- ASG 6: redoing Lab 5 &
  ASG 2 in Obj-C
  - camera_t object
  - find_closest function
- ASG 7: puts ASG 6,
  Labs 10, 11 together
  to give you a ray caster
  in Obj-C (like ASG 3
  before in C)

- ASG 8 is the big one
  - extends ASG 7:
    - introduces sphere
    - rendering equation
      (ambient, diffuse, specular
      components)
    - ray reflection
      (recursion)
- Note: Lab 12's doubly-linked
  list not used, but can be

- first, how to calculate ray/sphere intersection

- in cbox.txt (model file):

```
sphere left
{
  material chrome
  center 1.25 .75 -4
  radius .9
}
```

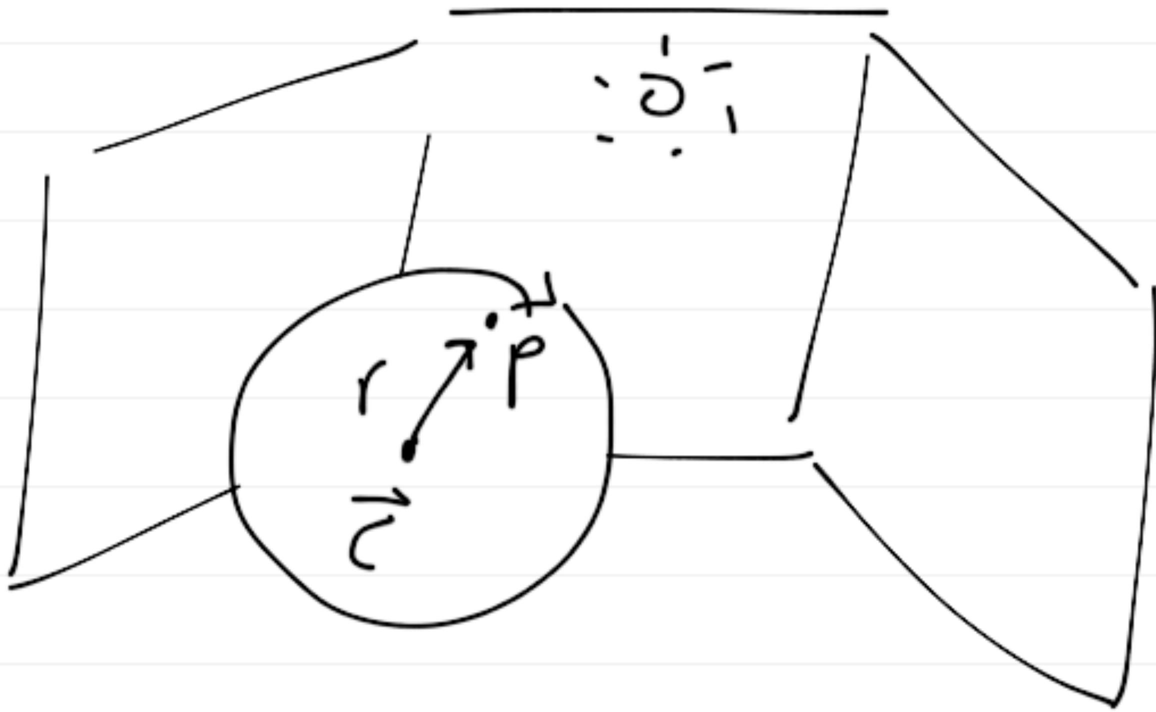
```
material chrome
{
  ambient 3 3 3
  diffuse .1 .1 .1
  specular .9 .9 .9
}

light top
{
  location 2.56 3.80 -1.5
  emissivity 1 1 1
}
```

} very shiny surface

} any number of these, like material or object

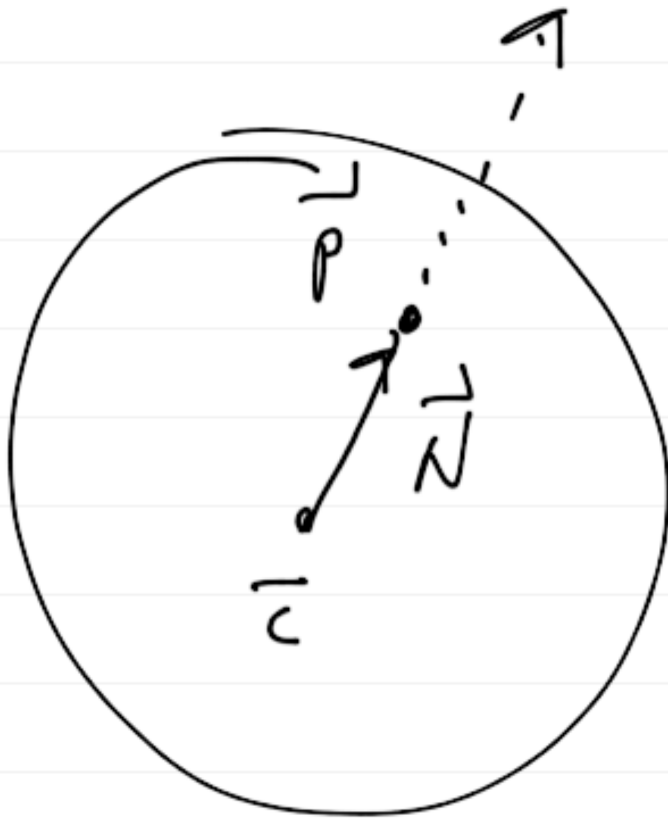
- world:



$\vec{c}$: sphere center (given)

$\vec{p}$: intersection point (of ray & sphere: what we calculate)

r: radius (given)



$\vec{N}$: normal at surface point (direction of surface at $\vec{P}$)

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

- take difference between $\vec{P}$ and $\vec{C}$ and normalize

- Also, given $\vec{c}$, $\vec{N}$,

$$\vec{p} = \vec{c} + r\vec{N}$$

any point on sphere is center + Normal scaled by radius

parametric equation of sphere

- just like ray is defined as:

$$ray = \vec{pos} + t\,\vec{dir}, t \geq 0$$

position (a point) + scaled direction

- another way to represent sphere mathematically:

$$(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) = r^2$$

any point on sphere surface (distance from center is r)

- express $\vec{p}$ parametrically,

$$\vec{p}(t) = \vec{pos} + t\,\vec{dir}$$

and substitute into this

- we get:

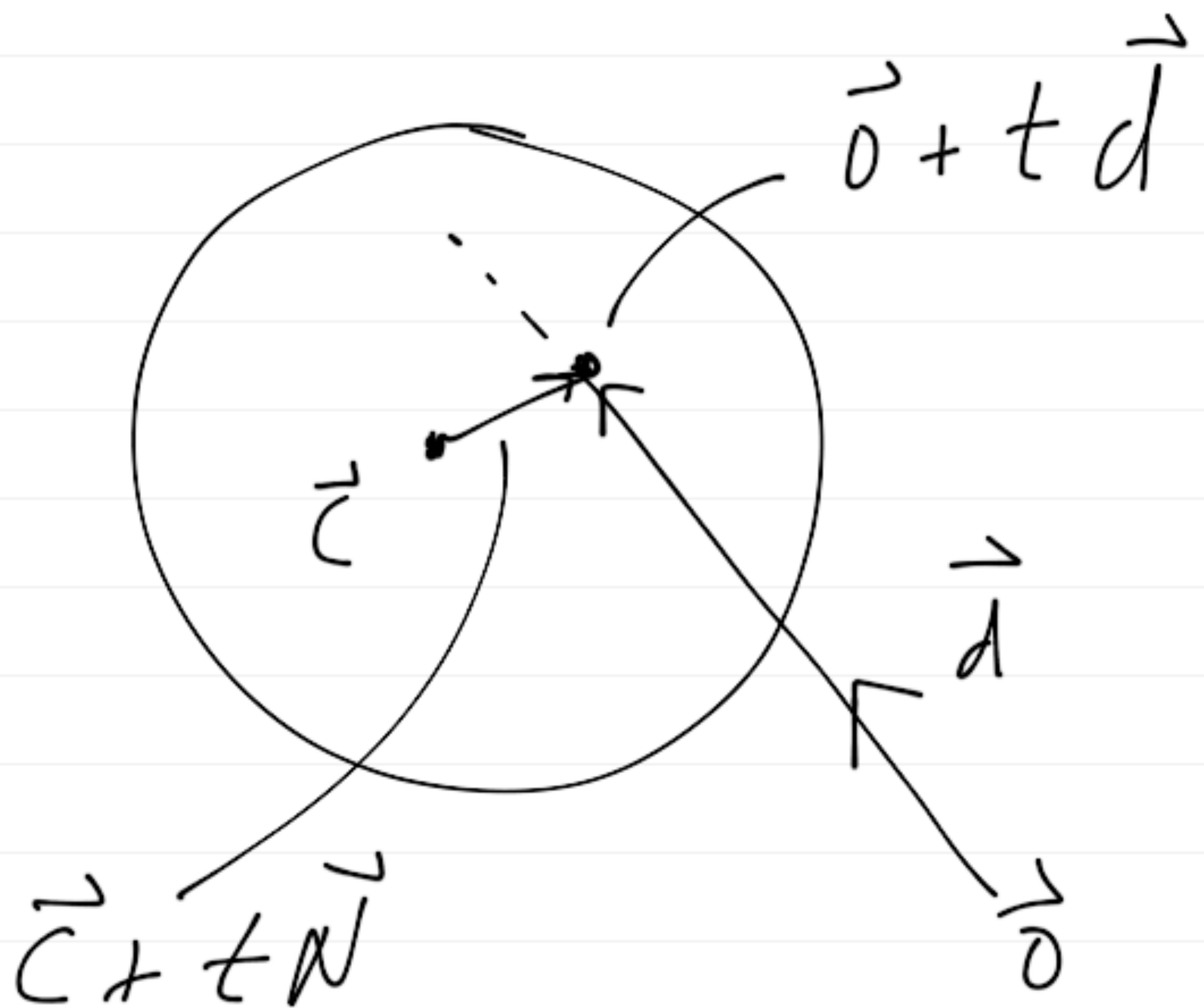
$$(\vec{pos} + t\,\vec{dir} - \vec{c}) \cdot (\vec{pos} + t\,\vec{dir} - \vec{c}) = r^2$$

(in the code I use $\vec{o}$ for ray origin and $\vec{d}$ for direction)

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

vectors (known, ray) — $\vec{o}$, $\vec{d}$

sphere center, r (known) — $\vec{c}$, r



- solve for t:

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

- mutiplying out and rearranging in terms of t,

$$(\vec{d} \cdot \vec{d})t^2 + [2(\vec{o} - \vec{c}) \cdot \vec{d}]t$$

$$+ (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2 = \emptyset$$

gives us $At^2 + Bt + C = 0$

where

$A = \vec{d} \cdot \vec{d}$

$B = 2(\vec{o} - \vec{c}) \cdot \vec{d}$

$C = (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2$

- which is basic quadratic formula:

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

- we have intersection between ray & sphere only if

$$B - 4AC > 0.0$$

which you calculate once and test for inequality

- in code, use vec_t class for convenience:

```
-(double) hits:
    (vec_t *)_pos:
    (vec_t *)_dir:
    (vec_t **)_hit:
    (vec_t **)_N
{
    :
}
```

sphere_t subscribes to intersecting protocol

```
vect *pc;
double a=1.0, b, c;
double dist=-1.0;

pc=[[_pos minus: center]
              retain];
```

↑ $(\vec{o} - \vec{c})$

retain ← keep in mem.

do not normalize!!

```
a = [_dir dot:_dir];
```

$(\vec{d} \cdot \vec{d})$

```
b = 2.0 * [pc dot:_dir];
```

$(2(\vec{o} - \vec{c}) \cdot \vec{d})$

```
c = [pc dot: pc] - radius * radius;
```

$((\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2)$

```
d = b * b - 4.0 * a * c;
```

```
if (d > 0.0) {
  t = (-b - sqrt(d)) / (2.0 * a);
  [last_hit autorelease];
  last_hit = [[_pos plus:
      [_dir scale: t]] retain];
```

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

```
[*_hit autorelease];

*_hit = [last_hit copy];
```

```
[normal autorelease];
normal =
  [[[last_hit minus: center]
  scale: 1.0/radius] retain];
```

$$\left(\vec{N} = \frac{1}{r}(\vec{p} - \vec{c})\right)$$

```
 [*_N autorelease];
 *_N = [normal copy];
} // end if (d > 0.0)
return t;
```

- hit point and normal are used to calculate color at surface

(in function ray_trace)

- recall main loop ⟶

```
for(y=h-1; y>=0; y--) {
 for(x=0; x<w; x++) {
   wx= x/(w-1) * ww;
   wy= y/(h-1) * wh;

   [pix set: wx: wy: wz];

   [dir autorelease];
   dir = [[pix minus: pos] retain];
   [dir autorelease];
   dir = [[dir unit] retain];
```

```
[color autorelease];

color = [[
ray_trace(model, pos, dir, 0.0)
    scale: 255.0] retain];
```

ray_trace returns a ptr to pixel object (pixel_t *) with r, g, b values ∈ [0, 1] (new "feature")

```
// after ray-trace call
imgloc = img + y*w + x;
for (i=0; i<3; i++)
  (*imgloc)[i]=[color get: i];
 } // for
} // for

// write image to file
     .
     .
     .
```

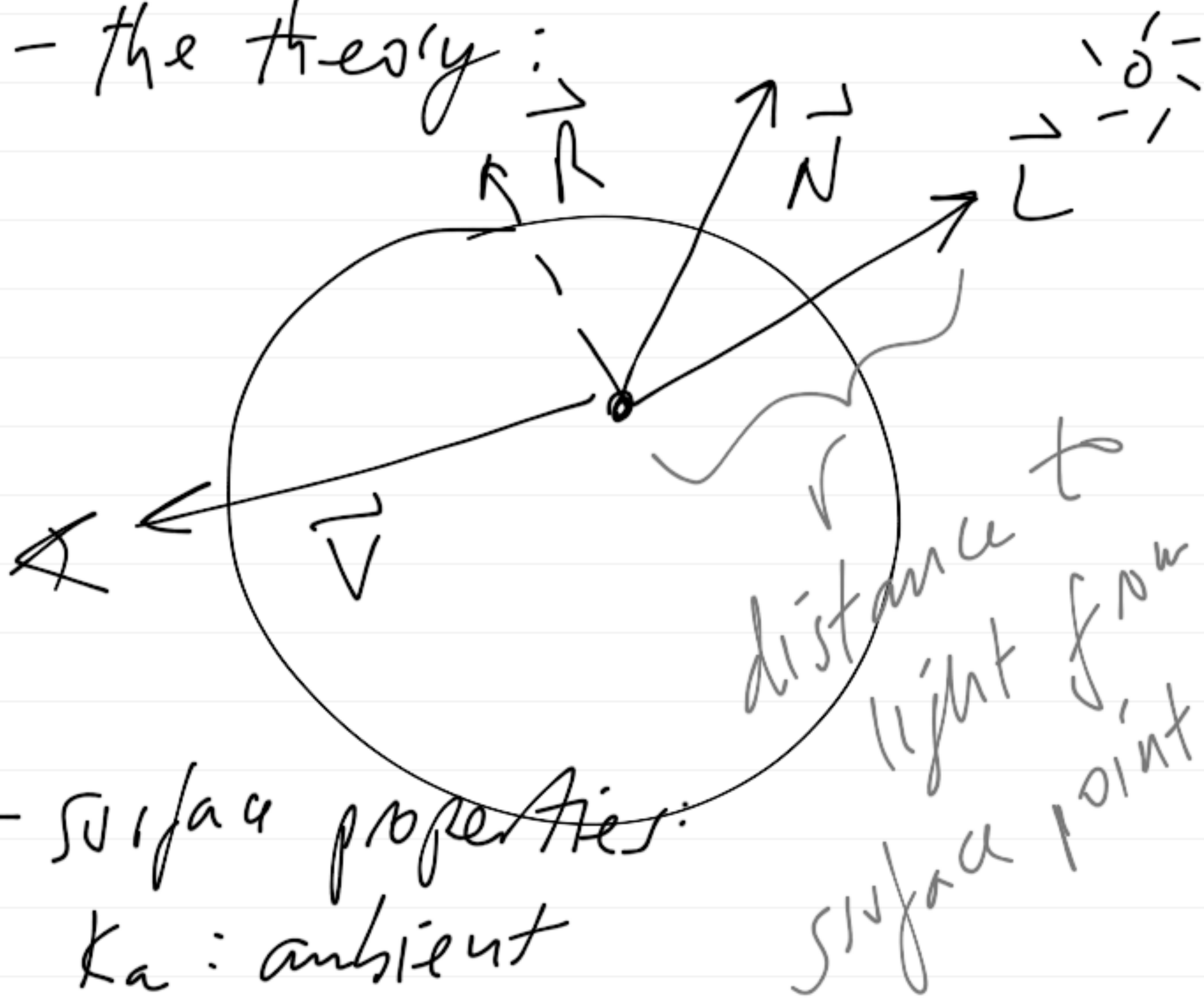
- in ray_trace routine, compute color at intersection point

- need light location & color

```
light name
{
    position  x y z        (vec_t)
    emissivity  r g b      (pixel_t)
}
```

} in model file

- the theory:



- surface properties:

  K_a : ambient

  K_d : diffuse

  K_s : specular

$$I = \frac{K_a I_a}{R} + \sum_{l} \frac{I_l}{r}\left(K_d (N \cdot L) + K_s (R \cdot V)^n\right)$$

$$I = \frac{K_a I_a}{R} + \sum_{l} \frac{I_l}{r} \left(K_d (N \cdot L) + K_s (R \cdot V)^n \right)$$

R ↑ ambient term, R is the ray distance (what you already have in ray caster)

$\sum_l$ ↑ for each light

I_l ↑ light color

r: div by dist. to light

$K_d (N \cdot L)$: diffuse term

$K_s (R \cdot V)^n$: specular term

N: surface normal

L: dir to light (from surf. point)

V: dir. to camera

R: L reflected about N

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

n: shininess, e.g. 1.0 or 32.0

$$I = \frac{K_a I_a}{R} + \sum_{l} \frac{I_l}{r}\left(K_d (N \cdot L) + K_s (R \cdot V)^n\right)$$

is called the Phong illumination model

- an estimate of physical light transport
- basically what OpenGL uses for direct illumination (local)

- ray tracer calculates reflections, giving global illumination, something GL doesn't do

- in ray_trace:

```
if (!(obj = [model find_closest:
    pos: dir: &dis: &hit: &N]) ||
    raydist > MAX_DIST)
    return NULL;
if (dis > 0) {
    raydist += dis;
    // calculate color
}
return [color autorelease];
}
```

update this ray's length

- calculate color:

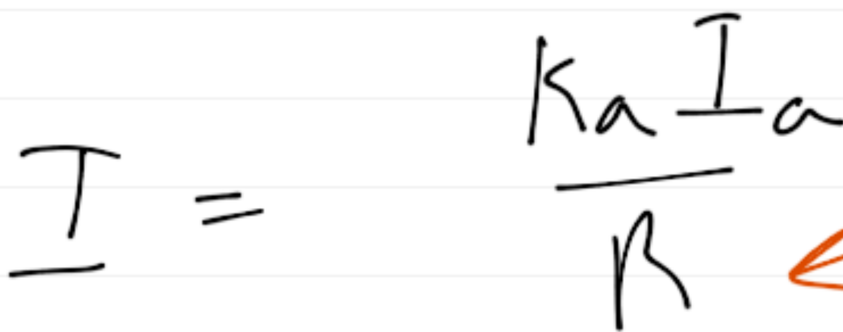
```
pixel_t *ambient = nil;
pixel_t *diffuse = nil;
pixel_t *specular = nil;

ambient = [[obj mat] getamb];
diffuse = [[obj mat] getdiff];
specular = [[obj mat] getspec];
```

```
[color autorelease];
color = [[ambient copy] retain];
```

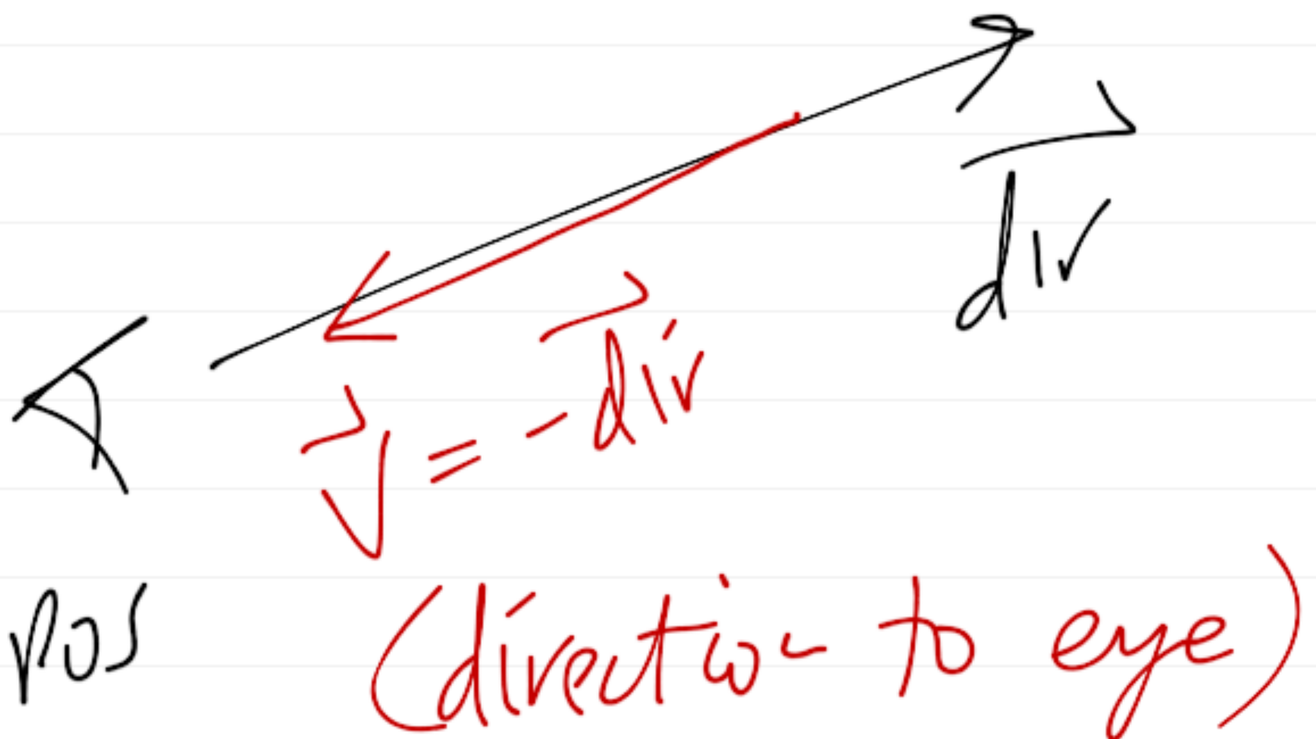
$$I = K_a I_a$$

```
[color autorelease];
color = [[color scale:1.0/raydist]
                retain];
```



- now the rest of the rendering equation:

```
[v autorelease];
v = [[dir scale:-1.0] retain]
```



```
if([diffuse nonzero]) {
//iterate thru lights in scene
for([[model lgts] reset];
 ![[model lgts] end];
 [[model lgts] next]) {

   lgt = [[model lgts] data];
   (pointer to light_t object)
```

```
[L autorelease];
L = [[[light getlocation] minus:
            hit] retain]

r = [L len];

[L autorelease];
L = [[L unit] retain];
```

(light direction and distance)

```
ndotl = [N dot: L];
```

(angle with light) →

```
if (0.0 < ndotl && ndotl < 1.0){
  [I_d autorelease];
  I_d =
[[[lgt getcolor] scale:
     1.0/r * ndotl] retain];
```

$$I_d = \frac{I_l}{r}(N \cdot L)$$

- now add into surface
  color

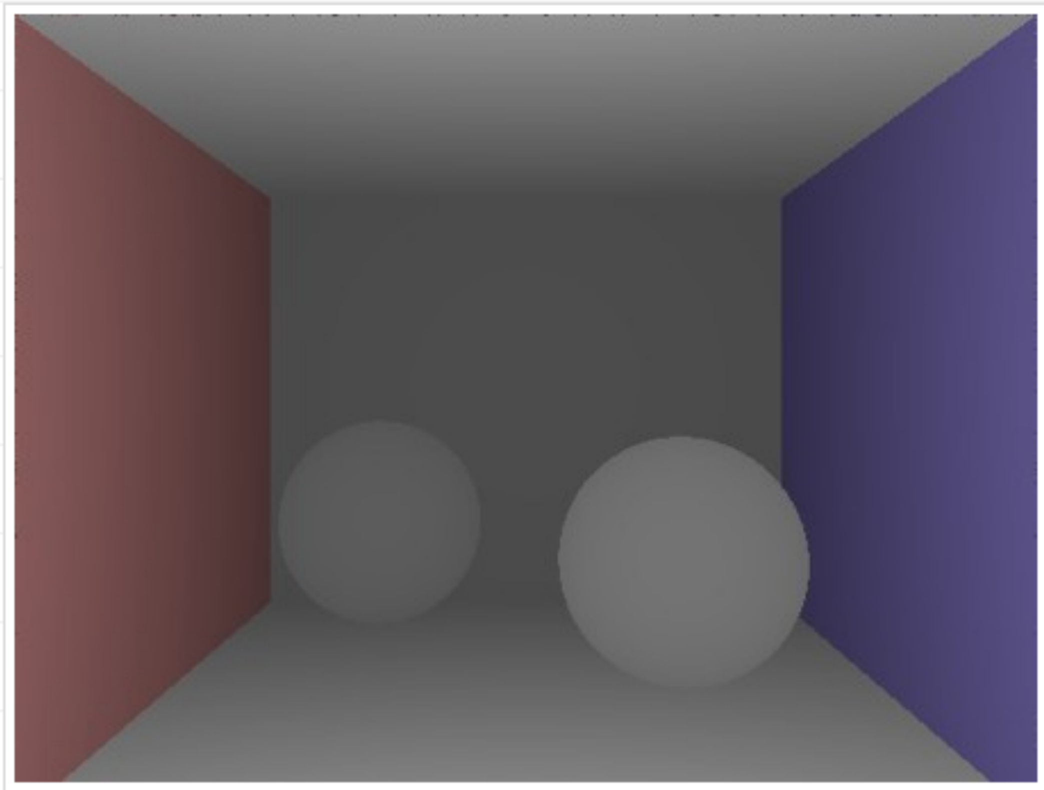
```
[color autorelease];
color = [[color add:
   [I_d mul: diffuse]] retain];
} // if (0 < N.L < 1)
} // for lgts loop
[color clamp: 0.0 : 1.0];
} // if diffuse nonzero
```

- pixel <u>clamp</u> function:

```
for each of r, g, b:
    if drgb[i] < min
        drgb[i] = min
    if drgb[i] > max
        drgb[i] = max
```

- clamp values so we don't go over 1.0 or below 0.0

- the above code segment should produce this image:



giving matte surface appearance

⇒ light arriving at surface scattered equally in all directions (diffuse scattering)

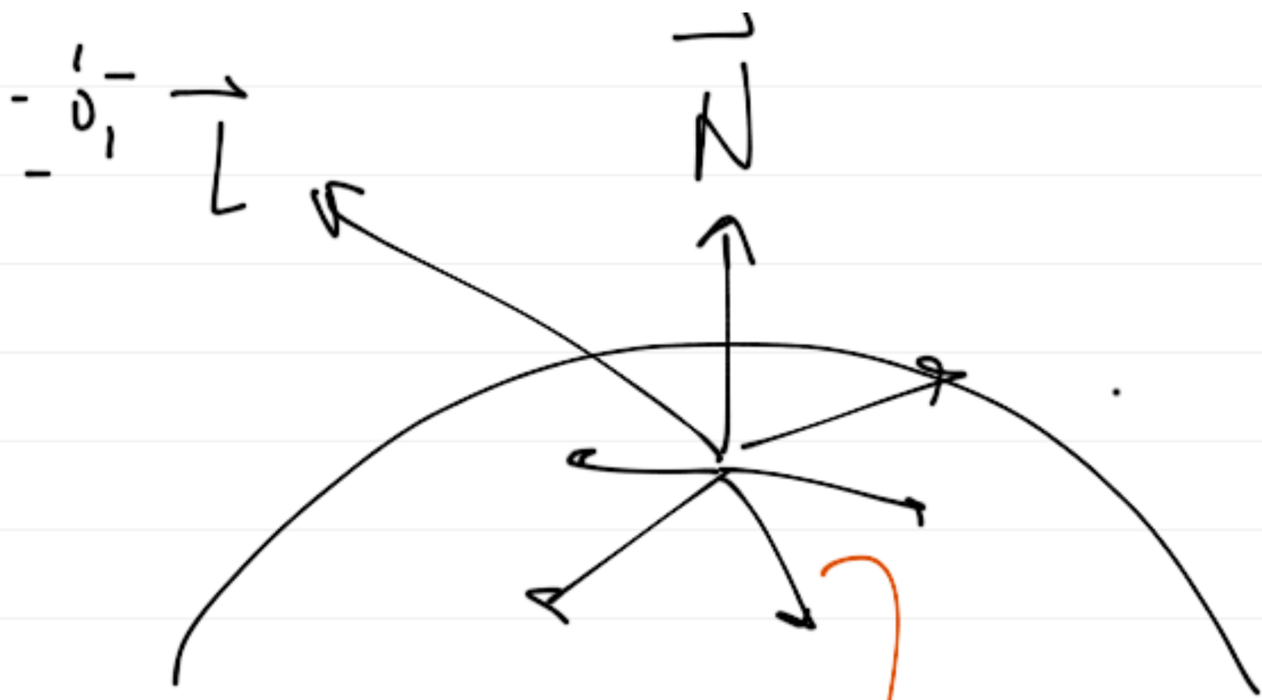
- diffuse scattering based on Lambert's Law modeling a perfect diffuse reflector

$$I \propto L \cdot N$$

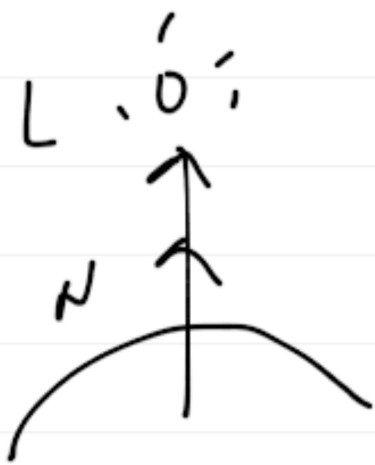
light intensity (→ I)

proportional to (→ $\propto$)

cosine of angle between light dir & surf. N (→ $L \cdot N$)



same reflected intensity in all directions



$\theta = 0°$

$\cos(\theta) = 1 = N \cdot L$

full illumination

$\theta = 90°$

$\cos(\theta) = 0 = N \cdot L$

no reflection

- if ($0 \leq N \cdot L \leq 1$)

    surface visible to light

  else

    surface hidden from light (only gets ambient contribution)

- Lambertian lighting model:

$$I = I_l \, k_d \, (N \cdot L)$$

diffuse surface prop.

with ambient term:

$$I = \frac{I_a k_a}{R} + \sum_l \frac{I_l}{r} k_d (N \cdot L)$$

- next time: specular component, reflection