Ray Tracer extensions:

- ray "objectification"
- vector reflect / refract functions
- transmission
- bounded planes
- depth of field
- shadow castings

ray objectification

- idea is to make a ray_t object
- ray_t contains data members: double dis
  and vec_t pos
  vec_t dir
  functions:

trace: (model_t *) Model

- I think that's all that's needed

- I have not implemented this in obj-c myself so I've not tested it

- point is: allows "spawning" of a ray →

```
ray_t * ray =
    [(ray_t *)[ray_t alloc]
    init: pos: dir ];
color = [ray trace: model];
[ray release];
```

- that's how I'd like to use it anyway

- note that the trace function would spawn its own reflection & refraction rays, e.g.,

```
ray_t * reflection
    [(ray_t *)[ray_t alloc]
    init: pos: dir ];
```

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

refcolor = [reflection trace: model];

[reflection release];

- same idea for transmission
  (of a ray through transparent object)

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

$$T = \frac{n_i}{n_t}(U - (U \cdot N)N) - \left(1 - \frac{n_i}{n_t}\right)^2 (1 - (U \cdot N)^2)N$$

$$T = \frac{n_i}{n_t}\left(U - (U \cdot N)N\right) - \left(1 - \frac{n_i}{n_t}\right)^2 \left(1 - (U \cdot N)^2\right)N$$

- want to provide both
  reflection (R) and
  refraction (T)
  constructors as part of
  vec_t object, so that

we can use:

vec_t *r = [div reflect : N];

and ⟶

Vector functions

```
vec_t *t =
  ndotd < 0 ?
    [dir refract: N: ior]:
    [dir reflect: [N: scale: -1.0]:
                  1.0/ior];

ray_t *reflection =
    [[ray_t] init: hit: r];

ray_t *refraction =
    [[ray_t] init: hit: t];
```

- what are ndotd and ior?

- ndotd is just N · dir
  (like N · L is ndotl)

- ior is index of refraction
  - from model file

material transparent

{

ambient 0 0 0

diffuse .1 .1 .1

specular .9 .9 .9

alpha .9 → indicates transparency

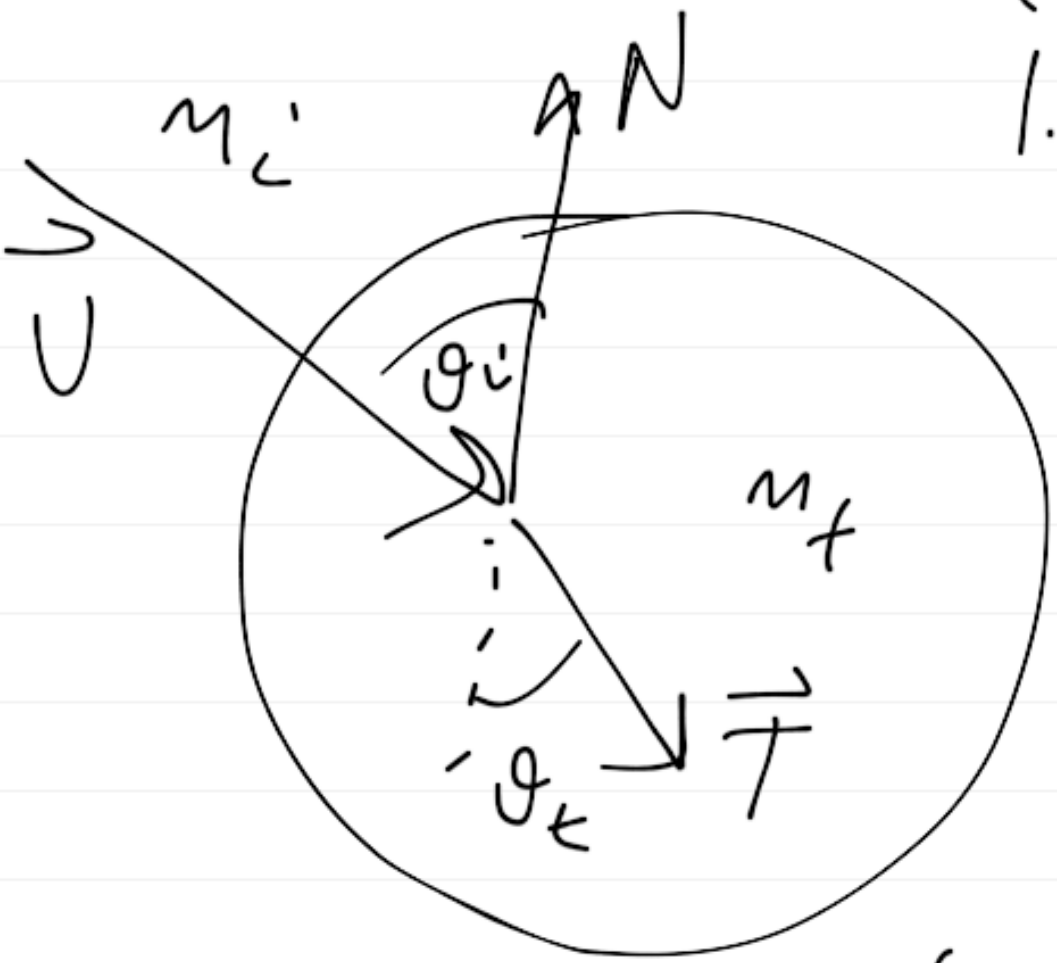
ior 1.83

}

↑ how much to refract (n_t)

transmission

n_i : air
1.000293

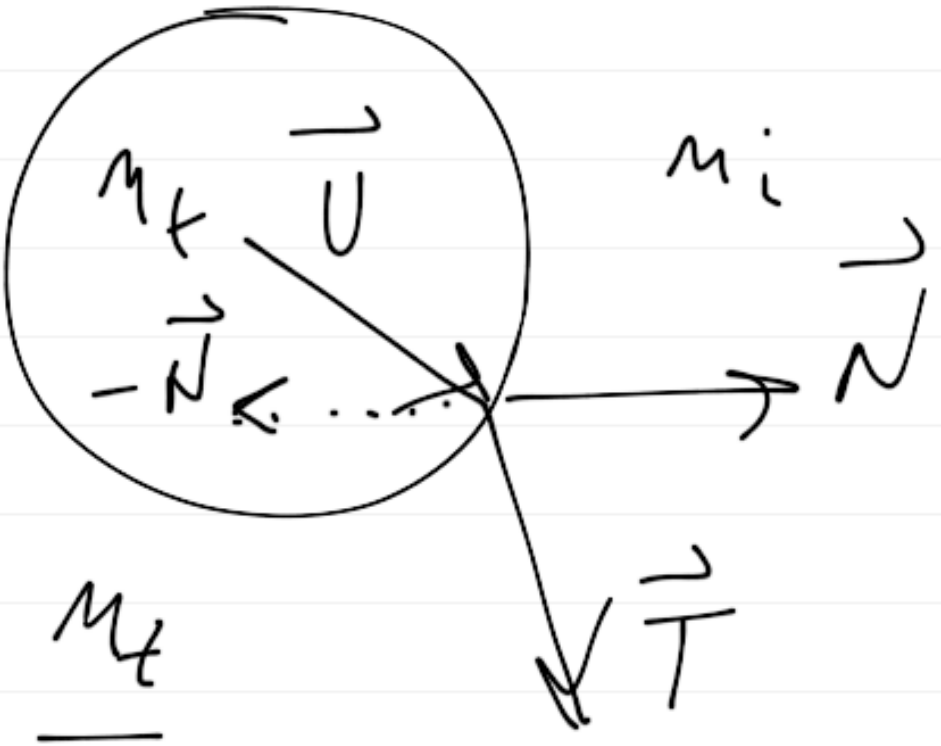


- ray going into sphere

Snell's Law says

$$\frac{n_i}{n_t} = \frac{\theta_i}{\theta_t}$$

ray leaving sphere



$$\frac{\sin\theta_t}{\sin\theta_i} = \frac{n_t}{n_i}$$

- in the first case want to call

[di refract: N: n_t]

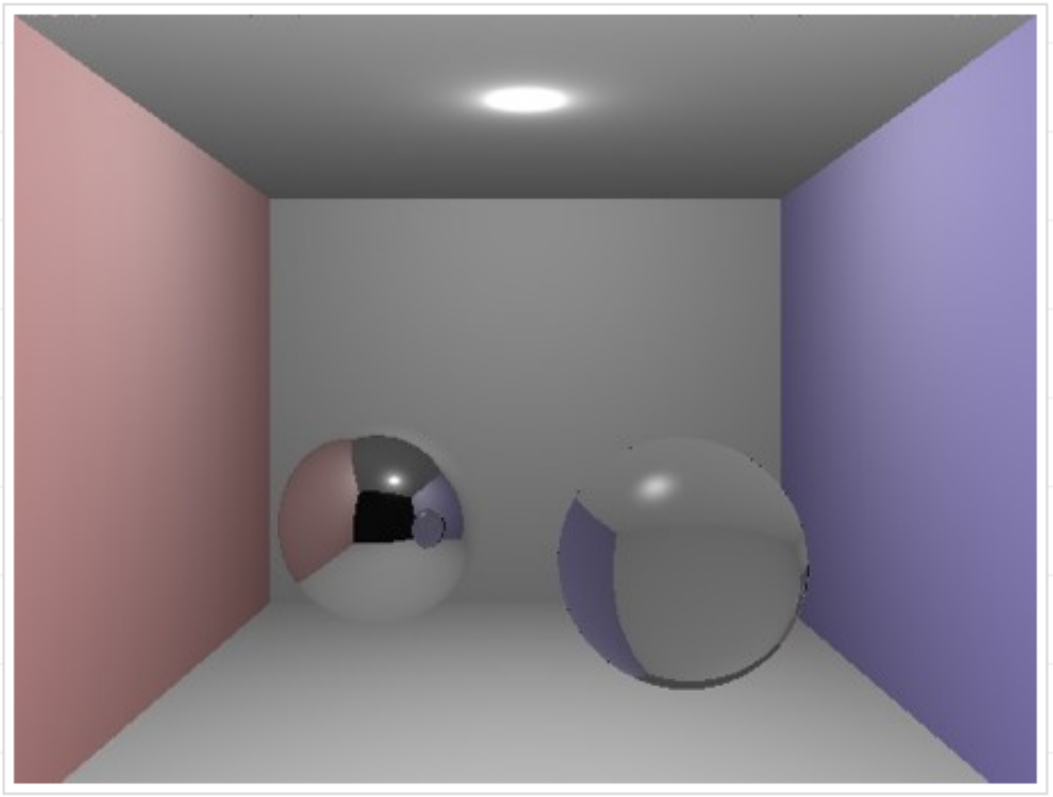
So that code uses $\frac{n_i}{n_t}$

- in the second case want to call

[dir refract: −N : $\frac{1}{n_t}$]

so that code use $\frac{\frac{n_i}{1}}{n_t} = \frac{n_t}{n_i}$

and uses the normal $-N$ pointing into the sphere (towards center)



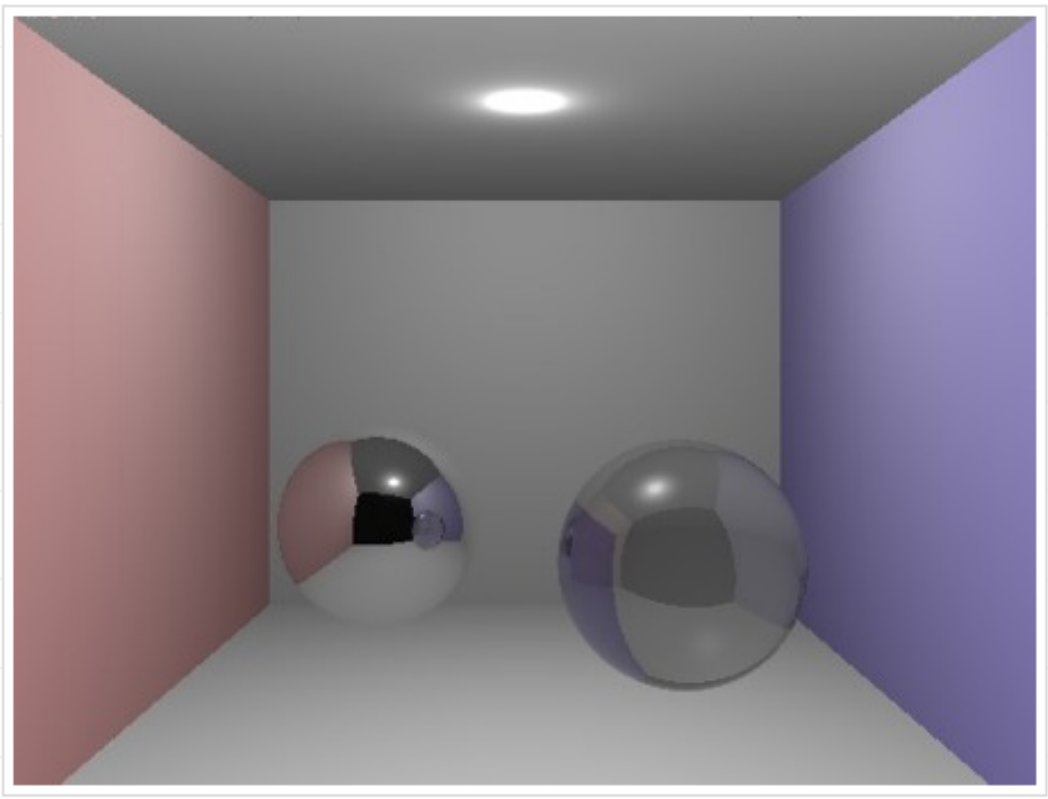
- basic transmission with model of highly refractive optical glass ($n_t = 1.83$) (flint glass)

- Schlick's approximation:

$$R = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

can be used to interpolate between reflected & refracted color at intersection point of two materials where refractive indices differ

$$R_0 = \frac{(n_t - 1)^2}{(n_t + 1)^2}$$

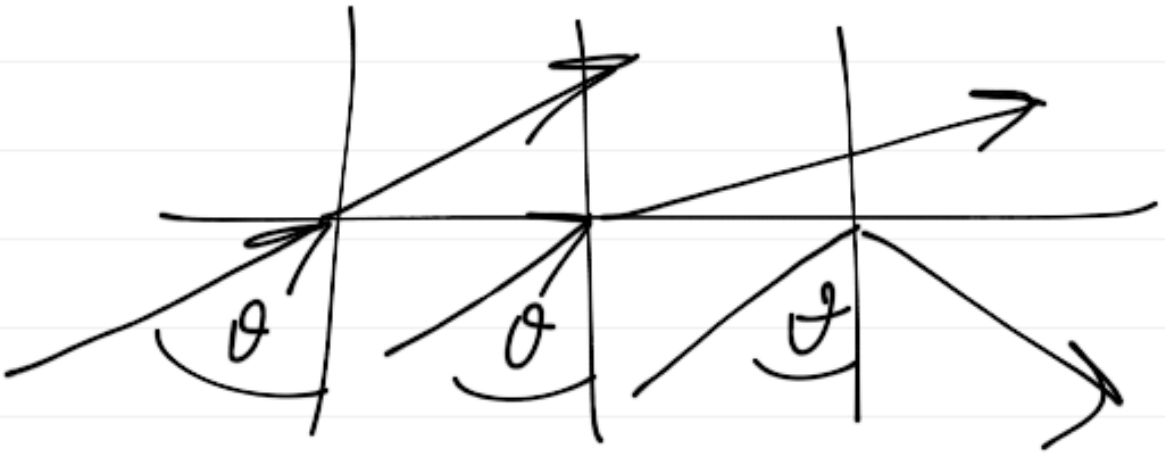


- Color = $(1-\alpha)$ color +
  $\alpha((1-R)$ trn color + (R) ref color$)$

- transparent sphere with Schlick's interpolation of reflection & refraction

- other issue with refraction:

TIR: Total Internal Reflection



when θ reaches critical angle, no refraction, just reflection

$$T = \frac{n_i}{n_t}\left(U - (U \cdot N)N\right) - \left(1 - \frac{n_i}{n_t}\right)^2 \left(1 - (U \cdot N)^2\right)N$$

if this < 0 then

TIR

- in this case refract vec_t function would just return reflect

- sphere_t: now needs to calculate both

$$t_0 = -b - \sqrt{b^2 - 4ac}$$

and

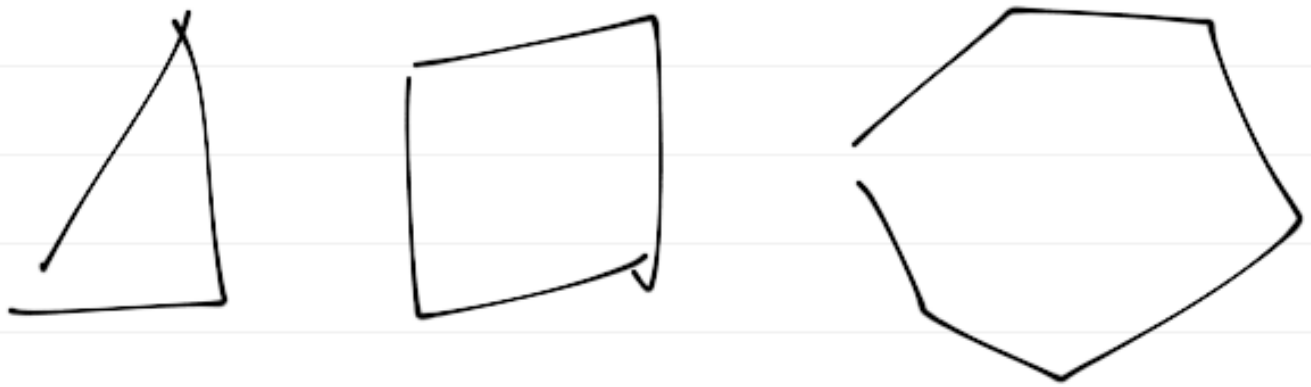
$$t_1 = -b + \sqrt{b^2 - 4ac}$$

and return t_0 if $t_0 < 0.0001$

otherwise return t_1

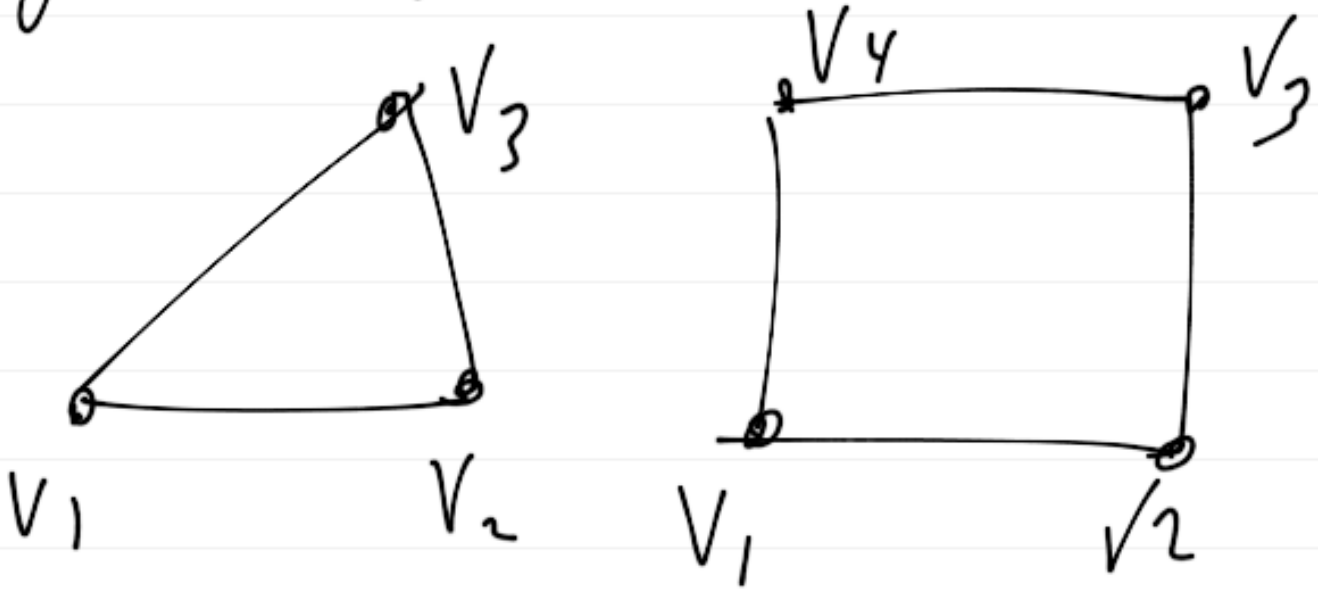
bounded planes

- what about other objects?
- the easiest to implement are bounded planes, or polygons, e.g.,

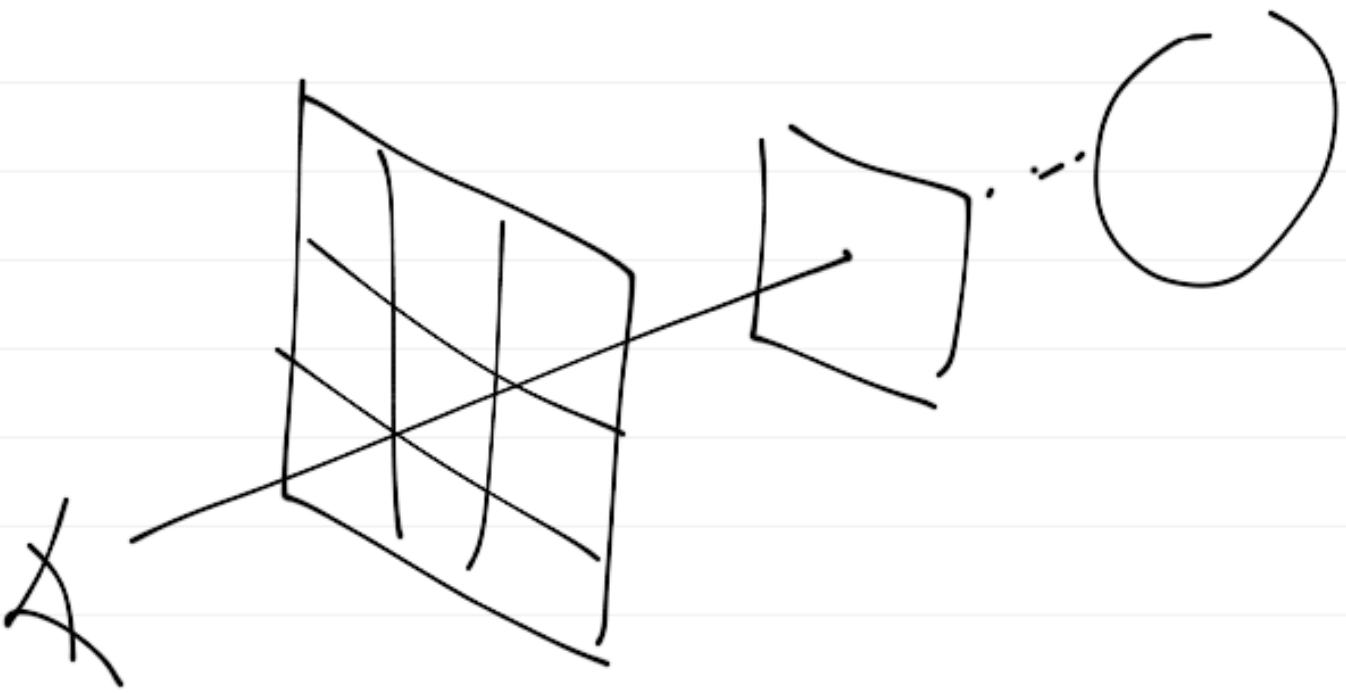


etc.

— in general, there are defined by vertices

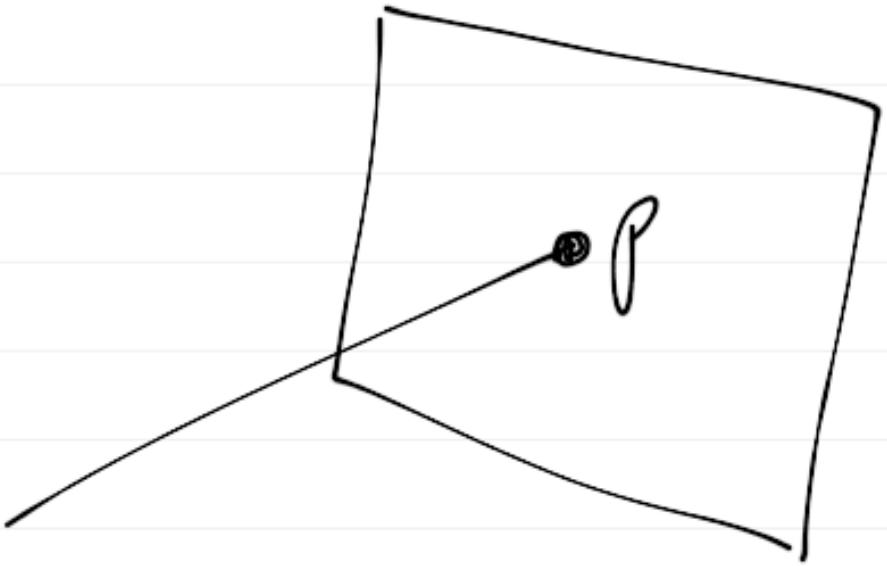


— so object file would likely contain lists of vertices



- ray/plane intersection easy (you're doing it already)
- all that remains is to test point-in-polygon

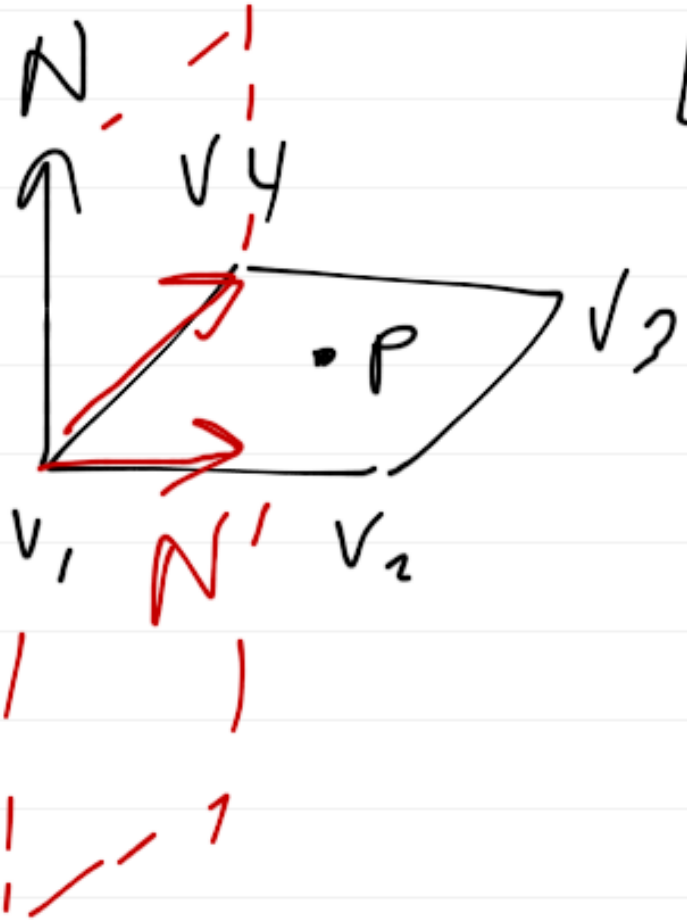
- point-in-polygon problem:



does point (hit point)
lie within bounded segments

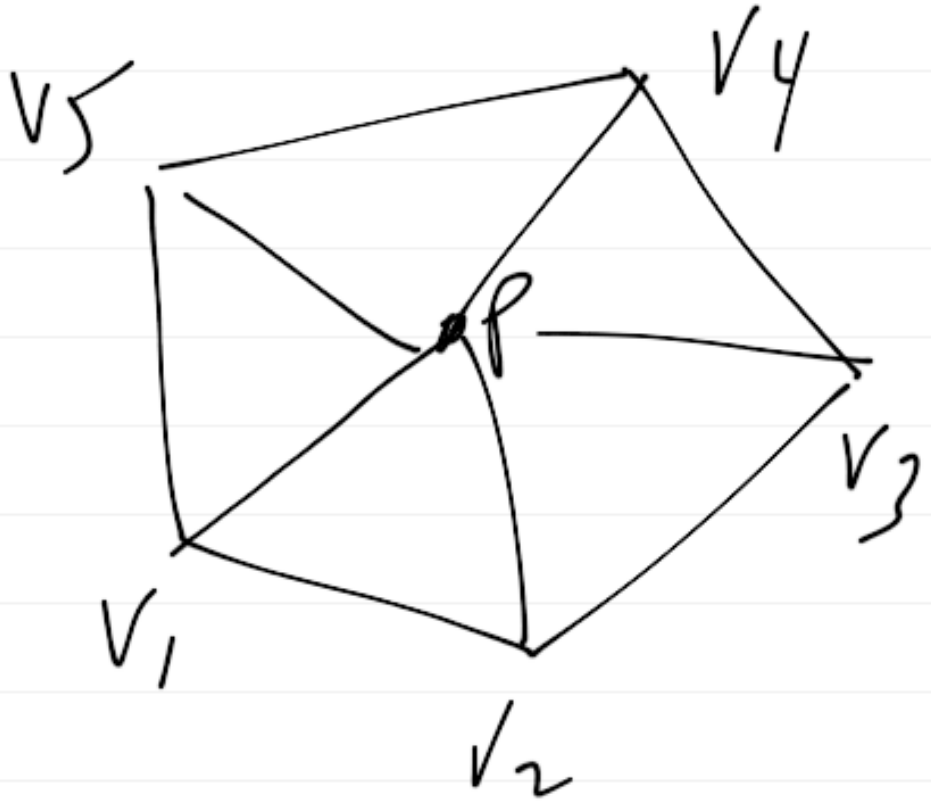
- several ways to compute this, e.g.:

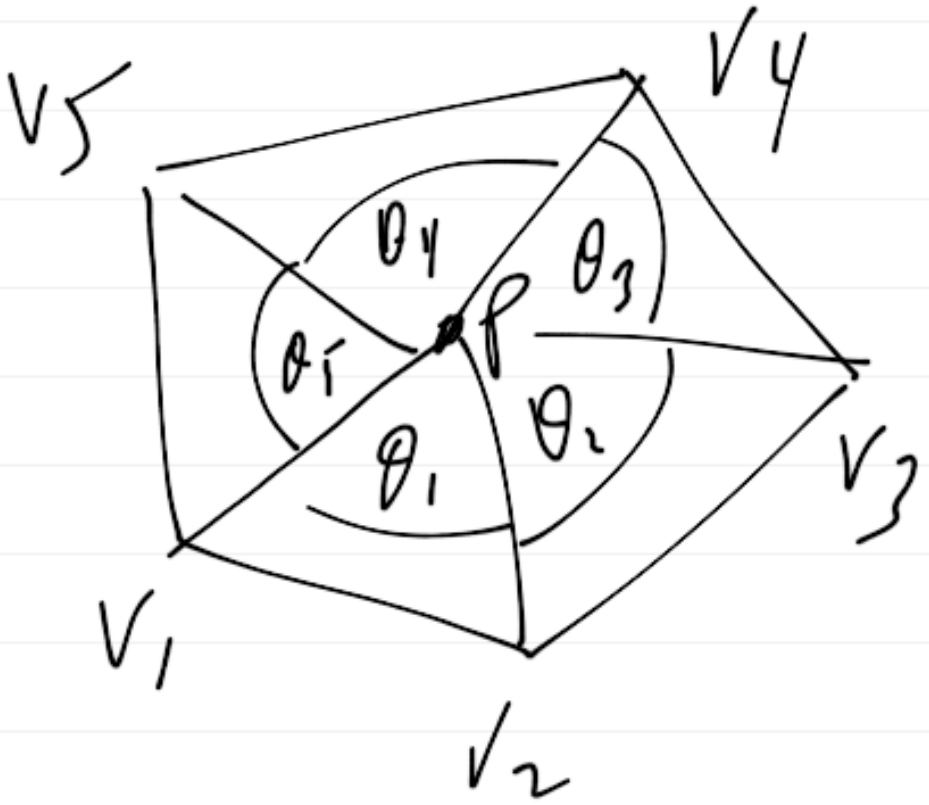
check to see if p above N'



$N' = (v_4 - v_1) \times N$ (right-hand rule)

- repeat for all edges
- this alg. can be slow
- possibly faster alg is summing angles



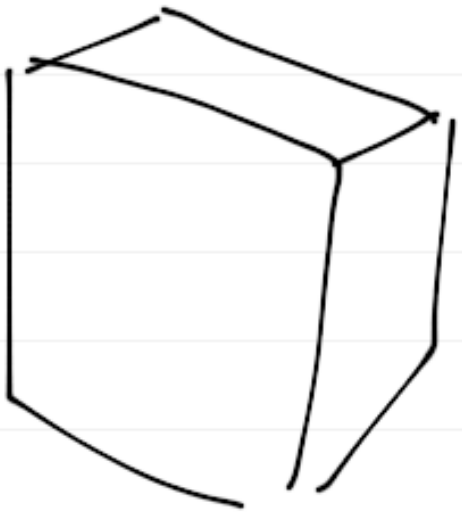


$$(v_{i+1} - p) \cdot (v_i - p) = \theta_i$$

- if $\sum_i \theta_i = 2\pi$, p inside

- this would allow creation of infinitesimally thin surfaces

- for more solid objects,



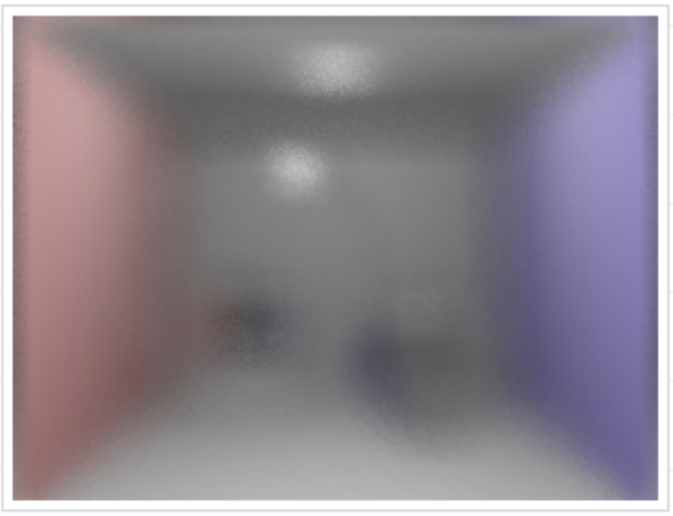
could use two goals

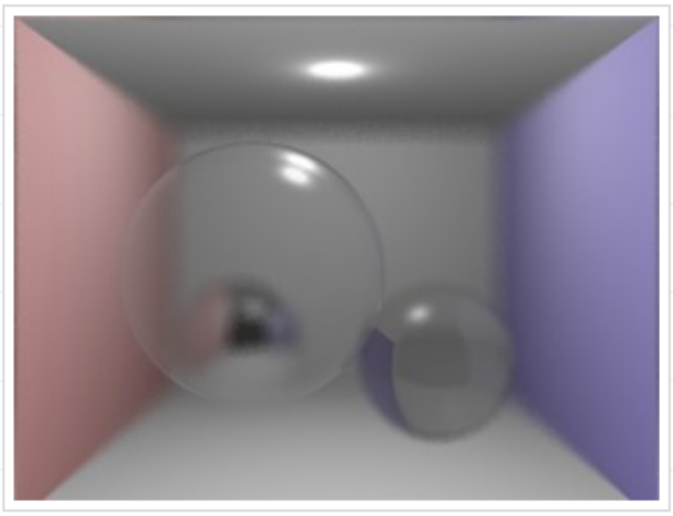
- for transmission thru such objects, ray would need to keep track which medium it was in

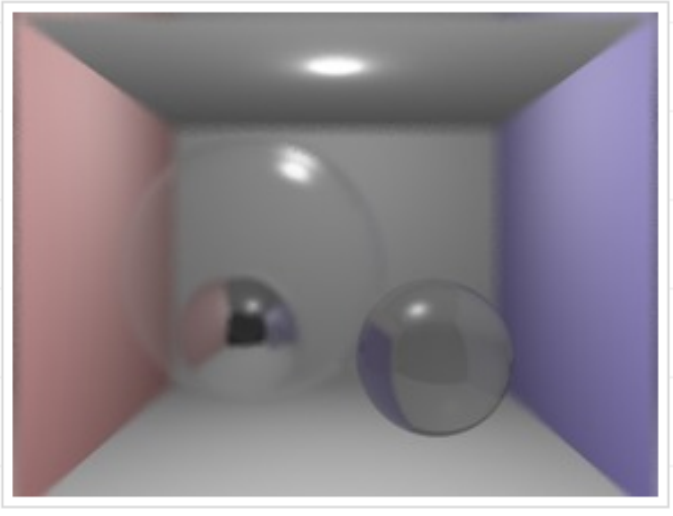
- ray object would keep track via something like inside flag (a boolean)

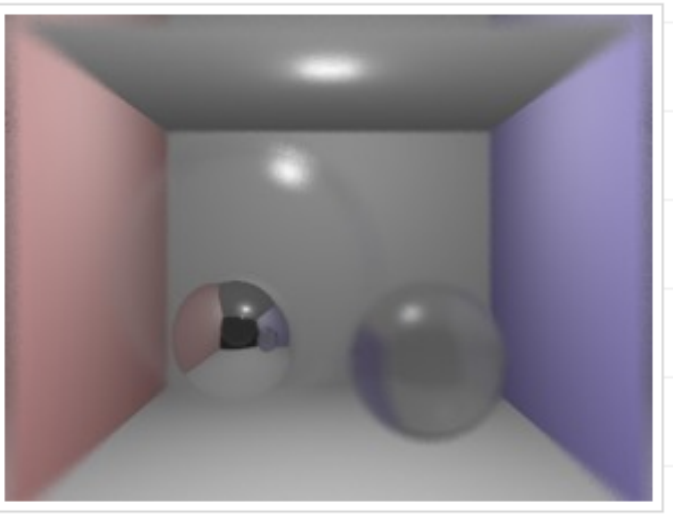
depth of field

— this is surprisingly easy









- need a random no. generator: rand()
  function

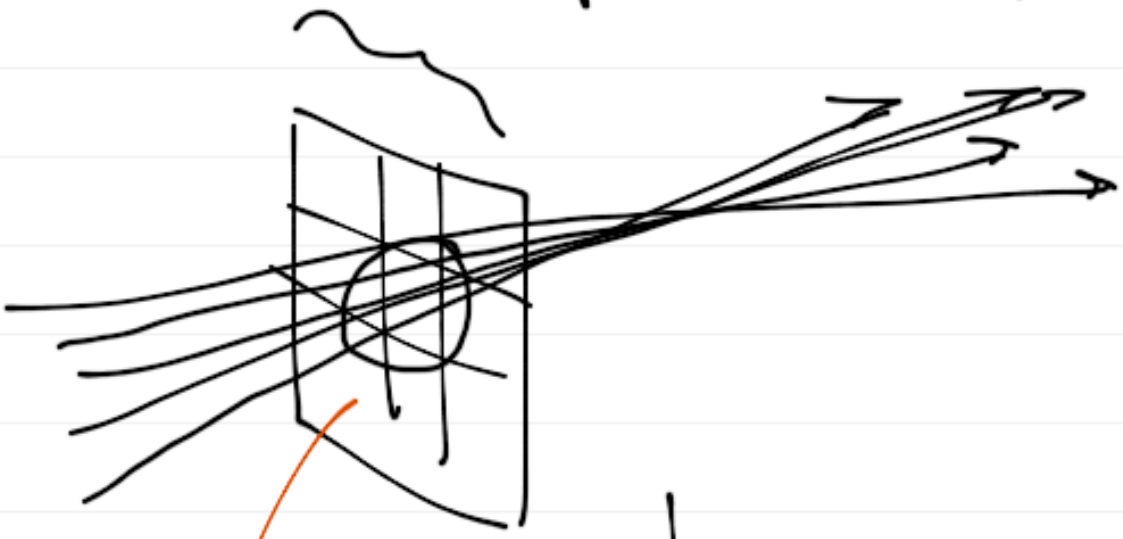
- set up camera parameters
  focus, e.g., 6.3
  fstop, e.g., 2.8
  exposure, e.g. 32

like on a real lens/camera

- aperture is a bit of a misnomer: just the number of rays to shoot per normal ray
- idea is to shoot many rays (1-32) per pixel so that they're focused at the focal distance

→

fstop = size of disc



focus

sample disc

du, dv
perturbations

orig. pixel x, y

```
double aperture = 1.0/fstop;

for(i=0; i<exposure; i++) {
    du = rand() ∈ [-aperture, aperture]

    dv = rand ∈ [...]

    pos[0] += du;
    pos[1] += dv;

    dir = fdepth * dir;

    dir[0] -= du;
    dir[1] -= dv;
```

- then ray trace as normal, accumulate color, dividing each ray's contribution by $\frac{1}{\text{exposure}}$

Shadow casting

- fairly straightforward to implement
- at hit point, after getting surface properties,
  for each light:
    - shoot ray towards light
    - if ray intersects, in shadow

- this hitpoint to light ray is called a shadow feeler

- if in shadow:

    Color += 0.1 * ambient

if not in shadow:

    proceed as normal

- shadow feeler just calls

find_closest

function to see if ray hits something (something non-transparent)