# 《Fundamentals of Computer Graphics》

## Lecture 8、Ray tracing

## Part 5: More realistic effects

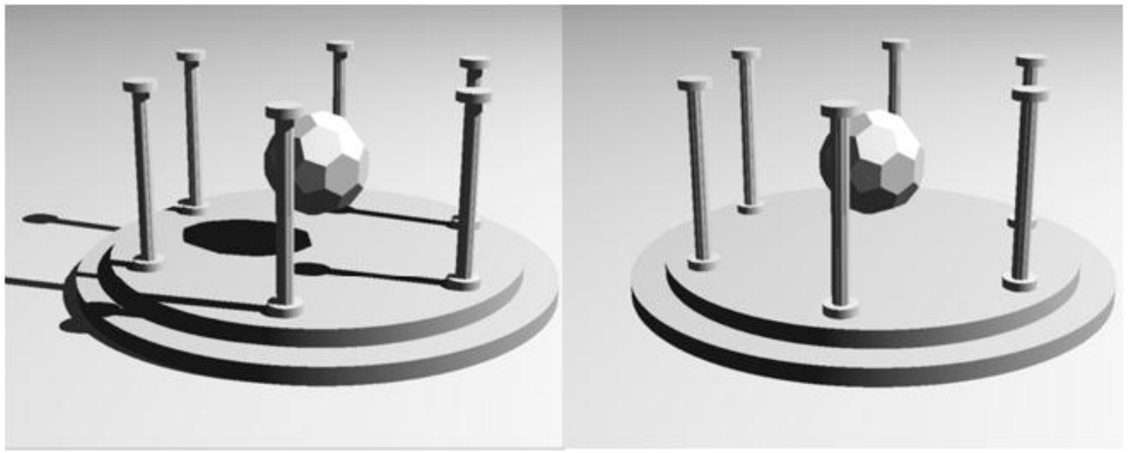
Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

Material by S.M.Lea (UNC)

# Adding Shadows

- Ray tracing produces shadows easily. Unfortunately, it slows down the ray tracing process.
- The figure shows a scene rendered with and without shadows. In part b, it is difficult to see how far above the platform the buckyball lies, whereas in part a our eye can see this immediately.

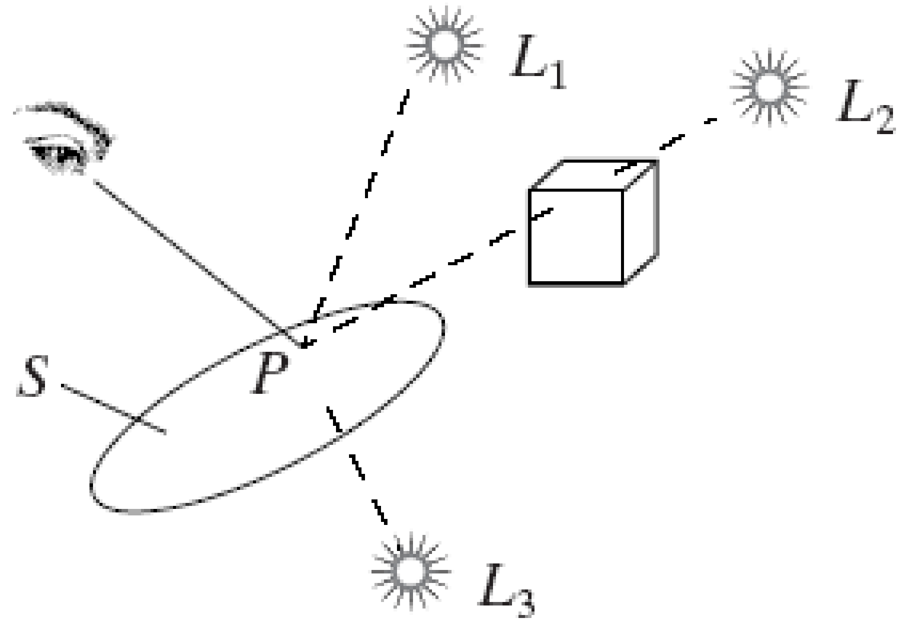


## Adding Shadows (2)

- The light intensities we have calculated up to now have assumed that the hit point, $P_h$, of the ray with the first object hit is in fact bathed in light from the various light sources.
- But this is not the case if some other object happens to lie between $P_h$ and a light source.
- In that case, $P_h$ is in shadow with respect to that light source, and both the diffuse and specular contributions are absent.
- This leaves only the ambient light component.

# Adding Shadows (3)

- **Shadowing situations:**
- Point *P* can see source $L_1$, so *P* is not in shadow with respect to $L_1$.
- But *P is* in the shadow of the cube with respect to source $L_2$. Further, the hit object itself hides the source $L_3$ from *P*; this is called **self-shadowing**.
- Therefore, the only light that *P* can see is the light from source $L_1$.

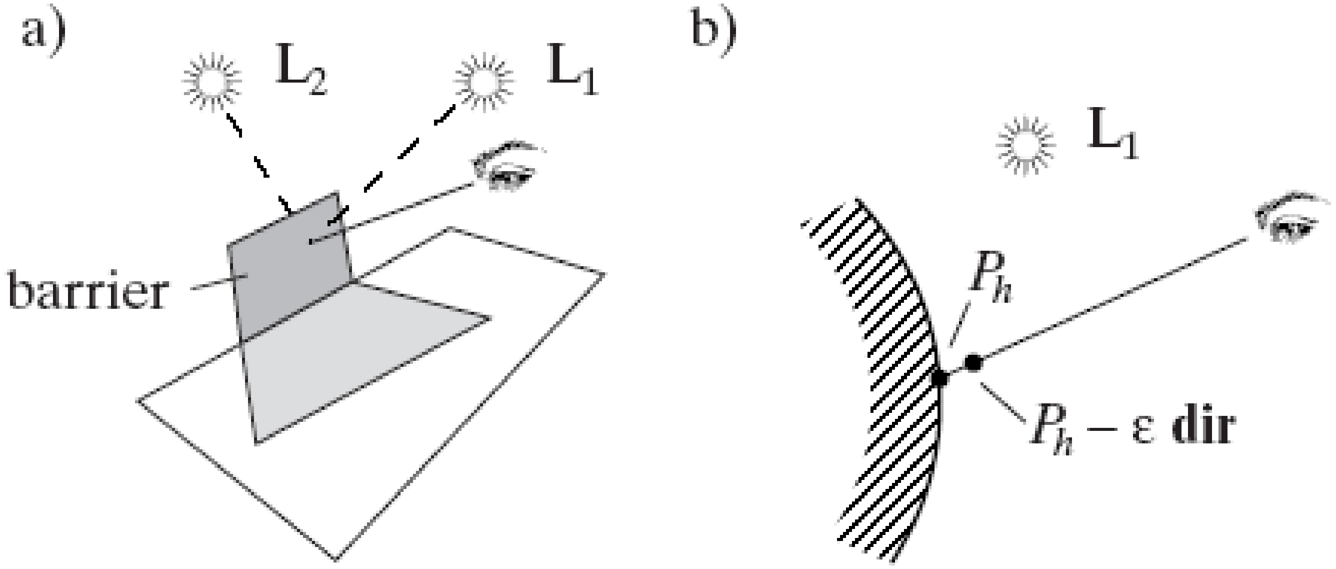


# Adding Shadows (4)

- In order to compute shadows accurately, we need to know when a hit point is in shadow with respect to a light source.
- So we need a routine, isInShadow(), that returns true if any part of *any* object lies between the hit point and a given source, and false otherwise.
- To do this, we spawn a new ray, often called a **shadow-feeler**, that emanates from $P_h$ at $t = 0$ and reaches $L$ at $t = 1$. The shadow-feeler thus has the parametric representation $P_h + (L - P_h)t$.
- To see if it hits anything, the entire object list is scanned, and each object is tested for an intersection with this ray. If any intersection is found to lie between $t = 0$ and $t = 1$, isInShadow() returns true.

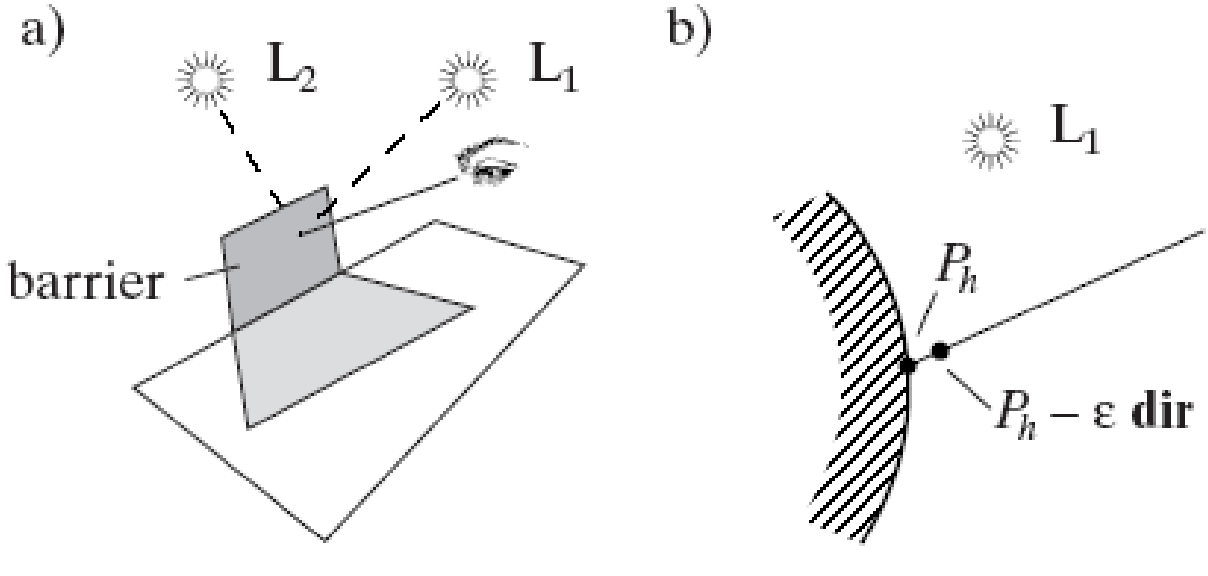
# Adding Shadows (5)

- Including self-shadowing: If the shadow feeler really starts at $P_{h,}$ then there is *always* an intersection between the feeler ray and the object itself, at $t$ = 0! Thus isInShadow() would always return true, which is clearly wrong.



# Adding Shadows (6)

- An adjusted shadow feeler is used, as shown in (b). The start point of the shadow feeler is shifted toward the eye by a small amount. If the ray has direction **dir** and hit at point $P_h$, the start point of the shadow feeler is offset to $P_h$ - ε **dir**, where ε is a small positive number.



# Adding Shadows (7)

- This puts the starting point slightly in front of (“on the eye side of”) the object that is hit by the ray.
- Using this start point for the shadow feeler, there is no intersection with the object at $t = 0$, and the self-shadowing is included.

# Adding Shadows (8)

- This approach fits into the Scene :: shade(ray) method in the following way.
- When getFirstHit() returns the best intersection record, the hit point and the normal vector at the hit point are determined. The ambient light color is found for this ray. Then a feeler ray is constructed and its start point is set to $P_h$ - ε **dir**. Its recurseLevel is set to 1 to disable projection extents in the various hit() methods.
- Then, for each light source L, the feeler direction is computed as L.pos – feeler.start and isInShadow(feeler) is called to see if the feeler hits any object.
- If it does, the computation of the diffuse and specular light contributions is skipped for this light source.

## Pseudocode for Shadows (9)

```
feeler.start = *hitPoint - e ray.dir;*
feeler.recurselevel = 1;
color = *ambient part*;
for (each light source, L)
{
feeler.dir = L.pos - hitPoint;
If (isInShadow(feeler))continue;
color.add(diffuse light);
color.add(specular light);              }
```

# Code for isInShadow()

- The routine simply scans through the object list looking for a hit, and if one is found, it returns false. If no hits are found it returns true.

```
bool Scene :: isInShadow(Ray& f)
{
   for(GeomObj* p = obj; p; p = p->next)
      if(p->hit(f))return true;
   return false;
}
```

# Code for isInShadow() (2)

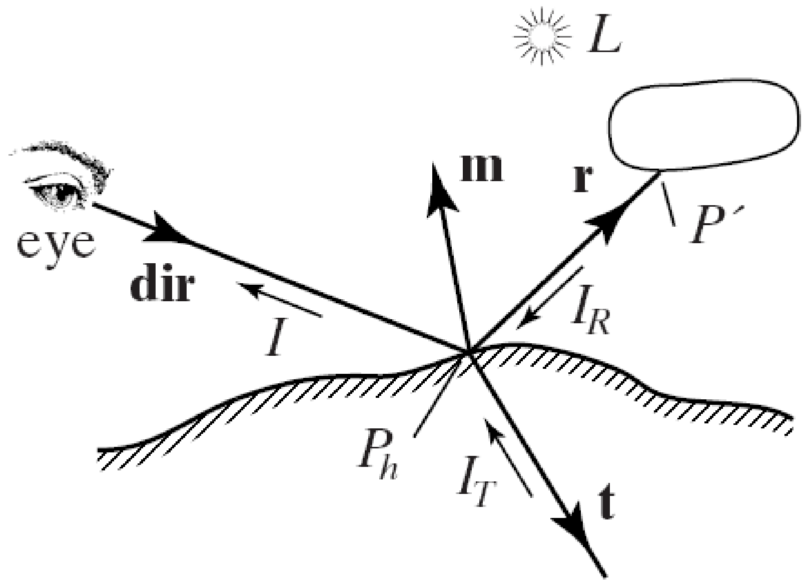
- The code uses a simplified version of hit() for each object type that takes only one argument – it doesn't need to build an intersection record. This version of hit() differs in three ways from the version used up to now.
- 1). It only accepts a hit for which the hit time lies between 0 and 1, since an object lying beyond the light source does not cast a shadow.
- 2). If it detects such a hit, it returns immediately without computing any data about the hit itself.
- 3). It cannot use projection extents since shadow feelers can originate anywhere in the scene. Therefore, it should perform some carefully selected combination of sphere and/or box extent tests for each object type.

## Reflections and Transparency

- One of the great strengths of the ray tracing method is the ease with which it can handle both the reflection and the refraction of light.
- This allows one to build scenes of exquisite realism containing mirrors, fish bowls, lenses, and the like.
- There can be multiple reflections in which light bounces off several shiny surfaces before reaching the eye, or elaborate combinations of refraction and reflection.
- Each of these processes requires the spawning and tracing of additional rays.

# Reflections and Transparency (2)

- The figure shows a ray emanating from the eye in the direction **dir** and hitting a surface at the point $P_h$.
- The figure is in 2D, which is acceptable because the nature of reflection and refraction causes all vectors to lie in the same plane.
- All formulas we develop operate in 3D.



# Reflections and Transparency (3)

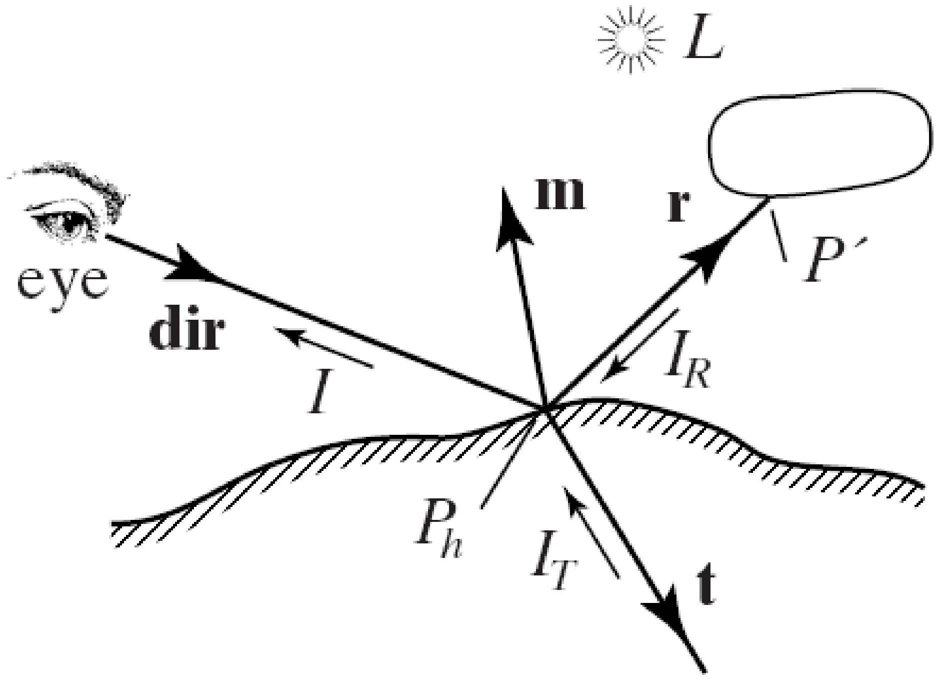
- When the surface is mirror-like or transparent, or both, the light $I$ that reaches the eye may have five components: $I = I_{\mathrm{amb}} + I_{\mathrm{diff}} + I_{\mathrm{spec}} + I_{\mathrm{refl}} + I_{\mathrm{tran}}$
- The first three are the familiar ambient, diffuse, and specular contributions. The diffuse and specular parts arise from light sources in the environment that are visible at $P_{\mathrm{h}}$.
- $I_{\mathrm{refl}}$ is the reflected light component, arising from the light, $I_{\mathrm{R}}$, that is incident at $P_{\mathrm{h}}$ along direction -**r**. This direction is such that the angles of incidence and reflection are equal, so **r** is given by $\mathbf{r} = \mathbf{dir} - 2\,(\mathbf{dir}\cdot\mathbf{m})\,\mathbf{m}$, where **m** is normalized.

# Reflections and Transparency (4)

- Similarly, $I_{\mathrm{tran}}$ is the transmitted light component, arising from the light, $I_{\mathrm{T}}$, that is transmitted through the transparent material to *P*h along direction **-t**.
- A portion of this light passes through the surface and in so doing is bent. It then continues its travel along **-dir**.
- The refraction direction, ***t***, depends on several factors, and its details are developed in the next section.

# Reflections and Transparency (5)

- $I_R$ and $I_T$ each arise from their own five components: ambient, diffuse, and so on.
- $I_R$ is the light that would be seen by an eye at $P_h$ along a ray from P' to $P_h$.
- To find $I_R$, we spawn a secondary ray from $P_h$ in the direction **r**, find the first object it hits, and compute the 5 light components.



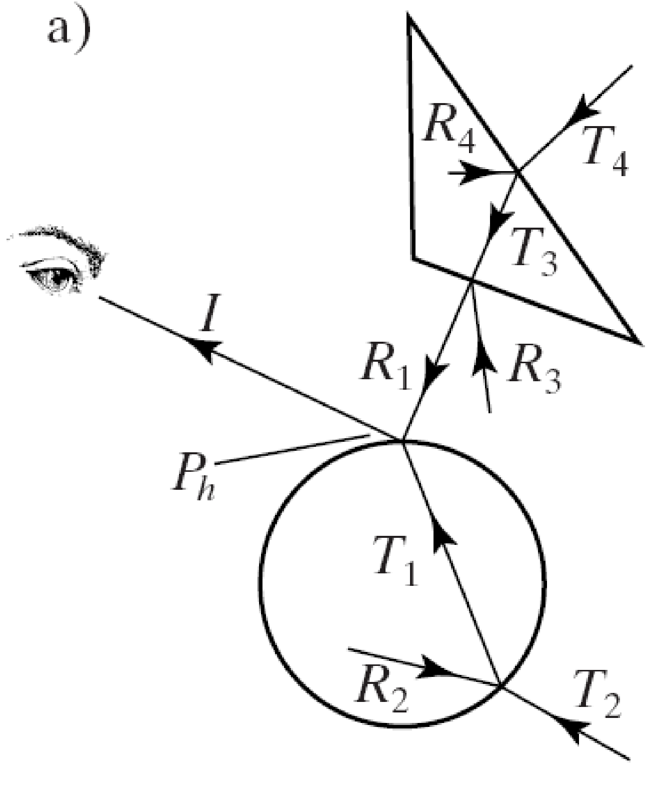
# Reflections and Transparency (6)

- This may in turn require spawning additional rays.
- Similarly, $I_T$ is found by casting a ray in direction ***t*** and seeing what surface is hit first, then computing the light contributions there, etc.
- The number of contributions of light grows at each contact point. The final light that reaches the eye, *I,* is a combination of a large number of contributions, consisting of reflected and refracted light from various points in the scene in addition to a number of ambient, diffuse, and specular components.

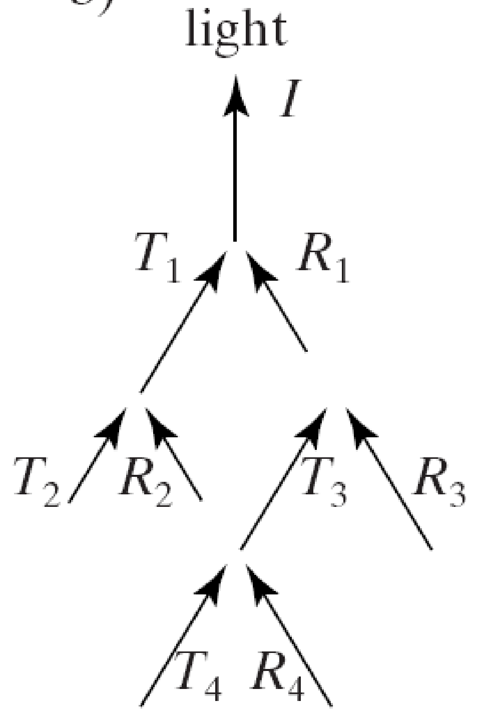
# Reflections and Transparency (7)

- The tree of light rays; local components are not shown.

a)



b)



## Reflections and Transparency (8)

- $I$ is the sum of three components: the reflected component $R_1$, the transmitted component $T_1$ from the refraction, and the local component $L_1$.
- The *local components* are simply the sum of the usual ambient, diffuse, and specular reflections at $P_\mathrm{h}$. Local components depend only on actual light sources; they are not computed based on casting secondary rays.
- The role of the ambient term is to approximate the effect of diffuse and specular reflections off other surfaces. $R_1$ is in turn the sum of $R_3$, $T_3$, and the local $L_3$.
- And $T_3$ is the sum of $R_4$, $T_4$ and $L_4$. Each contribution is the sum of three others, possibly ad infinitum.

## Reflections and Transparency (9)

- To incorporate these visual effects, Scene ::shade() is extended so that it can call itself recursively.
- Under the right conditions, shade() calls itself twice to accumulate reflected and transmitted light contributions.
- A skeleton for a recursive version of shade() is shown in Fig. 12.70.

## Reflections and Transparency (10)

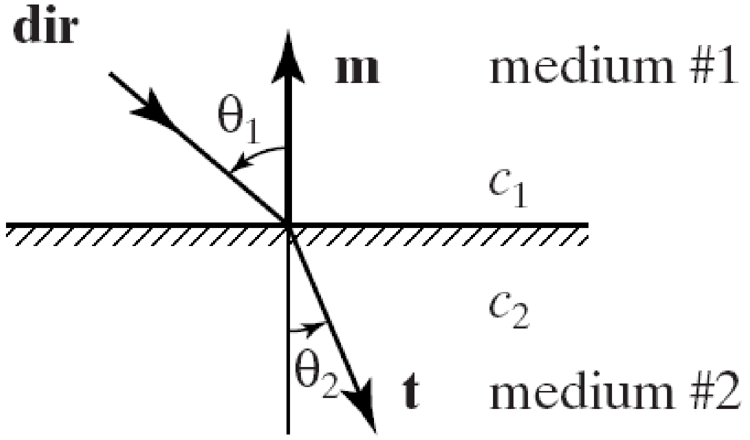
- If the hit object is shiny enough, a reflected ray is spawned, and shade() is used to compute how much light comes back along the reflection direction.
- The amount of light shade() finds is tempered by the reflection coefficient, shininess, of the hit object. This reflection coefficient is stored in one of the fields of the object.
- The user specifies it in an *SDL* file using, say, shininess 0.8. It is used for the "if shiny enough" test, as in if (shininess > 0.6).

# Reflections and Transparency (11)

- If the hit object is transparent enough, a transmitted ray is spawned, and shade() is used to compute how much light comes back along the transmitted direction.
- The amount of light found is scaled by the transmission coefficient, transparency, of the hit object. This coefficient is stored with the object, and it is used for the "if transparent enough" test, as in if (transparency > 0.5).
- There is the possibility that rays would keep spawning new reflected or transmitted rays forever. We include a maxRecursionLevel (which is stored in a field in the Scene object, and specified in an *SDL* file as maxRecursionDepth 5) to ensure the recursion stops.
- Usually a maximum recursion depth of 4 or 5 gives very realistic images.

# Refraction of Light

- When a light ray strikes a transparent object, a portion of the ray penetrates the object. The ray will change direction from **dir** to **t** if the speed of light is different in medium 1 and medium 2. Vector **t** lies in the same plane as **dir** and the normal **m**.



# Refraction of Light (2)

- If the angle of incidence of the ray is $\theta_1$, **Snell's law** states that the angle of refraction $\theta_2$ will be given by the equation $\sin(\theta_2)/c_2 = \sin(\theta_1)/c_1$, where $c_1$ is the speed of light in medium 1 and $c_2$ is the speed of light in medium 2.
- Only the ratio $c_2/c_1$ is important. It is often called the **index of refraction** of medium 2 with respect to medium 1.
- Note that if $\theta_1$ equals 0 so does $\theta_2$; light hitting an interface at right angles is not bent.

# Refraction of Light (3)

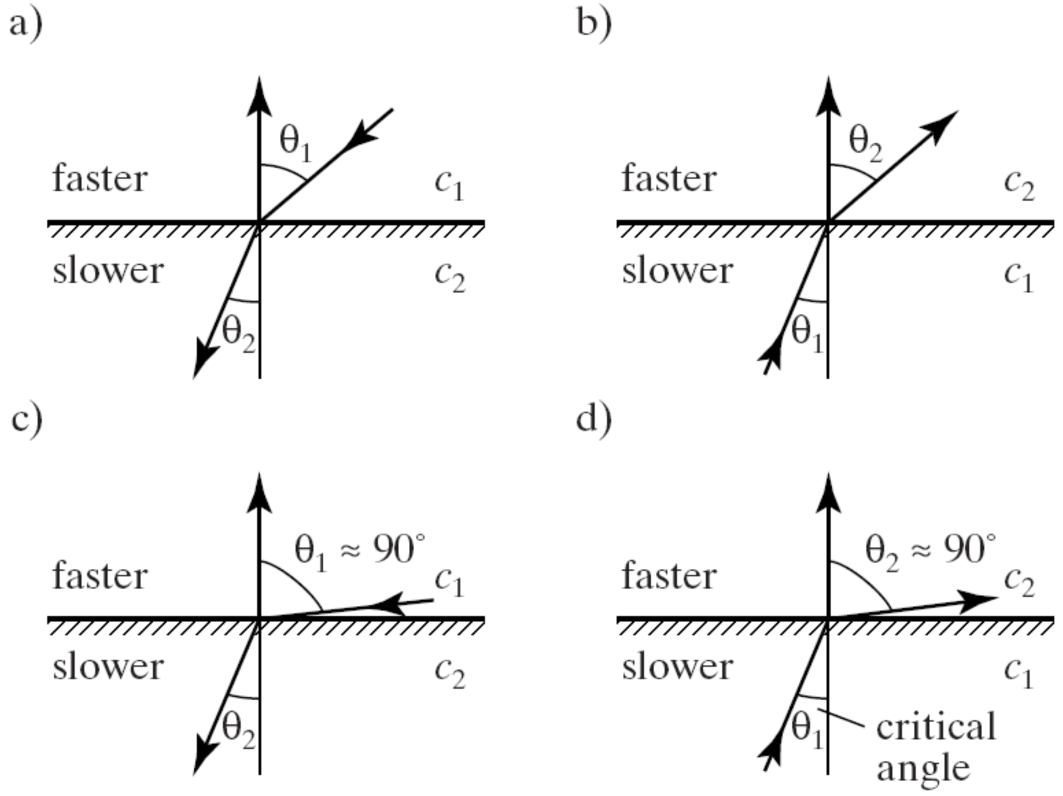
- Figure 12.72  provides a table showing the speed of light in various media relative to that in a vacuum (or air).
- This table is something of a simplification, because the speed of light generally varies with the wavelength of the light.
- For instance, the relative speed in fused quartz varies from 0.680 at l =400 nm (red) to 0.685 at l = 520 nm (green), to 0.687 at l = 680 nm (blue).
- This variation causes the familiar effect where a beam of white light is split into its rainbow of spectral colors when the beam is passed through a glass prism.

# Refraction of Light (4)

- Rays of light are bent more toward the normal direction when light enters a medium with a lower speed of light: (i.e. $c_2/c_1 < 1$).
- This is clear from Snell's Law because $sin(\theta_2)$,which equals $c_2/c_1\ sin(\theta_1)$, is less than $sin(\theta_1)$, so $\theta_2$ must be less than $\theta_1$.
- The reverse is true when light enters a medium with a higher speed of light: light is bent further away from the normal. Snell's law is completely symmetrical when subscripts 1 and 2 are interchanged.

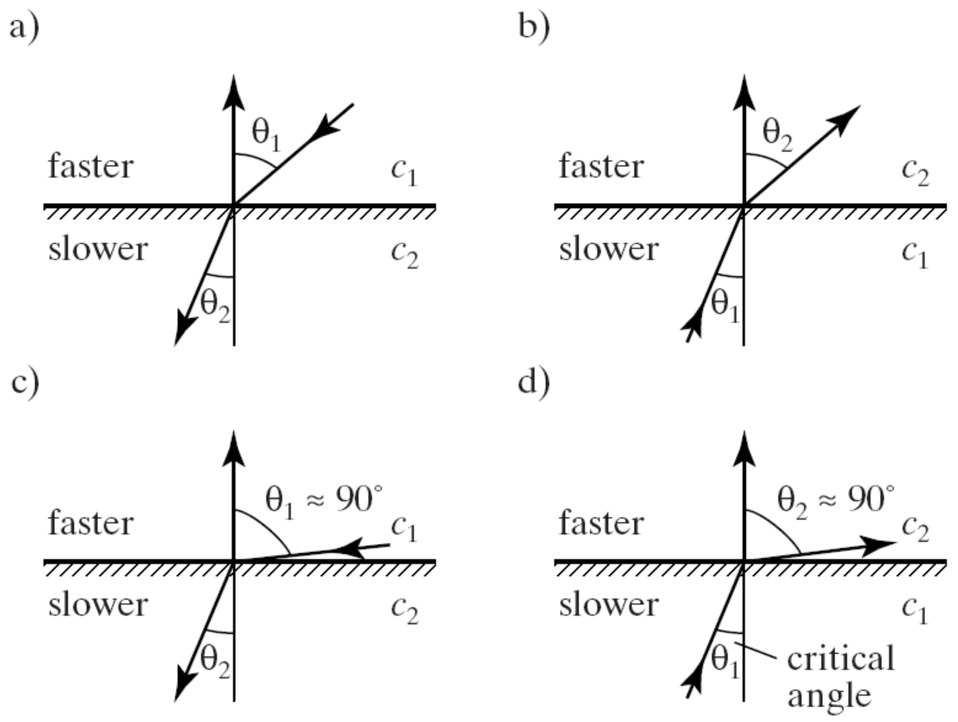
# Refraction of Light (5)

- The figure (a) shows light moving from the faster medium to the slower, and (b) shows light moving from the slower to the faster medium.
- The angles pair together in the same way in both cases; only the names change.



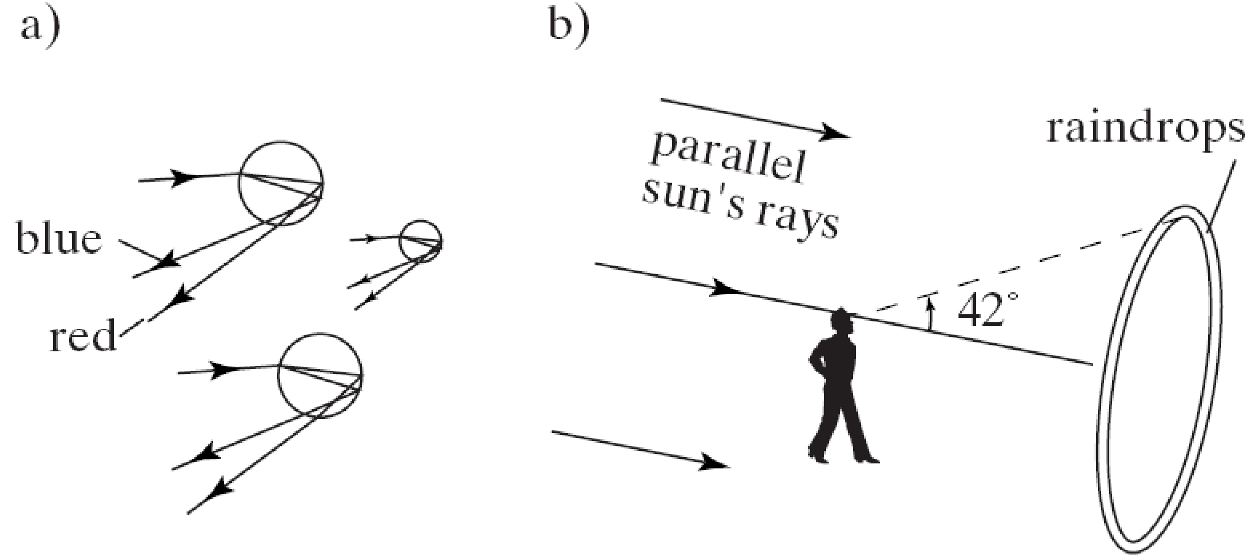
# Refraction of Light (6)

- In (c) and (d), the larger angle has become nearly $90^0$. The smaller angle is near the **critical angle**: when the smaller angle (of the slower medium) gets large enough, it forces the larger angle to $90^0$. A larger value is impossible, so no light is transmitted into the second medium. This is called **total internal reflection**.



# Example: Rainbows

- The speed of light in water varies with the light’s wavelength; and this is the genesis of rainbows. The figure shows several spherical droplets suspended in air. A ray of sunlight coming from the left is refracted slightly as it enters a droplet and experiences a total internal reflection before exiting. The angle between the incident and exiting rays is about $42^0$.

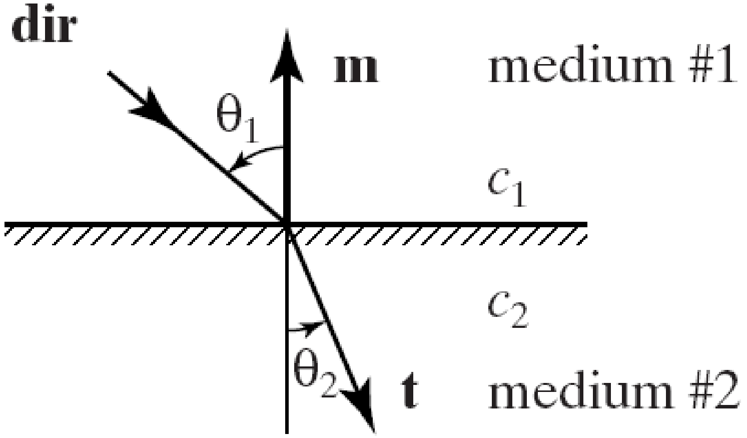


## Example: Rainbows

- Red light is refracted a little less than blue light, so the directions of the exiting rays are slightly different.
- Thus, red rays pour out of the myriad raindrops in one direction and blue rays in a slightly different direction.
- When the observer looks about $42^0$ off the sun's direction, she sees rings of light. Raindrops situated along a cone of a certain angle reflect back light of one color, change the angle slightly, and the color is slightly different.
- Thus, a rainbow is a collection of circular rings, and each observer has her own private rainbow.

# Refraction of Light (7)

- For ray tracing purposes we must find the direction, **t**, given the surface normal, **m**, and the ray direction, **dir**.
- We shall do this in a coordinate-free form using only dot products, so that it applies to any directions for **dir** and **m**.



# Refraction of Light (8)

- The resulting vector **t** is a linear combination of **m** and **dir** (assuming both have been normalized to unit length).
- Letting $R = c_2/c_1$,

  $\mathbf{t} = R\,\mathbf{dir} + [R(\mathbf{m}\cdot\mathbf{dir}) - \cos(\theta_2)]\mathbf{m}$, with $\cos(\theta_2)$ found from Snell's law,

$$\cos(\vartheta_2) = \sqrt{1 - R^2\left(1 - \left(m \bullet dir\right)^2\right)}$$

## Refraction of Light (9)

- There will be total internal reflection if the quantity in the square root of $cos(\theta_2)$ becomes negative, in which case **t** becomes irrelevant. This happens at the critical angle.
- Because the derivation of **t** uses only dot products, it is equally applicable to a 2D situation.

# Refraction of Light (10)

- When developing a ray tracer, it is simplest to model transparent objects so that their index of refraction does not depend on wavelength.
- In this case, the *same* rays are used to trace the red, green, and blue color components.
- To do otherwise would require tracing separate rays for each of the color components, as they would refract in somewhat different directions.
- This would be expensive computationally, and would still provide only an approximation, because an accurate model of refraction should take into account a large number of colors, not just the three primaries.

# Ray Tracing and Refraction

- When ray tracing scenes include transparent objects, we must keep track of the medium through which a ray is passing so that we can determine the value $c_2/c_1$ at the next intersection where the ray either exits from the current object or enters another one.
- This is most easily accomplished by adding a field to the ray that holds a pointer to the object within which the ray is traveling.
- How does shade() deal with rays that are inside objects? The answer depends on how much freedom is given to the modeler for describing scenes. We can think of several *design policies* the modeler might agree to.

# Ray Tracing and Refraction (2)

- **Design Policy 1: No two transparent objects may interpenetrate.**
- There can be no glass marble placed inside another, nor a cube of water inside a glass box. Then each ray is either in air alone, or it is inside a *single* object.
- a). Suppose the current ray in shade() is outside all objects, and upon hitting an object, say *A*, finds that *A* is transparent enough. It computes the direction **t** of the transmitted ray, using $c_1 = 1$ for air, and obtaining $c_2$ from the properties of *A*.
- The new ray is built, with its recurseLevel duly incremented, and containing a pointer to *A*. shade() is then called recursively and ultimately returns a color, which is scaled by the transparency of *A* and added to the colors accumulated so far.

# Ray Tracing and Refraction (3)

- b). Suppose, that the current ray is inside some object *A*, and hits another surface (of A). The ray is exiting into air.  The routine can check that the ray hits the same object that it is currently in.
- Because the ray is inside the object, the normal at the hit point must be reversed in sign; we want it to be pointing *into* the medium in which the ray is traveling.
- When a ray is inside an object it is usually considered not to be bathed in light, so no local ambient, diffuse, and specular intensities are computed.

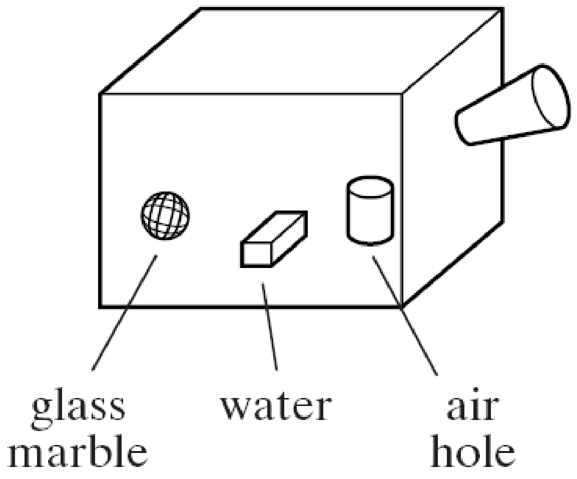
# Ray Tracing and Refraction (4)

- The inside wall of *A* might be considered shiny enough to warrant casting a reflected ray back into *A*. In that case the reflected ray is created and cast as usual.
- For the refracted ray, *A* is obviously transparent enough, so the value of $c_1$ is taken from the properties of *A*, and $c_2$ is set to 1 for air.
- If the angle of incidence is less than the critical angle, a new ray is spawned, (with its pointer set to NULL since it is not inside any object now), and sent on its way, to gather more light contributions.

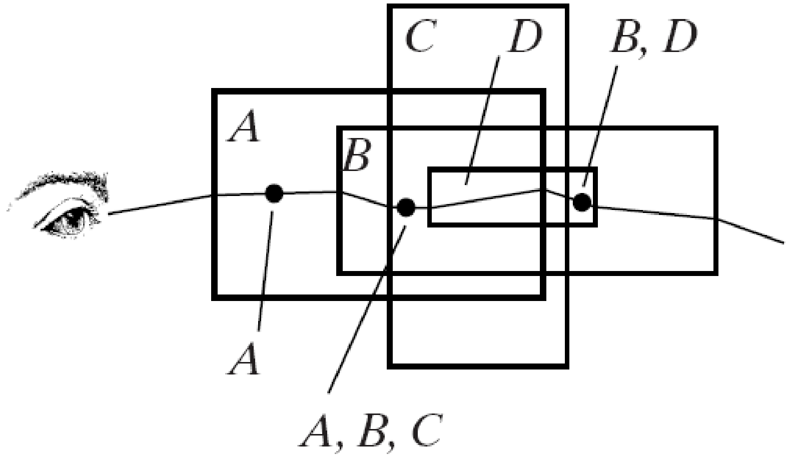
# Ray Tracing and Refraction (5)

- **Design Policy 2: Transparent objects may interpenetrate.**
- The figure shows a glass cube with several objects imbedded in it: a quartz spherical marble, a cylindrical air hole, a cube filled with water, and a partially imbedded glass cone. Each has its own speed of light.

a)



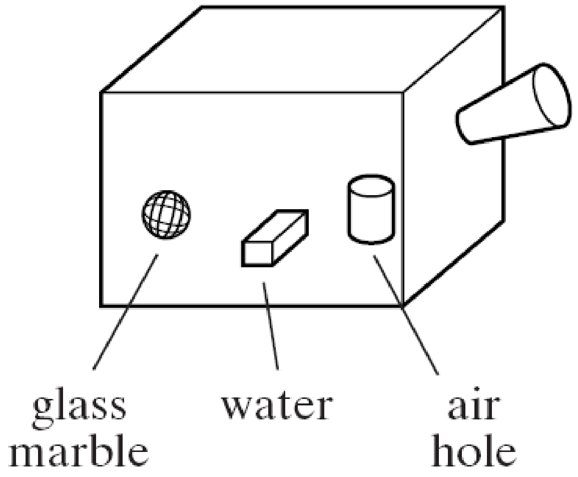
b)



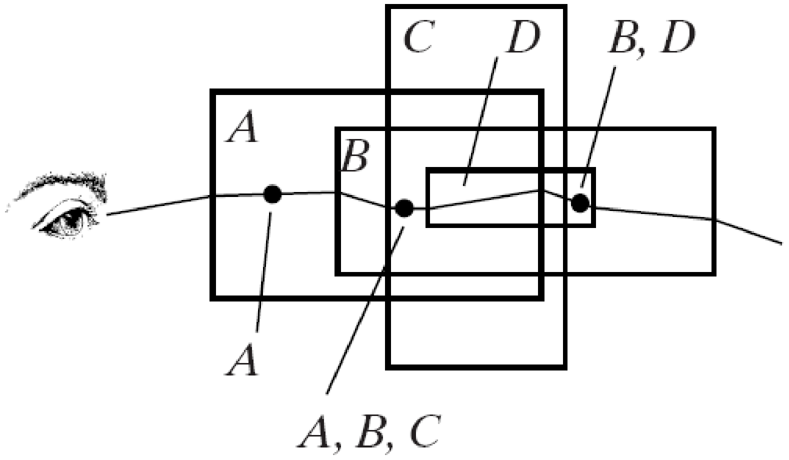
# Ray Tracing and Refraction (6)

- The figure (b) shows a ray traveling through a set of transparent objects, entering and exiting in a complex sequence. A list of objects is shown with each segment of the ray, reporting which objects the ray is traveling in.
- The complexity illustrates the costs to insure that shade() will always handle the situation correctly.

a)



b)



# Ray Tracing and Refraction (7)

- What does it mean to be inside two transparent objects at once, such that certain points in space are owned by two objects?
- In the modeling phase you have to decide what the nature of the material is inside each joint region. If a green marble is completely enclosed in a blue marble, it makes sense to say that the joint region belongs to the blue marble and assign it the color blue. But if objects partially interpenetrate, there is no obvious way to decide whose properties to use in the joint regions.
- The designer must assign a priority to each object with the understanding that the color of the object with the highest priority dominates in each joint region.

## Ray Tracing and Refraction (8)

- One way to handle this is to add a field to each object instance providing its priority and to augment the pointer in the Ray data structure so it becomes a list of pointers.
- At any instant, the ray is inside a certain set of objects, and the list reports this set. If the list is empty the ray is outside of all objects.
- So we might enhance the Ray type by adding a list, implemented as an array, to facilitate copying the whole object at once when making a new ray.

# Ray Class Changes

```
class Ray{
public:
   Point3 start;
   Vector3 dir;
   int recurseLevel;
   int row, col; // to assist with screen extents
   int numInside; // number of objects on the list
   GeomObj* inside[8]; // array of object pointers
   Ray(){recurseLevel = numInside = 0;} //constructor
   …other methods …
};
```

# Ray Tracing and Refraction (9)

- How is this inside list used in shade()? At any time, the current ray is inside some collection of objects (in the list).
- When it hits the next surface of some object, say *B*, we take different actions depending on whether the ray is entering or exiting *B,* which is determined from the isEntering field of the hit record.
- a). Entering *B*. If *B* is not transparent enough, stop spawning refracted rays, but do spawn a reflected ray if *B* is shiny enough.
- If *B* is transparent enough, use for $c_1$ the speed of light of the highest priority object currently on the list. If *B* has a higher priority then use its speed of light for $c_2$; otherwise set $c_2$ equal to $c_1$.

# Ray Tracing and Refraction (10)

- Add *B* to the list. Make a new transmitted ray, copying the current list (with *B* added) into it. Call shade() recursively.
- b) Exiting *B*: For $c_1$ use the speed of light of the highest priority object in the list.
- Remove *B* from the list.
- For $c_2$ use the speed of light of the highest priority object still on the list. Make a new transmitted ray, and copy the current inside list into it. Call shade() recursively.

## Example Scene

- The figure shows a scene containing a globe and jack which are transparent, whereas the floors and walls are not.
- The distortions produced when light is refracted through a transparent object are apparent.