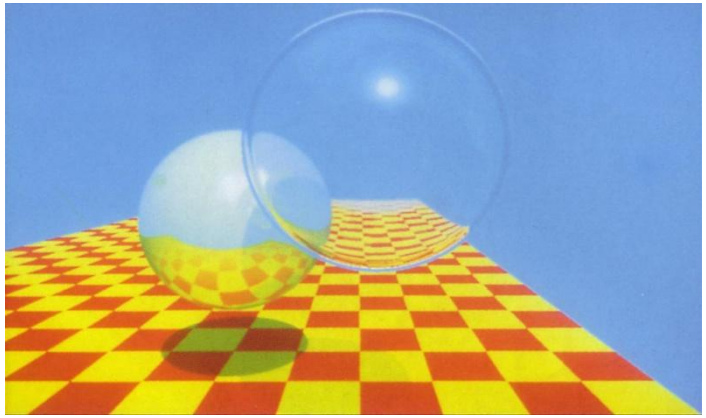
Computer Graphics

Shi-Min Hu

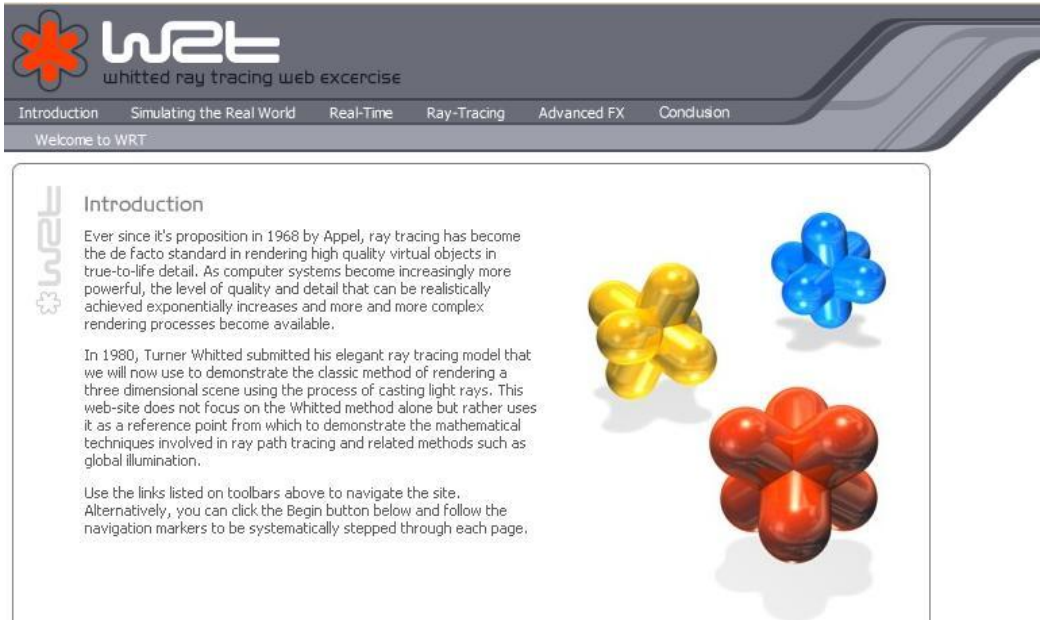
Tsinghua University

History of Ray Tracing (光线跟踪)

- **In 1980, Whitted proposed a ray tracing model, include light reflection and refraction effects. A Milestone of Computer Graphics.**



- Turner Whitted， An improved illumination model for shaded display， Communications of the ACM, v.23 n.6, p.343-349, June 1980.
- http://www.raytracing.co.uk/study/home.htm

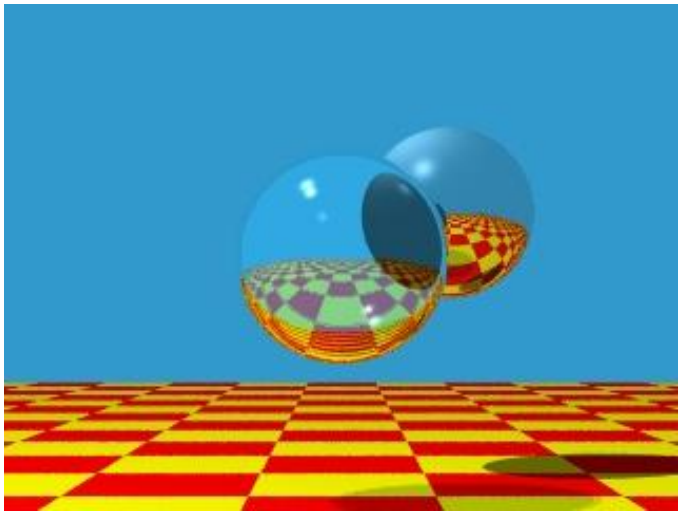


- After receiving his PhD from NCSU in 1978, Turner Whitted left for Bell Labs and proceeded to shake the CGI world with an algorithm that could ray-trace a scene in a reasonable amount of time.
- He only has 14 papers, and Ray tracing is his first paper.

- He was elected as member of National Academy of Engineering in 2003.

Ray Tracing(光线跟踪)

- Introduction of ray tracing
- Ray intersection(光线求交)
- shadows(阴影)
- Transparence and specular reflection (透明和镜面反射)
- textures(纹理)

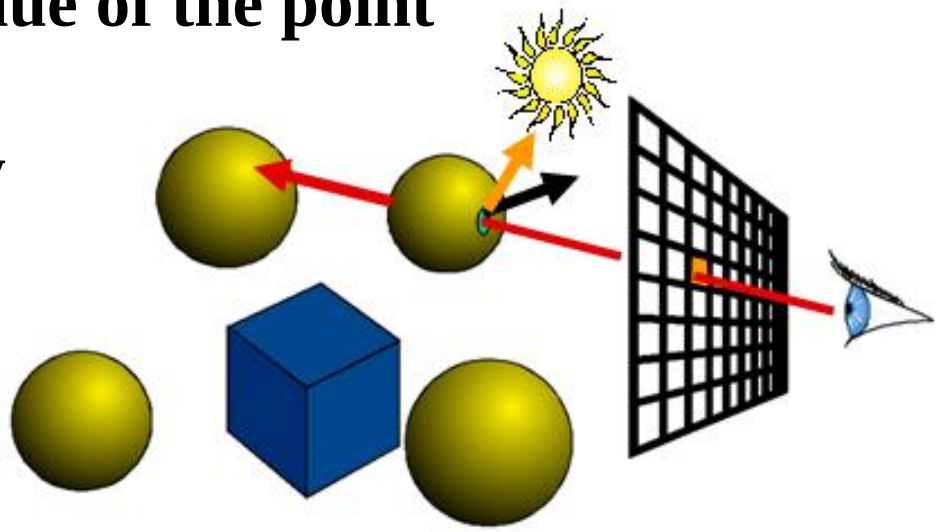


Introduction of ray tracing

- **Ray tracing**
  - **Because of its effectiveness, ray tracing is a widely used and very powerful rendering (drawing) technique**
  - **Why we see objects?**
    - **Light can be interpreted as a collection of rays that begin at the light sources and bounce around the objects in the scenes.**
    - **We see objects become rays finally come into our eyes.**

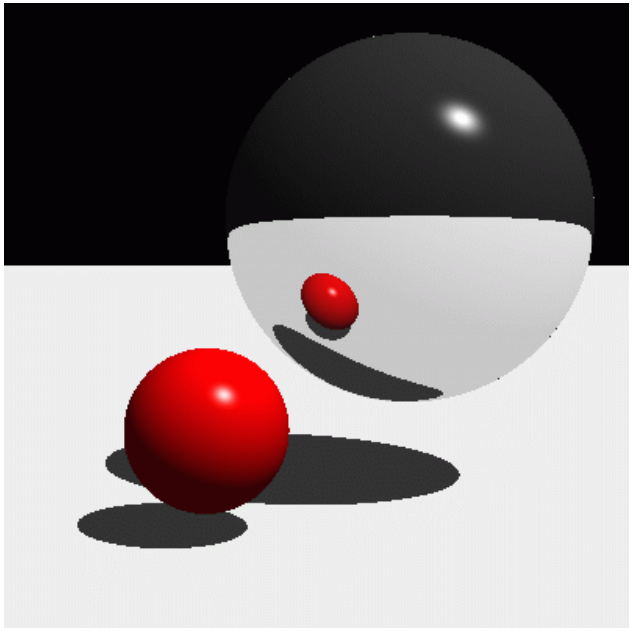
Basic Idea of ray tracing

- Think of the frame buffer as a simple array of pixels, with eye looking through it into the scene
- For each pixel, what can we “see”?
- a ray casting from the eye through the center of the pixel and out into the scene, its path is traced to see which object the ray hits first
- calculate the shading value of the point by the Phong model
- continue to trace the ray in the scene to achieve reflection, refraction..

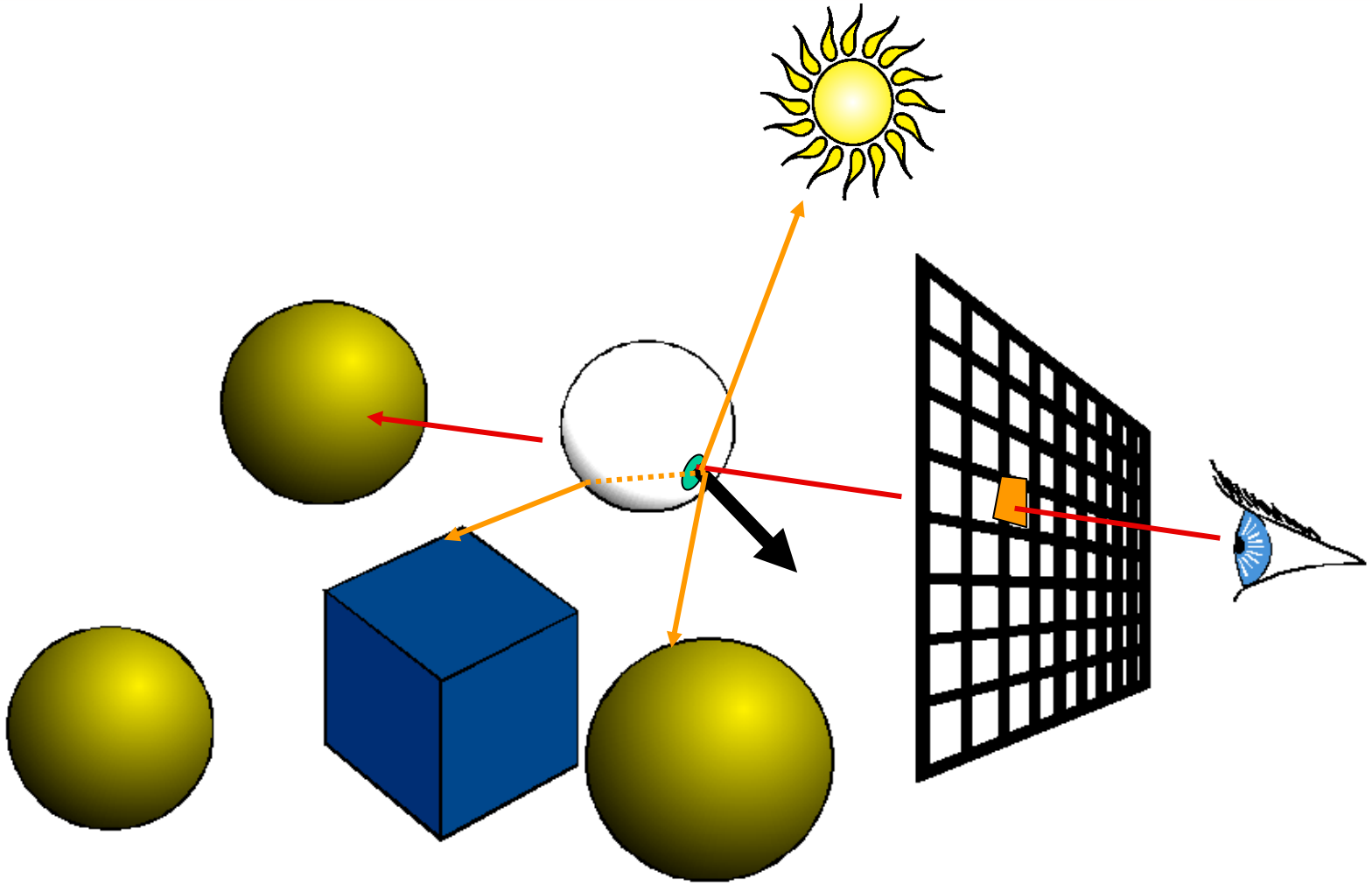


Introduction of ray tracing

- **Features**
  - **Easy to incorporate interesting visual effects, such as shadowing, reflection and refraction, since the path of a ray is traced through the scene**
  - **Besides geometric primitives (such as spheres, cones, cubes), easy to work with a richer class of objects, including polygonal meshes, compound objects.**



Recursive Ray Tracing



Recursive Ray Tracing

```
IntersectColor( vBeginPoint, vDirection)
{
   Determine IntersectPoint;
   Color = ambient color;
   for each light
        Color += local shading term;
   if(surface is reflective)
      color += reflect Coefficient *
           IntersectColor(IntersecPoint, Reflect Ray);
   else if ( surface is refractive)
     color +=  refract Coefficient *
           IntersectColor(IntersecPoint, Refract Ray);

   return color;
}
```

demos of ray tracing

- **DENG Jia (计2 邓嘉)’s Demo**

Play Video

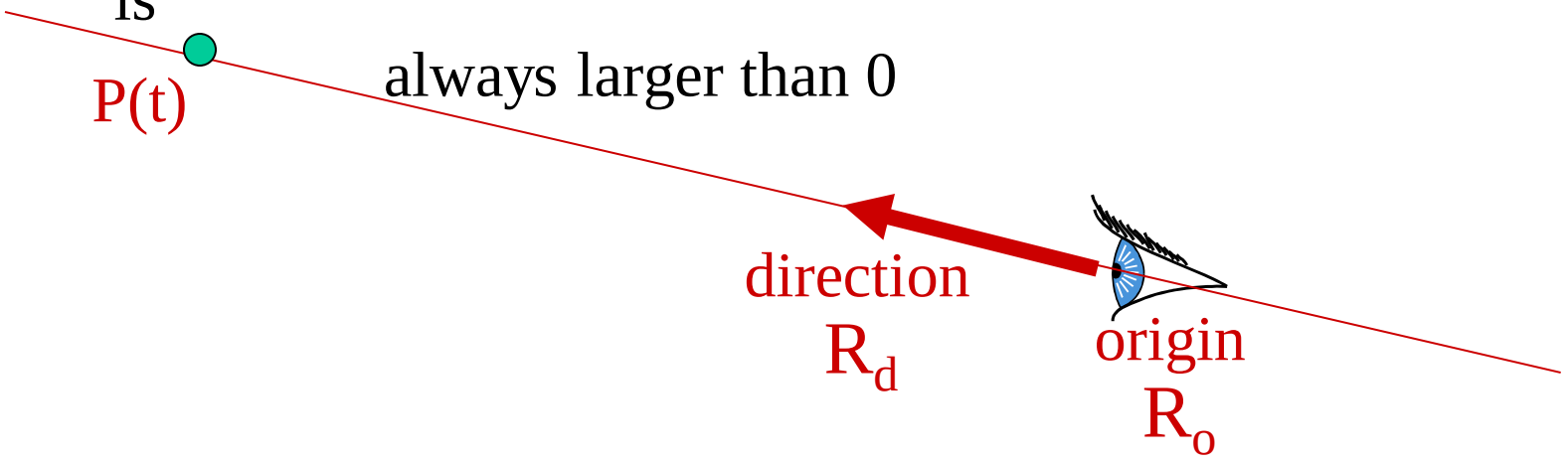
- **Deng Jia, 2002-2006, Tsinghua University, No. 1 in GPA.**
- **2006- , Princeton Univetrsity, published a paper on relief in ACM SIGGRAPH 2007**
- **Deng Jia’s story in Media computing (late)**

Ray intersection (光线求交)

- **Ray tracing**
  - **Ray Representation**
  - **Plane intersection**
  - **Triangle intersection**
  - **Polygon intersection**
  - **Sphere intersection**
  - **Box intersection**

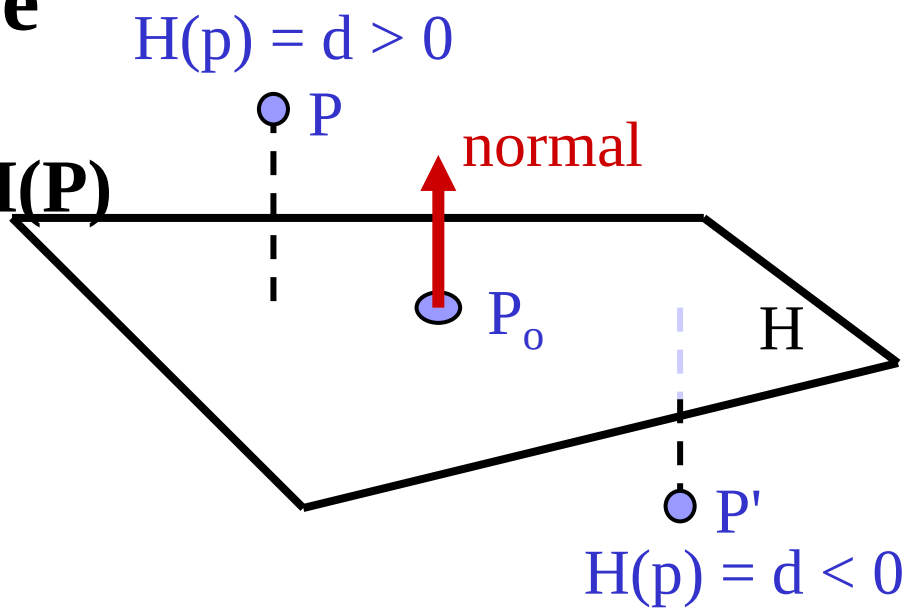
Ray representation

- **Parametric representation**
  - $P(t) = R_o + t * R_d$
  - where $R_o=(x_o,y_o,z_o)$ is the original point of the ray , $R_d=(x_d,y_d,z_d)$ is the direction the ray is going on, usually the direction is normalized
  - t value determines the point the ray arrives at, its value is always larger than 0



Plane Intersection

- **Plane Definition**
  - Explicit: $P_o=(x_o,y_o,z_o)$, $n=(A,B,C)$
  - Implicit: $H(P) = Ax+By+Cz+D = 0$
    $= n \cdot P + D = 0$
- **Point Plane Distance**
  - **If n is normalized, the distance is d = H(P)**
  - **But note that, d is signed distance**



Plane Intersection

- **Where does the ray intersect this plane?**
- **Given a plane with equation:**

  $\mathbf{n \cdot P + D = 0;}$
- **Intersection means satisfy both equations:**

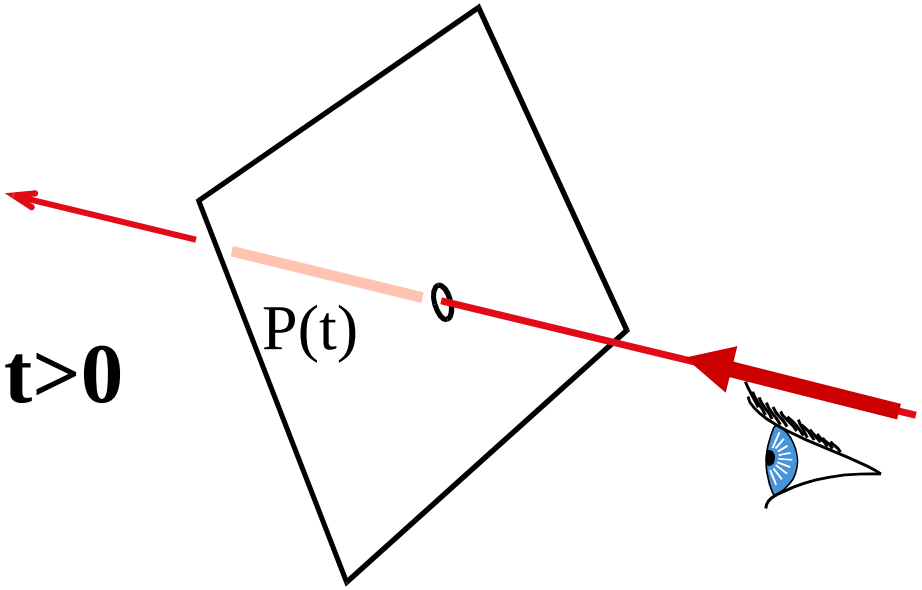
$P(t) = R_o + t * R_d$

$n \cdot P(t) + D = 0;$

So we have

$t = -(D + n \cdot R_o)/(n \cdot R_d)$

- **Just need to verify if t>0**



Triangle Intersection

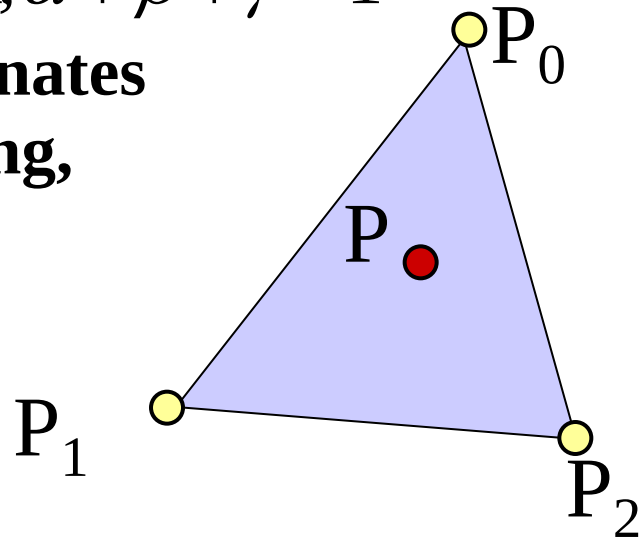
- **In real-time graphics, triangle geometry (triangle mesh) is usualy stored, and each triangle is defined by three vertices.**
- **There exists many different ray/triangle intersection methods, main steps are:**
  - **First compute the intersection point between the ray and the triangle's plane**
  - **Thereafter, project the intersection point onto the triangle's plane**
  - **Decide whether or not the point is inside the triangle**

Triangle Intersection

- **Barycentric coordinates:**
  - **A point P, on a triangle $P_0P_1P_2$, is given by the explicit formula:**

    $$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

    where (α, β, γ) are the barycentric coordinates, which must satisfy $0 \le \alpha, \beta, \gamma \le 1, \alpha + \beta + \gamma = 1$
  - **Note that the barycentric coordinates could be used for texture mapping, normal interpolation, color interpolation, etc.**



Triangle Intersection

- Since $\alpha + \beta + \gamma = 1$, we can write $\alpha = 1 - \beta - \gamma$, We have:

$$P = (1-\beta-\gamma)P_0 + \beta P_1 + \gamma P_2$$

- Set ray equation equal to barycentric equation:

$$R_o + tR_d = (1-\beta-\gamma)P_0 + \beta P_1 + \gamma P_2$$

- Rearrange the terms, gives:

$$\begin{pmatrix} R_d & P_0 - P_1 & P_0 - P_2 \end{pmatrix} \begin{pmatrix} t \\ \beta \\ \gamma \end{pmatrix} = P_0 - R_0$$

- The means the barycentric coordinate and distance t can be found by solving this linear system of equations

Triangle Intersection

- Denoting $E_1 = P_0 - P_1, \mathrm{E}_2 = P_0 - P_2, S = P_0 - R_0$ the solution to the equation above is obtained by Cramer's rule:

$$\begin{pmatrix} t \\ \beta \\ \gamma \end{pmatrix} = \frac{1}{\det(d, E_1, E_2)} \begin{pmatrix} \det(S, E_1, E_2) \\ \det(d, S, E_2) \\ \det(d, E_1, S) \end{pmatrix}$$

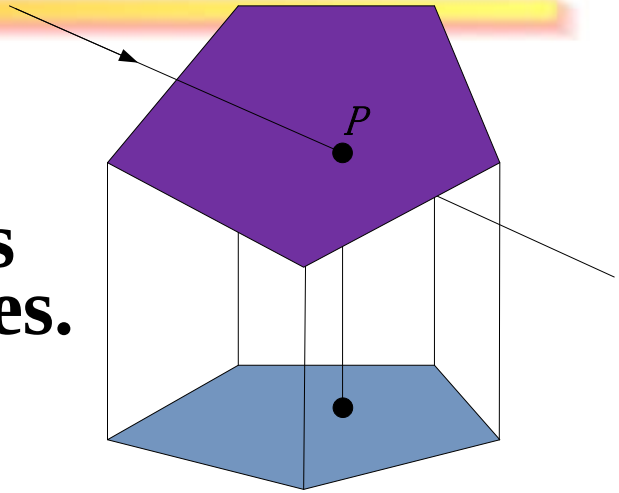
- Then check $0 \le \beta, \gamma \le 1, \beta + \gamma \le 1$ to determine whether or not the intersect point is inside the triangle

Polygon Intersection

- **Even though triangles are the most common rendering primitive, a routine to compute the intersection between a ray and a polygon is useful.**
- **A polygon of n vertices is defined by an ordered vertex list $\{v_0, v_1, \ldots, v_{n-1}\}$, where vertex v_i forms an edge with for $0 <= i < n-1$, and the polygon is closed by the edge from v_{n-1} to v_0. The plane of the polygon is denoted by**

$$\pi_p : n_p \cdot x + d_p = 0$$

Polygon Intersection



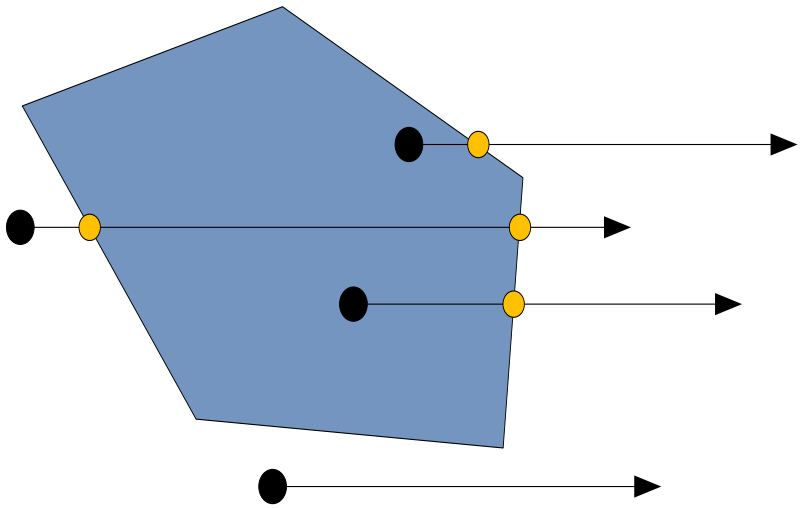
- **We first compute the intersection between the ray and the plane. The solution has been presented in previous slides.**
- **If there is an intersection, next we need to determine whether or not the intersection point P is inside the polygon.**
- **This is done by projecting all vertices and P to one of the xy-,xz-,yz planes, as shown in the figure**

Polygon Intersection

- **The question left is a 2D point-in-polygon problem. Here we review one of the most useful algorithm – the crossing test**
- **The crossing test is based on** *Jordan Curve Theorem*, **which says that a point is inside a polygon if a ray from this point in an arbitrary direction crosses an odd(奇数的) number of edges. This test is also known as the even-odd test.**

Polygon Intersection

- **The crossing test is illustrated in the right figure:**
  - **Two black points inside polygon (cross one edge)**
  - **two black points outside polygon (one crosses 2 edges and the other crosses 0 edge)**
- **The crossing test is the fastest test that does not use preprocessing (预处理)**



Polygon Intersection

- **The test point P can also be thought of as being at the origin, and the edges may be tested against the positive x-axis instead**
  - **If the y-coordinates of the edge have the same sign, then the edge does not cross x-axis**
  - **Else, compute the x-coordinate of the intersection between x-axis and the edge**
    - **If positive, the edge crosses x-axis, the number of crossing increases by 1**
    - **Else, the edge does not cross the x-axis**

- The other method

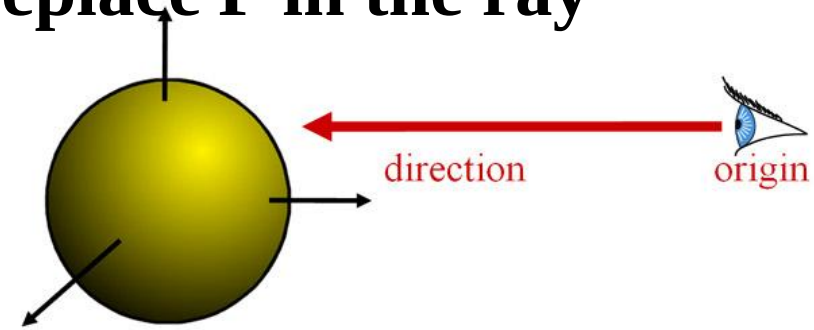
Sphere Intersection

- **Now, let's look at the intersection test between a ray and a sphere**
- **Mathematical solution**
  - **sphere defined as:**
    - **a center point P_c , and a radius *r* ; the implicit formula for the sphere is :**

$$f(P) = \|P - P_c\| - r = 0$$

  - **To solve for the intersection between a ray and a sphere, simply replace P in the ray equation to yield:**

$$\|P(t) - P_c\| - r = 0$$

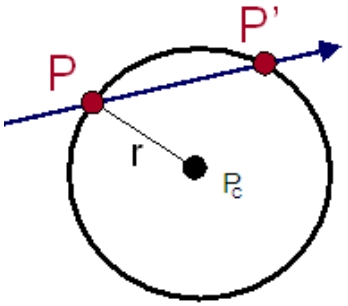


Sphere Intersection: Algebra method

- **The equation of last page is simplified as follows:**

$$\|P(t) - P_c\| - r = 0$$

$$\|R_0 + tR_d - P_c\| = r$$



$$(R_0 + tR_d - P_c) \cdot (R_0 + tR_d - P_c) = r^2$$

$$t^2(R_d \cdot R_d) + 2t(R_d \cdot (R_0 - P_c)) + (R_0 - P_c) \cdot (R_0 - P_c) - r^2 = 0$$

Since Rd is normalized:

$$t^2 + 2t(R_d \cdot (R_0 - P_c)) + (R_0 - P_c) \cdot (R_0 - P_c) - r^2 = 0$$

Rewrite as: $t^2 + 2tb + c = 0$

Solution is: $t = -b \pm \sqrt{b^2 - c}$

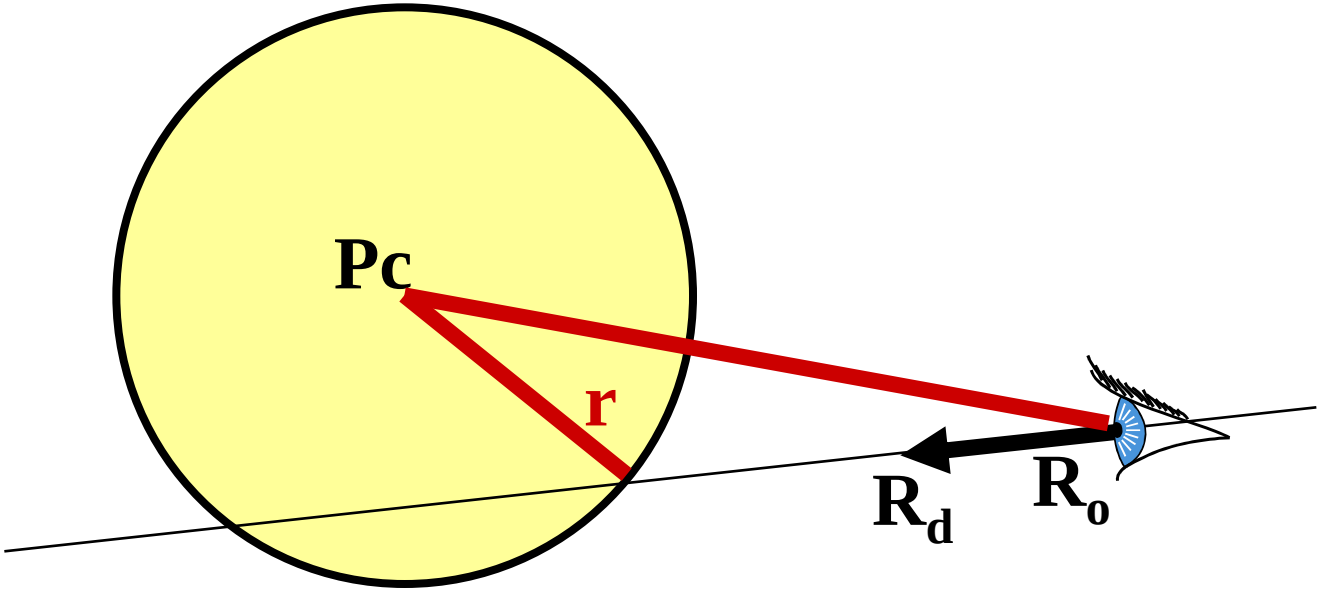
Sphere Intersection

- **Previous algebra solution could be improved, e.g., observing that intersections behind the ray origin are not needed**
- **An optimized Solution: Geometric method**
  - **Easy reject (No intersection testing)**
  - **Easy to check ray origin inside or outside the sphere**
  - **Easy to check which point is closed to ray from sphere origin**
  - **Ray direction: pointing to or away from sphere**

Sphere Intersection: Geometric Method

- **Optimized Solution**
  - **We first compute the vector $\vec{l}$ from the ray origin to the center of the sphere:**

$$\vec{l} = P_c - R_0$$



Sphere Intersection

- **Optimized Solution**
  - **compute the vector $\vec{l}$ from the ray origin to the center of the sphere:**

$$\vec{l} = P_c - R_0$$

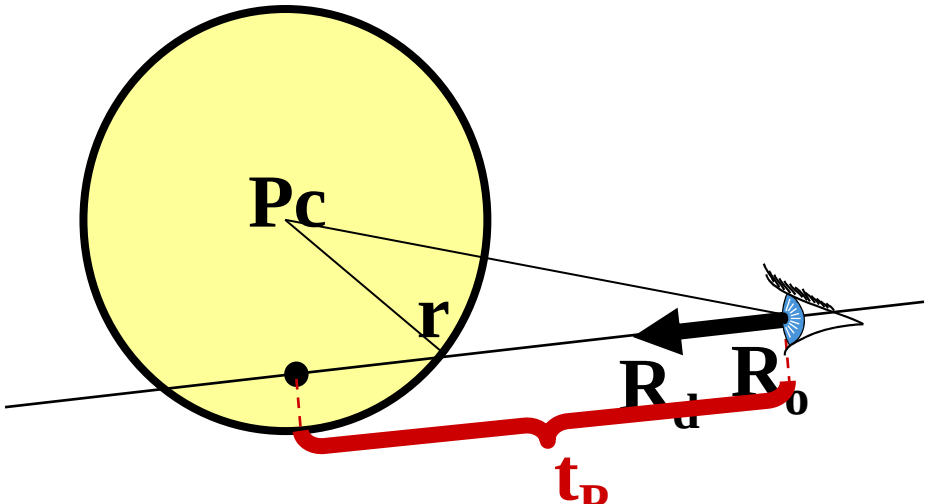
  - **Is ray origin inside/outside the sphere?**
    - **Inside the sphere:** $\vec{l}^2 < r^2$
    - **Outside the sphere:** $\vec{l}^2 > r^2$
    - **On the sphere:** $\vec{l}^2 = r^2$
  - **If ray origin is on the sphere, be careful about degeneracy**

Sphere Intersection

- **Next, Find closest point to sphere center:**

$$t_p = \vec{l} \cdot R_d$$

  - **If origin outside & $t_P < 0$ → no hit**
    - **the sphere is behind the ray origin and we can reject the intersection**

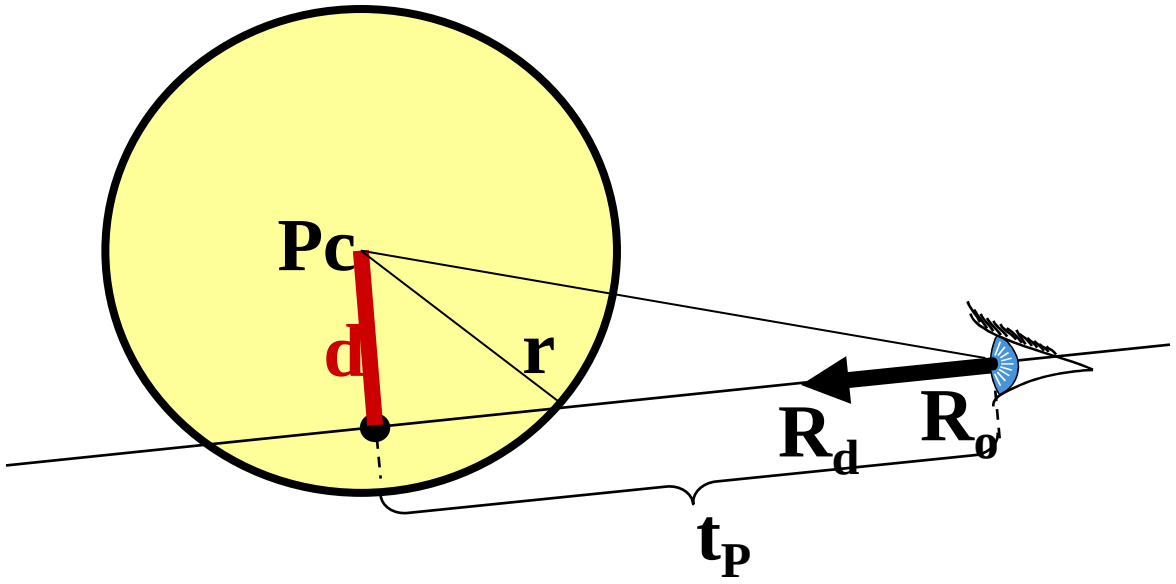


Sphere Intersection

- **Next, find squared distance d from the sphere center to the closest point**

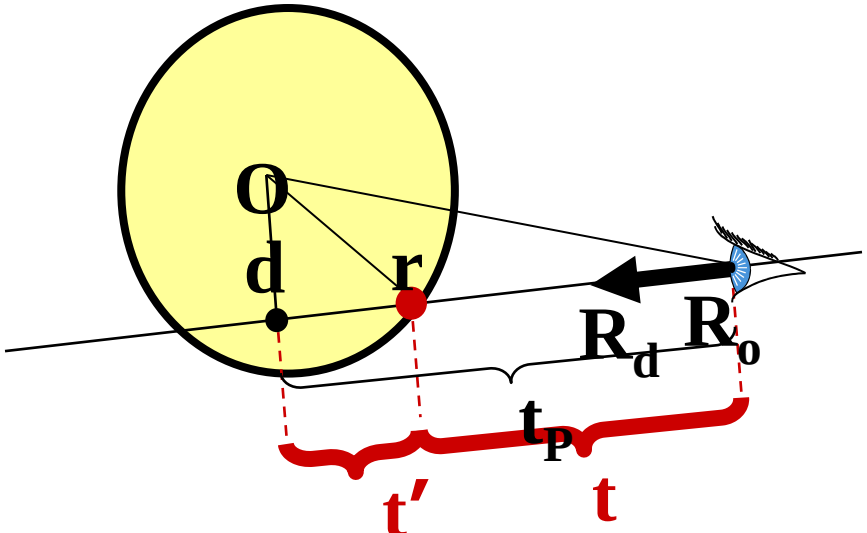
$$d^2 = \vec{l}^2 - t_p^2$$

- **If d>r , no hit**



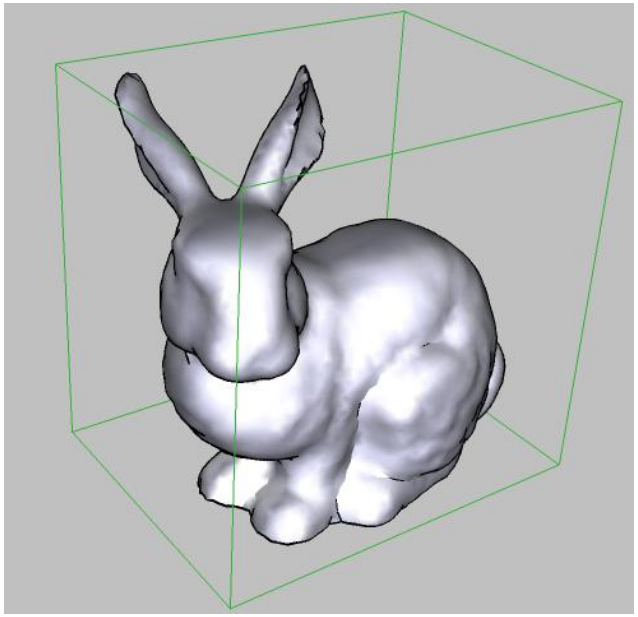
Sphere Intersection

- **Find distance (t') from closest point (t_P) to correct intersection: $t'^2 = r^2 - d^2$**
- **Then the solution will be**
  - **If origin outside sphere $\rightarrow t = t_P - t'$ ()**
  - **If origin inside sphere $\rightarrow t = t_P + t'$**



Box Intersection

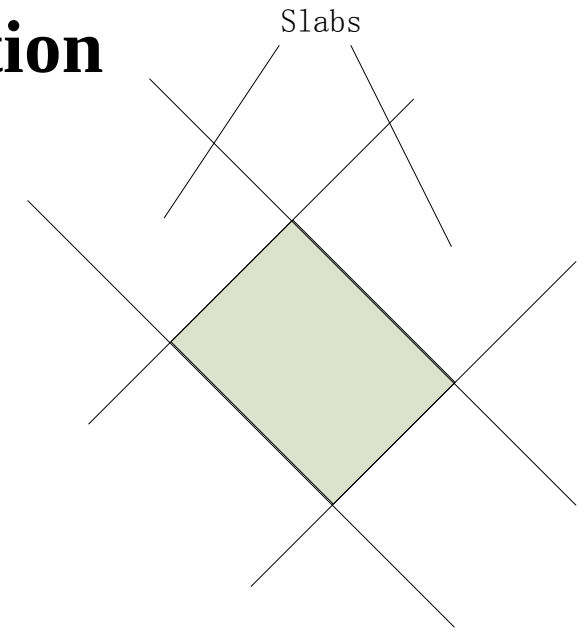
- **Ray/Box intersection test are important in graphics, since we often bound geometric objects with boxes, which is called bounding box(包围盒).**
  - **With bounding box, when testing a ray intersect with an object, we first test the ray intersect with the bounding box, if not intersected, then the ray will not intersect with the object definitely.**



Box Intersection

- **Here, we introduce a slab based method for box intersection test, proposed by Haines**
  - **A slab is simply two parallel planes, which are grouped for faster computation**
  - **A box is composed of 3 slabs**

    As illustrated in the right figure in 2D



Box Intersection

- **For each slab, intersecting with the ray, there is a minimum t value and maximum t value, these are called $t_i^{\min}$ and $t_i^{\max}$. (i=0,1,2)**
- **The next step is to compute:**

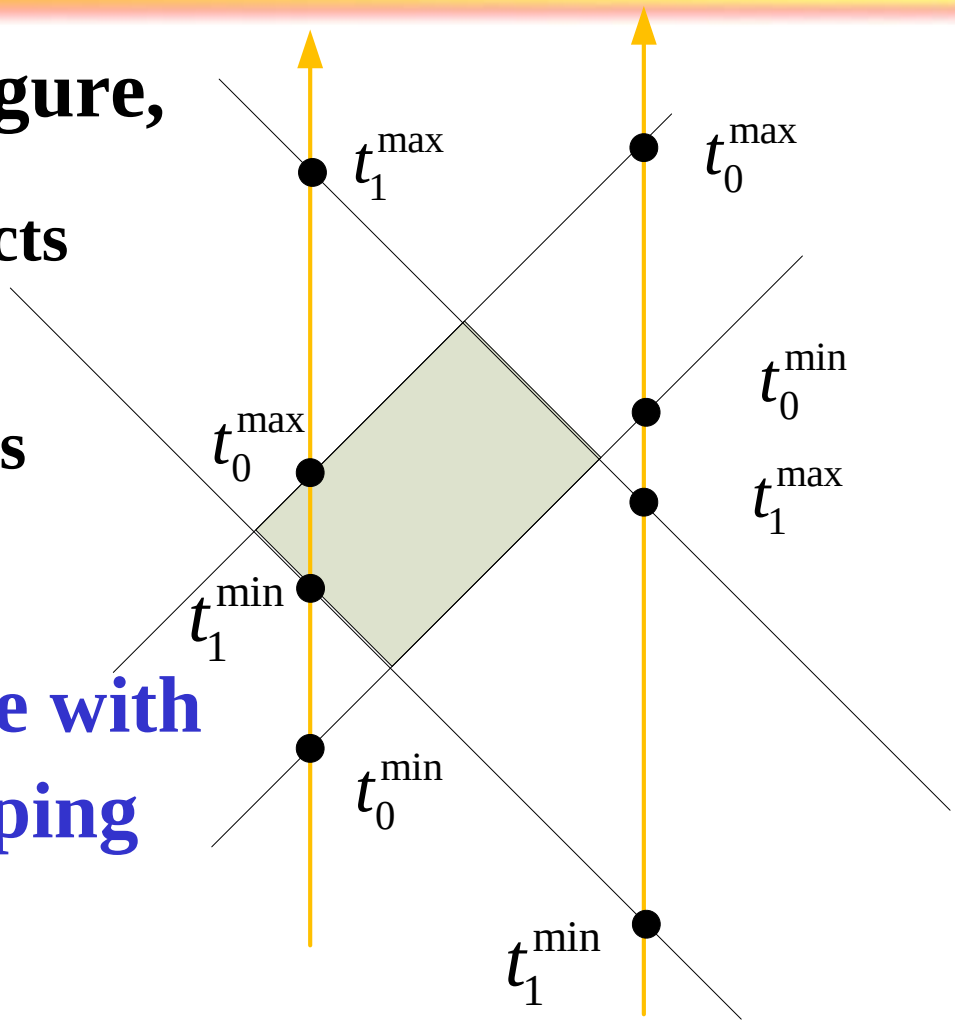
$$t^{\min} = \max(t_0^{\min}, t_1^{\min}, t_2^{\min});$$

$$t^{\max} = \min(t_0^{\max}, t_1^{\max}, t_2^{\max});$$

- **Now, the clever test:**
  - **if $t^{\min} < t^{\max}$,then the ray intersects the box $t^{\min}$ is the enter-point, $t^{\max}$ is the exit point**
  - **Otherwise, no intersections**

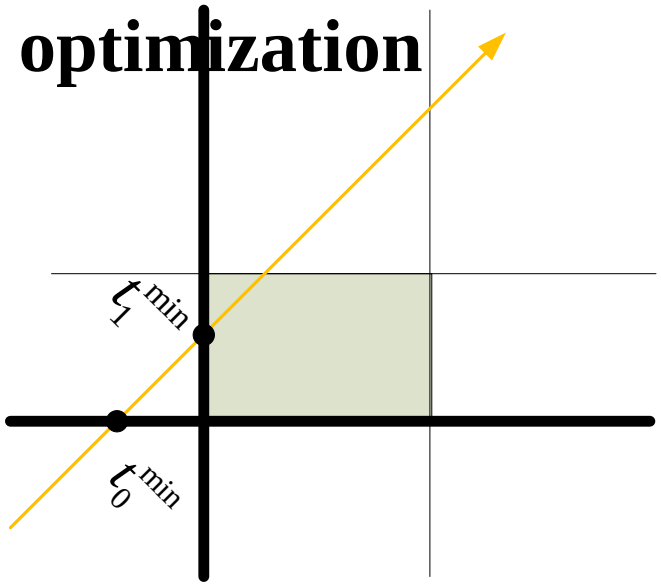
Box Intersection

- **Look at the right figure, there are 2 rays:**
  - **The left ray intersects with the box**
    it has $t^{\min} < t^{\max}$
  - **The right ray misses**
    it does not have
    $t^{\min} < t^{\max}$
- **The idea is the same with Liang-Barsky Clipping**



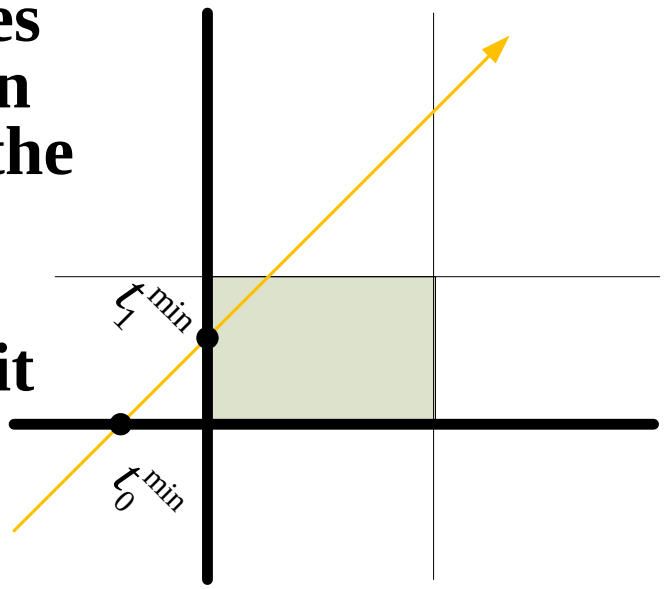
Box Intersection

- **Woo's Method for intersection between a ray and an axis-aligned box(和坐标轴平行)**
  - **Woo introduced some smart optimization**
  - **specific for axis-aligned box**



Box Intersection

- **Woo's Method**
  - **First, identity three candidate planes out of the six planes. For each pair of parallel planes, the back–facing plane can be omitted for further consideration.**
  - **After finding the three planes we computed the intersection distances (t-value) between the ray and the planes.**
  - **The largest of the t-values corresponds to a potential hit**



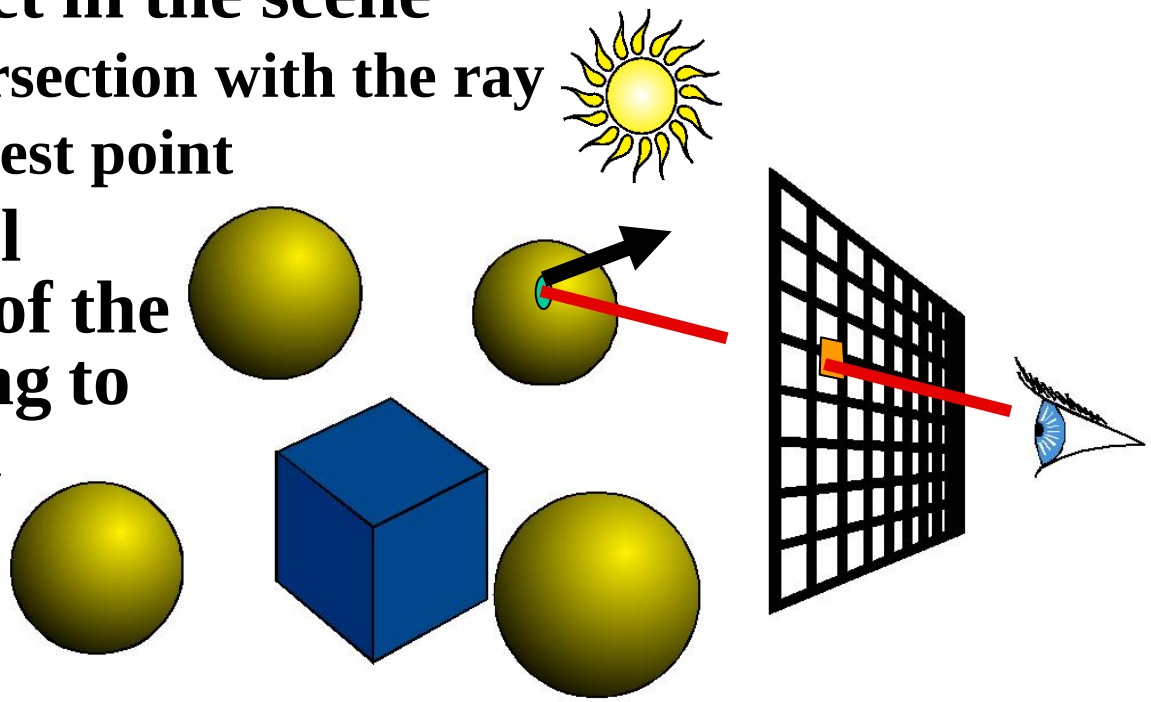
Box Intersection

- **Woo's Method**
  - **Use the potential hit t-value to compute the intersection point**
  - **if the intersection point is located on the face of the box, then it is a real hit**
- **Slabs method vs Woo's Method**
  - **Comparable in performance**

- **With above discussion of ray intersection, let's ray tracing**

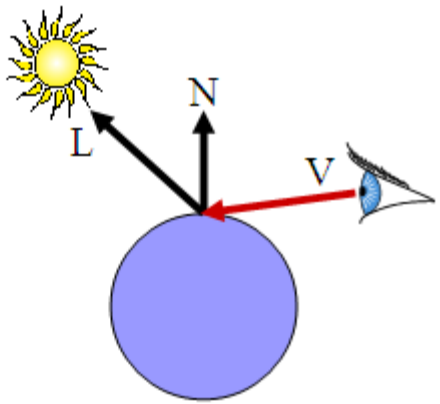
The Simplest Ray Tracing: Ray Casting

- For each pixel
  - Cast a ray from the eye, through the center of pixel, to the scene
  - For each object in the scene
    - Find the intersection with the ray
    - Store the closest point
  - Calculate local shading term of the point according to light, material and normal

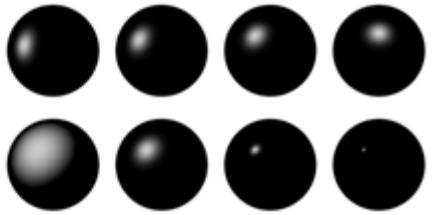


The Simplest Ray Tracing: Ray Casting

- **Shading results depend on surface normal, light direction, light intensity, view direction, material property and so on**
- **Do not account for secondary ray, so do not have shadow, reflection and refraction effects**



Diffuse sphere

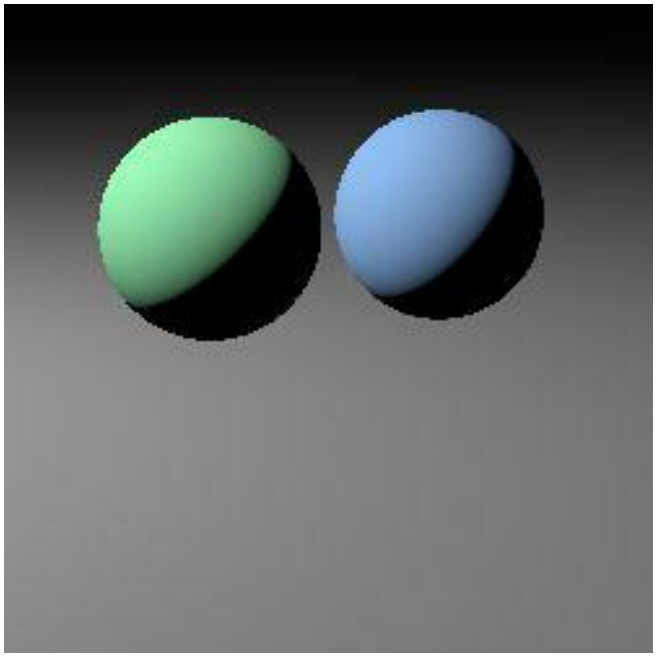


Specular spheres

Sample Code of Ray casting

- for each pixel
  - cast a ray and find the intersection point
  - if have intersection
    - color = ambient
    - for each light

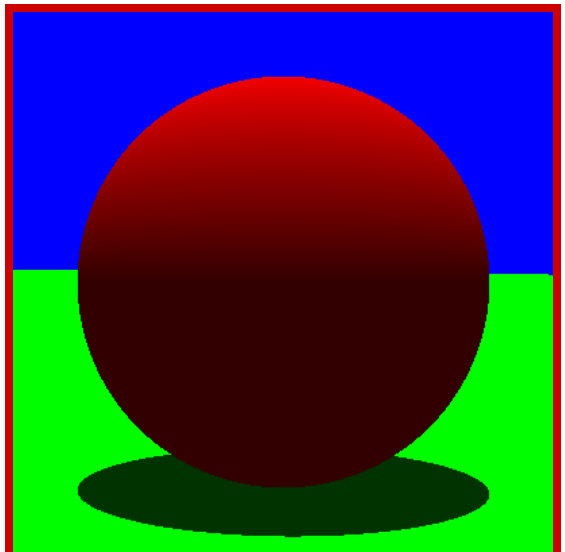
      color += shading from this light (depending on light property and material property)
    - return color
  - Else
    - return background_color



Add Shadows

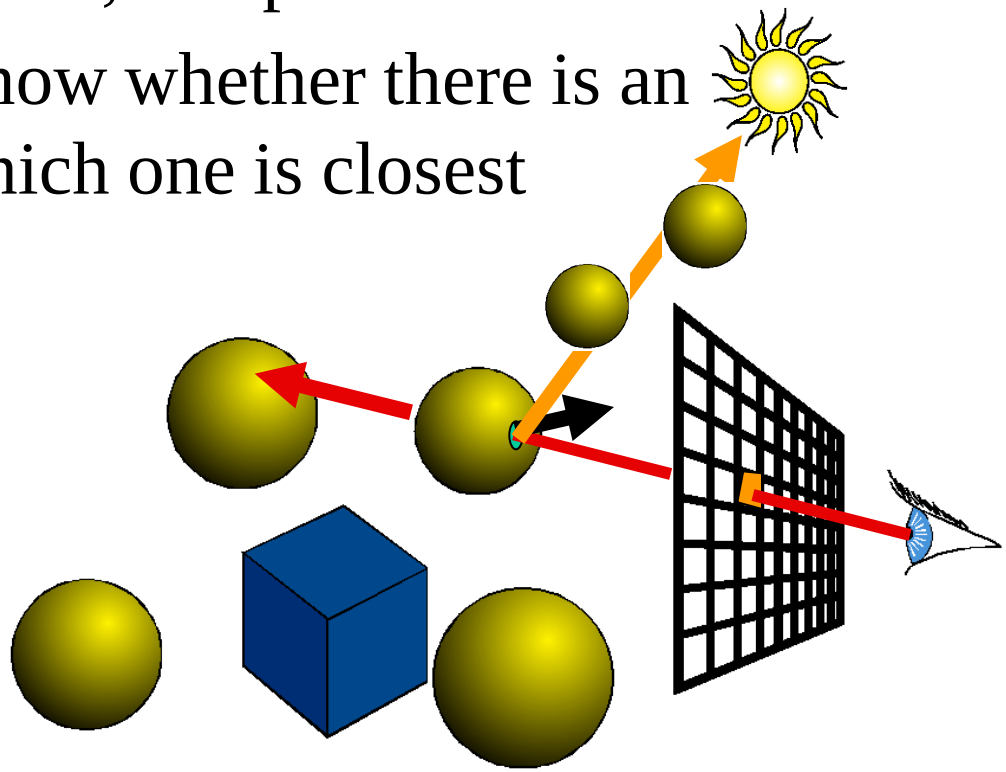
- For each pixel
  - Cast a ray and find the intersection point
  - color = ambient
  - for each light
    - if intersection point is not in shadow area of the light (evaluated by shadow rays)

      color += shading from this light
  - return color



Add Shadows

- As illustrated in the figure below, casting a shadow ray from intersection point to the light, if there is an intersection, this point is in shadow
- We only want to know whether there is an intersection, *not* which one is closest



Add Ray reflection and refraction

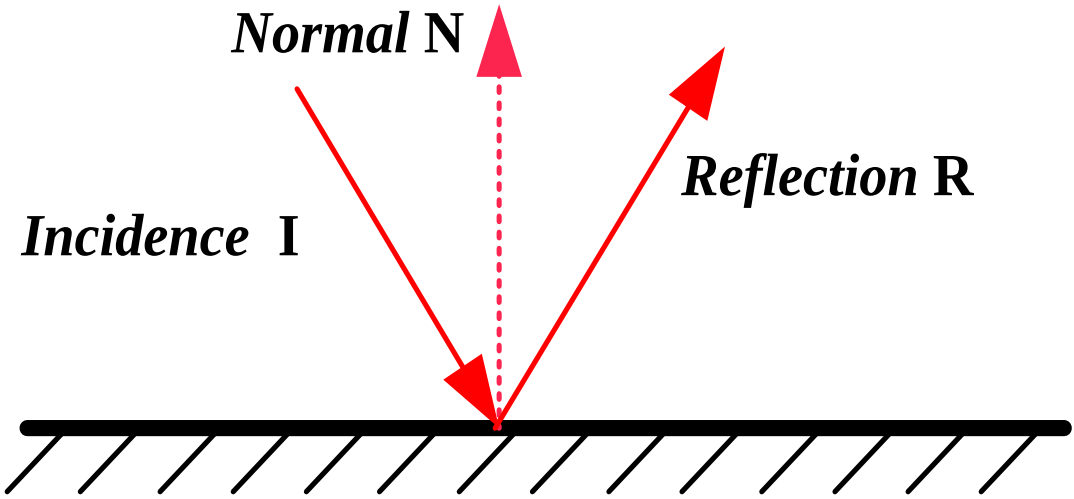
- Ray tracing gives the ability to have objects with mirror reflections or objects with refractions.
  - The first step is to determine where a ray intersects the object.
  - The next step is to determine the direction the ray will travel when it reflects off the surface or refracts through the object.
  - a new ray direction is calculated based on the incoming ray and the surface normal.

Ray Reflections

- The law of reflection:
  - the angle of incidence = the angle of reflection
  - incoming ray, reflection ray and normal is in the same plane
- The reflection ray is calculated as:

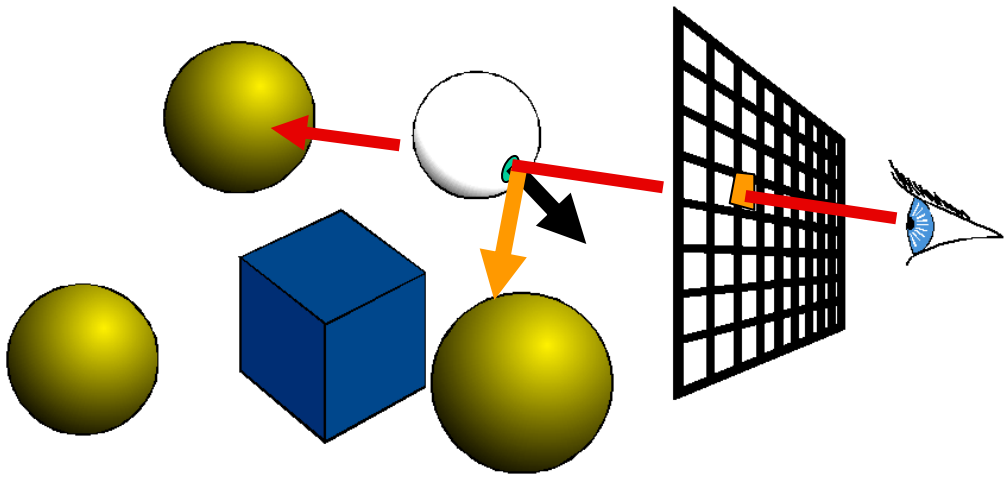
$$R = I - 2(I \cdot N)N$$

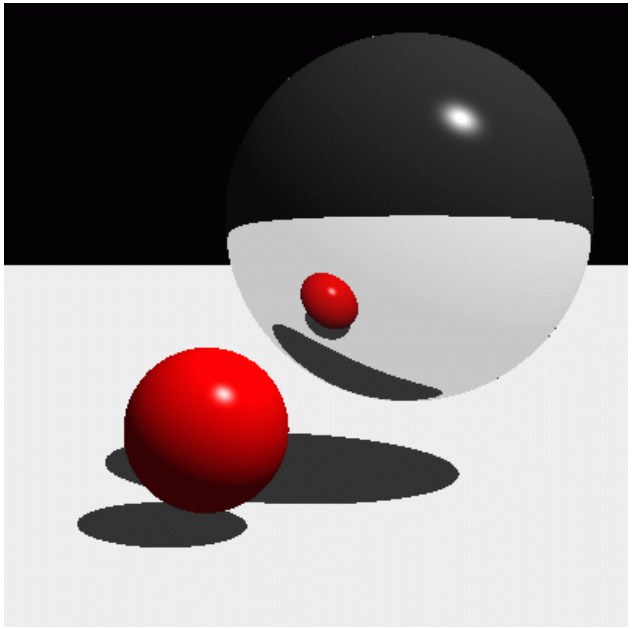
  - where I,N and R are all unit vectors



Ray Reflections

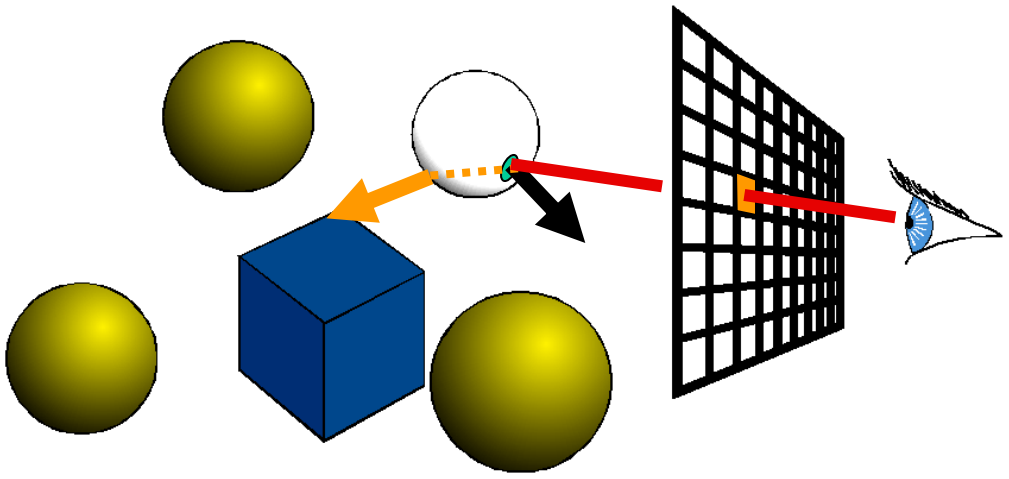
- **As in the figure, reflection ray is symmetric with respect to the normal from the view direction**





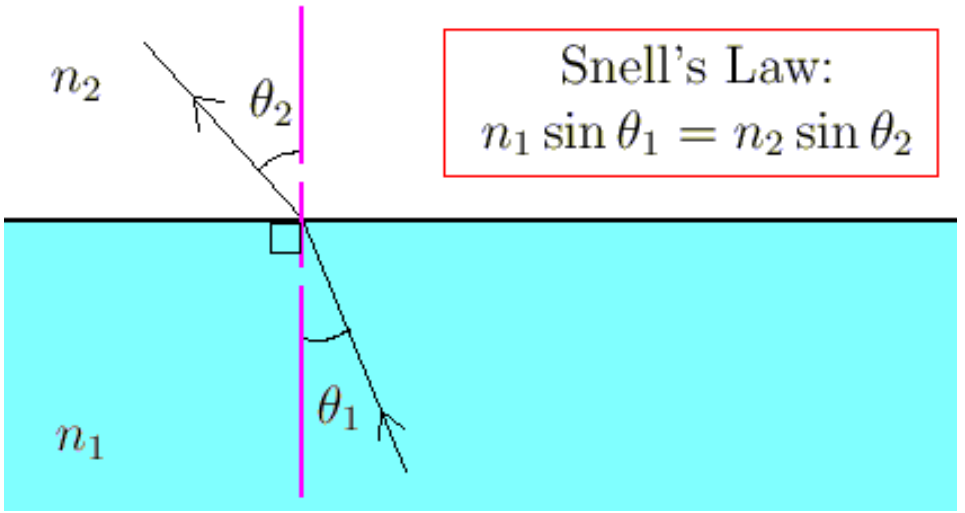
Ray Refraction

- When light travels from one transparent medium into another, the direction of light can change because of the relative densities of the media



Ray Refraction

- The law of refraction (also called Snell's Law):
  - the ratio of the sines of the angles of incidence and of refraction is a constant that depends on the media
  - The constant is called the relative refractive index



Ray Refractions

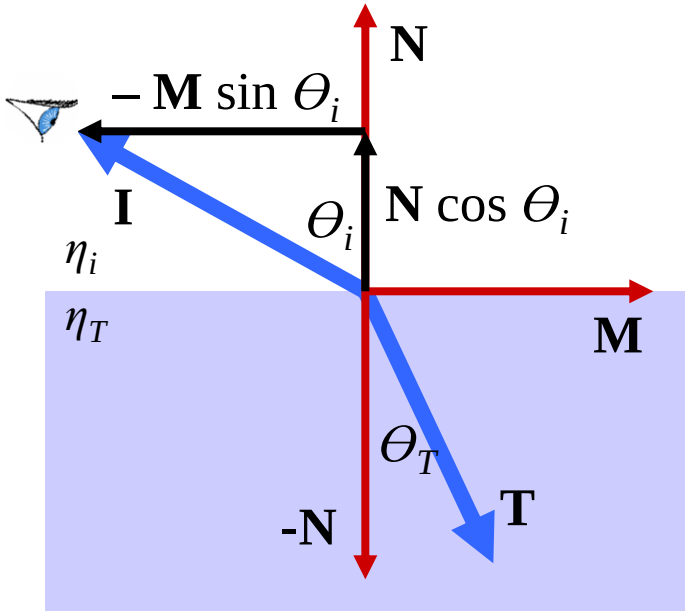
- **Snell's law gives:**

$$\eta_i \sin\theta_i = \eta_T \sin\theta_T$$

$$\eta_i^2 \sin^2\theta_i = \eta_T^2 \sin^2\theta_T$$

$$\eta_i^2(1-\cos^2\theta_i) = \eta_T^2(1-\cos^2\theta_T)$$

$$\cos\theta_T = \sqrt{1-\frac{\eta_i^2(1-\cos^2\theta_i)}{\eta_T^2}}$$

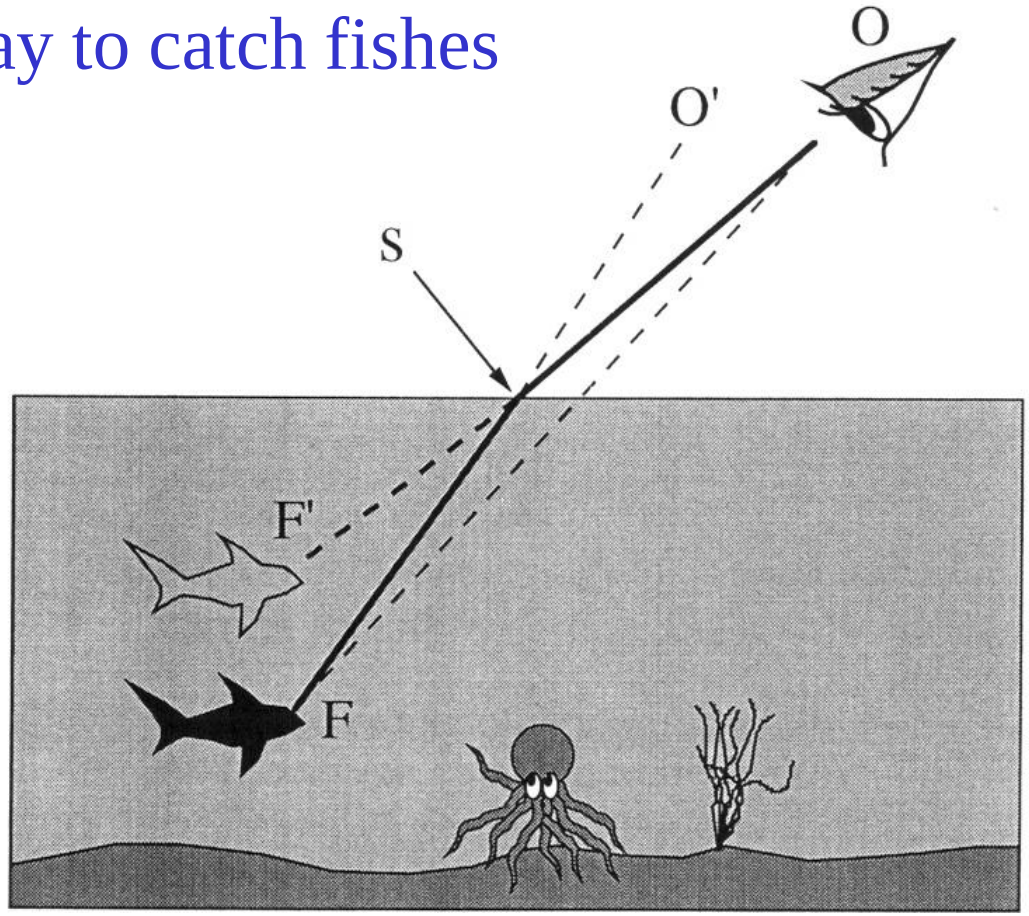


The transmitted ray direction can now be calculated by :

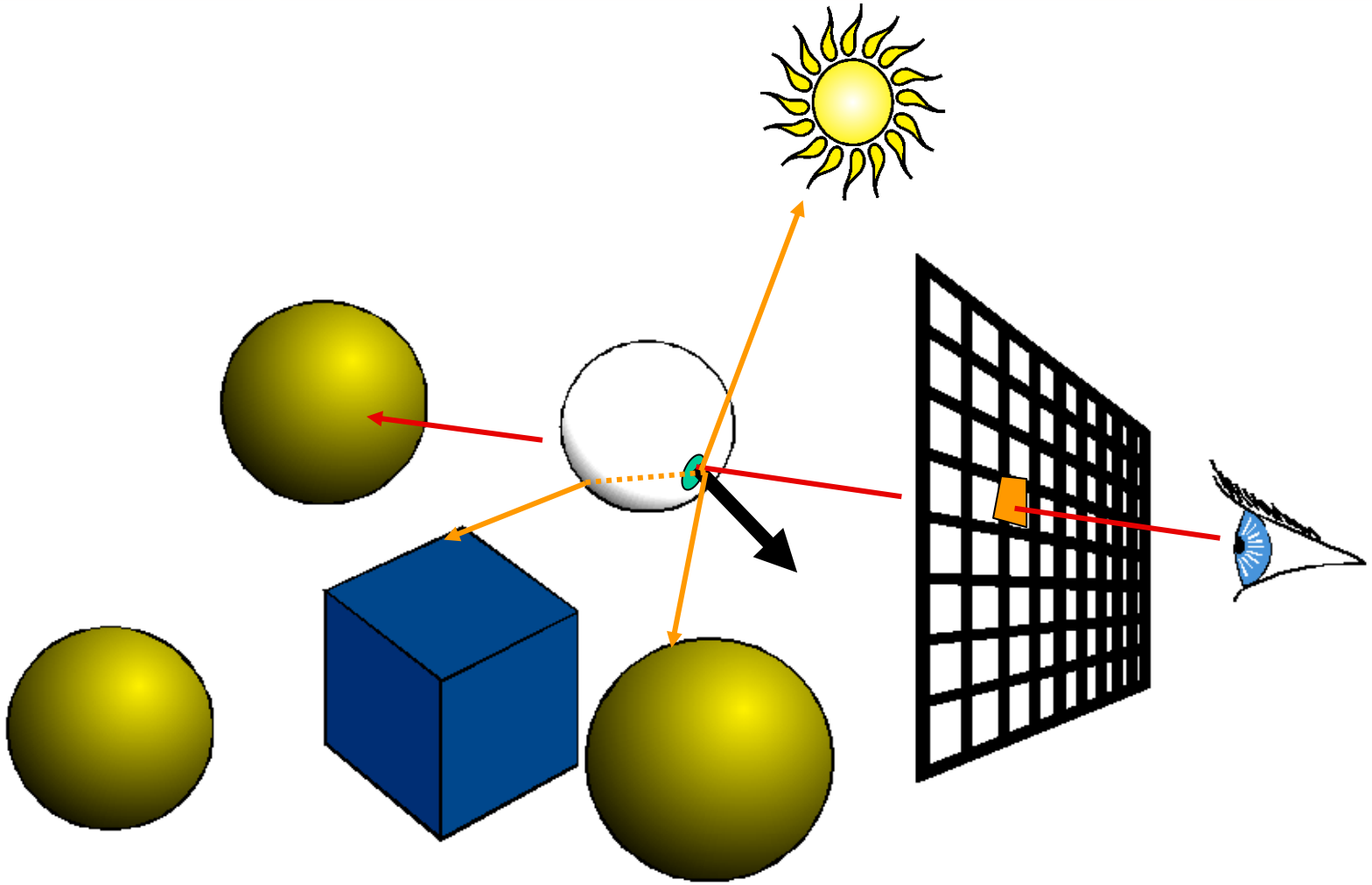
$$T = \frac{\eta_i}{\eta_T} * I + (\cos\theta_T \frac{\eta_i}{\eta_T}) * N$$

Ray Refractions

The right way to catch fishes



Recursive Ray Tracing



Recursive Ray Tracing

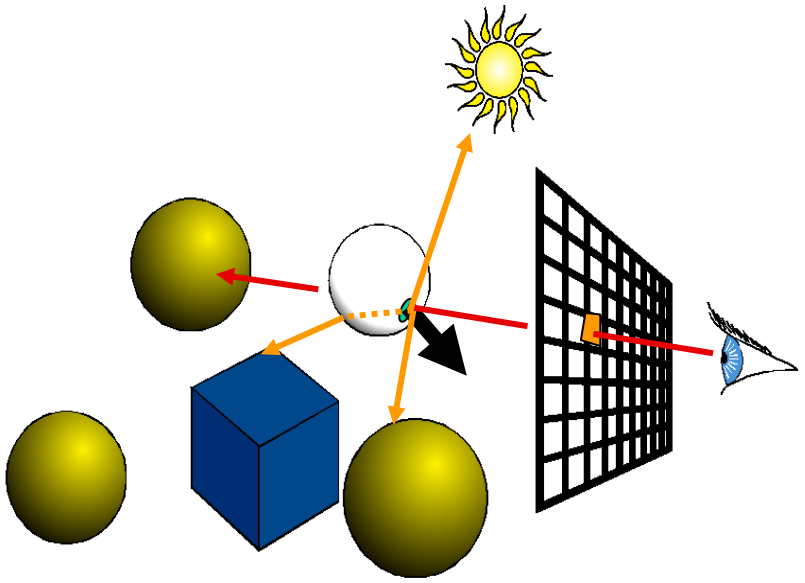
```
IntersectColor( vBeginPoint, vDirection)
{
   Determine IntersectPoint;
   Color = ambient color;
   for each light
        Color += local shading term;
   if(surface is reflective)
      color += reflect Coefficient *
           IntersectColor(IntersecPoint, Reflect Ray);
   else if ( surface is refractive)
     color +=  refract Coefficient *
           IntersectColor(IntersecPoint, Refract Ray);

   return color;
}
```

Recursive Ray Tracing

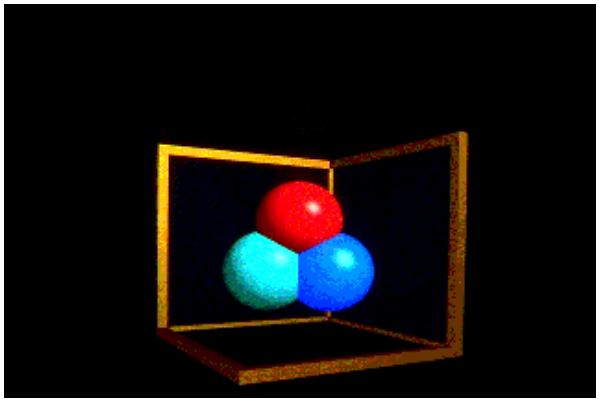
- **Does it ever end?**

  Stopping criteria:

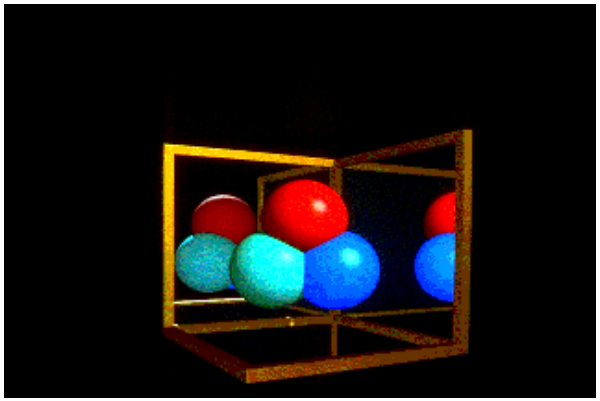
- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / refracted contribution becomes too small



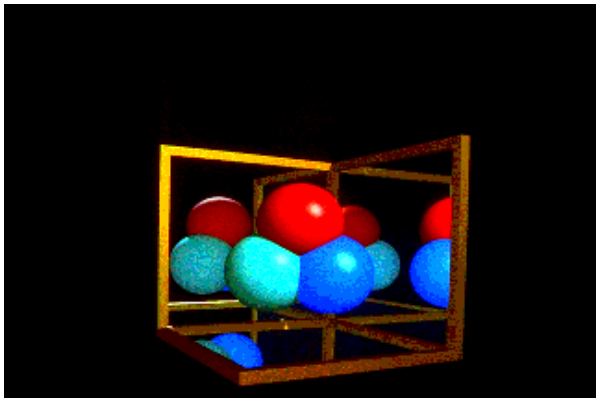
Recursion ray tracing results



0 recursion



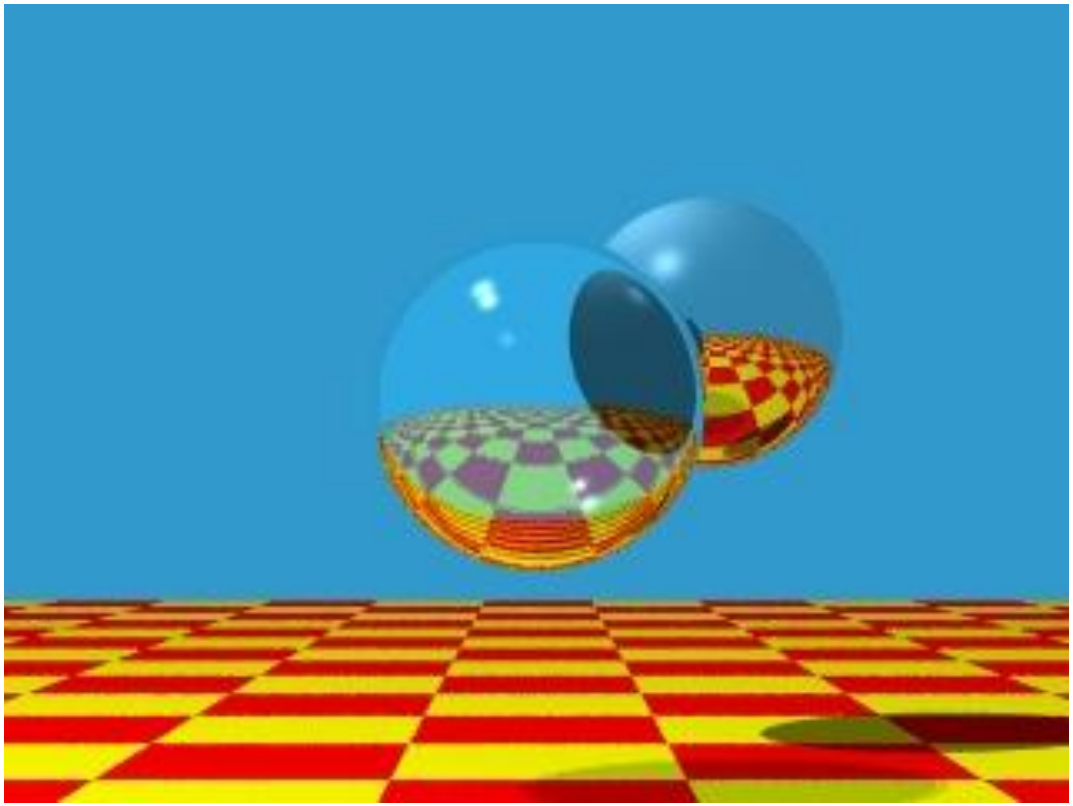
1 recursion



2 recursions

Add Texture(纹理)

- **Computer generated images can be more realistic by painting textures on various surfaces.**
  - **2D Texture**
  - **3D Texture**
  - **As shown in the figure, the floor is textured by a chess board**



Add Texture(纹理)

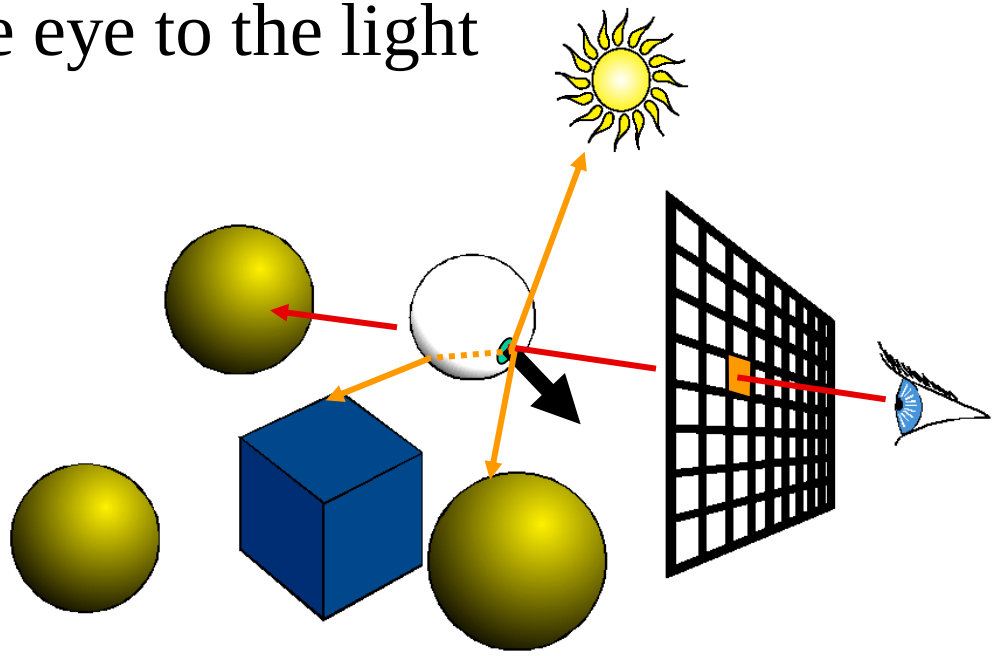
- **2D texture**
  - **Take a rectangle for example**
    - **Specify 2D texture coordinate(纹理坐标) for 4 corner points**
    - **calculate the 2D texture coordinate of the intersection point**
    - **Use this 2D texture coordinate to look-up the texture image, assign this value to the intersection point**
  - **We will discuss more at “Texture course”**

Consideration of Ray Tracing

- **Epsilon problem.**
  - **when a ray is tangent(相切) to a plane/sphere, a ray intersects with a polygon at its vertex …**
- **Acceleration**
  - **Bounding box**
  - **Hierarchical Structure**

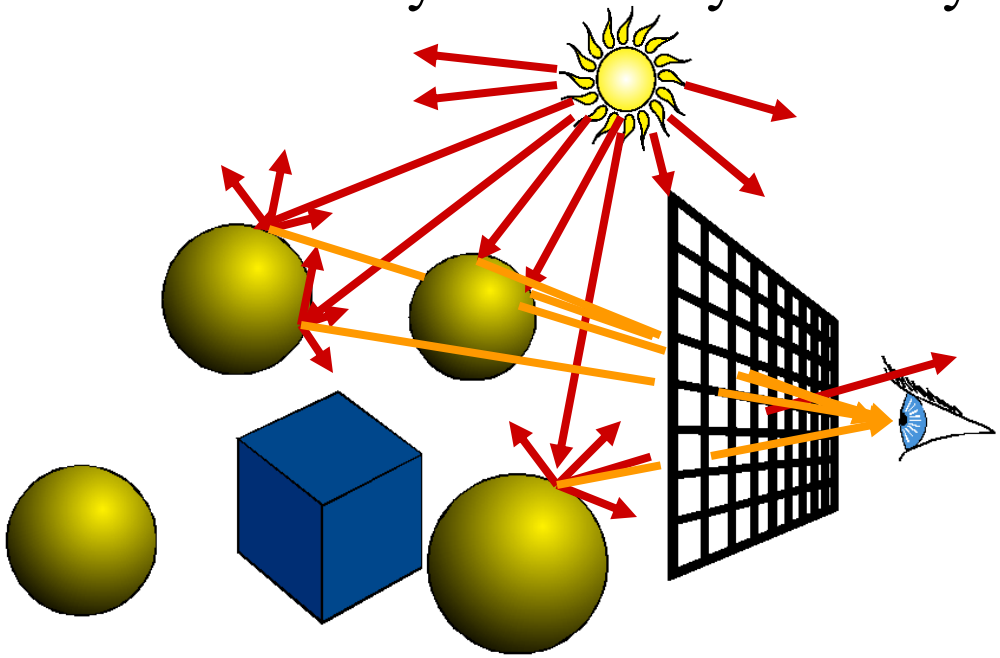
Does Ray Tracing simulate physics?

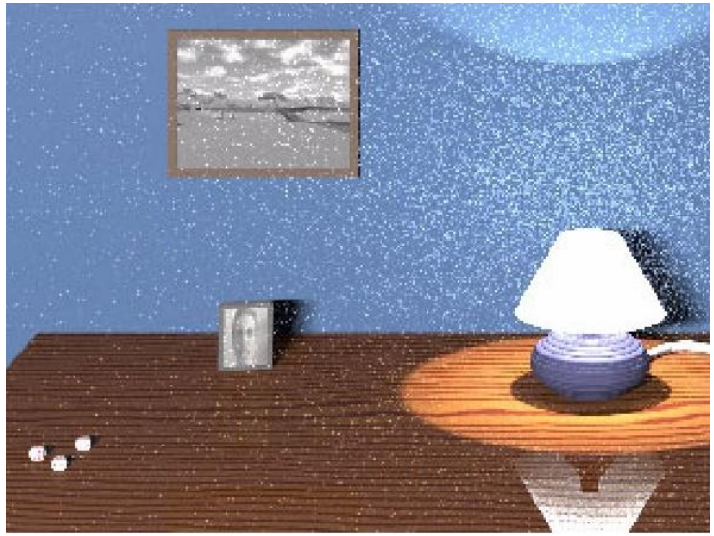
- Photons go from the light to the eye, not the way we used in ray tracing algorithm
- What we do is *backward ray tracing*, photon go from the eye to the light



Forward Ray Tracing

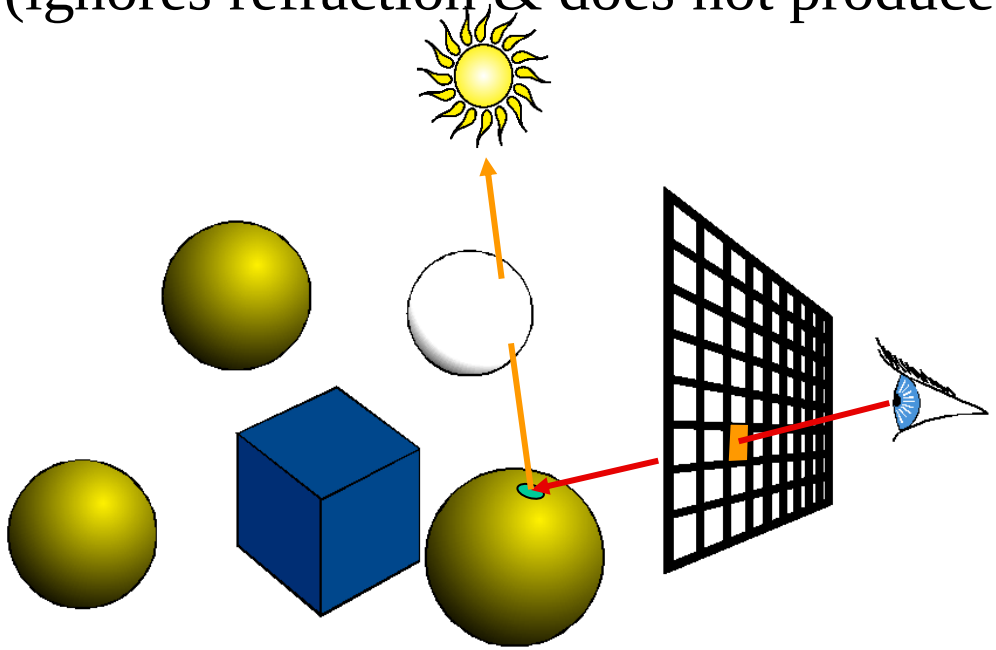
- Start from the light source
  - But low probability to reach the eye
- What can we do about it?
  - Always send a ray to the eye…. still not efficient



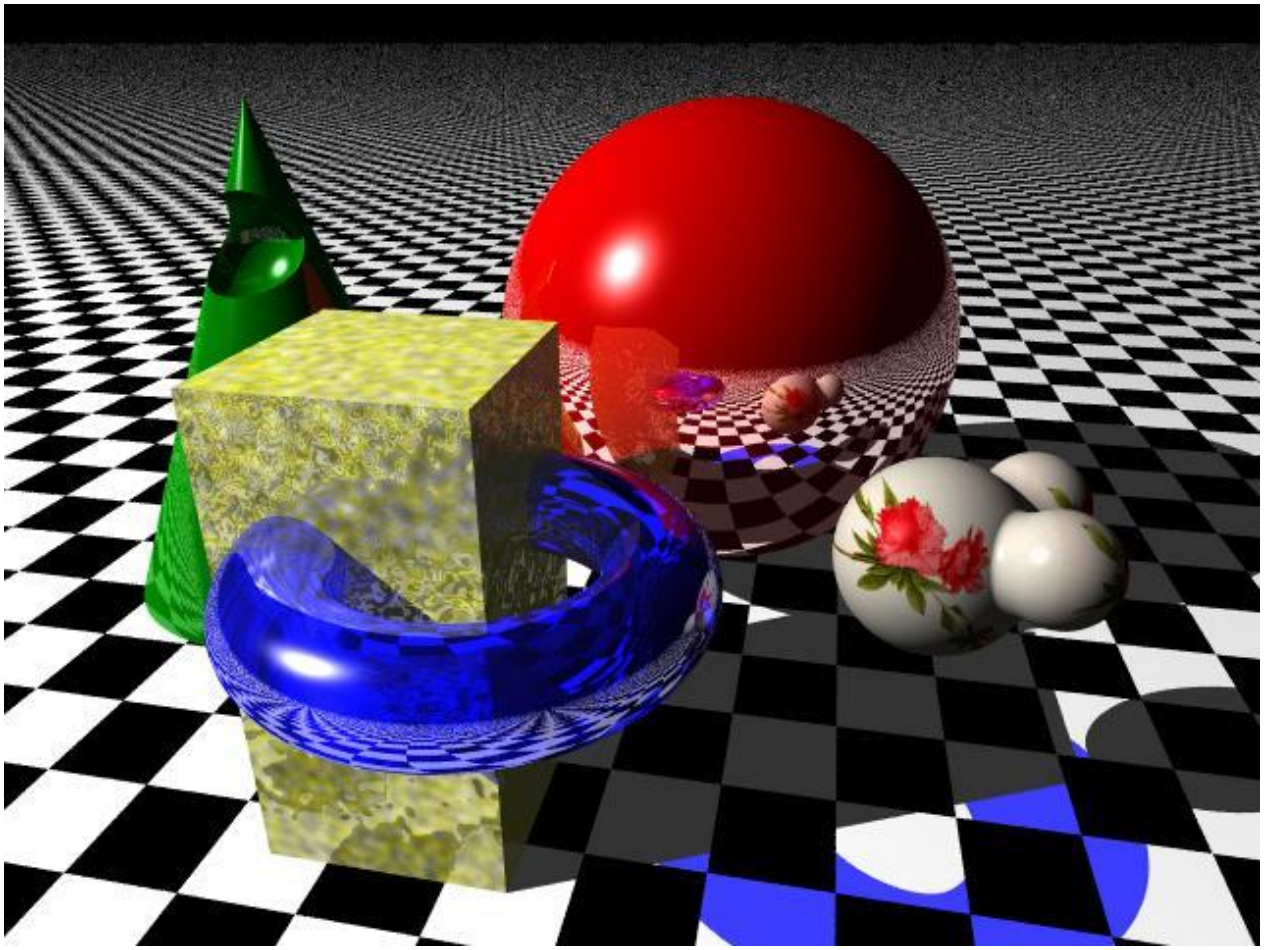


Does Ray Tracing Simulate Physics?

- Ray Tracing is full of dirty tricks
- For example, shadows of transparent objects:
  - opaque?
  - multiply by transparency color?
    (ignores refraction & does not produce caustics)



Some demos of ray tracing



Some demos of ray tracing

Some demos of ray tracing

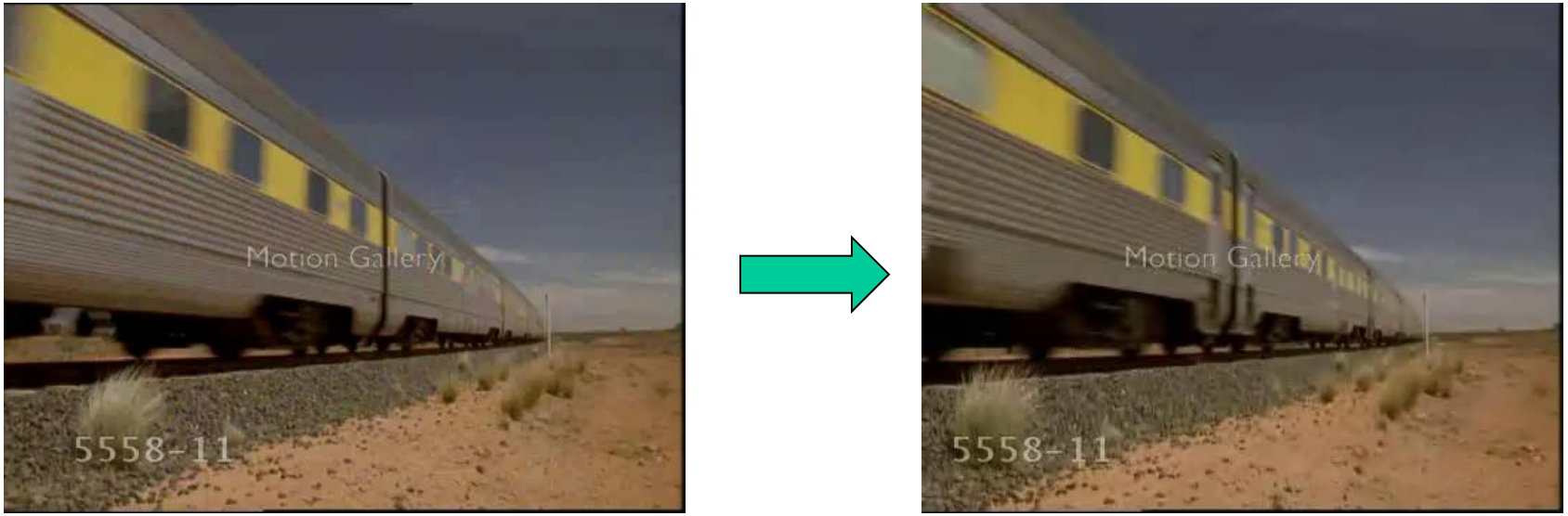
Some demos of ray tracing

- **DENG Jia (计2 邓嘉)’s Demo**

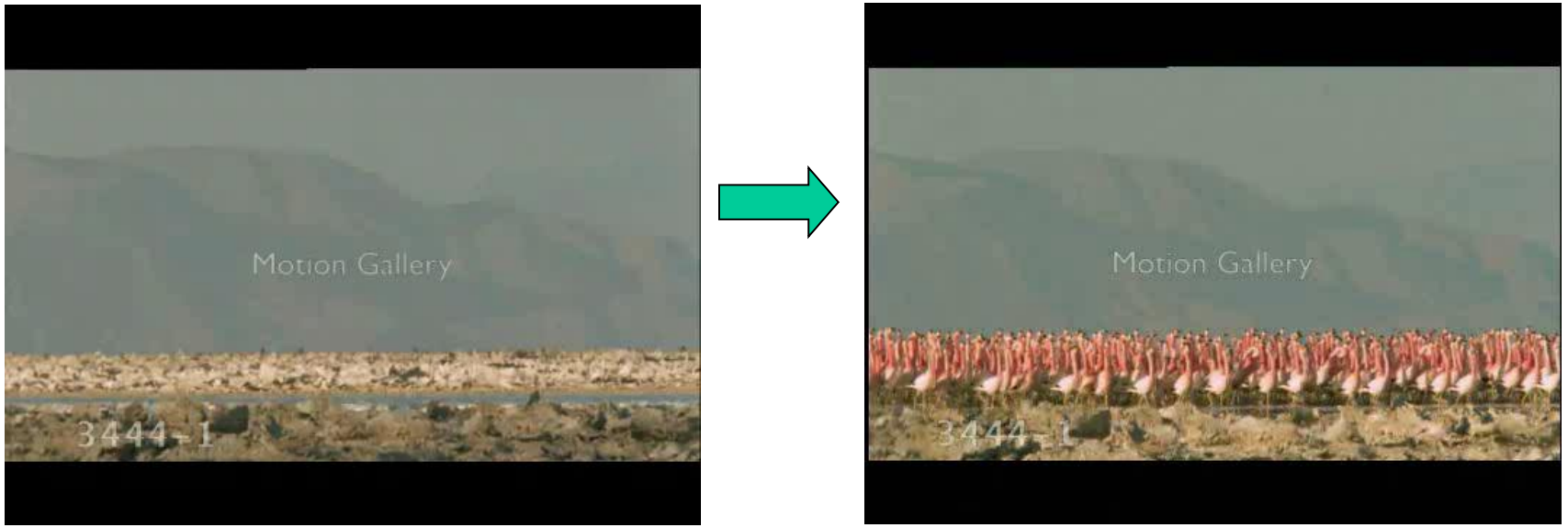
Play Video

- **Deng Jia’s story in Media computing**

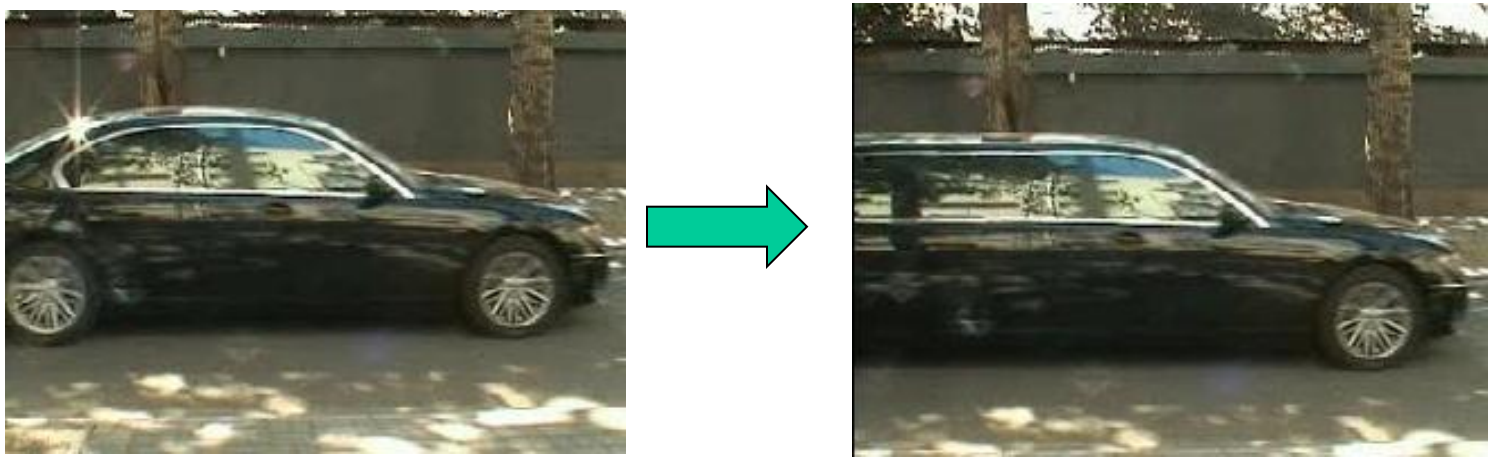
Video Repetition



- Flamingo coming



- I want a super car

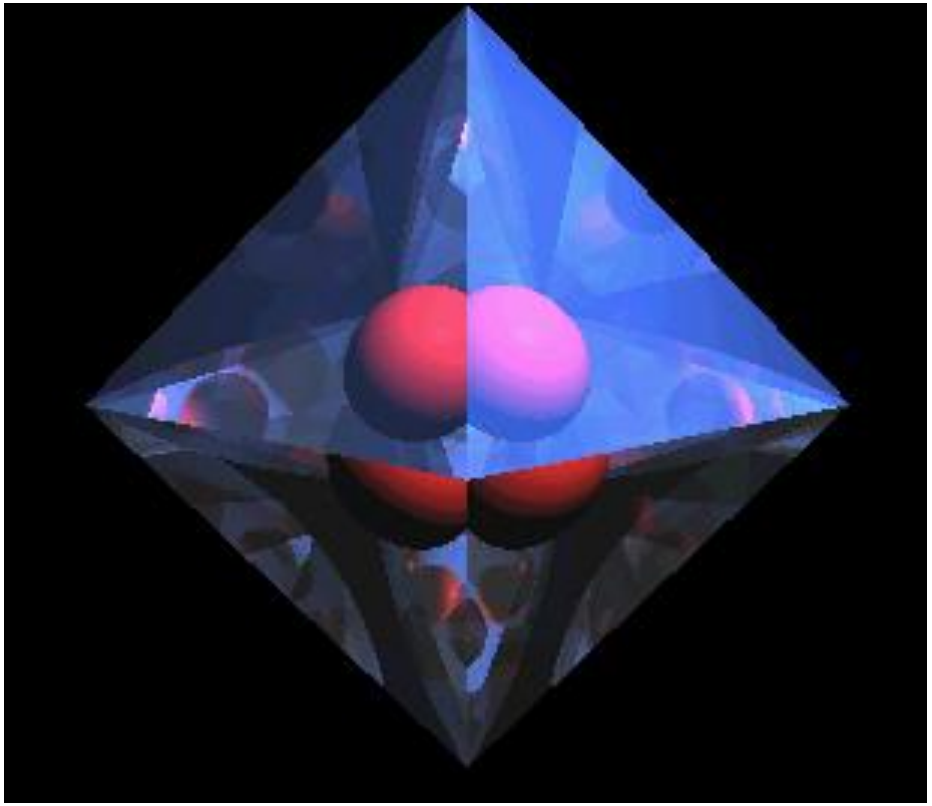


Some demos of ray tracing

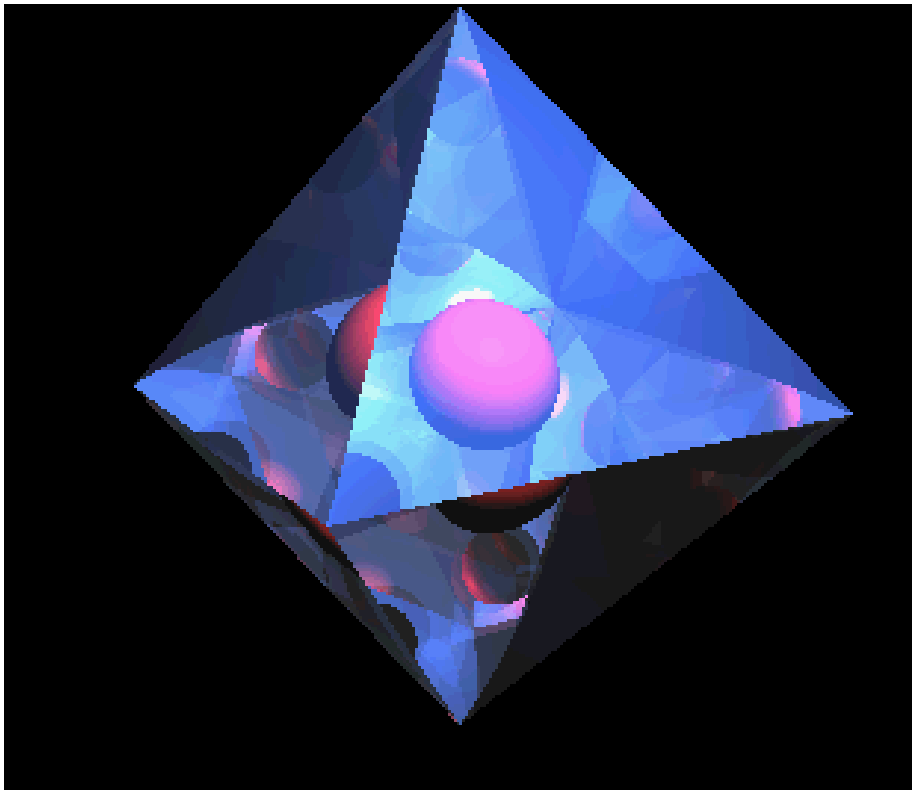
- **GAO Yue(计2 高岳)'s Demo**
- **Gao Yue is a PhD in GCC Group**

Launch Application

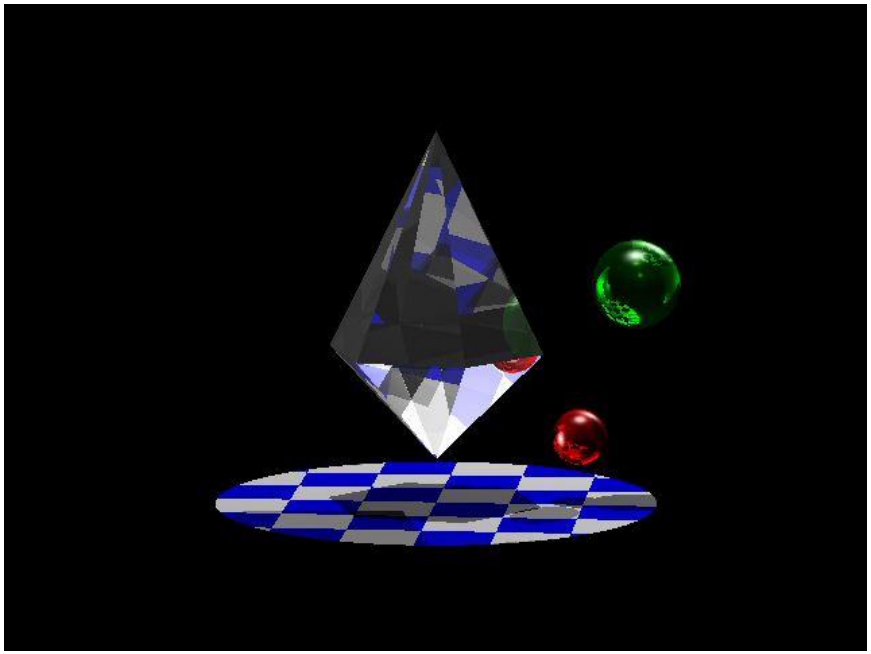
- Further Demo 1



- Further Demo 2



- 另一个大作业



- http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm

Assignments

- **Projects (60%)**
  - **Project 1 (Simple Ray Tracing) (30%)**
    - **Contain primitives of cube, polyhedron, sphere**
    - **Effects : phong model, texture, mirror, transparent, shadow**
    - **Optional: other BRDF models, acceleration techniques, high dimensional texture, soft shadows …**
    - **CANNOT use OpenGL**

Thanks!