Computer Graphics and Programming

Lecture 11

Ray Tracing

Jeong-Yean Yang
2020/12/8

1

Basic Concept of Ray Tracing

Light, Color, and Magic with Math.

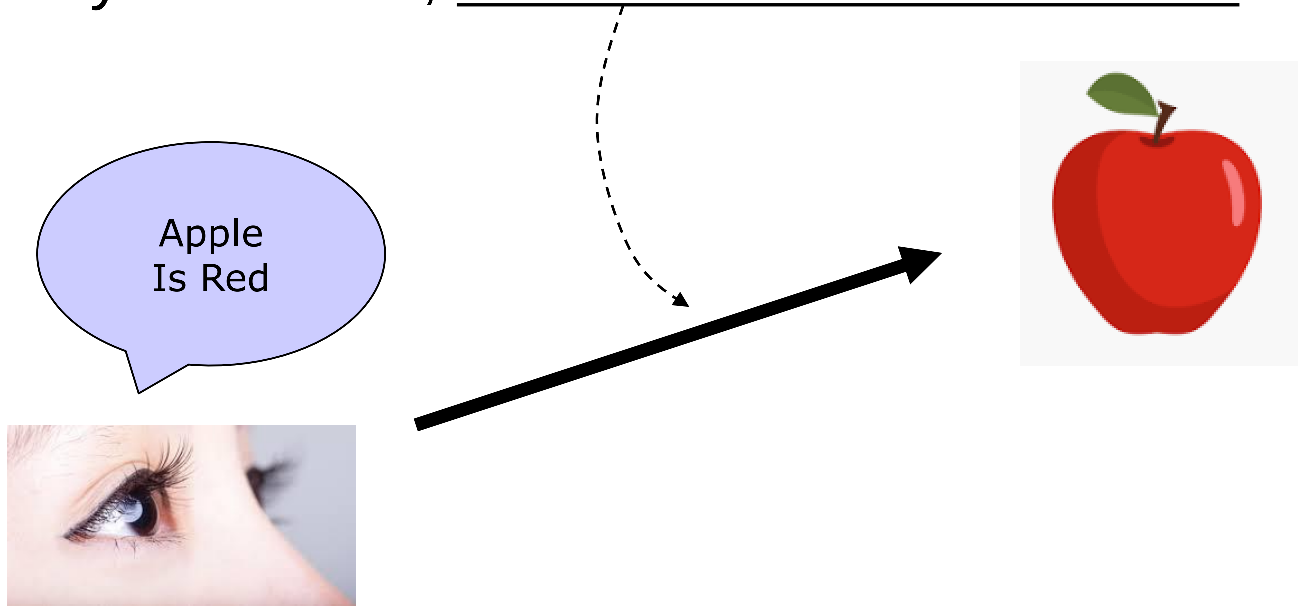
What is a Ray Tracing?

- **Calculate Color as we see in a Physical world.**
- Everything in a Ray tracing is Math and Math.
- 3D Geometries such as sphere, cylinder, plane and line are Perfectly Calculated.
- Ray tracing is Entirely 3D Euclidean Mathematics.
- Thus, it shows very **Realistic** scene.

How to Calculate Colors?

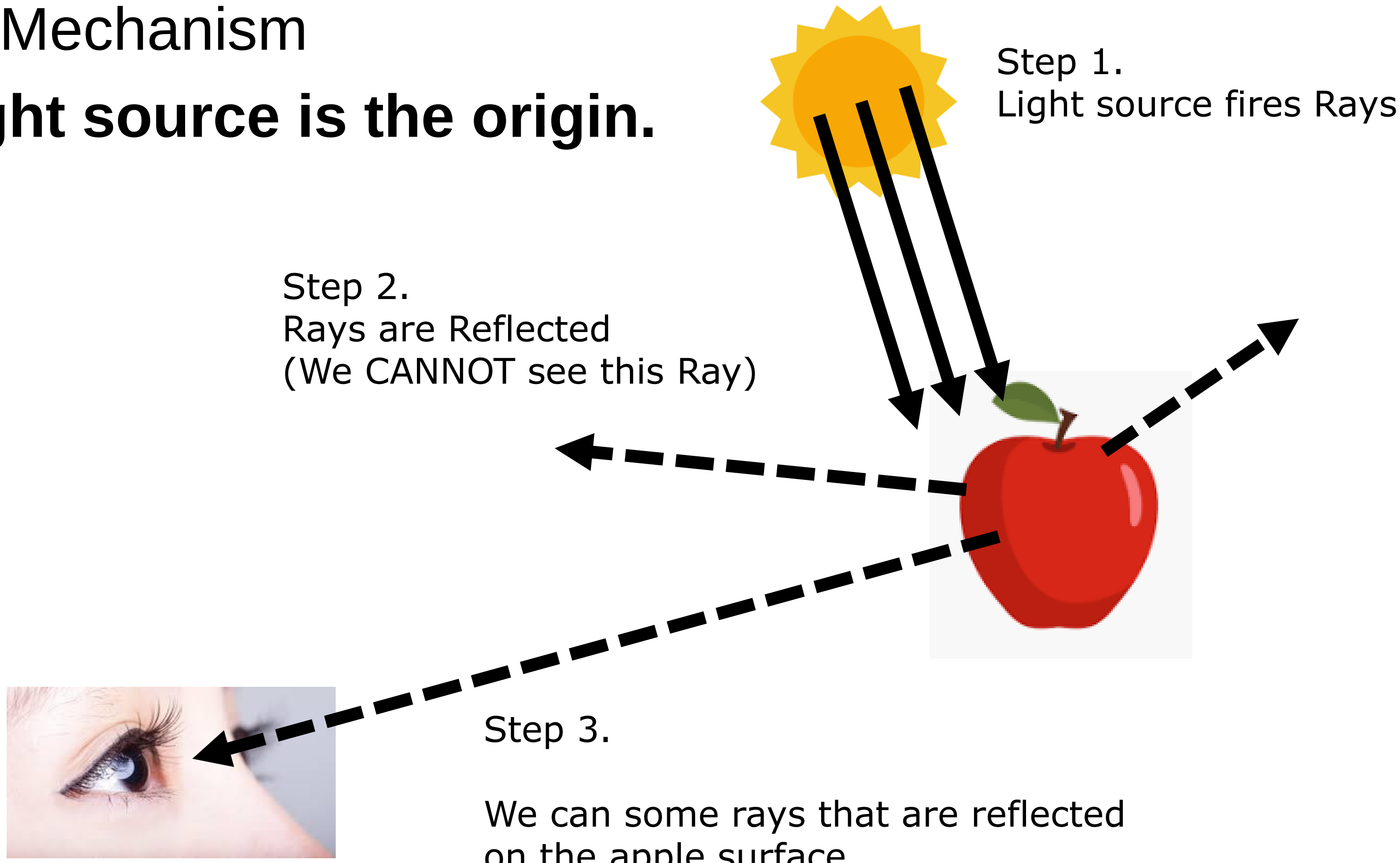
- What we see in everyday is What?
  - We can see an Apple. It is red.

- In a Physical world, the Arrow direction is True?

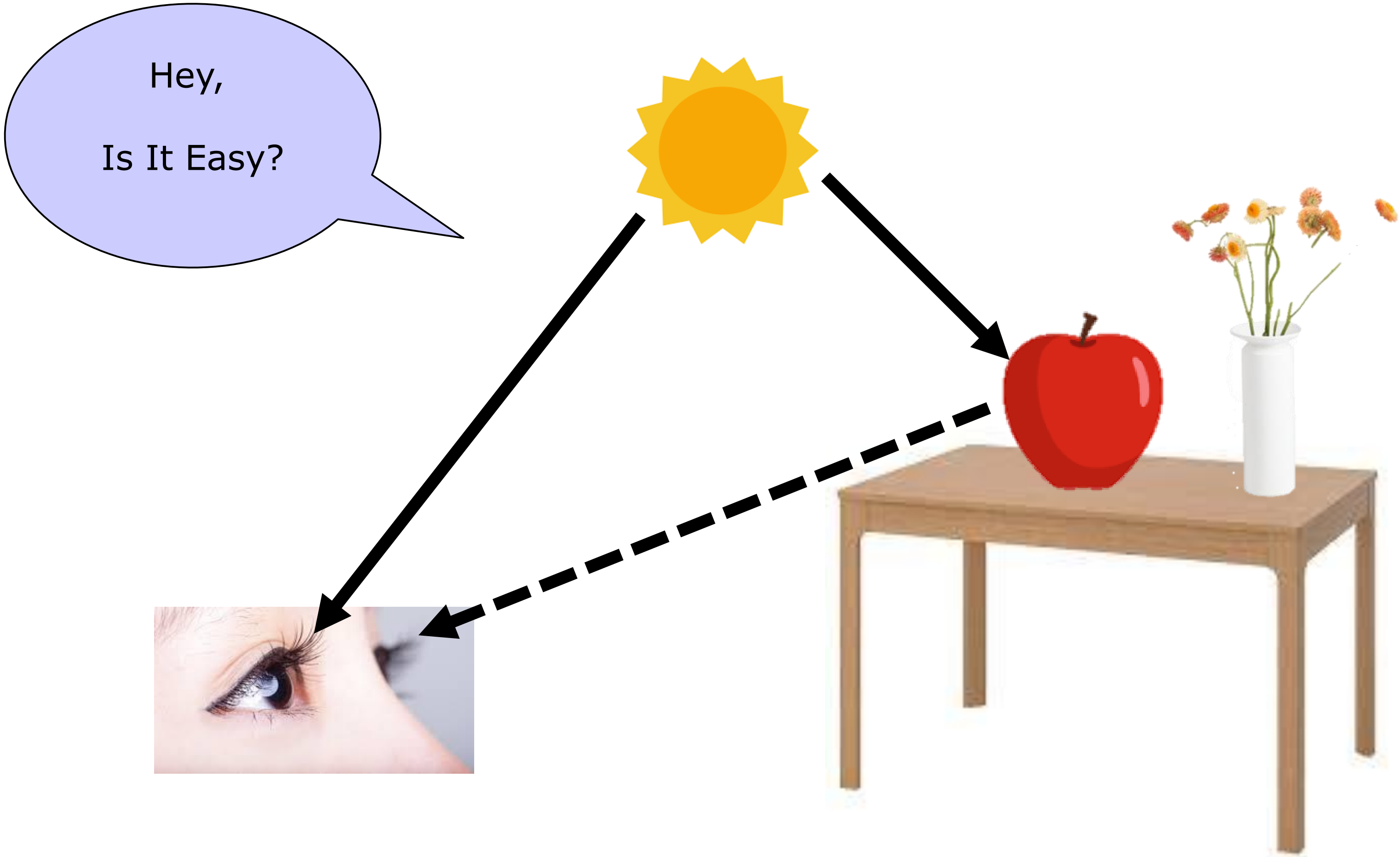


What We Intend to See is Not the Truth.
The Light is coming on Our Eyes

- Seeing Mechanism
- **The Light source is the origin.**

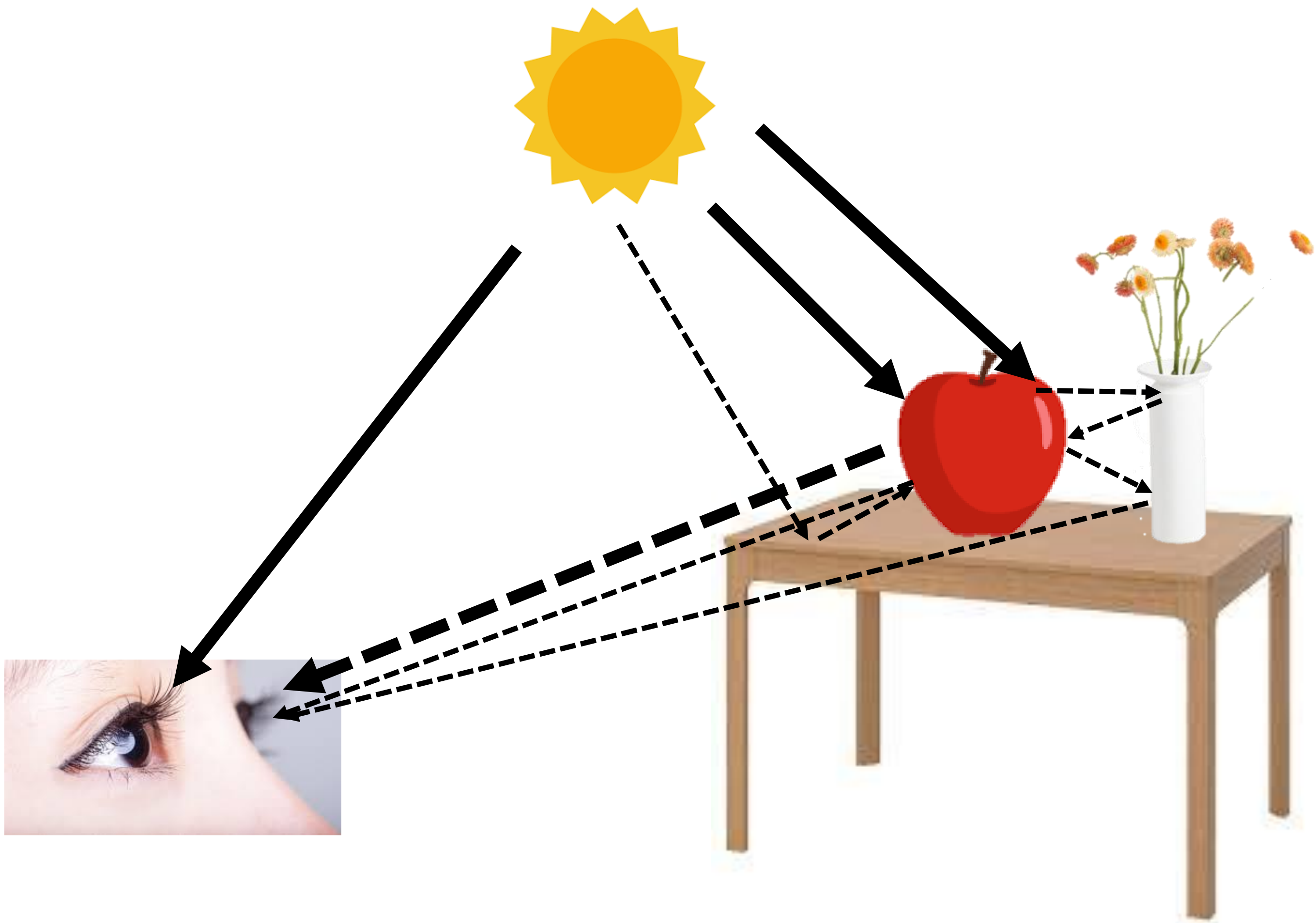


What We See is a Set of Reflected Rays from Light Sources



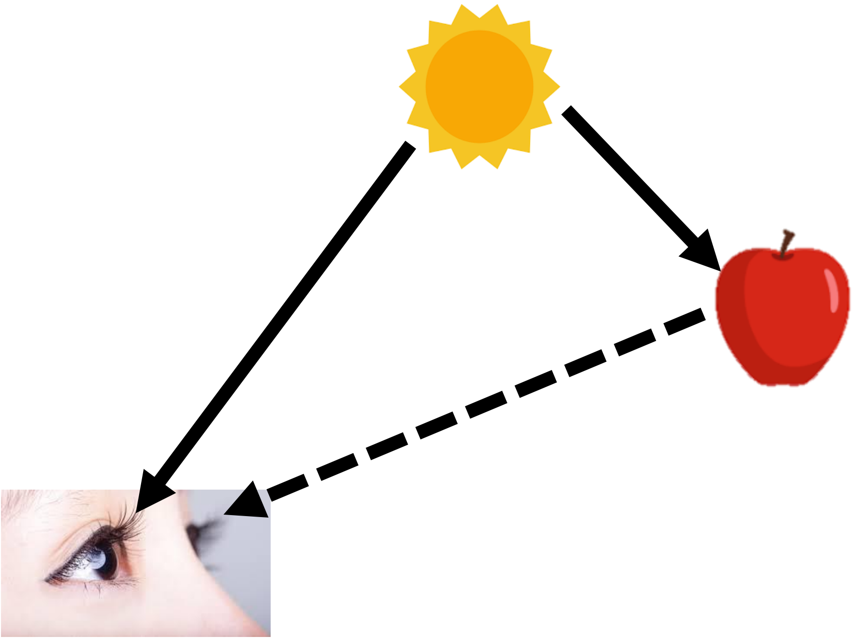
Our World is more complex than you think

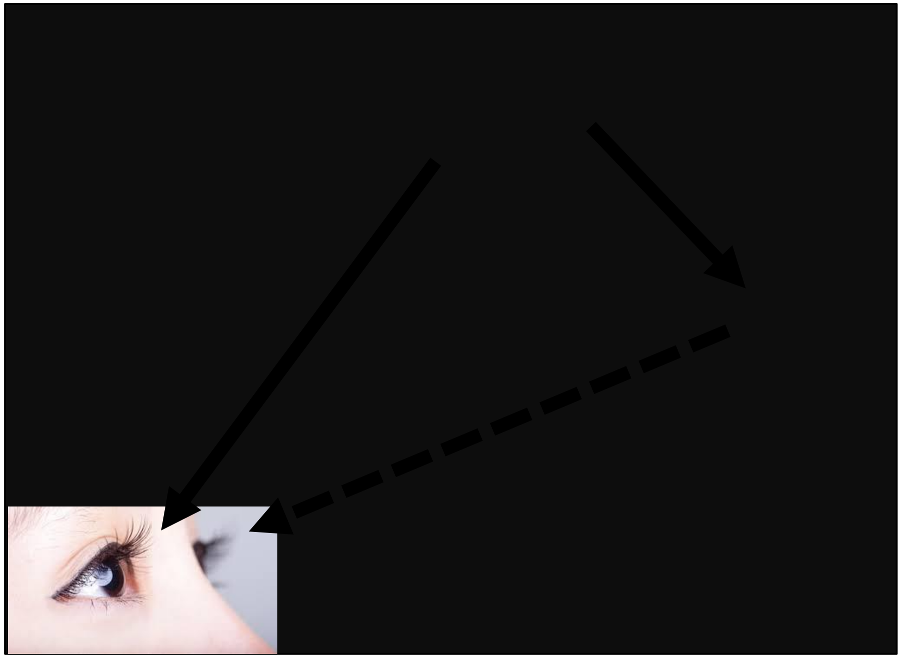
Everything Reflects Rays.
(Without a Black hole)



What You See is
Reflected Rays from Light Source

- Can you believe it? Think the Sun Disappears

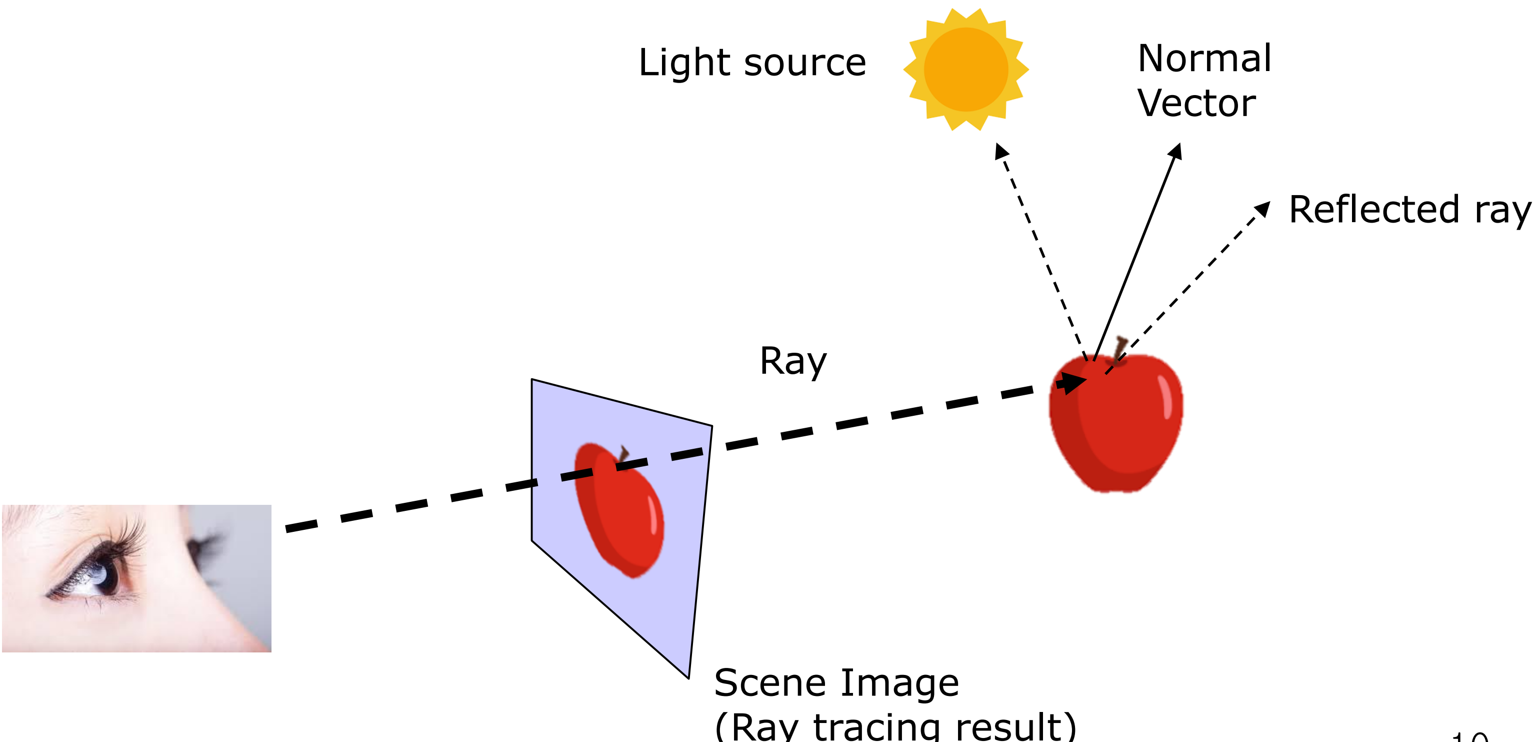




2 Ray Tracing in the Reversed Way

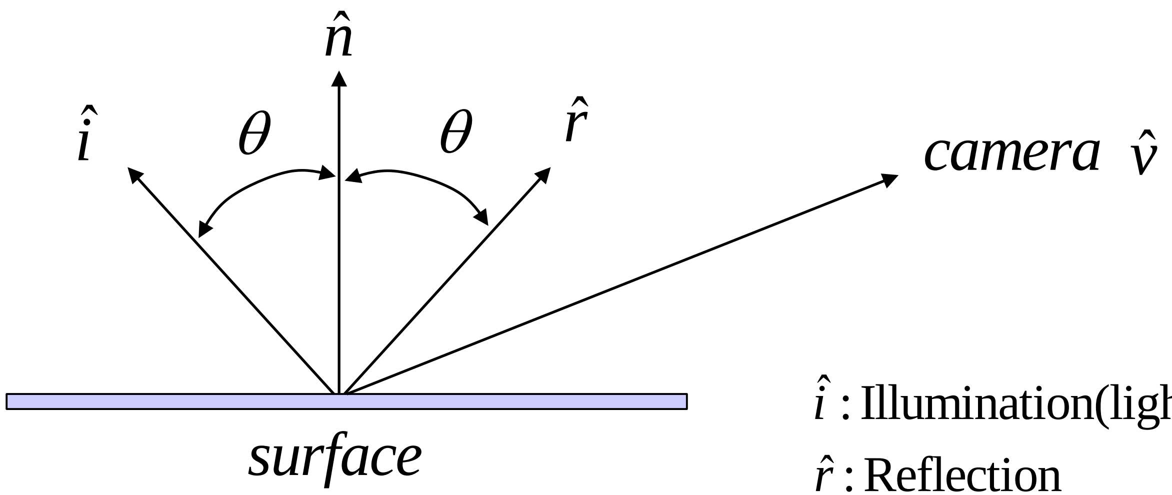
Ray + Tracing

- What is a Tracing?
  - Tracing follows where the color comes from?



Remind
pp. 53
Lecture 8

Lambertian Reflection Model



$\hat{i}$: Illumination(light source)
$\hat{r}$: Reflection
$\hat{n}$: *normal*

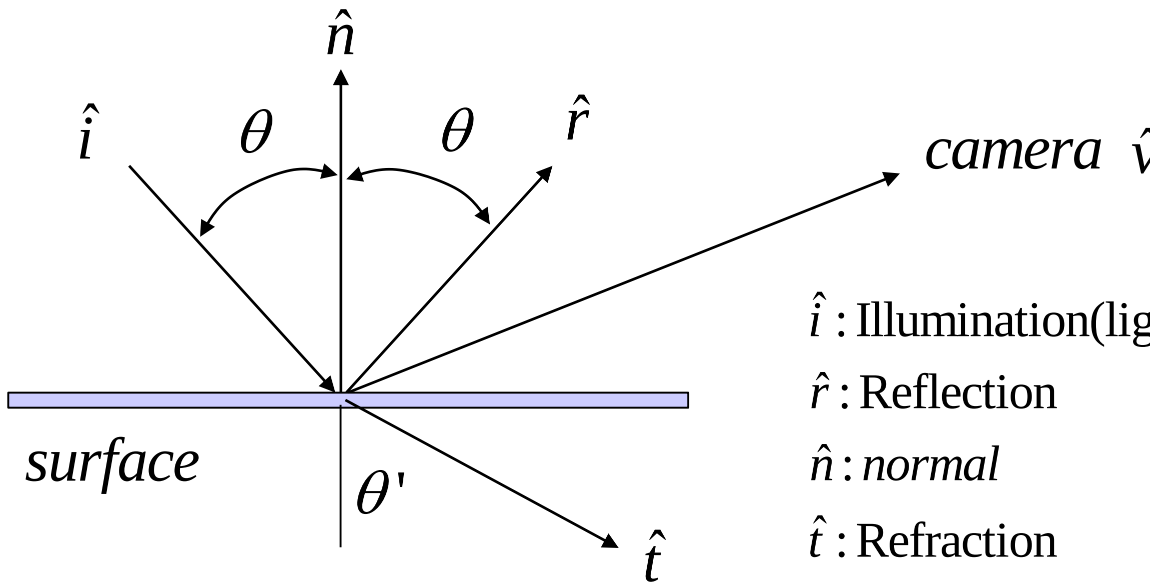
- Lambertian model defines Diffuse color
  - by Only Normal vector

$$\cos\theta = \hat{i} \circ \hat{n}$$

- OpenGL rendering calculates cosine for diffuse color

Illumination Model with Reflection and Refraction

Lambertian model (lecture8 pp.53) + Refraction



$\hat{i}$: Illumination(light source)

$\hat{r}$: Reflection

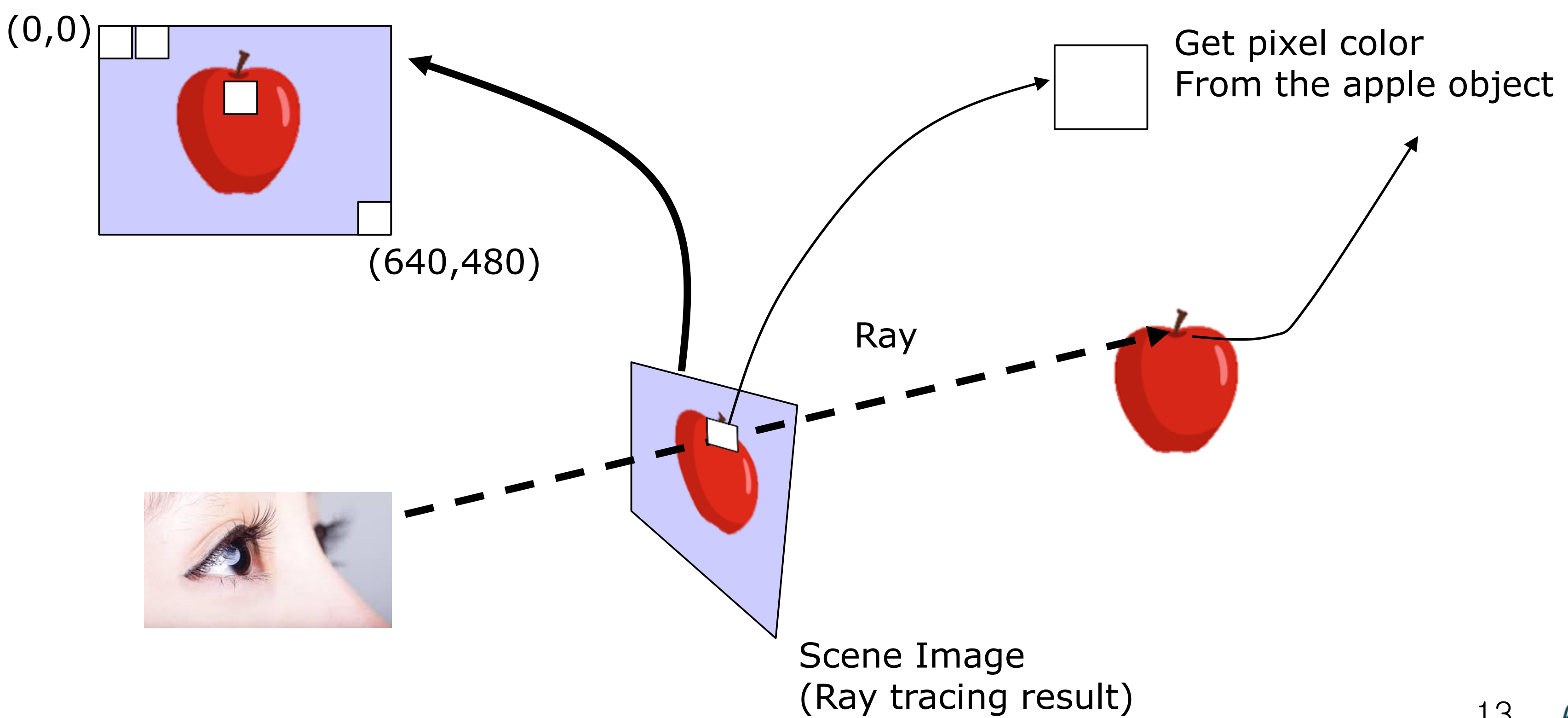
$\hat{n}$: *normal*

$\hat{t}$: Refraction

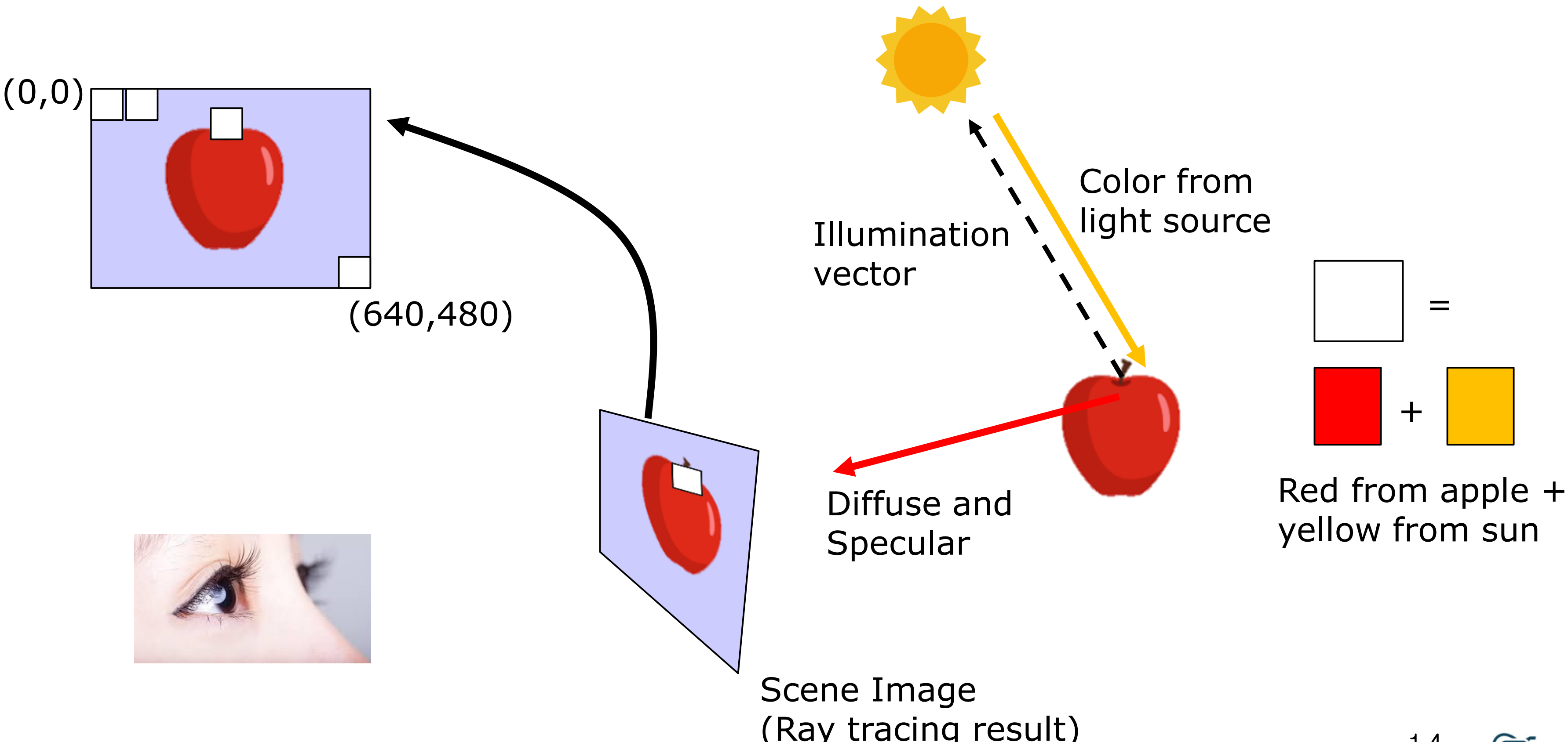
- Illumination model in Ray tracing
  - Reflection and Refraction

Ray Tracing finds Colors
Step 1. Eye fires Ray

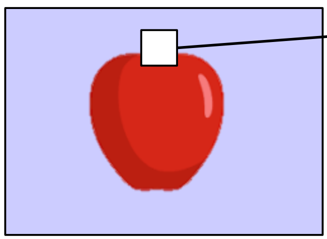
- Scene image = width x height
- Eye fires rays for each pixel

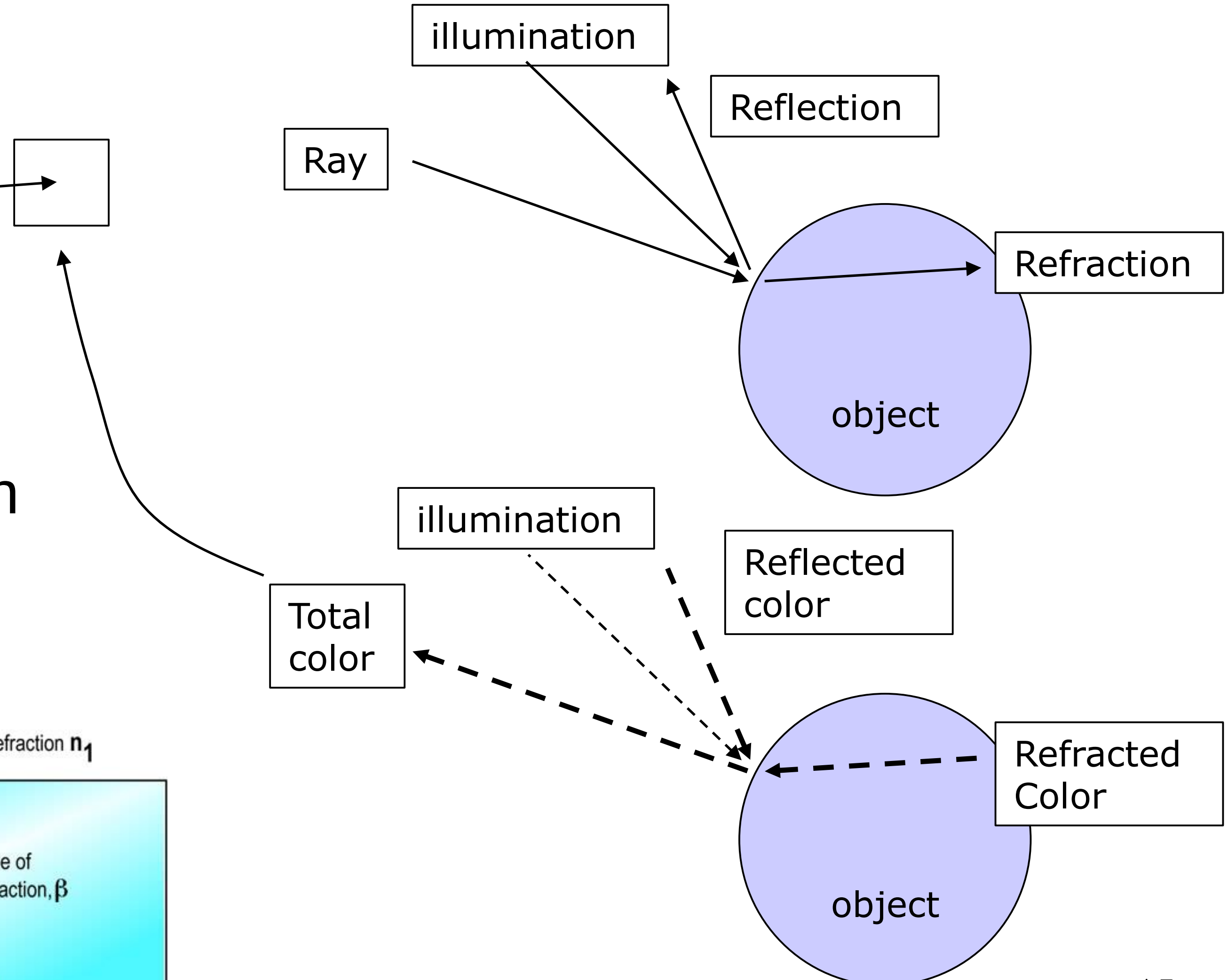


Ray Tracing finds Colors
Step 2. Calculate **Reflection** and Refraction

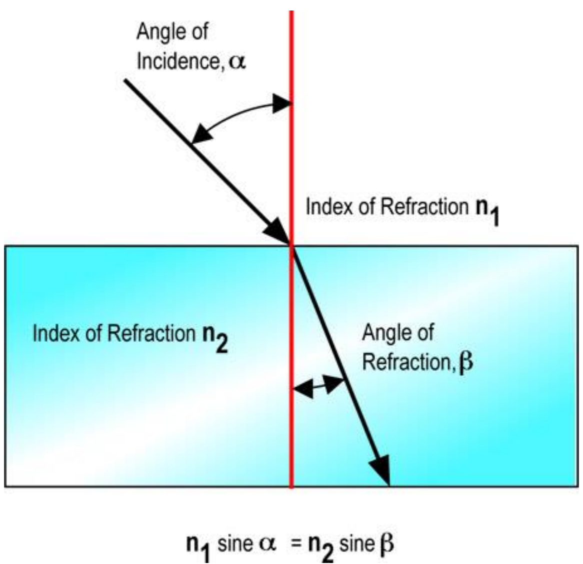


Ray Tracing finds Colors
Step 3. Calculate Reflection and **Refraction**



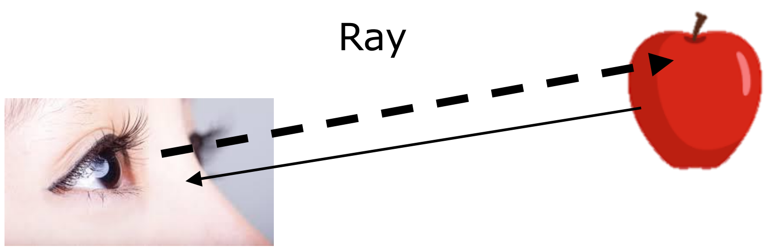


- Refraction

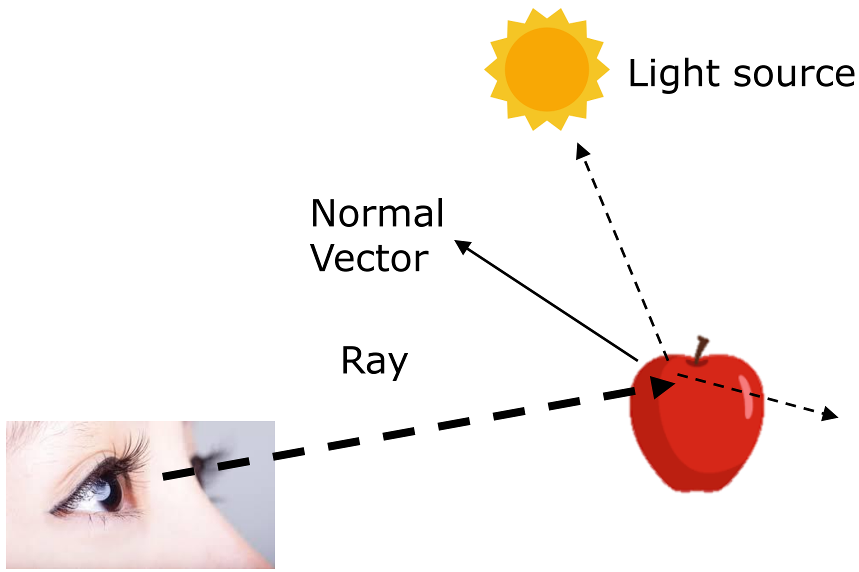


Ray Casting Vs. Ray Tracing

- Ray Casting has NO Reflection and Refraction
- Ray Tracing does with Reflection and Refraction

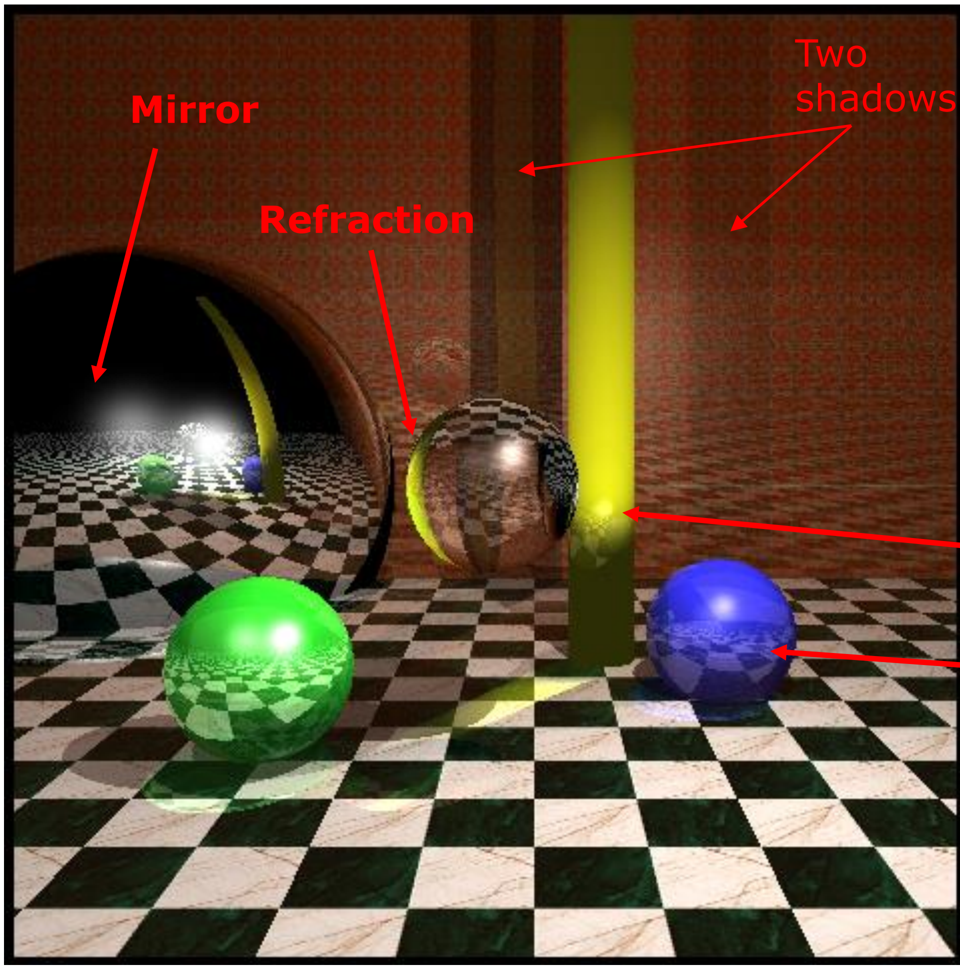


Ray Casting



Ray Tracing

Example of Ray Tracing



What is the difference with OpenGL?

- There are two lights.

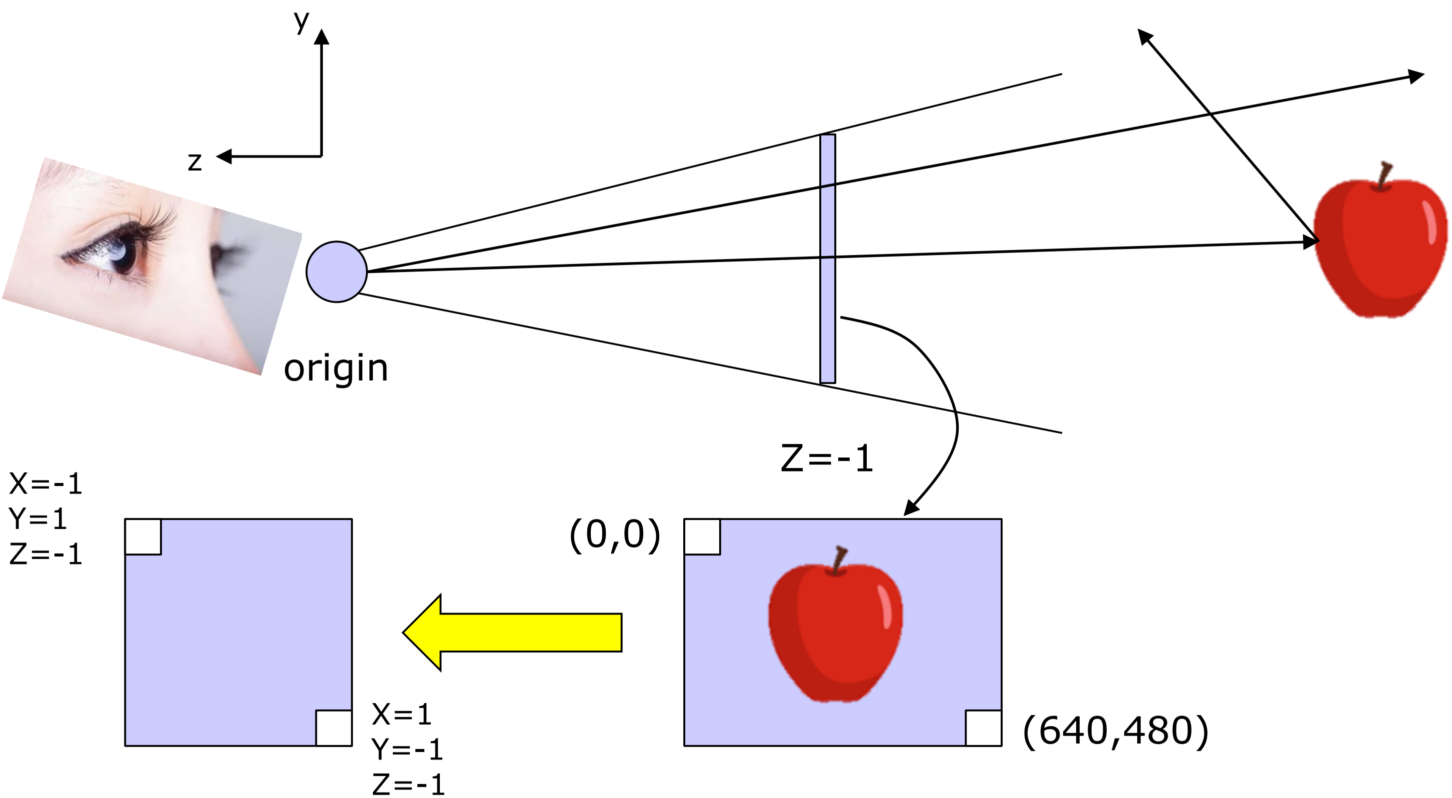
- Shadow, Transparency, Refracted image, and mirror

- Realistic scene rather than polygon-based OpenGL

Lenz effect from a sphere

Reflected image from floor

Ray through Z= -1 (Near) plane



- Calculate Ray

$$(x, y) \in R^2 \qquad x = [0, w) \qquad y = [0, h)$$

$\Rightarrow$

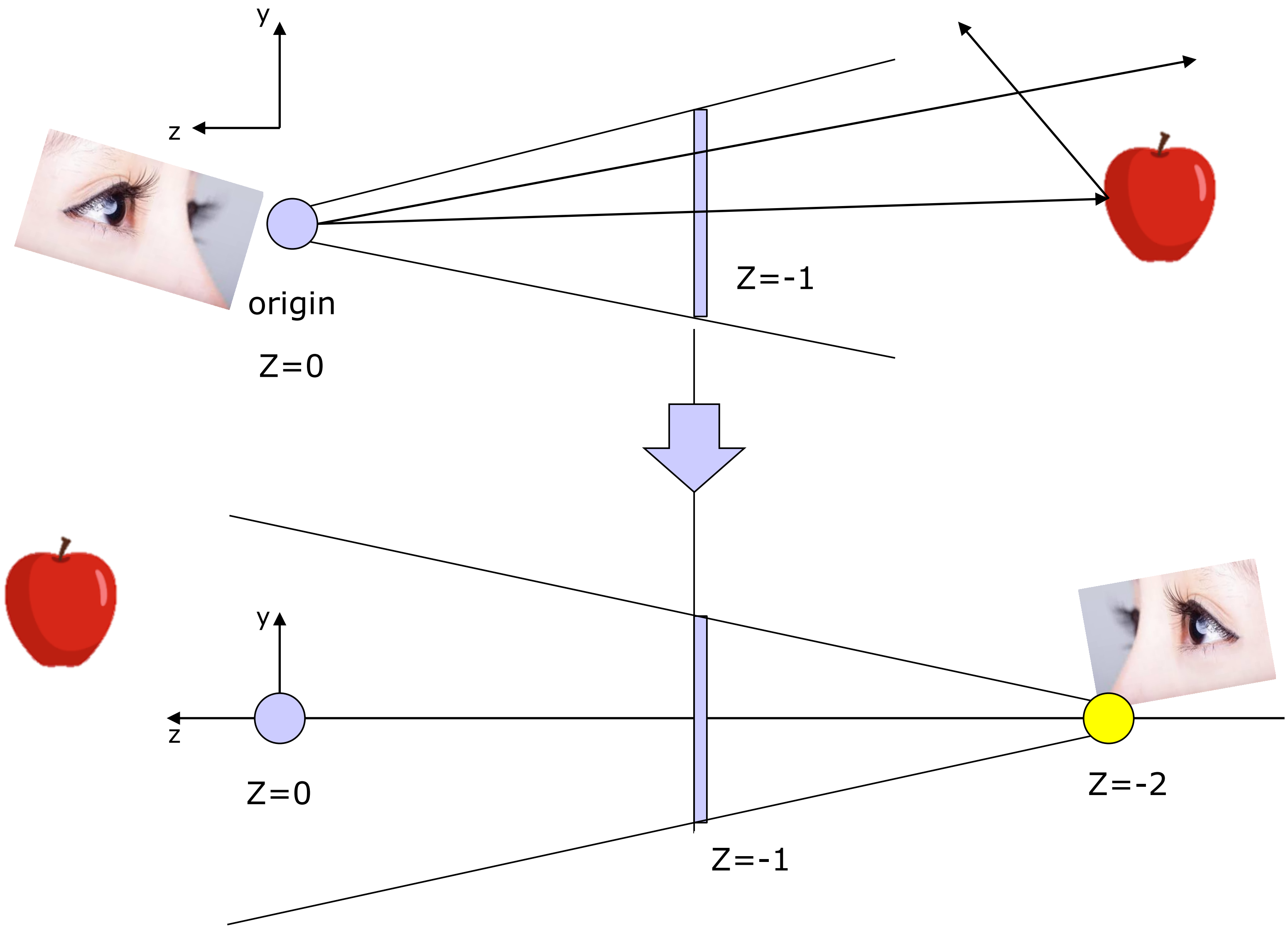
$$X = (x - 320) / 320 \qquad Y = -(y - 240) / 240 \qquad Z = -1$$

$\Rightarrow$

$$Ray, \hat{v} = [X, Y, Z] - o$$

Modifying Viewpoint in RT Example

((-)Z is somewhat confused..)

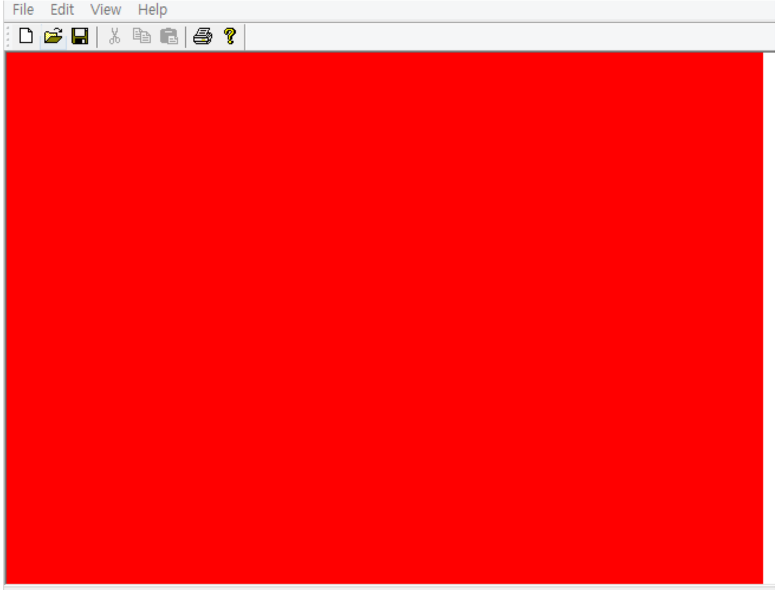


uRT-05-RT1-Buffer
Basic Buffering

- Load 640x480 image, “dummy.jpg”

```
void uRT::Update()
{
    int w,h;
    w= screen.w;
    h= screen.h;

    BYTE *p = img.GetBuffer();
    for (int j=0;j<h;j++)
    for (int i=0;i<w;i++)
    {
        *p++ = 255;  Red
        *p++ = 0;    Green
        *p++ = 0;    Blue
    }
    img.BGR2RGB();
}
```

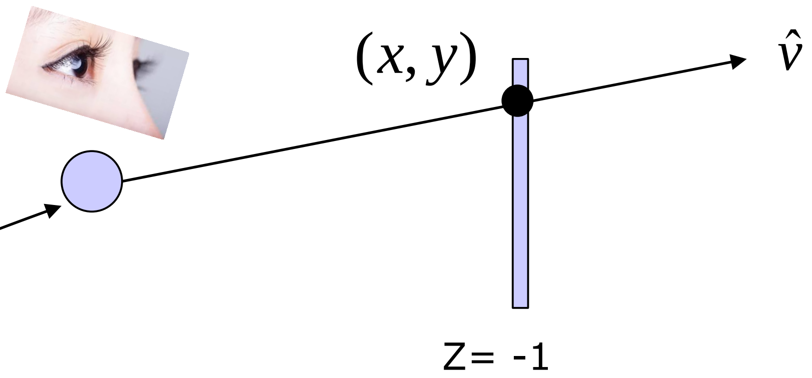
JPG is based on BGR color map



Result
640x480 red screen

Ray Vector Calculation
640x480 = 307200 rays

- Ex) uRT-06-RT2-2DLight




```
uRT::uRT()
{
    screen.w     = 0;
    screen.h     = 0;
    screen.o     = uVector(0,0,-2);
}
```

```
uVector uRT::Ray(int x,int y)
{
    float dx,dy;
    dx  = 2./screen.w;
    dy  = 2./screen.h;

    uVector ret;
    ret.x    = -1+ dx*x;
    ret.y    = -(-1+ dy*y);
    ret.z    = -1;

    ret.x    = ret.x*screen.w/screen.h;

    ret      = ret-screen.o;
    ret      = ret.Unit();
    return ret;
}
```

$$X = (x - 320)/320$$

$$Y = -(y - 240)/240$$

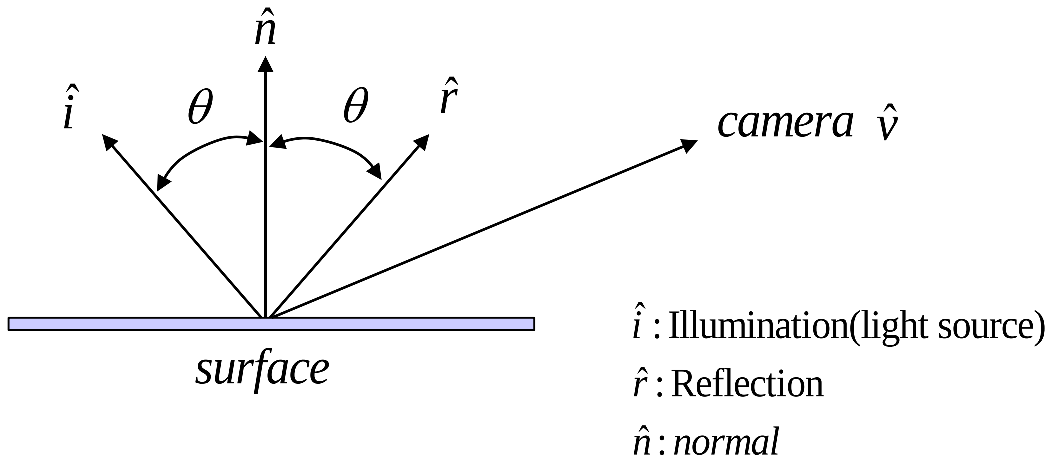
$$Z = -1$$

$$X' = X \cdot 640/480$$

$$Ray, \hat{v} = [X', Y, Z] - \hat{o}$$

uRT-06-RT2-2DLight
Normal vector calculation

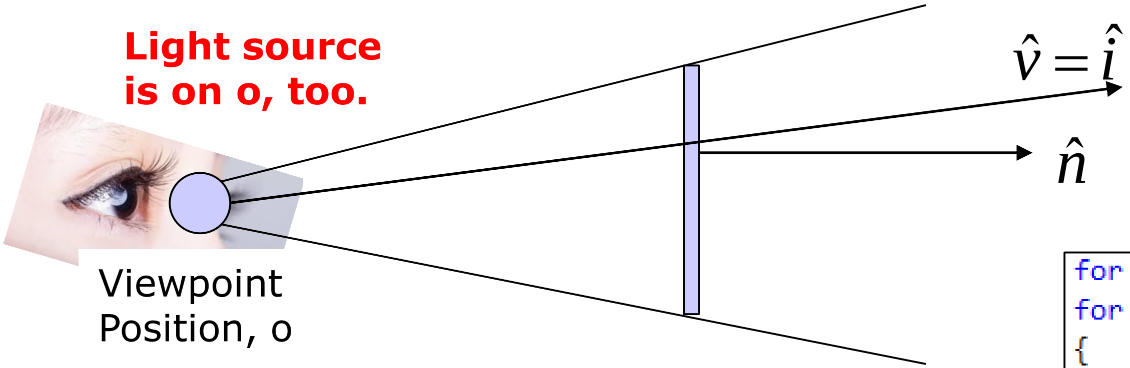
- Remind Lambertian Diffuse Model



- Use normal vector (0,0,1)
- Lambertian diffuse uses illumination source, i. $\cos\theta = \hat{i} \circ \hat{n}$
- If we use a ray, v, what will happen?

$$\hat{v} = \hat{i}, \qquad \cos\theta = \hat{v} \circ \hat{n}$$

uRT-06-RT2-2DLight
Normal vector calculation

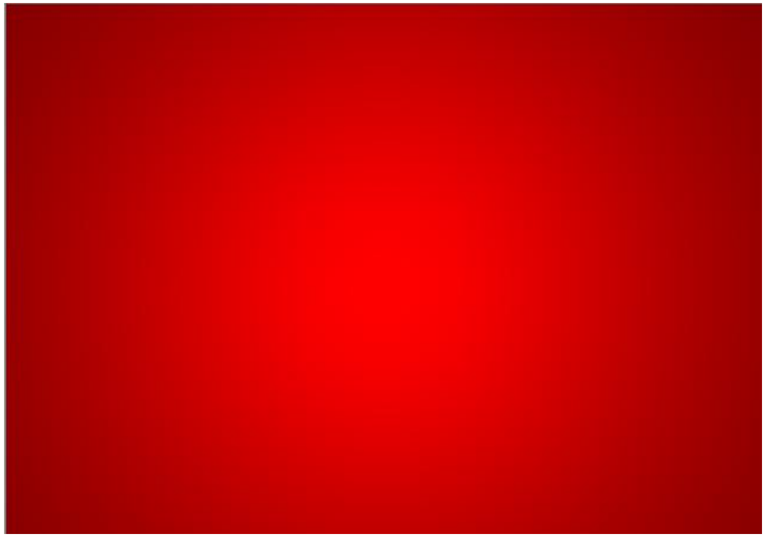


$\hat{v} = \hat{i}$

$\hat{n}$

$$\cos\theta = \hat{v} \circ \hat{n}$$

Result



```
for (int j=0;j<h;j++)
for (int i=0;i<w;i++)
{
    uVector v = Ray(i,j);

    uVector n(0,0,1);
    float f = v.Dot(n);

    c.r = f;
    c.g = 0;
    c.b = 0;

    c.r *=255;
    c.g *=255;
    c.b *=255;

    if (c.r>255)      c.r=255;
    if (c.g>255)      c.g=255;
    if (c.b>255)      c.b=255;
```

Object Modeling in uObj

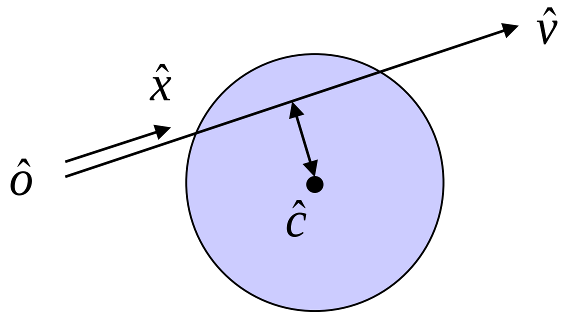
- 3D graphics uses Object Modeling in two ways.

- 1. Polygon-based modeling
  - Vertices and faces

- 2. Parametric Modeling
  - Sphere (radius and center position)
  - Plane(normal vector and position)
  - and so on
- Ray tracing uses Parametric Modeling

Example of Sphere
Math of 3Dim. Vector Space

- The minimum distance is easy



$\hat{o}$: *starting position*

$\hat{v}$: *direction*

$\hat{c}$: *center*

$$|\hat{v}| = 1$$

$$\hat{x} = \hat{o} + \lambda\hat{v}$$

$$\mathrm{d}^2 = |\hat{x} - \hat{c}|^2 = |\hat{o} + \lambda\hat{v} - \hat{c}|^2 = |\hat{o} - \hat{c} + \lambda\hat{v}|^2 = |\hat{b} + \lambda\hat{v}|^2$$

$$= |\hat{b}|^2 + \lambda^2 |\hat{v}|^2 + 2\lambda\hat{b} \circ \hat{v} = |\hat{b}|^2 + \lambda^2 + 2\lambda\hat{b} \circ \hat{v}$$

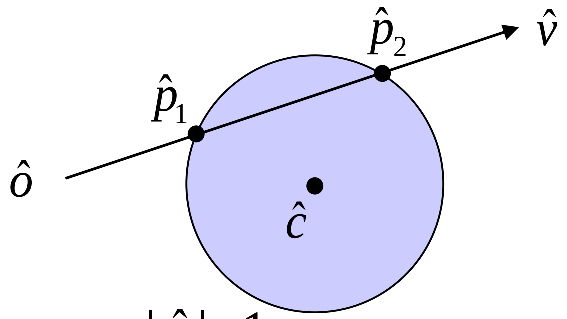
$$\frac{d}{d\lambda}\mathrm{d}^2 = 2\lambda + 2\hat{b} \circ \hat{v} = 0$$

$$\therefore \lambda = -\hat{b} \circ \hat{v} = -(\hat{o} - \hat{c}) \circ \hat{v}$$

$$\therefore \hat{x} = \hat{o} - \left((\hat{o} - \hat{c}) \circ \hat{v}\right)\hat{v}$$

Example of Sphere Intersection

- Get Intersection point for Ray Tracing



$$|\hat{v}| = 1$$

$$\hat{x} = \hat{o} + \lambda \hat{v}$$

$$radius, r^2 = |\hat{x} - \hat{c}|^2 = |\hat{o} - \hat{c} + \lambda \hat{v}|^2 = |\hat{b} + \lambda \hat{v}|^2$$

$$= |\hat{b}|^2 + \lambda^2 |\hat{v}|^2 + 2\lambda \hat{b} \circ \hat{v}$$

$$\therefore \lambda^2 + 2\lambda \hat{b} \circ \hat{v} + |\hat{b}|^2 - r^2 = 0$$

$$\lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-2\hat{b} \circ \hat{v} \pm \sqrt{4\left(\hat{b} \circ \hat{v}\right)^2 - 4\left(|\hat{b}|^2 - r^2\right)}}{2}$$

If Ray passes the Sphere or Not

$$\hat{x} = \hat{o} + \lambda \hat{v}$$

$$\lambda_{1,2} = -\hat{b} \circ \hat{v} \pm \sqrt{\left(\hat{b} \circ \hat{v}\right)^2 - \left(|\hat{b}|^2 - r^2\right)}$$

- 3Dim space is in a REAL Space

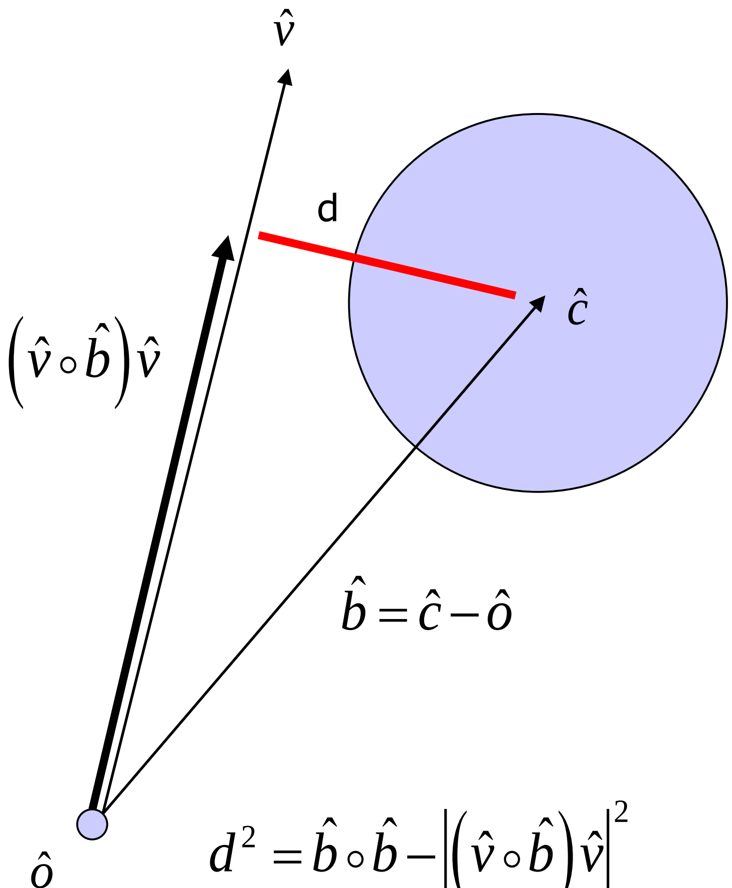
$$D = \left(\hat{b} \circ \hat{v}\right)^2 - \left(|\hat{b}|^2 - r^2\right) < 0 : No \text{ Intersection}$$

$$D = \left(\hat{b} \circ \hat{v}\right)^2 - \left(|\hat{b}|^2 - r^2\right) \geq 0 : \text{Intersection}$$

$$\lambda_{1,2} = -\hat{b} \circ \hat{v} \pm \sqrt{\left(\hat{b} \circ \hat{v}\right)^2 - \left(|\hat{b}|^2 - r^2\right)}$$

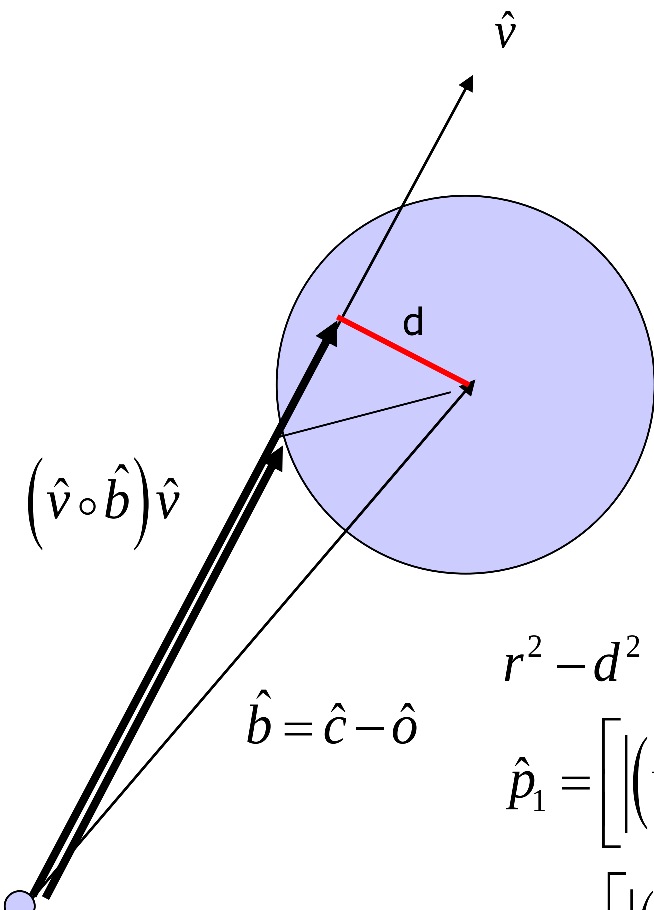
$$\hat{p}_1 = \hat{o} + \lambda_1 \hat{v}, \quad \hat{p}_2 = \hat{o} + \lambda_2 \hat{v}$$

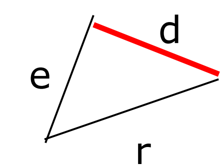
Alternative Solution by Geometry



$$d^2 = \hat{b} \circ \hat{b} - \left| (\hat{v} \circ \hat{b})\hat{v} \right|^2$$

$d > r$: *No Intersection*



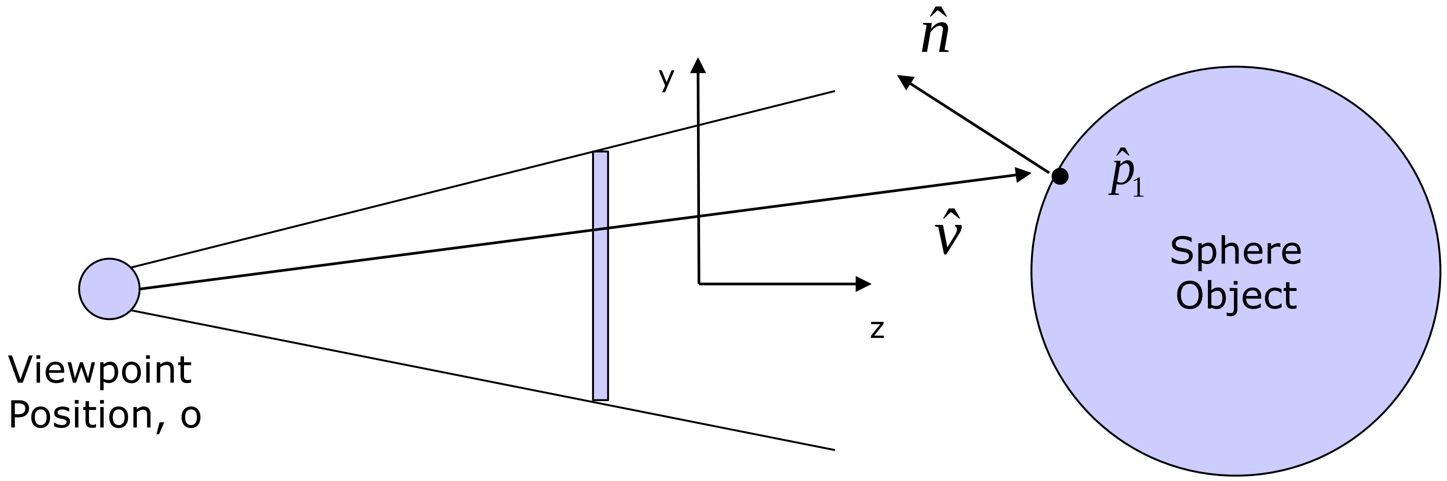


$$r^2 - d^2 = e^2$$

$$\hat{p}_1 = \left[\left| (\hat{v} \circ \hat{b})\hat{v} \right|^2 - e^2 \right] \hat{v}$$

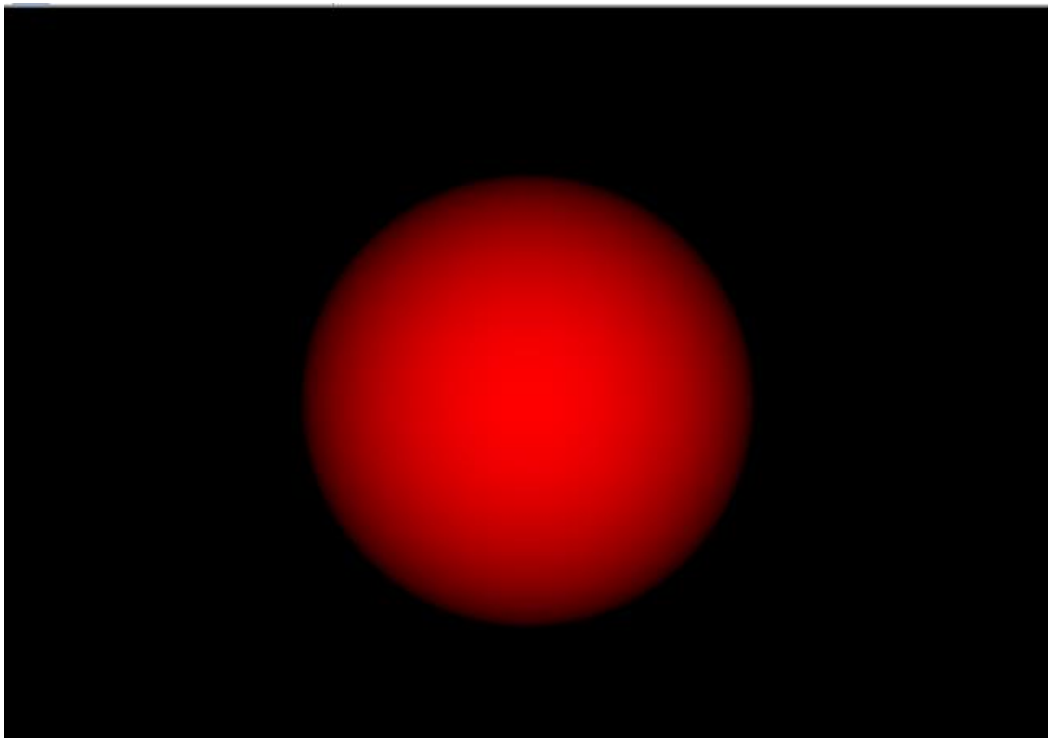
$$\hat{p}_2 = \left[\left| (\hat{v} \circ \hat{b})\hat{v} \right|^2 + e^2 \right] \hat{v}$$

Ray Tracing with a Sphere
uRT-07-RT3-Object



- Assume that **Light source is on Viewpoint, o.**
- Step 1) Fire ray
- Step 2) Find the intersection point, p1
- Step 3) Get Unit Normal $\hat{n} = (\hat{p}_1 - \hat{c})_u$
- Step 4) Color = $\cos\theta = \hat{i} \circ \hat{n}$

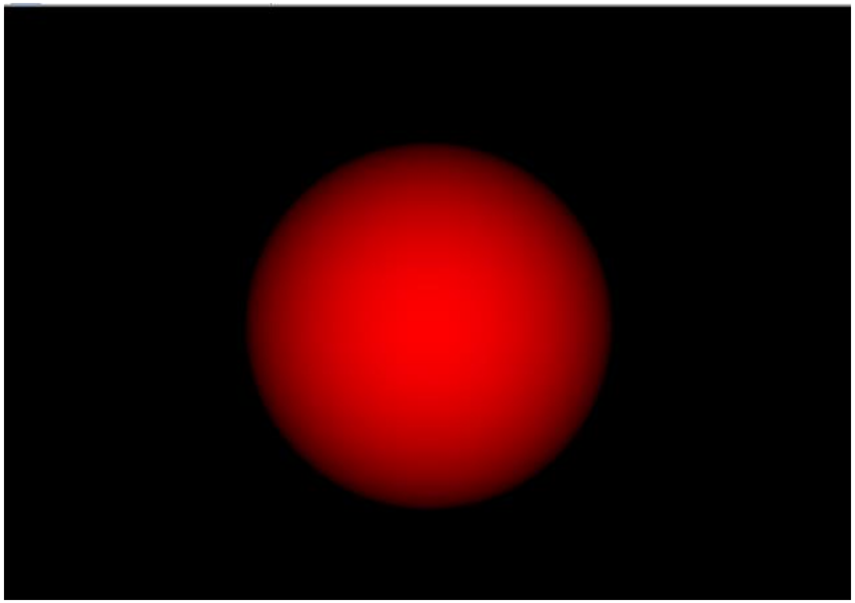
uRT-07-RT3-Object

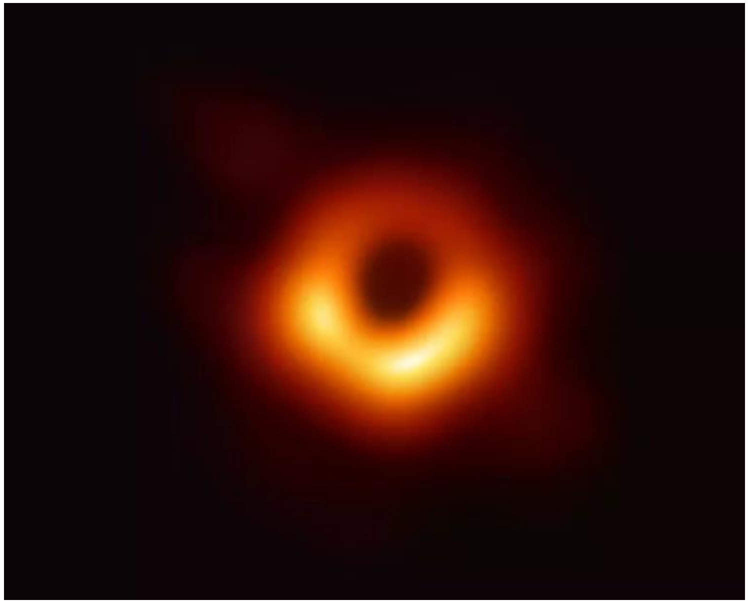


```
for (int j=0;j<h;j++)
for (int i=0;i<w;i++)
{
    uVector v = Ray(i,j);
    c    = FindRGB(screen.o,v);
```

```
uColor uRT::FindRGB(uVector o, uVector v)
{
    uColor ret;

    float fmin=1e10;
    for (int i=0;i<m_objs.GetSize();i++)
    {
        // Get minimum.
        uObj *p = &m_objs[i];
        float f = p->Distance(o,v);
        if (f<fmin) fmin = f;

        // get intersection point
        if (f<0)    continue;
        uVector pt= o+v*f;
        uVector n = p->Normal(pt);

        ret.r    = -v.Dot(n);
        ret.g = 0;
        ret.b = 0;;
    }

    return ret;
}
```

Break Time





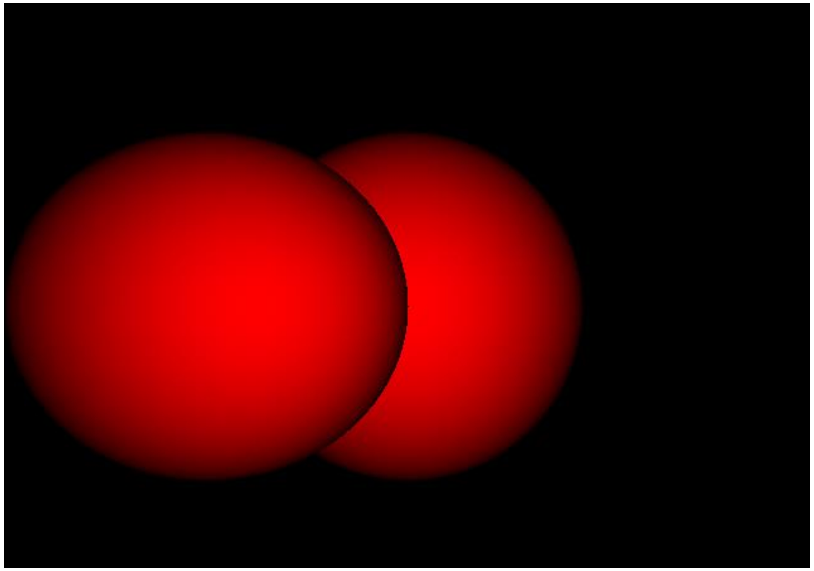
Black hole by X ray images

- Lambertian Model works as Smoothing effect
- Black hole images are Rendered by Mathematical Calculation(It is NOT an optical image)

Ray Tracing Depth Sorting

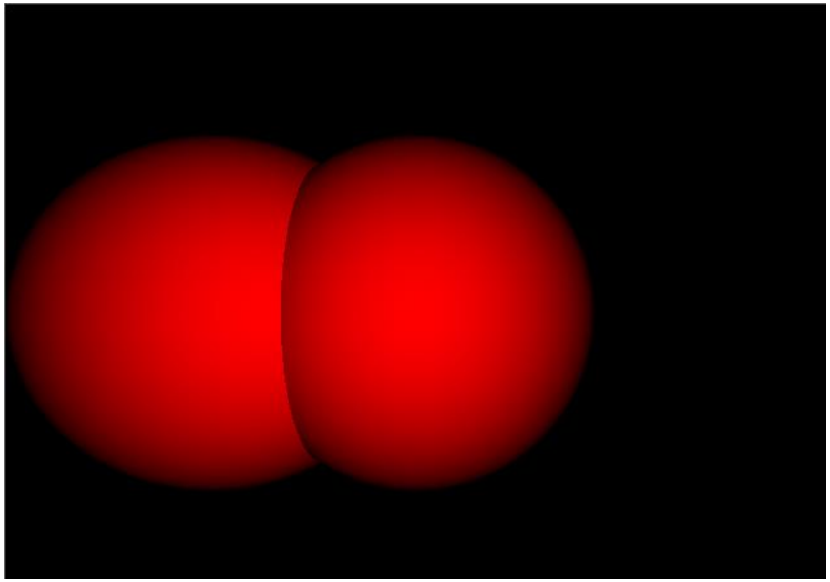
- It is similar to Z-buffer method
- Depth sorting finds which one is nearest to viewpoint.

uRT-08-RT4-MultiObject-ZProb



Bad Case

uRT-09-RT4-MultiObject-ZOrder

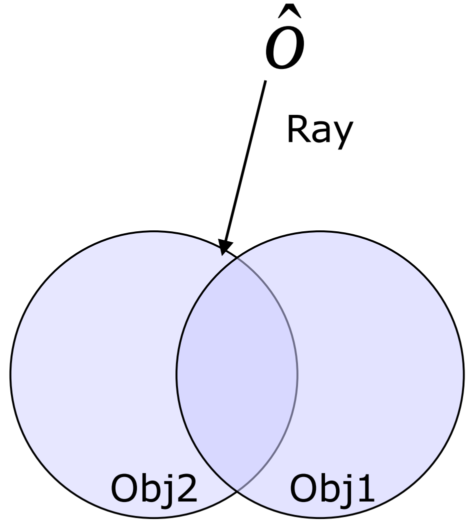


Good case

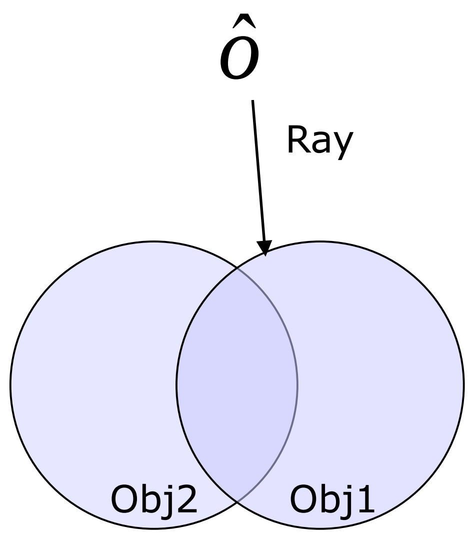
Modified FindRGB

```
uColor uRT::FindRGB(uVector o, uVector v)
{
    uColor ret;

    float   fmin=1e10;
    int     mini=-1;
    for (int i=0;i<m_objs.GetSize();i++)
    {
        // Get minimum.
        uObj *p = &m_objs[i];
        float f = p->Distance(o,v);
        if (f<0)    continue;

        if (f<fmin)
        {
            fmin = f;
            mini = i;
        }
    }
    // draw the nearest object which has the minimum distance
    if (mini>=0)
    {
        // get intersection point
        uObj *p = &m_objs[mini];
        uVector pt= o+v*fmin;
        uVector n = p->Normal(pt);

        ret.r    = -v.Dot(n);
        ret.g = 0;
        ret.b = 0;;
    }

    return ret;
}
```

Find which one has minimum distance from viewpoint, o



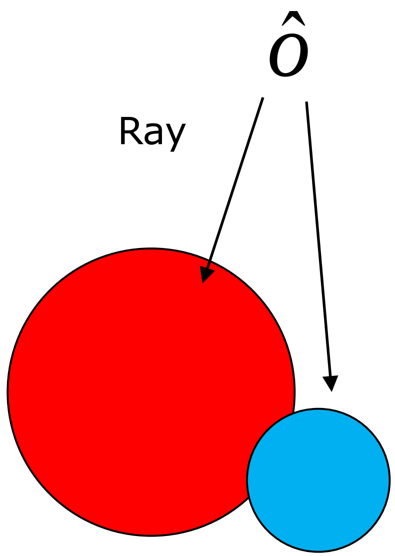
Obj2 is closed to position, o

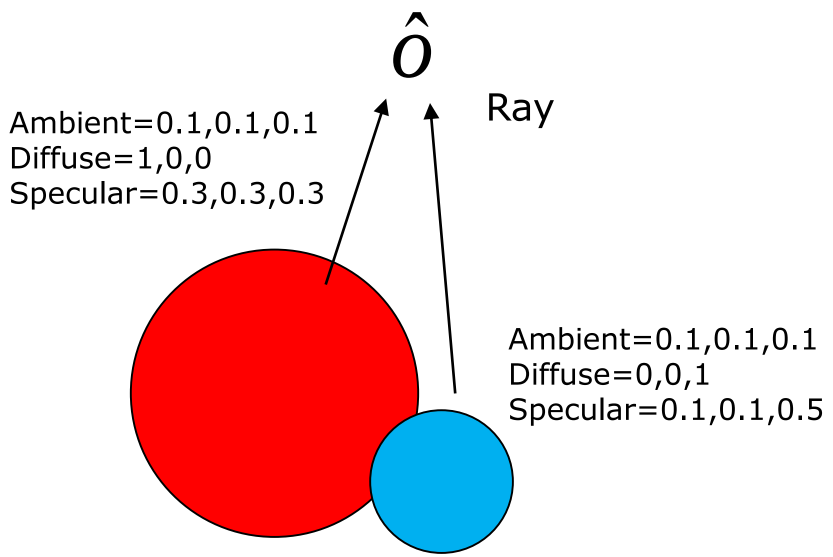


Obj1 is closed to position, o

Diffuse, Ambient, and Specular in Ray Casting

- Ray get color from Object



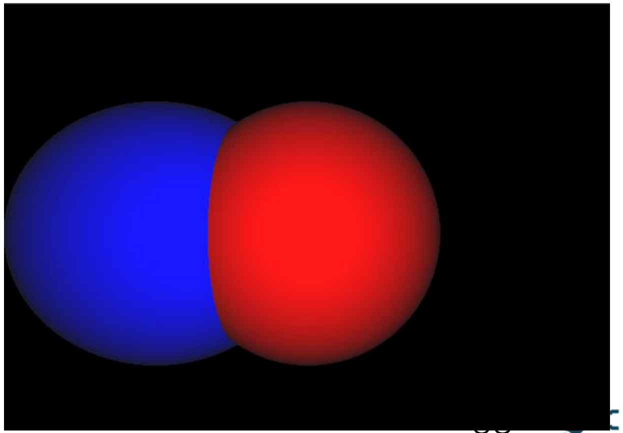


uRT-10-RT5-Colors-Ambient_diffuse

```
uColor uRT::FindRGB(uVector o, uVector v)
{
    uColor ret;

    float   fmin=1e10;
    int     mini=-1;
    for (int i=0;i<m_objs.GetSize();i++)
    {
        // Get minimum.
        ...
    }

    // draw the nearest object which has the minimum distance
    if (mini>=0)
    {
        // get intersection point
        uObj *p = &m_objs[mini];
        uVector pt= o+v*fmin;
        uVector n = p->Normal(pt);

        float dot = -v.Dot(n);

        ret      = p->ambient+ p->diffuse*dot +p->specular;
    }

    return ret;
}
```

One Step Ray Tracing

$$RGB=ambient+diffuse\cdot\cos\theta+specular$$

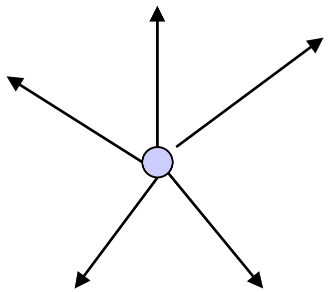


Light Position

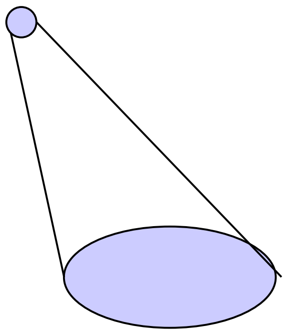
- Light object: uLight
  - The previous examples has the light at the viewpoint

```
// Illumination source(light)
class uLight
{
public:
        uLight();
public:

public:
        uVector o;
};
```

Light position



Point light

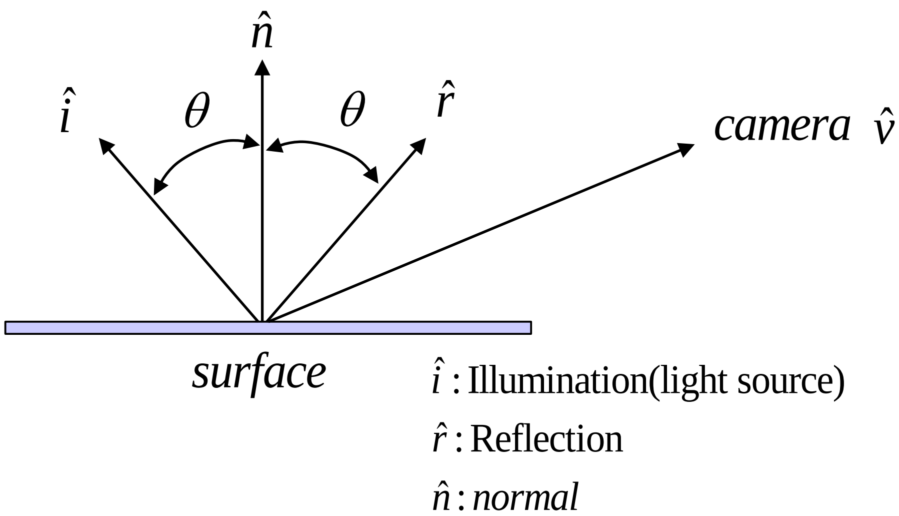


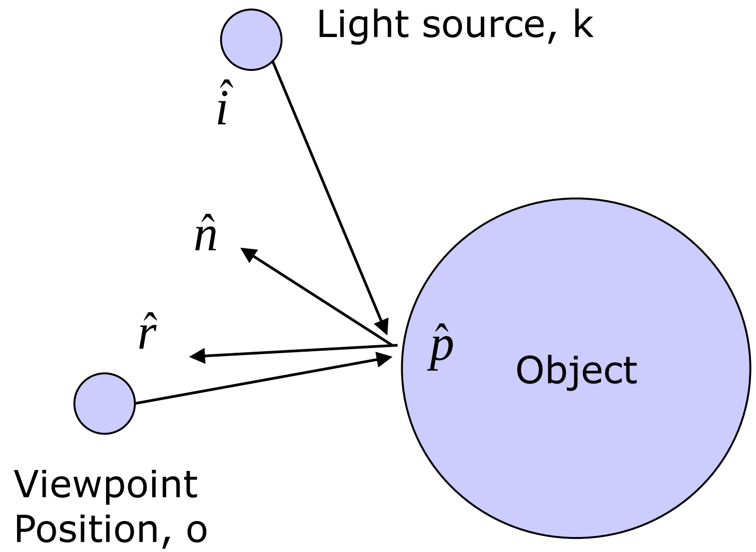
Directional light

$$Attenuation, \alpha = func(\text{distance})$$

$$RGB' = RGB \cdot \alpha$$

How to Calculate the Distance to Point Lights





- Step 1. calculate intersection point, p
- Step 2. calculate illumination vector, $\hat{i} = \hat{k} - \hat{p}$
- Step 3. calculate normal vector
- Step 4. calculate Reflection vector $\hat{r} = 2\hat{n} - \hat{i}$

Math of Reflection Vector
pp. 54 in Lecture 8

Reflection vector

$$\frac{\hat{i}+\hat{r}}{2}=\left(\hat{i}\circ\hat{n}\right)\hat{n}$$

$$\therefore \hat{r}=2\left(\hat{i}\circ\hat{n}\right)\hat{n}-\hat{i}$$

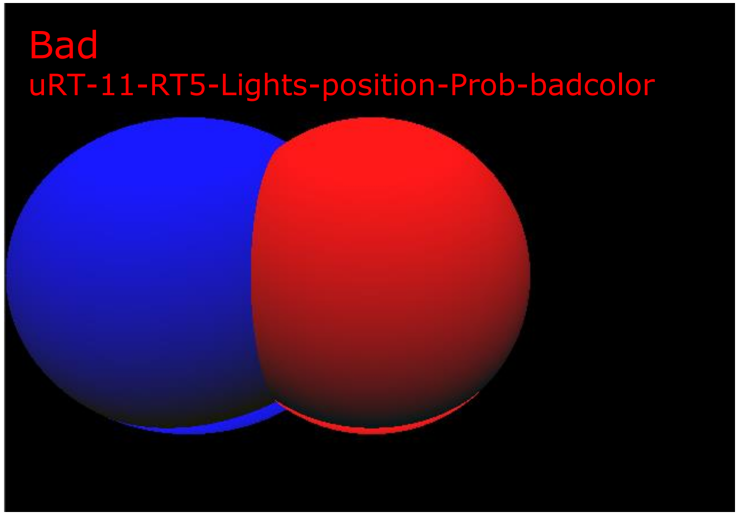
Reflection vector

$$\frac{\hat{i}+\hat{r}}{2}=\hat{n}$$

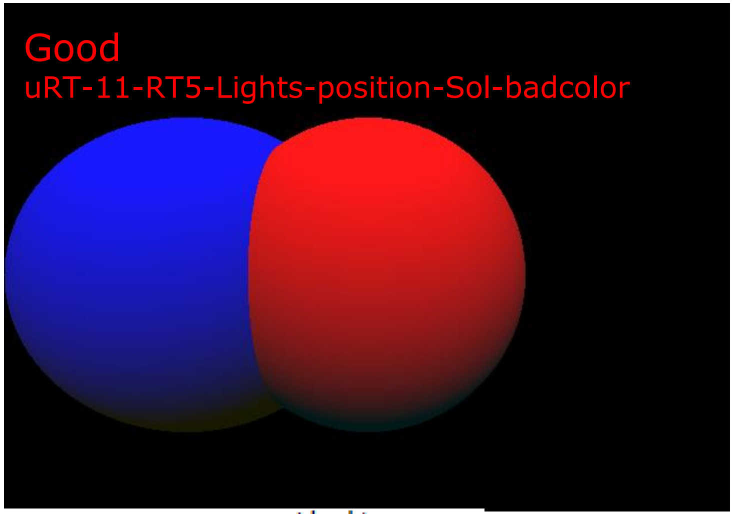
$$\therefore \hat{r}=2\hat{n}-\hat{i}$$

- What is the difference?
- Think if illumination vector $\hat{i}$ is normalized,
  - The result is same.

Ex) Bad and Good Case

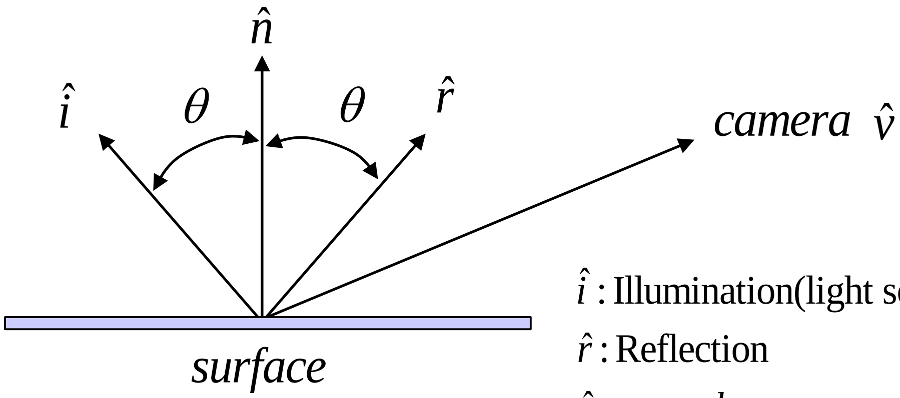



```
uVector v = Ray(i,j);
c    = FindRGB(screen.o,v);

c.r *=255;
c.g *=255;
c.b *=255;

if (c.r>255)      c.r=255;
if (c.g>255)      c.g=255;
if (c.b>255)      c.b=255;

r = (BYTE)c.r;
g = (BYTE)c.g;
b = (BYTE)c.b;
```




```
uVector v = Ray(i,j);
c    = FindRGB(screen.o,v);

c.r *=255;
c.g *=255;
c.b *=255;

if (c.r>255)      c.r=255;
if (c.g>255)      c.g=255;
if (c.b>255)      c.b=255;

if (c.r<0)    c.r=0;
if (c.g<0)    c.g=0;
if (c.b<0)    c.b=0;

r = (BYTE)c.r;
g = (BYTE)c.g;
b = (BYTE)c.b;
```

Phong Effect
pp. 54 in lecture 8



$\hat{i}$: Illumination(light source)
$\hat{r}$: Reflection
$\hat{n}$: *normal*

$$\therefore \cos\alpha = \hat{r} \circ \hat{v}$$

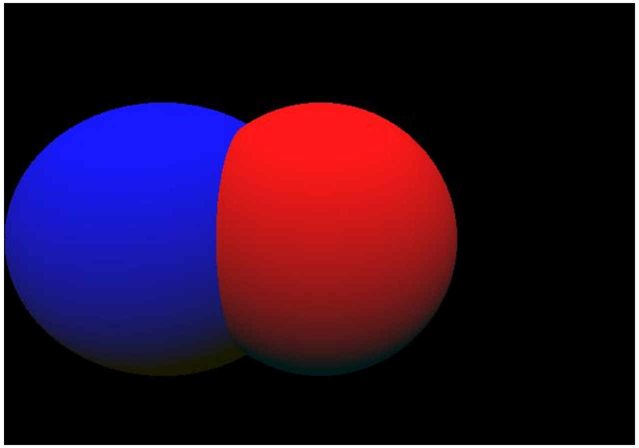
$$S(\alpha) = \cos\alpha^{s}$$

- Color is determined by

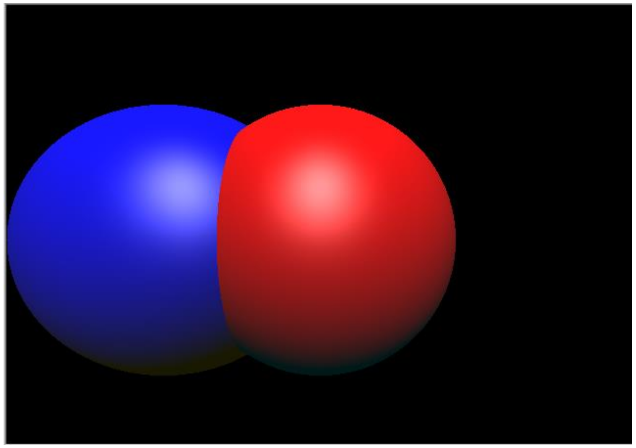
$$RGB = Ambient + Diffuse \cdot \cos\theta + Specular \cdot S(\alpha)$$
$$= Ambient + Diffuse \cdot (\hat{i} \circ \hat{n}) + Specular \cdot (\hat{r} \circ \hat{v})^{s}$$

Phong’s effect Result

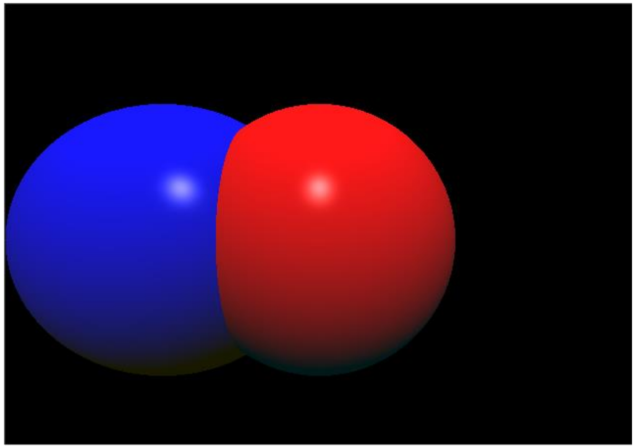
uRT-12-RT6-Colors-Specular-Phong



specular



Phong’s specular $(\hat{r} \circ \hat{v})^{10}$



Phong’s specular $(\hat{r} \circ \hat{v})^{100}$

```
if  (l.Dot(n)<0)       r = uVector(0,0,0); // No reflection
else                   r = (n*2-l).Unit();

//dot = -v.Dot(n);
dot      = l.Dot(n);
sdot     = -v.Dot(r);
sdot     = pow(sdot,100);
ret      = ret + p->diffuse*dot + p->specular*sdot;
```

OpenGL with GLSL is same with Ray Tracing with Phong’s effect

What you learn here
is close to
Ray Casting

OpenGL GLSL is
Nearly
Semi Ray Tracing
(or Ray casting)

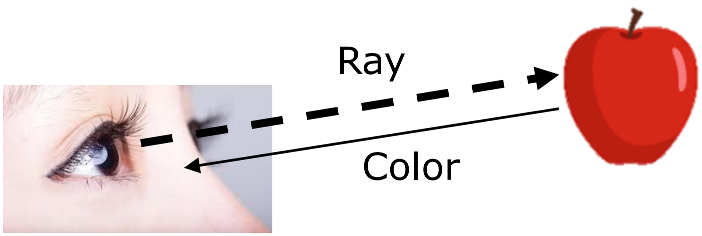
- Ray casting does not cover Phong’s effect.
- You finished Background Knowledge of OpenGL by learning Ray Tracing

3

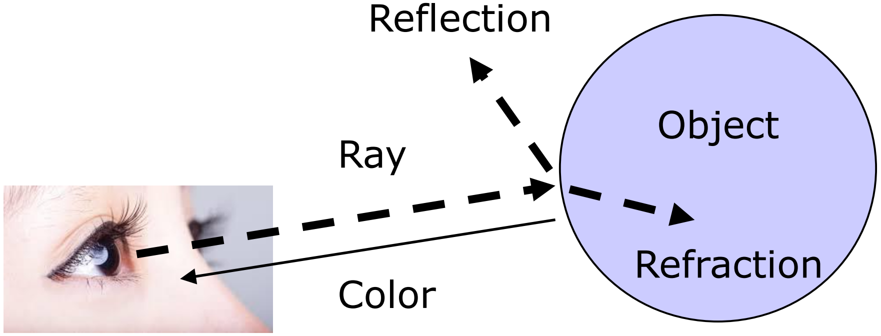
Over OpenGL Technologies

Ray Tracing with Transparency

Ray Tracing with Transparent Ray



Ray Casting

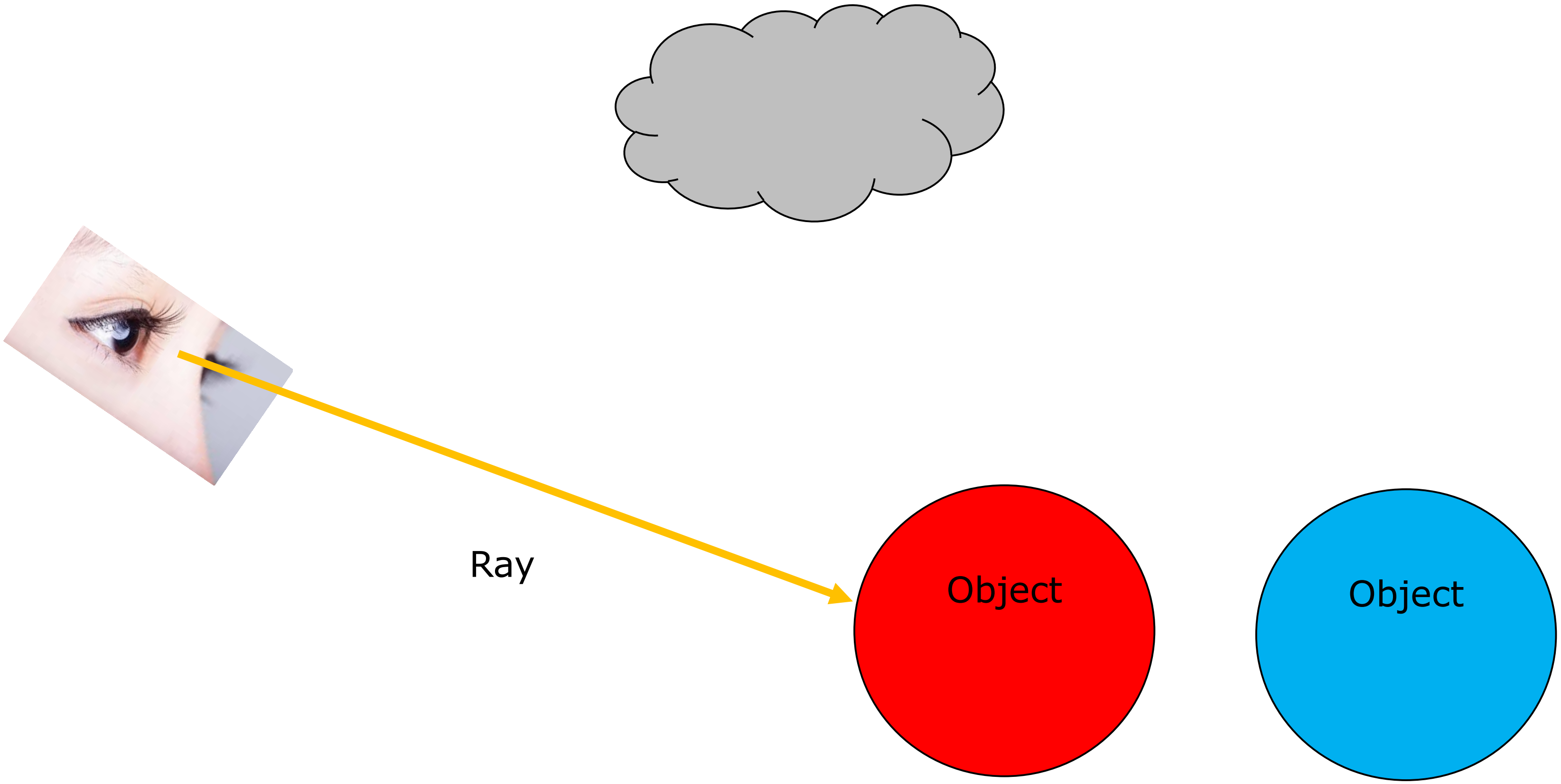


Ray Tracing

- Each Ray is divided by Two Rays, such as Reflection and Refraction

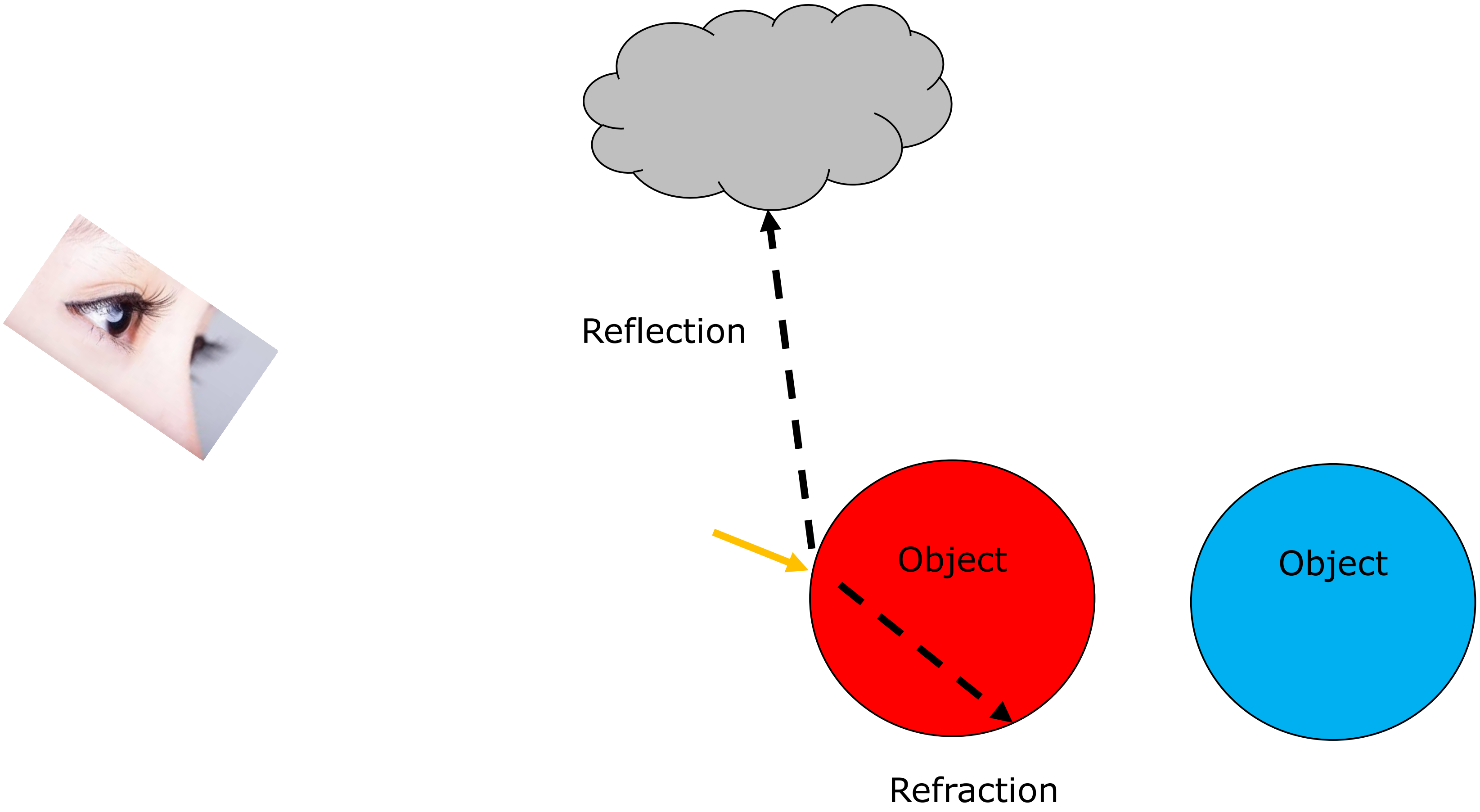
- Reflected and Refracted rays are repeatedly divided by other two rays such as reflection and Refraction

Ray Tracing Concept.
Step1. Ray Start from Eye



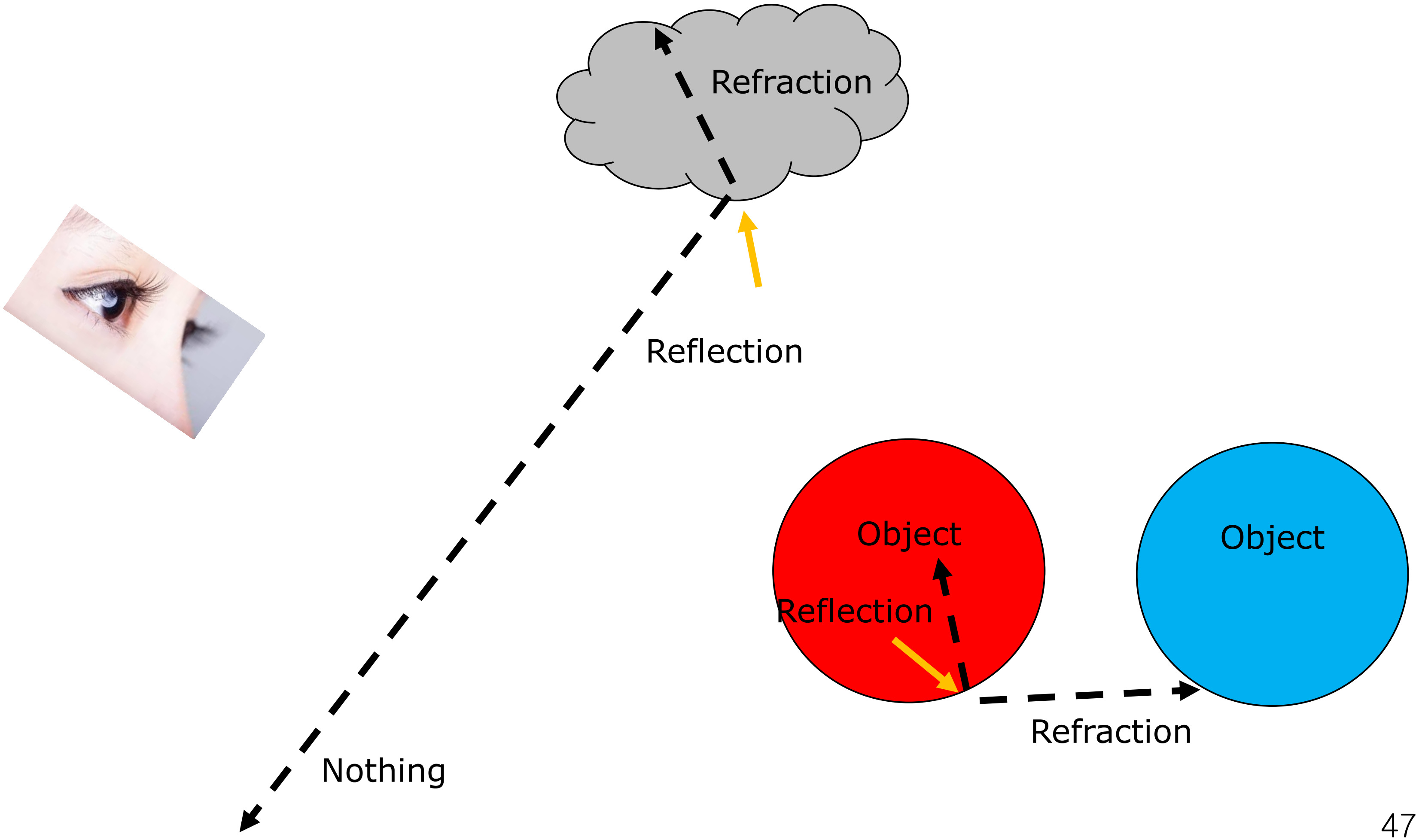
Ray Tracing Concept.

Step2. Ray generates Reflection and Refraction



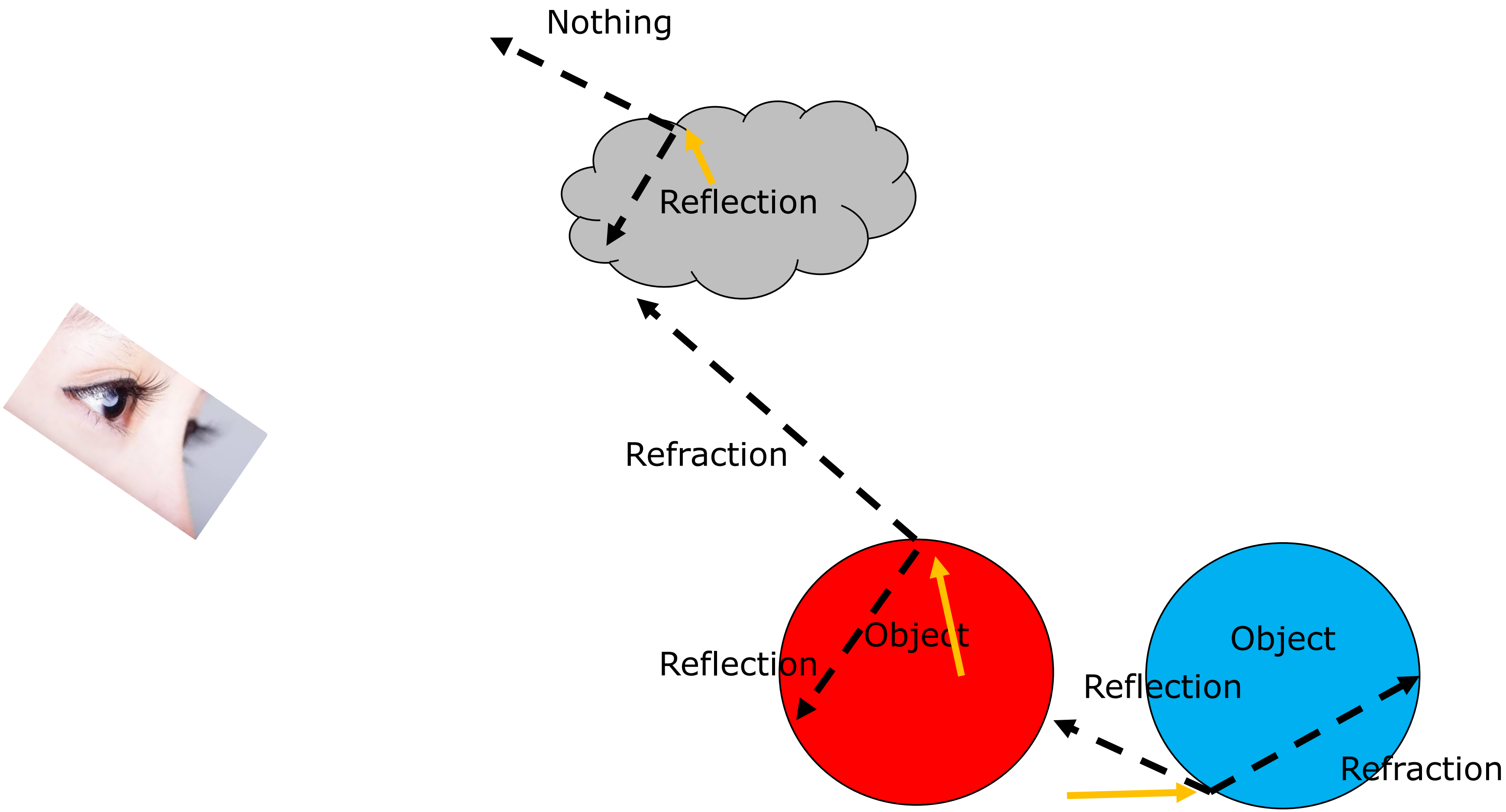
Ray Tracing Concept.

Step3. Each Ray generates Reflection and Refraction



Ray Tracing Concept.

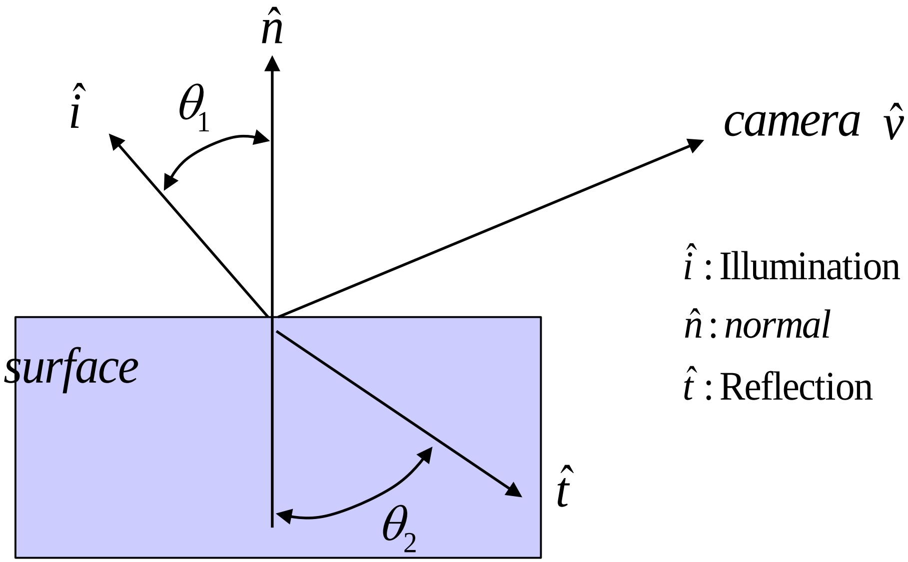
Step4. More and More Rays



Ray Tracing has three features

- 1. Nothing to meet an object or a light
  - The Ray goes astray.
  - I means that the ray meets No light → It is removed
- 2. It is called, Infinite Ray Generation → Infinite Loop
  - One ray is divided into Reflection and Refraction.
  - Set limitation of the Ray Separation(or Generation)
  - Three for simple example → Low quality.
  - One ray with three steps generates 8 rays.
- 3. Light Intensity is needed.
  - Light CANNOT go infinite distance.
  - Light intensity is Inverse proportional to the distance

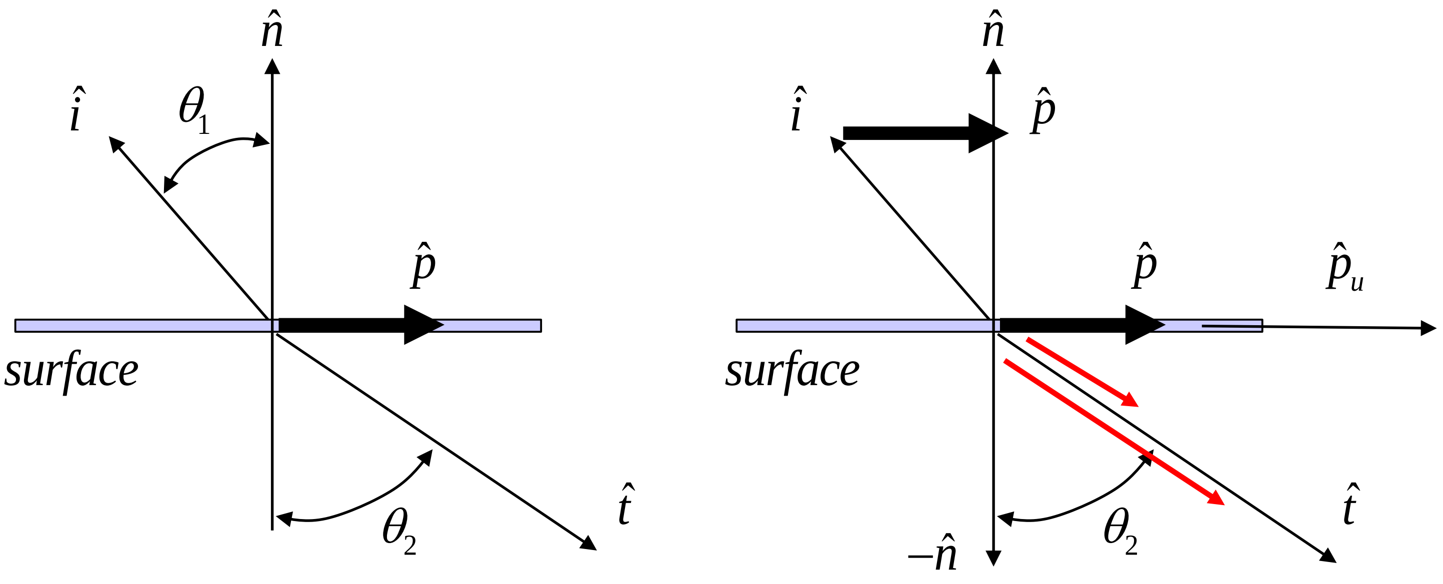
Refraction



- Snell's Law
  - Light velocity varies when passing materials

$$\text{Refractive Factor} = n = \frac{v_1}{v_2} = \frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

Refraction Vector Calculation



$$\hat{p} = \left(\hat{i} \circ \hat{n}\right)\hat{n} - \hat{i}$$

$$\hat{p}_u = \hat{p} / |\hat{p}|$$

$$\hat{t} = \sin\theta_2 \hat{p}_u + \cos\theta_2 (-\hat{n})$$

Get Refraction Vector

```
uVector uObj::Refraction(uVector ray,uVector n)
{
    uVector ni    = ray*-1;
    float d       = ni.Dot(n);
    uVector p     = n*d+ray;
    p             = p.Unit();

    float s1      = sqrt(1-d*d);
    float s2      = s1/geo.tr;

    float c2;
    if (s2>=1)  return uVector(0,0,0);

    c2  = sqrt(1-s2*s2);
    uVector t;

    t   = p*s2 -n*c2;
    t   = t.Unit();
    return t;
}
```

$$|ray, \hat{v}| = 1$$

$$\hat{i} = -\hat{v}$$

$$\hat{p} = \left(\hat{i} \circ \hat{n}\right)\hat{n} - \hat{i}$$

$$\hat{p}_u = \hat{p} / |\hat{p}|$$

$$\hat{p} = \left(\hat{i} \circ \hat{n}\right)\hat{n} - \hat{i} = \left(-\hat{v} \circ \hat{n}\right)\hat{n} + \hat{v}$$

$$\hat{p}_u = \hat{p} / |\hat{p}|$$

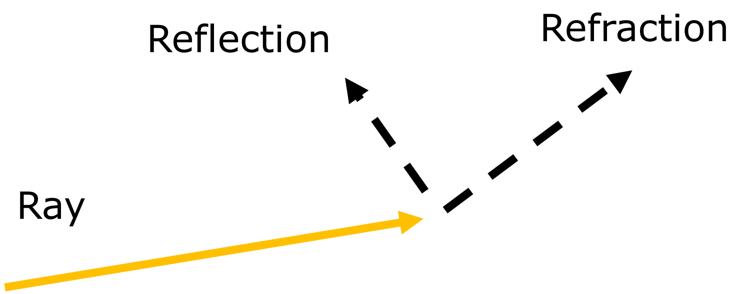
$$\text{Refractive Factor} = \frac{\sin\theta_1}{\sin\theta_2} = geo.tr$$

$$\hat{t} = \sin\theta_2 \hat{p}_u + \cos\theta_2(-\hat{n})$$

$$\hat{t}_u = \hat{t} / |\hat{t}|$$

Refraction with geo.tr vector

```
if (nCount<3)
{
    nCount++;
    int nr = nCount;
    int nt = nCount;

    uVector t   = p->Refraction(v,n);
    uColor ref  = FindRGB(pt,r,nr)*p->specular;
    uColor tra  = FindRGB(pt,t,nt)*p->transparent;
    ret = ret + ref +tra;
}
```



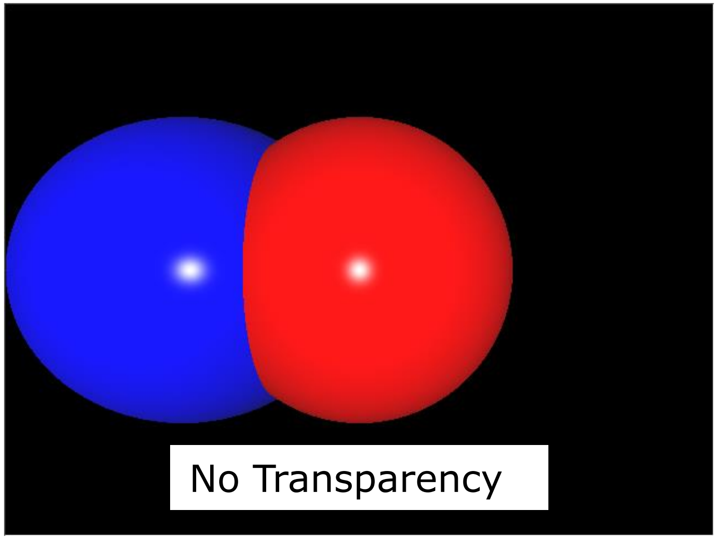
- Three steps Ray tracing.

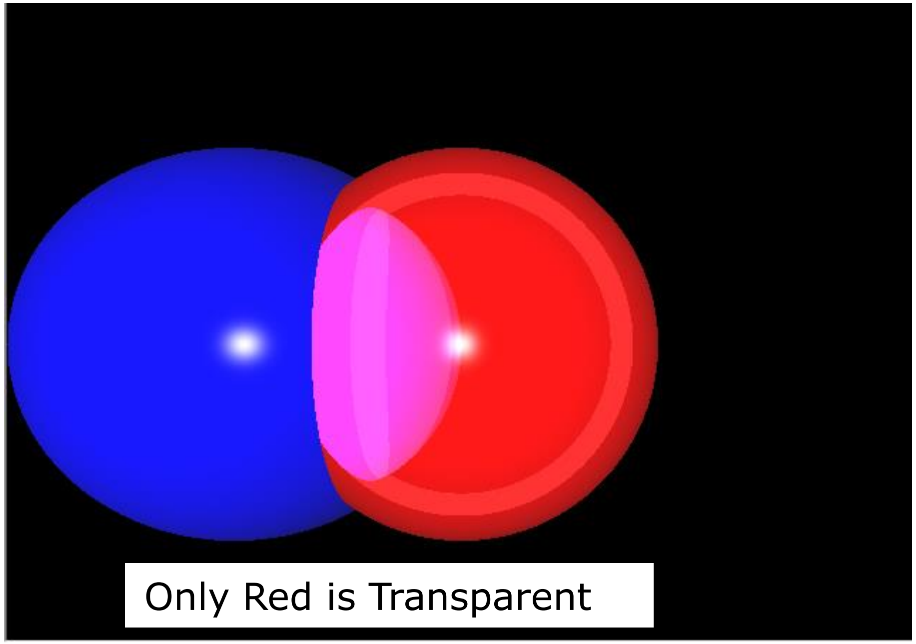
Ret = Diffuse + Ambient

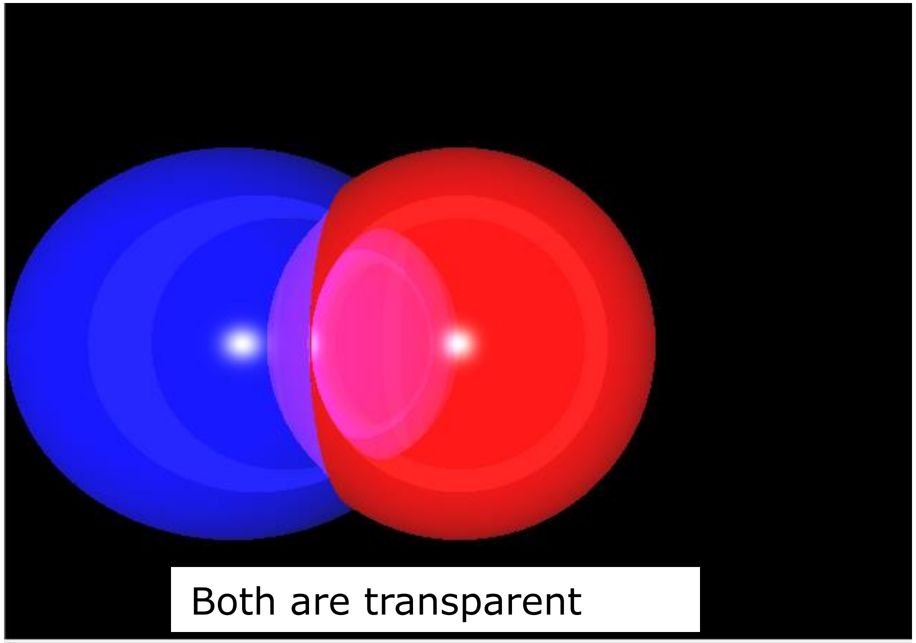
Ref = Reflected Color * Specular
Tra = Transparent Color * Transparent

→ Final Ret = ret+ Ref + Tra

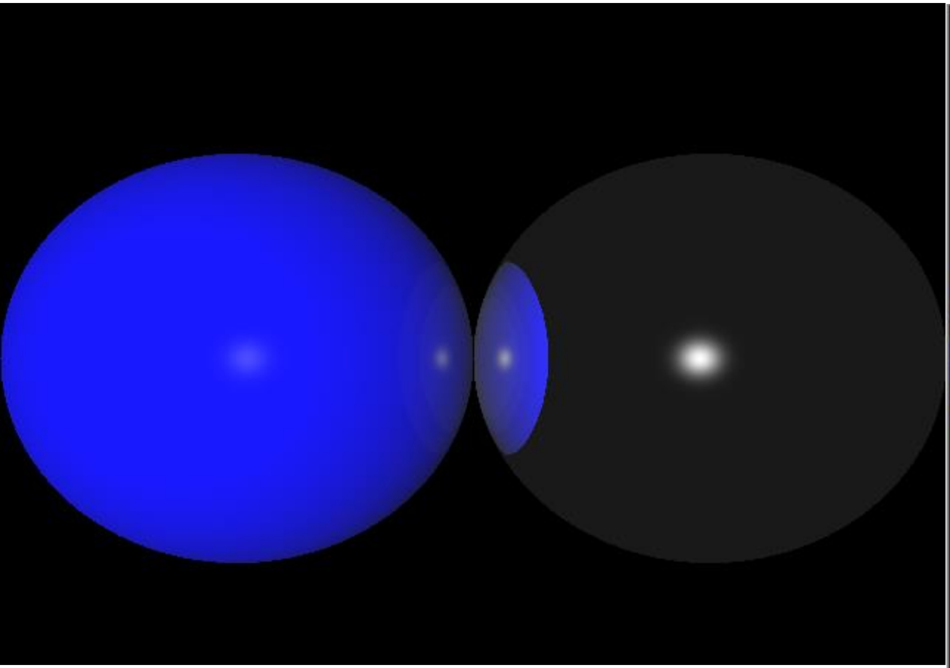
Various Types of Transparency



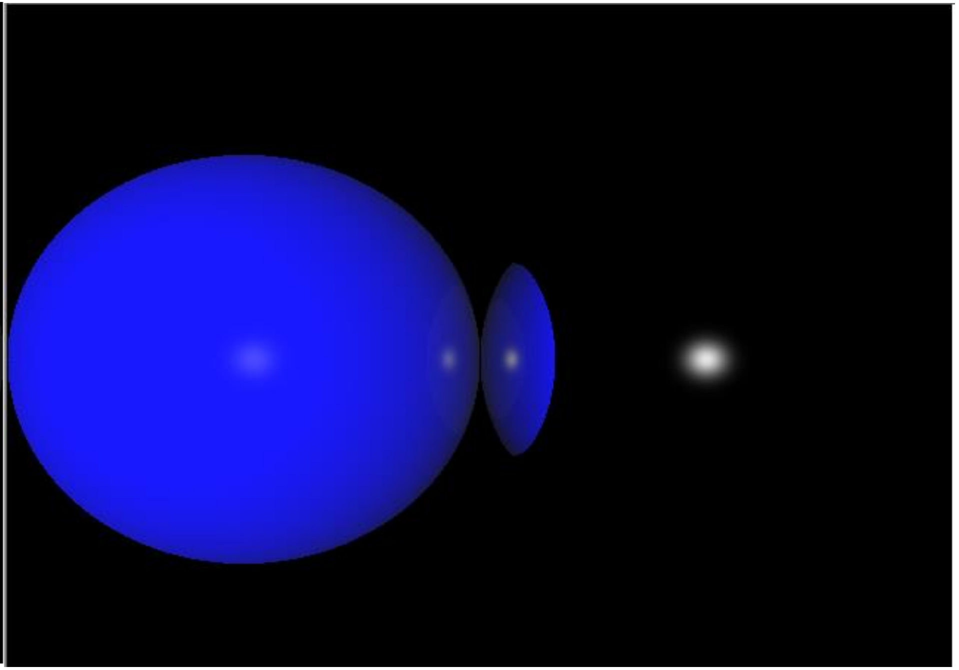




Mirror with No Transparency



It is Not a Perfect Mirror



It is a Perfect Mirror