Basic Ray Tracing

Why Build a Ray Tracer?

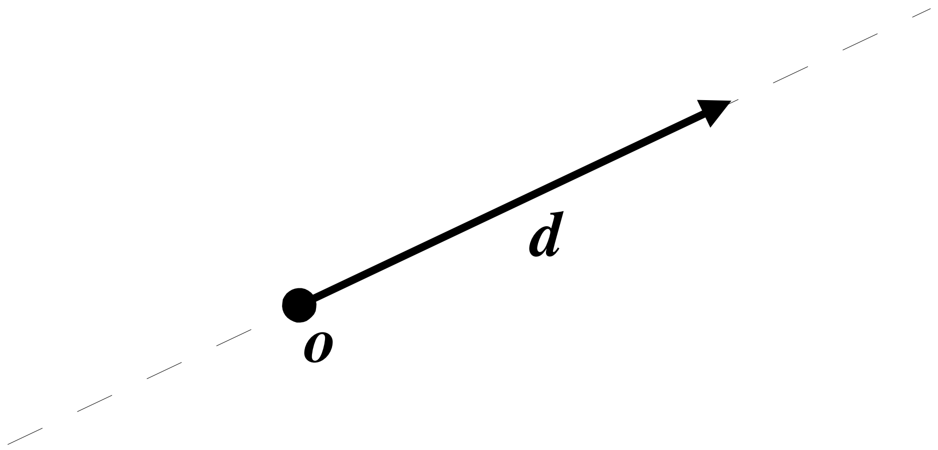
- Ray tracing is more elegant than polygon scan conversion.
- Testbed for lots of effects in
  - modelling
  - rendering
  - texturing
  - animation
- Ray tracers are the easiest renderers to implement.
- Ray tracers are used in practically *all* other photorealistic renderers.

Rays are Parametric Lines

- Describe a line with a parameter *t*:

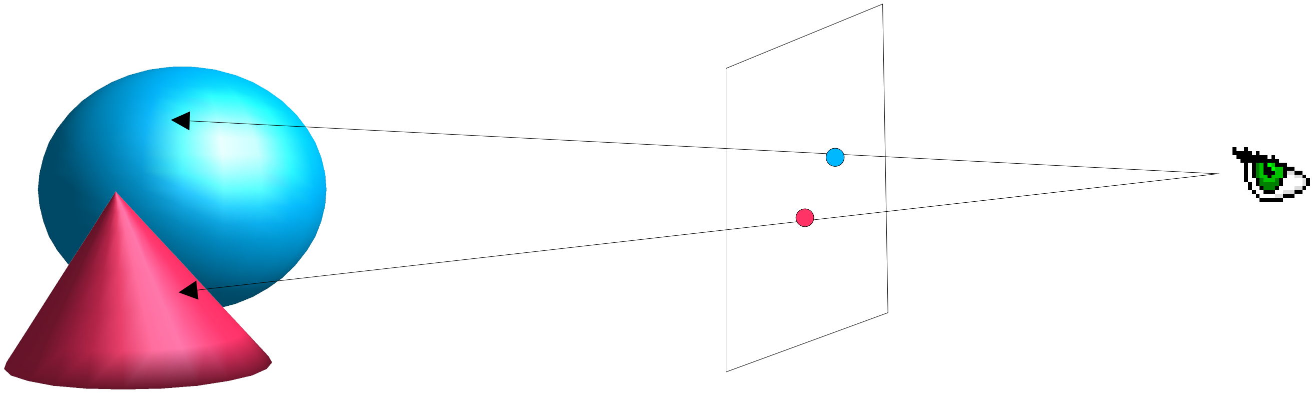
$$\boldsymbol{p}(t)=\boldsymbol{o}+\boldsymbol{d}\,t$$

- ***o*** is one point on the line, ***d*** is a direction
- If we restrict t ≥ 0, ***p***(*t*) describes a ray from origin ***o***.



Why We Call it Ray Tracing (or Casting)

- Cast rays from the eye through a viewing plane (pixel) into a scene.
- Compute intersection with objects in the scene.
- Compute lighting at the intersection.
- Set pixel to computed color (or background if no object is hit).



Ray/Implicit Surface Intersections

- Implicit surfaces are given by $f(x,y,z) = f(\mathbf{p}) = 0$.
- Substitute $\boldsymbol{p} = \boldsymbol{o} + \boldsymbol{d}t$ into it, you get $f(\mathbf{p}(t)) = 0$.
- What kind of mathematical problem does this become?
- In how many dimensions?
- Are these easy to solve, in general?
- How can we compute the normal ***n*** of an implicit surface?

Ray/Implicit Surface Intersections: Planes

- For a plane $f(\boldsymbol{p}) = \boldsymbol{n} \cdot \boldsymbol{p} + D$.
- (If you know one point $\boldsymbol{a}$ on the plane, D = $-\boldsymbol{n} \cdot \boldsymbol{a}$.)
- Substitute $\boldsymbol{p} = \boldsymbol{o} + \boldsymbol{d}t$ into it, you get $(\boldsymbol{n} \cdot \boldsymbol{o}) + (\boldsymbol{n} \cdot \boldsymbol{d})t + D = 0$.
- Solve to get

$$t = \frac{-D - (\boldsymbol{n} \cdot \boldsymbol{o})}{\boldsymbol{n} \cdot \boldsymbol{d}}$$

- What can go wrong?
- Normal is immediately available.

Ray/Parametric Surface Intersections

- Parametric surface:

$$\boldsymbol{p}(u,v)=\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}f(u,v)\\g(u,v)\\h(u,v)\end{bmatrix}$$

- Can you think of an example?
- So we have to solve

$$\boldsymbol{o}+\boldsymbol{d}\,t=\begin{bmatrix}f(u,v)\\g(u,v)\\h(u,v)\end{bmatrix}$$

- How many equations? How many unknowns? What are they?
- Is this easy? Are multiple solutions possible?
- Normal computed via

$$\boldsymbol{n}(u,v)=\frac{\frac{\partial\boldsymbol{p}}{\partial u}\times\frac{\partial\boldsymbol{p}}{\partial v}}{\left|\frac{\partial\boldsymbol{p}}{\partial u}\times\frac{\partial\boldsymbol{p}}{\partial v}\right|}$$

Ray/Sphere Intersections: Problem

- Like planes, spheres are implicit surfaces: $f(\boldsymbol{p}) = |\boldsymbol{p} - \boldsymbol{c}|^2 - R^2$.
- Substitute $\boldsymbol{p} = \boldsymbol{o} + \boldsymbol{d}t$ into it and you get $|\boldsymbol{o} + \boldsymbol{d}t - \boldsymbol{c}|^2 - R^2 = 0$.
- Rearranging, you get

$$(\boldsymbol{d} \cdot \boldsymbol{d})t^2 + 2\boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c})t + |\boldsymbol{o} - \boldsymbol{c}|^2 - R^2 = 0$$

- What kind of equation is this?
- Is it easy to solve?

Ray/Sphere Intersections: Solution

- solution:

$$t = \frac{-B \pm \sqrt{B^2 - 4\mathrm{AC}}}{2\mathrm{A}}$$

where

$$A = |\boldsymbol{d}|^2$$
$$B = 2\,\boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c})$$
$$C = |\boldsymbol{o} - \boldsymbol{c}|^2 - R^2$$

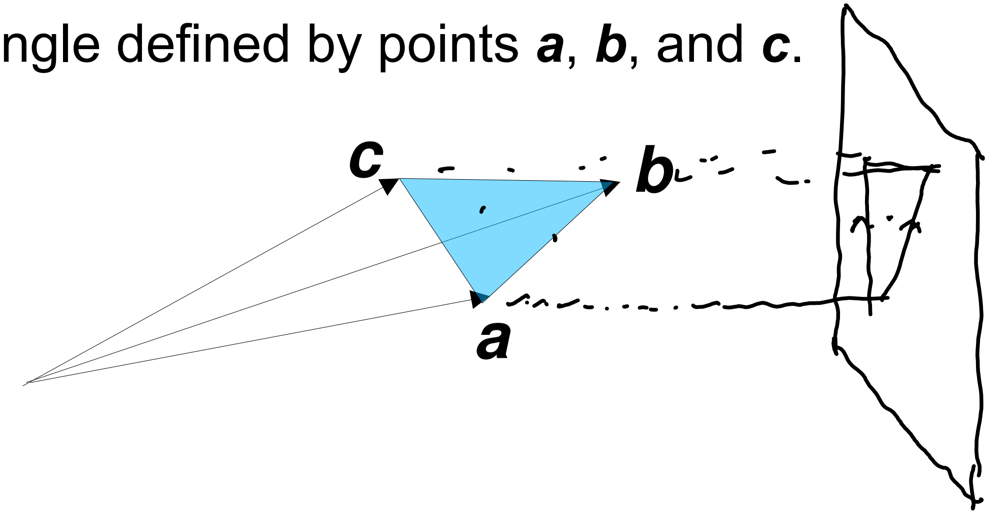
- What do the solutions mean?
- Spheres can be used as bounding volumes.

Ray/Box Intersections

- see book

Ray/Triangle Intersections: Problem

- Triangle defined by points ***a***, ***b***, and ***c***.



- Barycentric coordinates: $\boldsymbol{p} = \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}$ x = constant
- If $\alpha + \beta + \gamma = 1$, p lies in plane defined by points.
- If (α,β,γ) all lie between 0 and 1, point is within triangle.
- Apply both constraints to get $\boldsymbol{p}(\alpha,\beta,\gamma) = \boldsymbol{a} + \beta(\boldsymbol{b}-\boldsymbol{a}) + \gamma(\boldsymbol{c}-\boldsymbol{a})$.
- Solve $\boldsymbol{p} = \boldsymbol{o} + \boldsymbol{d}t = \boldsymbol{a} + \beta(\boldsymbol{b}-\boldsymbol{a}) + \gamma(\boldsymbol{c}-\boldsymbol{a})$
- How many equations? How many unknowns? What are they?

Ray/Triangle Intersections: Solution

- How do we solve this?
- Rewriting ***o*** + ***d****t* = ***a*** + β***(b-a)*** + γ***(c-a)*** in matrix form:

$$\begin{bmatrix} (\boldsymbol{b}-\boldsymbol{a}) & (\boldsymbol{c}-\boldsymbol{a}) & -\boldsymbol{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \boldsymbol{M} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \boldsymbol{o}-\boldsymbol{a} \end{bmatrix}$$

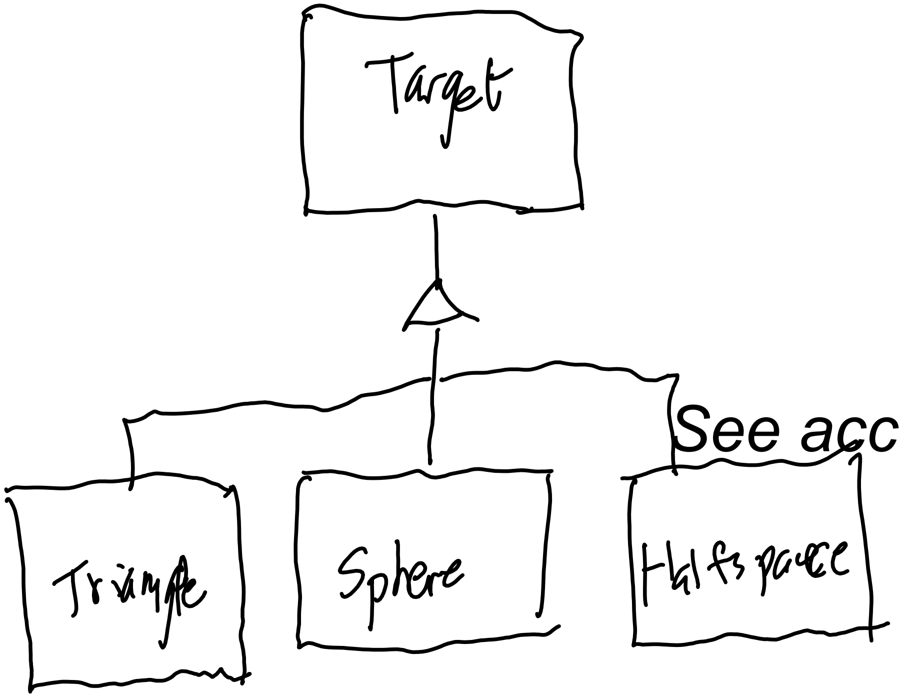
$$\Rightarrow \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \boldsymbol{M}^{-1} \begin{bmatrix} \boldsymbol{o}-\boldsymbol{a} \end{bmatrix}$$

- How do we interpret results? What if ***M*** cannot be inverted? What ranges of β, γ, and *t* are important?
- Do we always need to solve for all three unknowns?

Generalizing Ray/Object Intersections

- This is an embarassingly simple example of object orientation.
- Create a "superclass" `RtObject`. (Shirley calls it a "surface".)
- Add virtual methods `rayIntersects()`, `normal()`, etc. to it.
- Add modelling transforms to it.
- Create subclasses for each kind of object (`Plane`, `Sphere`, `Triangle`, etc.).
- Add specific methods to each subclass.
- Your raytracer works on a collection of `RtObject`s.
- All geometry-specific knowledge is in subclasses.

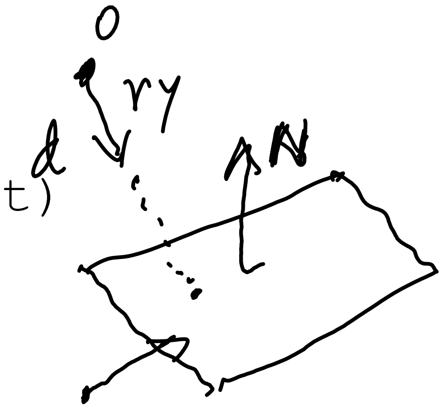
The Initial Target Hierarchy



See accompanying writeup

Target Example: The Halfspace Class

```
class Halfspace(Target):
   ...    intersection          object
   def ixFirst(hlfsp, ry):
      ry = ry.transform(hlfsp.tfScnToTgt)
      eqnAtO = ry.o[2] # z-component
      dDotN = ry.d.unit()[2]
      if dDotN == 0:
         if abs(eqnAt0) < EPSILON:
            p = ry.o
         else:
            return None
      else:
         p = ry.o - ry.d * eqnAtO / dDotN
      return Intersection(hlfsp,
            p.transform(hlfsp.tfTgtToScn))
   def uN(hlfsp, p):
      return Vector3D(hlfsp.tfTgtToScn[0:2][2]).unit()
```



Camera.raytrace(): A High-Level Ray Tracer

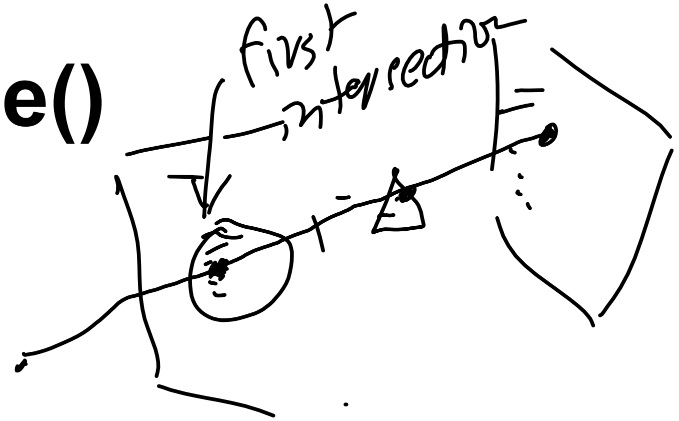
camera scene image size

in middle of pixel

```
class Camera:
...
   def raytrace(cam, scn, w, h):
       img = Image(w, h)
       for i in range(w):
           for j in range(h):
               ry = cam.ray(w, h, i+0.5, j+0.5)
               img[i][j] = scn.trace(ry)
```

- (red indicates methods we have yet to cover.)
- Note that the ray goes through the middle (0.5,0.5) of the pixel.
- We could combine the two function calls into one, but we don't, for good reason.

Scene.trace()



```
class Scene:
    ...
    def trace(scn, ry):
        ix = scn.srf.ixFirst(ry)
        if ix != None:
            uN = ix.uN()
            return ix.prim.mtl.illuminate(scn, ry, ix)
        else:
            return scn.background
```

scene

ray

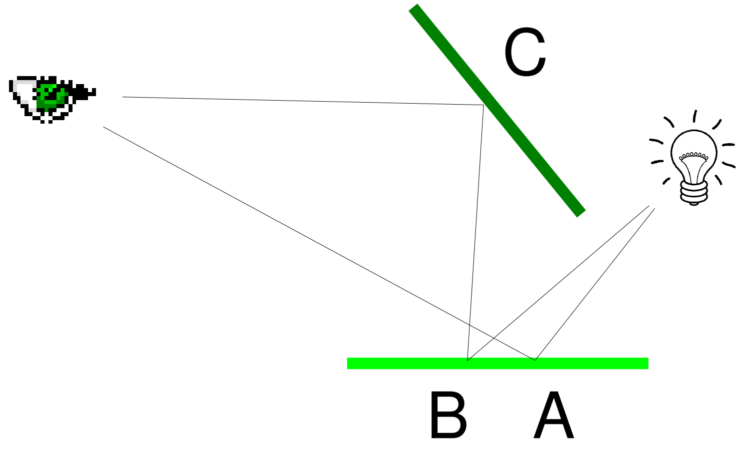
primitive

target

material

- use float RGB vectors (0 ≤ (R,G,B ≤ 1) usually

Lighting: Direct vs. Indirect

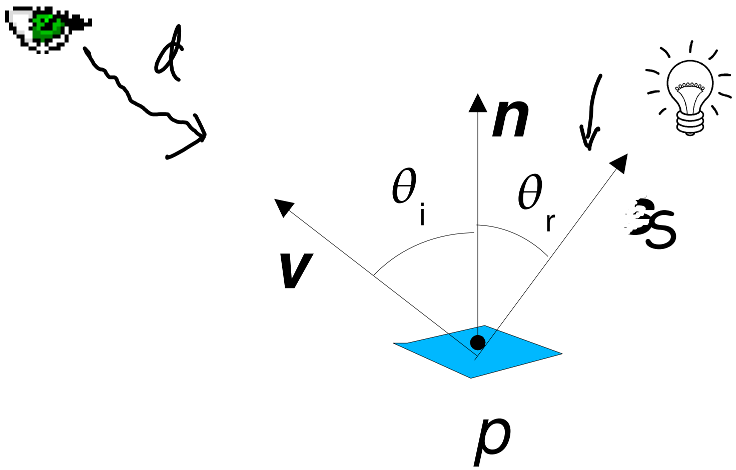


- *Direct* light bounces off a (non-mirror) surface *once* before it reaches the eye (e.g. bulb-A-eye).
- *Indirect* light bounces off more than one surface before it reaches the eye. (e.g. bulb-B-C-eye)
- The room you're in is probably lit mostly by indirect light.
- Direct light is easy to do with ray tracing.
- Indirect light is hard -- except for a few special cases -- so we use some hacks and wait to improve things later.

Lighting: Lambert's Law

- Lambert's Law defines a diffuse surface as one for which reflected light obeys the formula L = E R cos θ, where
  - E is the incident irradiance (power per unit area) intrinsic to the light source
  - R is the reflectance intrinsic to the surface
  - θ_i is the incident angle (between the normal and the light direction)
  - Note that there is no dependency on the reflected angle θ_r.
- Many surfaces in nature (and man-made) are diffuse, or almost so.
- This is the easiest illumination model (aka. *shader*) to implement.

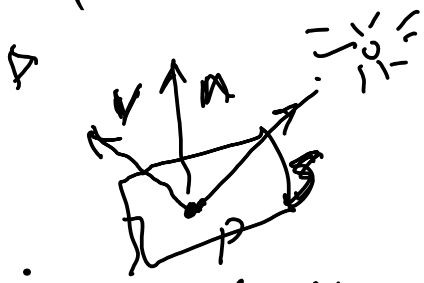
Implementing Diffuse Surfaces



- For an intersection point p, we have
  - the eye vector ***v*** (= -***d***/|***d***|)
  - the surface normal ***n***
  - a vector ***s*** in the direction of the light source
- So can we replace the cosine in Lambert's Law?

Shader.rdncDirect() and Diffuse.brdf()

The *Shader* class is for materials with a bidirectional reflectance distribution function (BRDF). The simplest of these is the Diffuse (i.e. Lambertian) shader:



radiance (physical term for light)

```
class Shader(Material):
   ...
   def rdncDirect(shdr, ix, ryIn, lum):
      uV = -ryIn.d.unit()
      uS = lum.uS(ix.p) # points towards source
      return lum.irr(p, uN) * shdr.brdf(uS, uV)
```

luminaire (light source)

(x,y,z)

irradiance

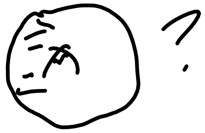
~E cos θ

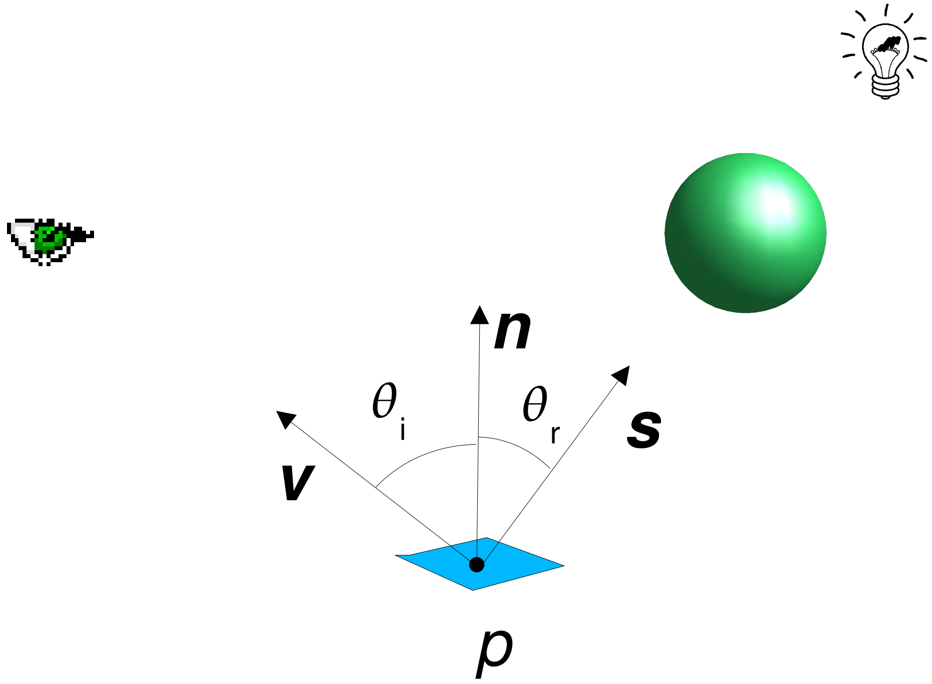
for diffuse

```
class Diffuse(Shader):
   ...
   def brdf(dff, uS, uV):
      return dff.kd
```

(RGB)

Shadows

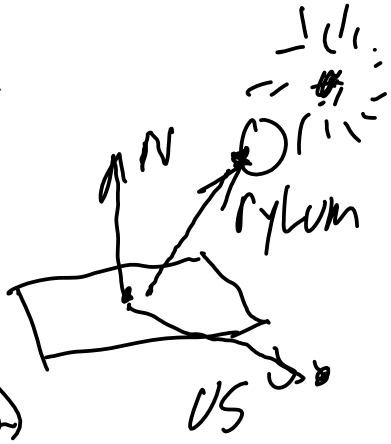




- What if there's an object between the light source and ***p***?
- Is there some tool we could use to find out if the path from ***p*** to the light source is blocked?

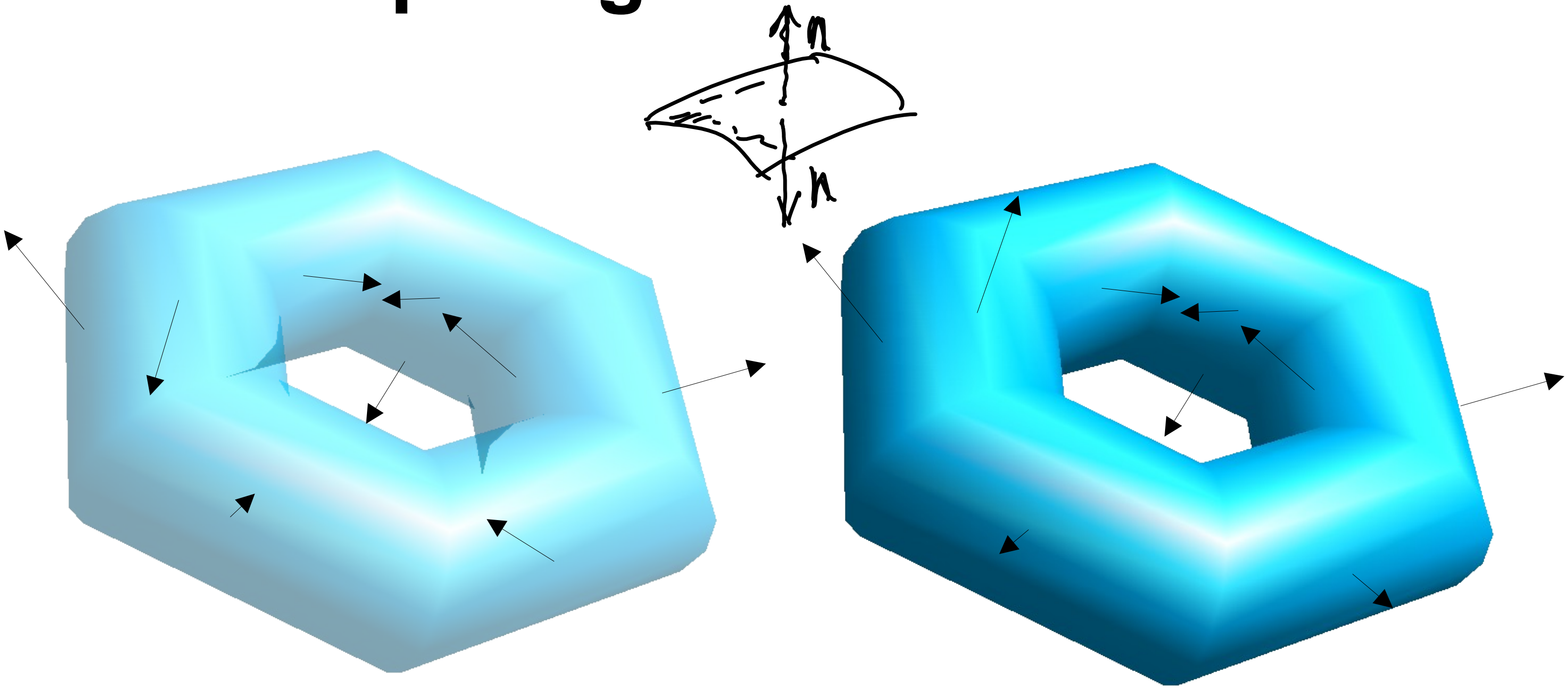
Material.illuminate():

```
class Material:                   material  intersection  incident ray
   ...                                                     scene
   def illuminate(mtl, ix, ryIn, scn):
      L = mtl.rdncIndirect(ix, ryIn, scn)
      uV = -ryIn.d.unit()
      for lum in scn.lums:
         uS = lum.uS(p)
         if uN.dot(uS) > 0: (lum is above horizon)
            ryLum = Ray(ix.p, uS, ry.nRefr, ry.depth)
            ixLum = scn.tgt.ixFirst(ryLum) # better: ixAny?
            if ixLum == None \(complete miss)
                  or lum.isCloser(ixLum.p, ix.p):
               L += mtl.rdncDirect(uV, ix.uN(), lum)
      return L
```



"Any" means you can look for any intersection, not just the closest ("First"). You can stop at the first intersection you find.

Computing Surface Normals

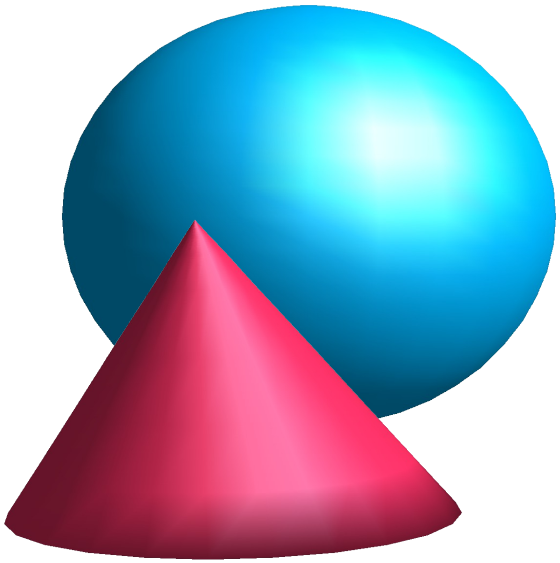


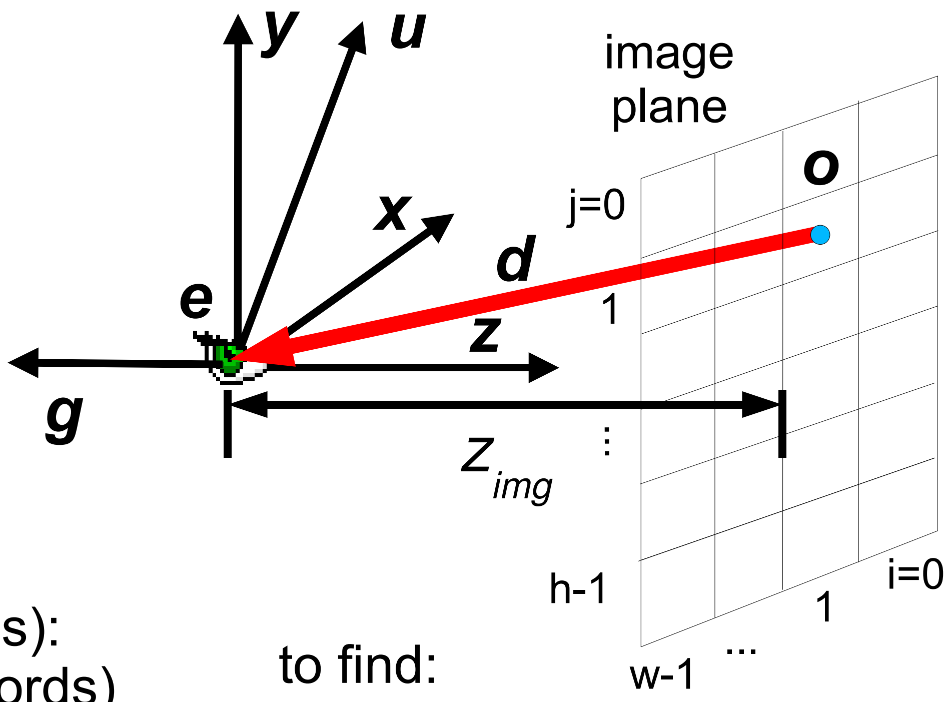
ambiguously-facing

outward-facing

- Normals have a sign ambiguity: both ***n*** and ***-n*** are perpendicular to the surface. (This is true for all surfaces.)
- If we adopt the convention that all surfaces are closed and ***n*** always faces *outward*, our computations get easier.

Viewing Geometry





given (in scene coordinates):

- ***e***: eye position (world coords)
- ***g***: "gaze" direction
- ***u***: up vector

and also

- *w* and *h*: image dimensions
- *i* and *j*: pixel indices

to find:

- z_{img}: distance of image plane from origin
- ***o***: ray origin (in the pixel)
- ***d***: ray direction

Note the reversal of image plane and eye point compared to OpenGL. There's a reason for this we'll see in the "distribution raytracing" unit.

Viewing Coordinate Bases

These are the same (symbology notwithstanding) as they were in CptS 442/542:

$$\hat{\boldsymbol{z}} = -\frac{\boldsymbol{g}}{|\boldsymbol{g}|}$$

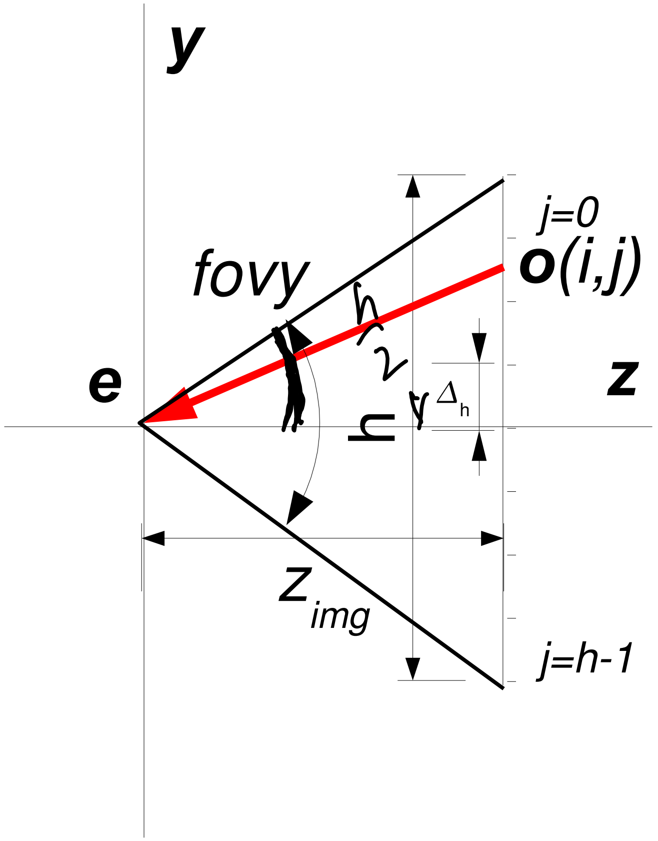
$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{u} \times \hat{\boldsymbol{z}}}{|\boldsymbol{u} \times \hat{\boldsymbol{z}}|}$$

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{z}} \times \hat{\boldsymbol{x}}$$

Combined with **e**, we have the transforms between world and viewing coordinates.
Is this a right-handed or left-handed coordinate system?

The Ray Direction I

Looking sideways at the viewplane...



We want to convert image coordinate *j* to a *y* lying between *-h/2* and *h/2*. Likewise for *i* to an *x* lying between *-w/2* and *w/2*.

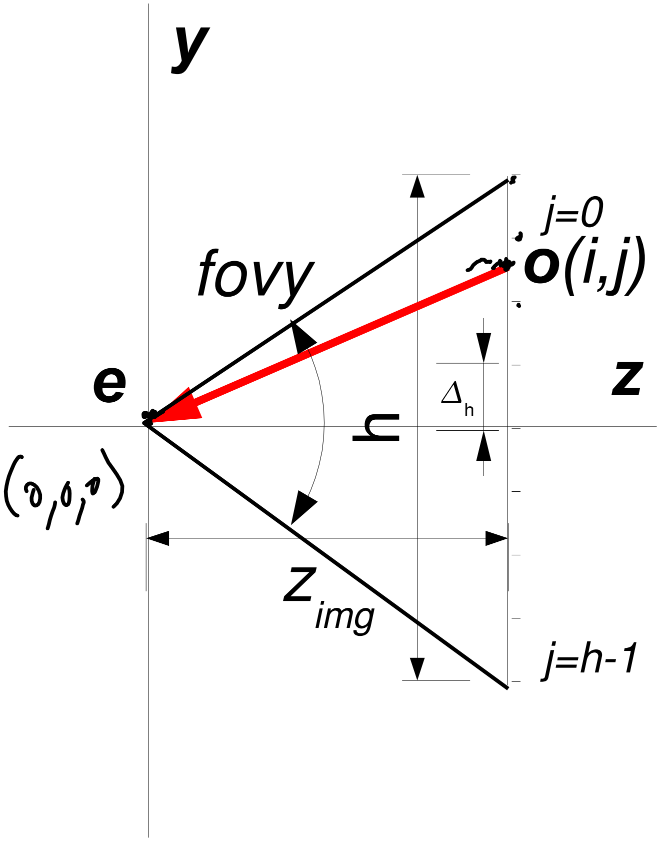
First, we need z_{img}.

By trig, we have:

$$\tan\left(\frac{fovy}{2}\right) = \frac{\frac{h}{2}}{z_{img}} \rightarrow z_{img} = \frac{h}{2\tan\left(\frac{fovy}{2}\right)}$$

and likewise for *fovx* and *w*.

The Ray Direction II



So the formula for ***o***(i,j) in camera coordinates is:

$$\boldsymbol{o}(i,j)=\begin{bmatrix}\frac{w}{2}-i\\\frac{h}{2}-j\\z_{img}\end{bmatrix}$$

and since **e** is the camera coordinate origin:

$$\boldsymbol{d}(i,j)=\boldsymbol{e}-\boldsymbol{o}(i,j)=-\boldsymbol{o}(i,j)$$

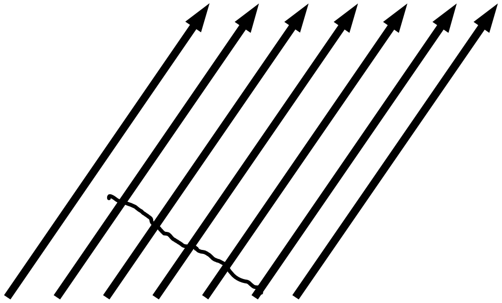
The Ray Direction III

Having ***d***(i,j) in camera coordinates, we need to turn it back into world coordinates, in which our objects are defined, so we multiply each component of d by its corresponding vector (this is actually a matrix transform):

$$\boldsymbol{d}'(i\,,j)=\left[\hat{\boldsymbol{x}}\quad \hat{\boldsymbol{y}}\quad \hat{\boldsymbol{z}}\right]\boldsymbol{d}=-\left(\frac{w}{2}-i\right)\hat{\boldsymbol{x}}-\left(\frac{h}{2}-j\right)\hat{\boldsymbol{y}}-\left(z_{img}\right)\hat{\boldsymbol{z}}$$

and we use ***e*** (in its original world coordinates) as the ray origin.

Directional Light Sources

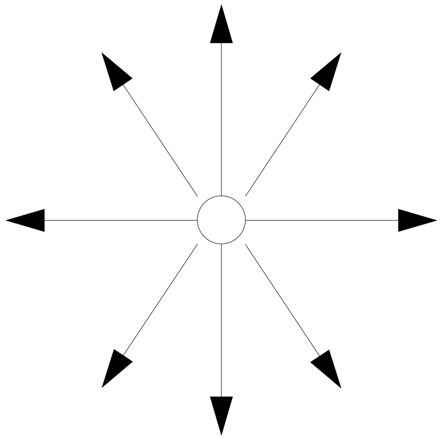


parameterized by

- direction ***l***
- irradiance $E_{\perp}$

both of which are constant

Point Light Sources



parameterized by

- position ***P***
- power Φ

power per unit area

irradiance is $$E(\boldsymbol{p}) = \frac{\boldsymbol{\Phi}}{4\pi\left|\boldsymbol{p}-\boldsymbol{P}\right|^2}$$

direction is $$\boldsymbol{l}(\boldsymbol{p}) = \frac{\boldsymbol{P}-\boldsymbol{p}}{\left|\boldsymbol{P}-\boldsymbol{p}\right|}$$

Ambient Light Sources

?

parameterized by

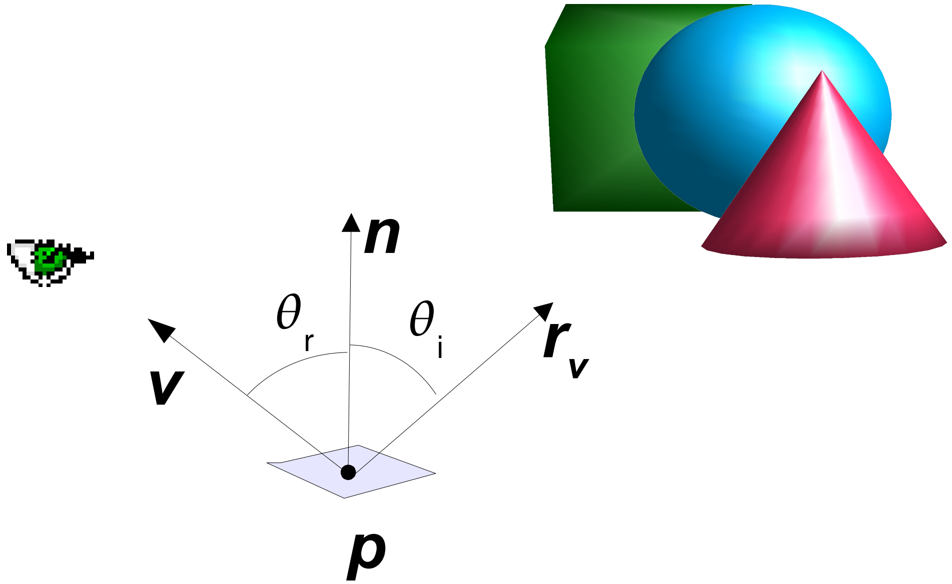
- ambient light L_a

Each object then gets a (multichannel) "ambient color".

This is an egregious hack!

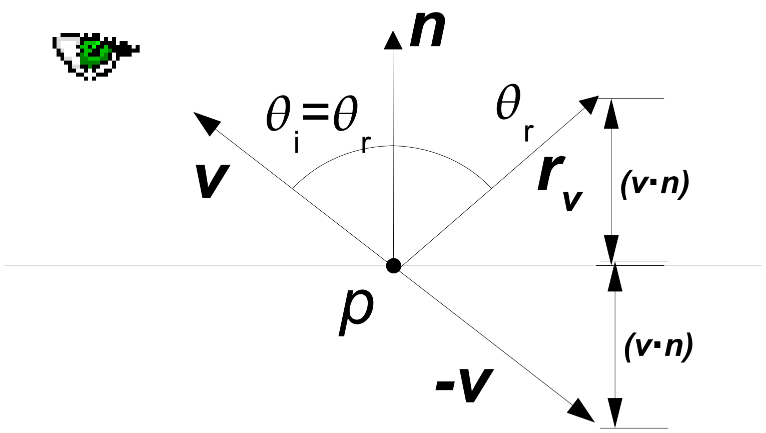
Non-Diffuse Materials

Materials: Smooth Metal



If the intersection ***p*** is smooth metal, we want a mirror effect, so we cast a mirror ray starting at ***p*** in the reflected direction $\boldsymbol{r_v}$ and incorporate the color we get for that ray into the color at ***p***. We allow for less than 100% reflectivity -- no mirror is perfect!

The Reflection Vector



The reflection vector is given by:

$$\boldsymbol{r}_{\boldsymbol{v}} = -\boldsymbol{v} + 2(\boldsymbol{v}\cdot\boldsymbol{n})\boldsymbol{n}$$

But we can also use the ray direction ***d*** (∝ -***v***) instead of ***v***:

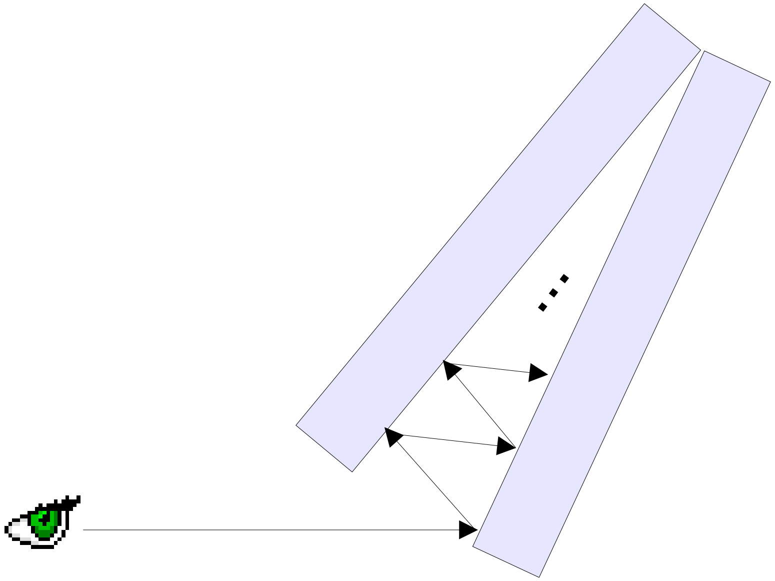
$$\boldsymbol{r}_{\boldsymbol{v}} = \boldsymbol{d} - 2(\boldsymbol{d}\cdot\boldsymbol{n})\boldsymbol{n}$$

Reflector.rdncIndirect()

```
class Reflector(Material):
   ...
   def rdncIndirect(mtl, ix, ryIn, scn):
      uV = -ryIn.d.unit()
      ry = Ray(p, uV.reflect(ix.uN()),
               ryIn.nRefr, depth+1)
      return mtl.kr * scn.trace(ry)
```

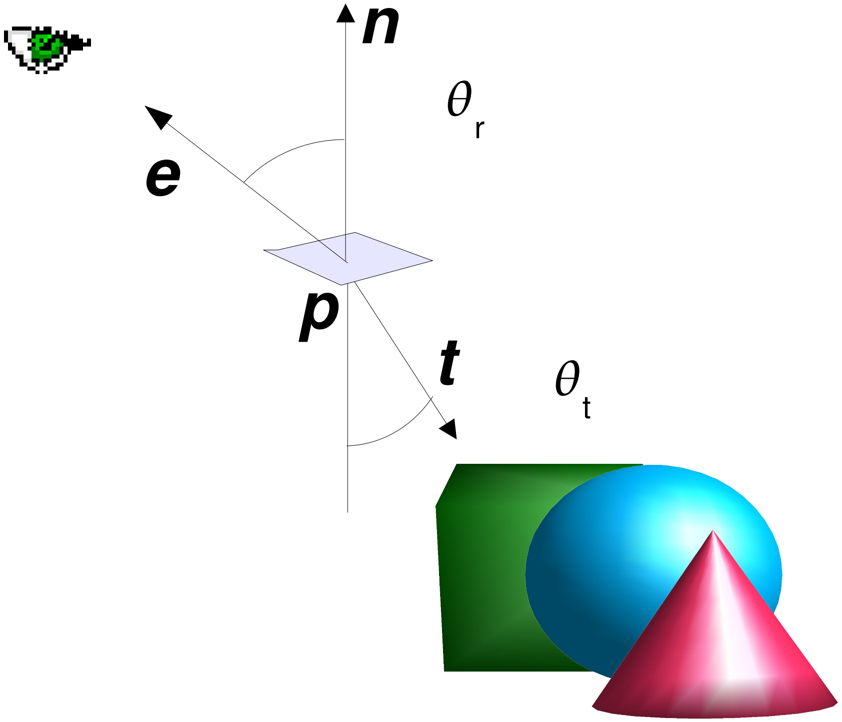
What `depth` is For

Suppose we have two perfect mirrors placed like this:



The depth parameter is one way to limit the number of reflections. There are more accurate possibilities, but this is the easiest. Note, however, that this is one of the shortcomings of raytracing.

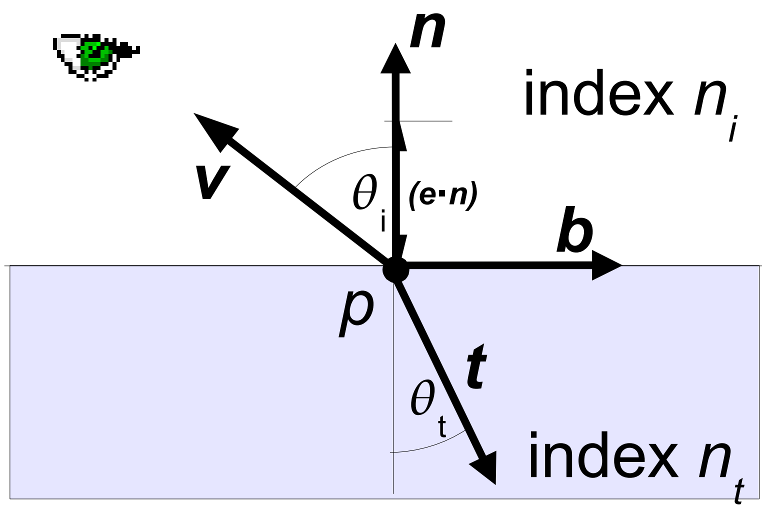
Materials: Dielectrics



If the surface at intersection ***p*** is dielectric ((semi-)transparent and light-refracting, like glass or water), we want to show refraction, so we cast a ray starting at ***p*** in the transmitted refracted direction ***t*** and incorporate the color we get for that ray into the color at ***p***. This may be combined with reflection.

Snell's Law

$$n_t \sin\theta_t = n_i \sin\theta_i$$



Where n_t and n_i are indices of refraction of the two media.

Rewrite as

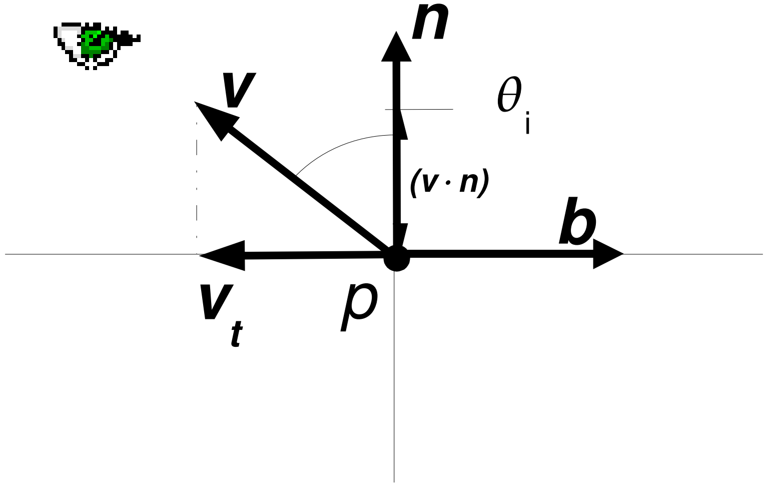
$$\sin\theta_t = \frac{n_i}{n_t}\sin\theta_i \equiv \eta\sqrt{١-(\boldsymbol{v}\cdot\boldsymbol{n})^{٢}}$$

If we had the tangent (unit) vector ***b***, we could write:

$$\boldsymbol{t} = \sin\theta_t\,\boldsymbol{b} - \cos\theta_t\,\boldsymbol{n}$$

$$\Rightarrow \boldsymbol{t} = \eta\sqrt{1-(\boldsymbol{v}\cdot\boldsymbol{n})^2}\,\boldsymbol{b} - \sqrt{1-\left(\eta\sqrt{1-(\boldsymbol{v}\cdot\boldsymbol{n})^2}\right)^2}\,\boldsymbol{n}$$

Computing the Tangent Vector



v has both a normal (parallel to ***n***) and a tangential (perpendicular to ***n***) component. We extract the tangential component

$$\boldsymbol{v}_t = \boldsymbol{v} - (\boldsymbol{n} \cdot \boldsymbol{v})\,\boldsymbol{n}$$

b is this vector, negated and normalized:

$$\boldsymbol{b} = \frac{-\boldsymbol{v}_t}{|\boldsymbol{v}_t|} = \frac{(\boldsymbol{n}\cdot\boldsymbol{v})\,\boldsymbol{n} - \boldsymbol{v}}{\sqrt{\left(\boldsymbol{v} - (\boldsymbol{n}\cdot\boldsymbol{v})\,\boldsymbol{n}\right)\cdot\left(\boldsymbol{v} - (\boldsymbol{n}\cdot\boldsymbol{v})\,\boldsymbol{n}\right)}} = \frac{(\boldsymbol{n}\cdot\boldsymbol{v})\,\boldsymbol{n} - \boldsymbol{v}}{\sqrt{(\boldsymbol{v}\cdot\boldsymbol{v}) - 2(\boldsymbol{n}\cdot\boldsymbol{v})^2 + (\boldsymbol{n}\cdot\boldsymbol{v})^2(\boldsymbol{n}\cdot\boldsymbol{n})}}$$

$$= \frac{(\boldsymbol{n}\cdot\boldsymbol{v})\,\boldsymbol{n} - \boldsymbol{v}}{\sqrt{1 - (\boldsymbol{n}\cdot\boldsymbol{v})^2}} = \frac{\cos\theta_i\,\boldsymbol{n} - \boldsymbol{v}}{\sin\theta_i}$$

The Transmitted Ray Direction

Substituting

$$\boldsymbol{b}=\frac{(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}-\boldsymbol{v}}{\sqrt{1-(\boldsymbol{n}\cdot\boldsymbol{v})^2}}$$

into

$$\boldsymbol{t}=\eta\sqrt{1-(\boldsymbol{n}\cdot\boldsymbol{v})^2}\,\boldsymbol{b}-\sqrt{1-\left(\eta\sqrt{1-(\boldsymbol{n}\cdot\boldsymbol{v})^2}\right)^2}\,\boldsymbol{n}$$

we get (whew!)

$$\boldsymbol{t}=\eta\left((\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}-\boldsymbol{v}\right)-\sqrt{1-\left(\eta\sqrt{1-(\boldsymbol{n}\cdot\boldsymbol{v})^2}\right)^2}\,\boldsymbol{n}$$

What could go wrong?

Indices of Refraction

Medium	Index of Refraction
Methylene Iodide	1.74
Glass, dense flint	1.66
Carbon bisulfide	1.63
Sodium chloride	1.53
Glass, crown	1.52
Fused Quartz	1.46
Ethyl alcohol	1.36
Water	1.33
Air (1 atm, 20° C)	1.0003
Vacuum	1.00

The Fresnel Term

When light reflects off a dielectric, its reflectivity has an angular dependence.

$$R(\boldsymbol{v}, \boldsymbol{s}, \eta) = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left\{1+\frac{(c(g+c)-1)^2}{(c(g-c)-1)^2}\right\}$$

where

$$c \equiv \boldsymbol{v}\cdot\boldsymbol{h}$$
$$h \equiv \widehat{\boldsymbol{v}+\boldsymbol{s}}$$
$$g^2 \equiv \eta^2 + c^2 - 1$$

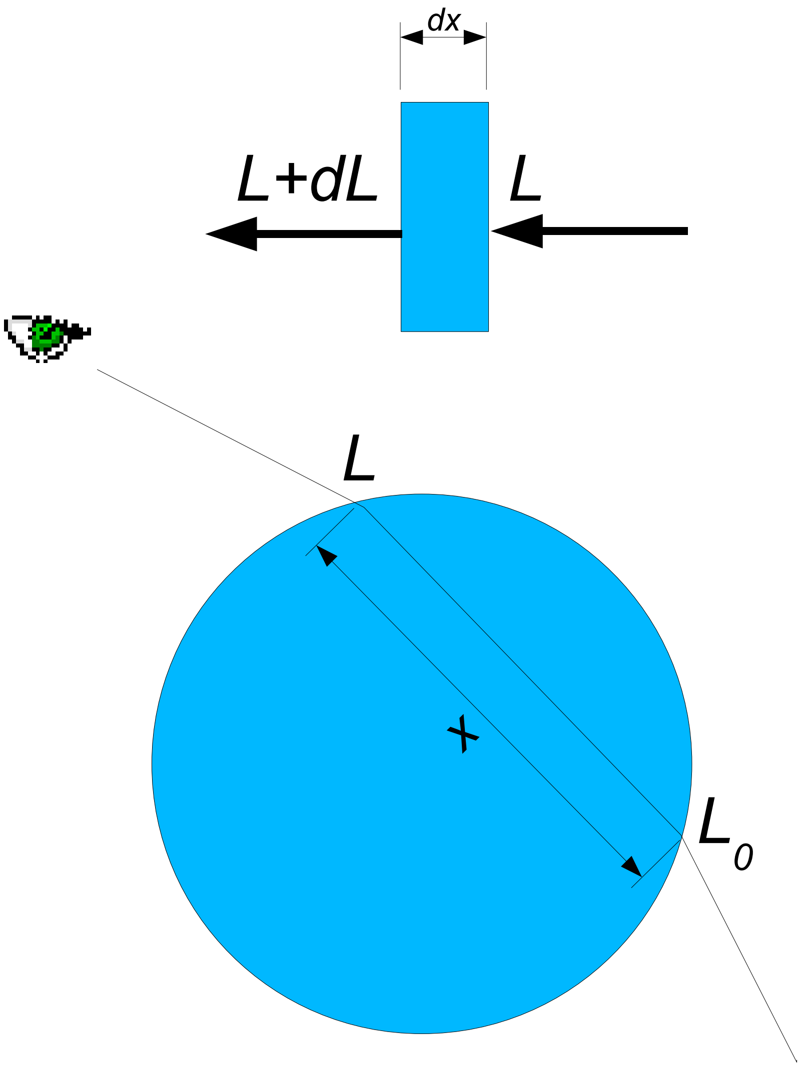
This is time-consuming. A reasonable approximation (Shirley) is

$$R(\boldsymbol{v}, \boldsymbol{n}, \eta) = R_0 + (1-R_0)(1-(\boldsymbol{v}\cdot\boldsymbol{n}))^5 \qquad R_0 \equiv \left(\frac{\eta^{-1}-1}{\eta^{-1}+1}\right)^2$$

Dielectric.rdncIndirect()

```
class Dielectric(Material):
   ...
   def rdncIndirect(dlct, ix, ryIn, scn):
      uV = -ryIn.d.unit()
      ryRefl = Ray(ix.p, uV.reflect(ix.uN), ryIn.nRefr,
                   ryIn.depth+1)
      ryRefr = Ray(ix.p, uV.refract(ix.uN, dlct.nRefr),
                   dlct.nRefr, ryIn.depth+1))
      f = dlct.fresnel(uV, ix.uN, ryIn.nRefr)
      if f == 1: # total internal reflection
         rdnc = scn.trace(ryRefl)
      else:
         rdnc = (f * scn.trace(ryRefr)
            + (1 - f) * scn.trace(ryRefl))
      return dlct.kAbs * rdnc
```

Attenuating Media



Beer's Law: Light passing through an absorbing medium is diminished by a fractional amount proportional to the thickness of the medium.

$$dL = -C\,L\,dx$$

$$\Rightarrow \frac{dL}{dx} = -C\,L \Rightarrow L = L_{.}e^{-Cx}$$

So e^{-Cx} is the *attenuation factor*.

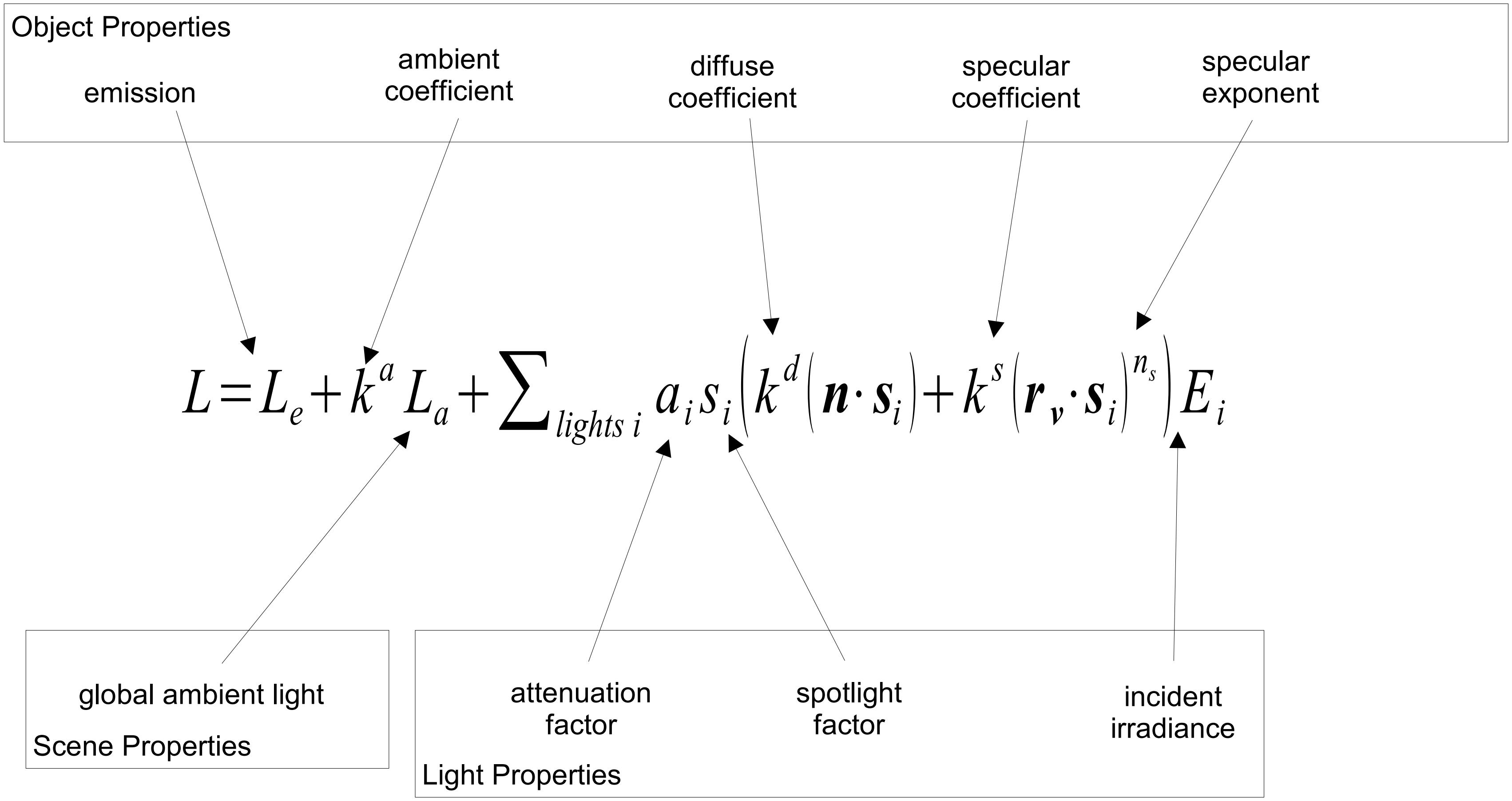
This is the same idea as fog in OpenGL. How would you incorporate this into the schema?

Materials: Polished Surfaces

This is an alternative shader developed by Shirley:

$$L = R\left(1 - \left(1 - (\boldsymbol{n}\cdot\boldsymbol{v})\right)^5\right)\left(1 - \left(1 - (\boldsymbol{n}\cdot\boldsymbol{s})\right)^5\right)E$$

A Conventional Ray Tracing Shader



Object Properties

emission

ambient coefficient

diffuse coefficient

specular coefficient

specular exponent

$$L = L_e + k^a L_a + \sum_{lights\ i} a_i s_i \left(k^d \left(\boldsymbol{n} \cdot \boldsymbol{s}_i \right) + k^s \left(\boldsymbol{r}_{\boldsymbol{v}} \cdot \boldsymbol{s}_i \right)^{n_s} \right) E_i$$

global ambient light

Scene Properties

attenuation factor

spotlight factor

incident irradiance

Light Properties

Negative dot products are ignored.

DiffusePhongAmbient.rdncDirect()

This is the traditional reflectance model used in OpenGL. Note that we must divide the irradiance by the incident cosine to recover the original (non-physically plausible) formula.

```
class DiffusePhongAmbient(Material):
   ...
   def rdncDirect(dpa, ry, ix, lum):
      uV = -ry.d.unit()
      uN = ix.uN()
      uS = lum.uS(ix.p) # points towards source
      uRv = uV.reflect(uN)
      # officially, "irradiance" includes S.N factor
      irr = lum.irr(p, uN)
      # but the specular term (non-physically) ignores it
      irrSpec = irr / uS.dot(uN) # so we remove it
      return dpa.kd * irr \
         + dpa.ks * (uS.dot(uRv)**dpa.expo) * irrSpec \
         + dpa.ka * lum.irrAmb
```

So What's Wrong with Ray Tracing (so far)?

- It takes a long time.
- We're always using rays to sample.
- We don't handle interreflecting diffuse surfaces.
- Our light sources are limited to point or unidirectional.