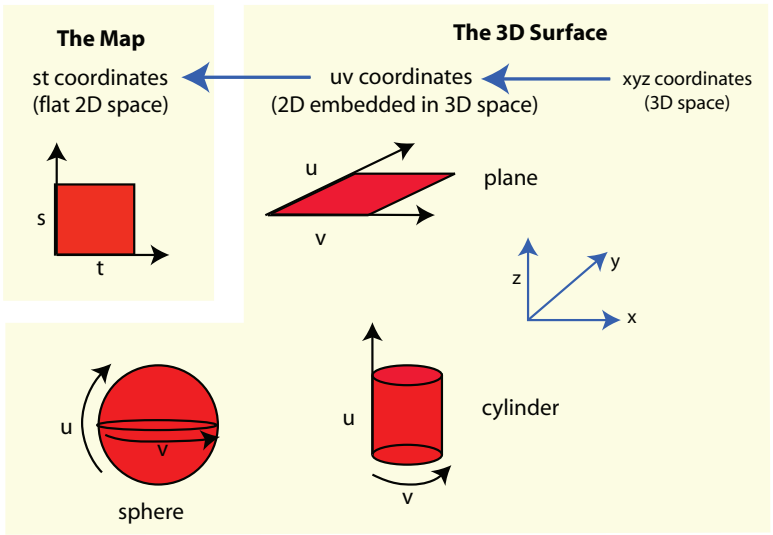
4.6 Textures

4.6.1 Texture Types

- The texture data can be a 2D image or a 3D solid texture.
- The coordinates of the texture are called the *texture parameters* and the assignment of the texture parameters to the surface is called *parameterization*. Texture parameters are commonly denoted u, v for 2D and u, v, w for 3D.
- Texture images, *texture(u,v)* are good when the surface has a natural parameterization and/or you want a "decal" or "projected" look. This is the most common type of texture mapping.
- Solid textures, texture(u,v,w), are good when you want a surface to appear carved out of a material (e.g. wood, marble).
- Textures can be stored as a raster image, a 3D discrete grid, or can be computed on the fly procedurally. In procedural mapping (surface or solid): the "image" is defined mathematically.

4.6.2 Texture Parameterizationi: 2D Image Textures

2D texture is mapped to a 2D surface embedded in 3D space. See figure below. In this process we assume that the image is stored in some nxm array. Each point in the array is called a texel. WLOG, we can scale this so that the image fits in a range $0 \leq u, v \leq 1$.

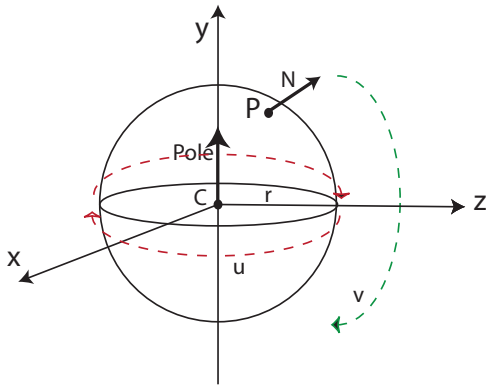


Each intersection point, gets mapped back to the (u,v) coordinates of the shape, which then is mapped to the (s,t) coordinates of the texture in order to determine the color at that intersection point.

Parameterization is somewhat straight forward for spheres, planes, cylinders,and rectangular surfaces (see sphere below).

Parameterization is somewhat less obvious for arbitrary polygons or volumes. In the case of volumes, one can do a 2-step process of first mapping to an easy to parameterize intermediate shape such as a sphere or cylinder. The second step is to map from the intermediate shape to the surface of the actual object.

4.6.3 Spheres



Assume center is at C and radius = r.

We use a left handed coordinate system. v is zero at north pole and 1 at south pole. The angle that is made is called θ which ranges between 0 and π.

u moves around in the xz plane around y (use left-hand rule to get direction). The angle is ϕ which ranges between 0 and 2π.

Let P be the intersection point. Then the normal at P is

$$N = \frac{P - C}{||P - C||} = \frac{P - C}{r}$$

1. Compute v:

$$v = \frac{\theta}{\pi}$$

But $cos\theta = N \cdot Pole$ so that

$$v = \frac{cos^{-1}(N \cdot Pole)}{\pi}$$

Letting $Pole = (0, 1, 0)$ then

$$v = \frac{cos^{-1}((p_y - C_y)/r)}{\pi}$$

2. Compute u: Note, if $v = 0$ or 1 then u can be anything.

$$u = \frac{\phi}{2\pi}$$

But $\tan\phi = x/z = \frac{p_x - C_x}{p_z - C_z}$ so that

$$u = 0.5 + \frac{1}{2\pi}\tan^{-1}\frac{p_x - C_x}{p_z - C_z}$$

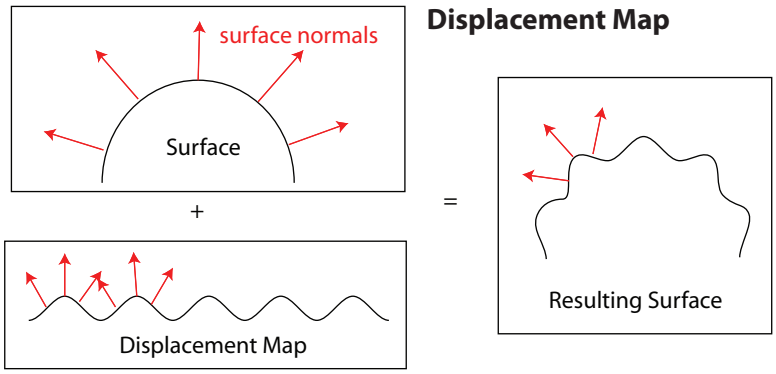
Note, we add .5 because $\tan^{-1}$ returns a value between $-\pi$ and π rather than 0 and 2π .

4.6.4 3D Parameterization: 3D Solid textures

(u,v,w) are usually taken to be the world space or object space. Often these are defined procedurally (wood, marble). If we want the texture to move with the object, then one needs to use the object space.

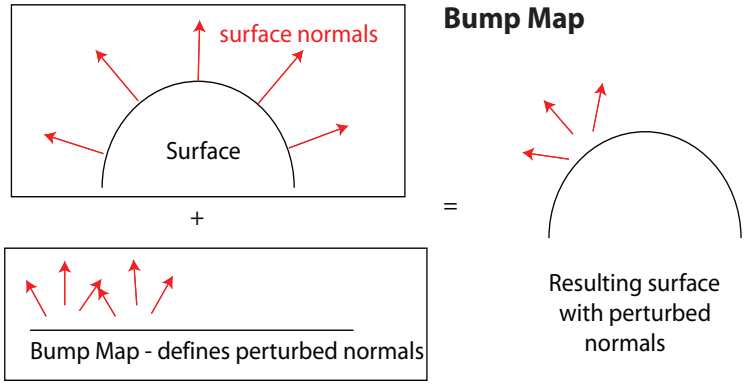
4.6.5 Bump Mapping

If we want fine grained surface texture, we can define a base surface (e.g. a sphere) together with the small displacement from the sphere (e.g. the small bumps on the surface of an orange). The combined result is called a displacement map.

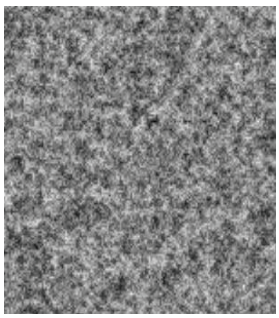


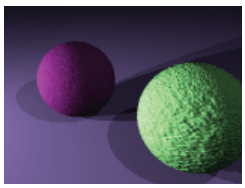
This process can be very expensive because it requires adding many, many tiny triangles in order to model the displacement.

A cheaper alternative is called bump mapping which mimics the normals of the displacement map but without actually changing the surface itself. The surface perturbations can be stored as a texture. Since the surface is not changed, no geometry needs to be added. One just needs to adjust the surface normal at the time the lighting calculation is done. This is a trick of the eye. If the displacement is small, the viewer won't be able to tell that the surface has not actually changed.



Below is an example of the bump map (texture file) together with several resulting bump mapped spheres. It is difficult to tell that the surface is actually just a smooth sphere.


Texture file


Bump mapped surface