太极图形课

第07讲 Rendering: Implementation Details of a Ray Tracer

taichi

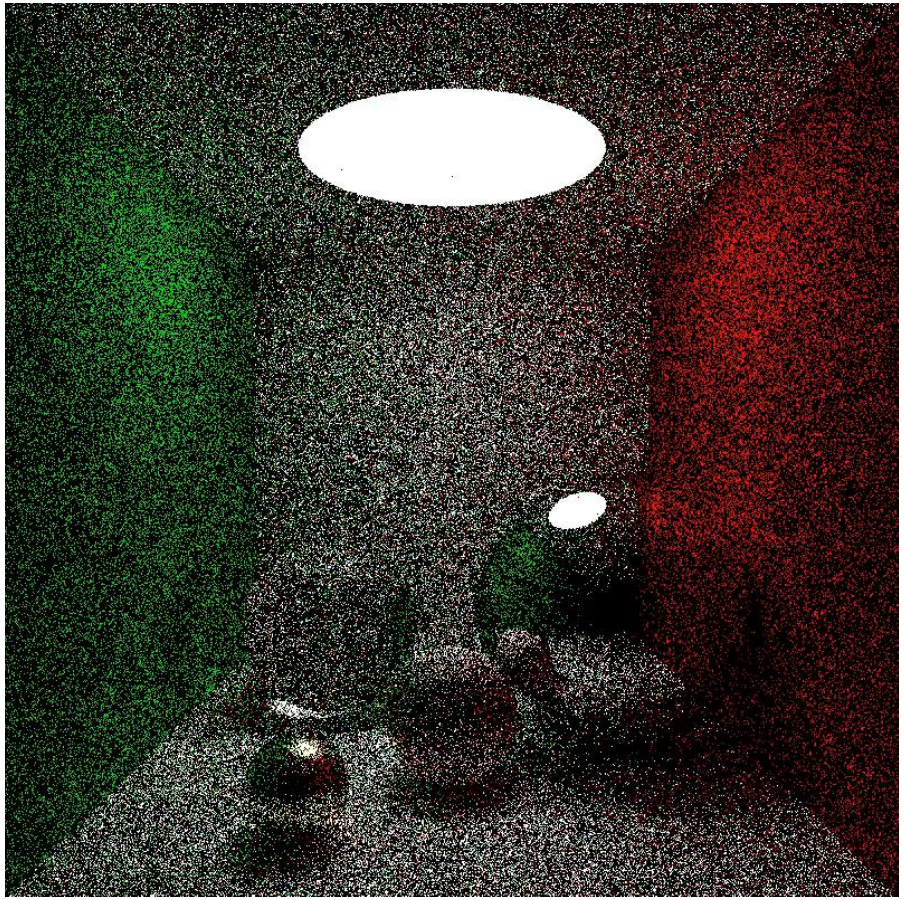
太极图形课

第07讲 Rendering: Implementation Details of a Ray Tracer

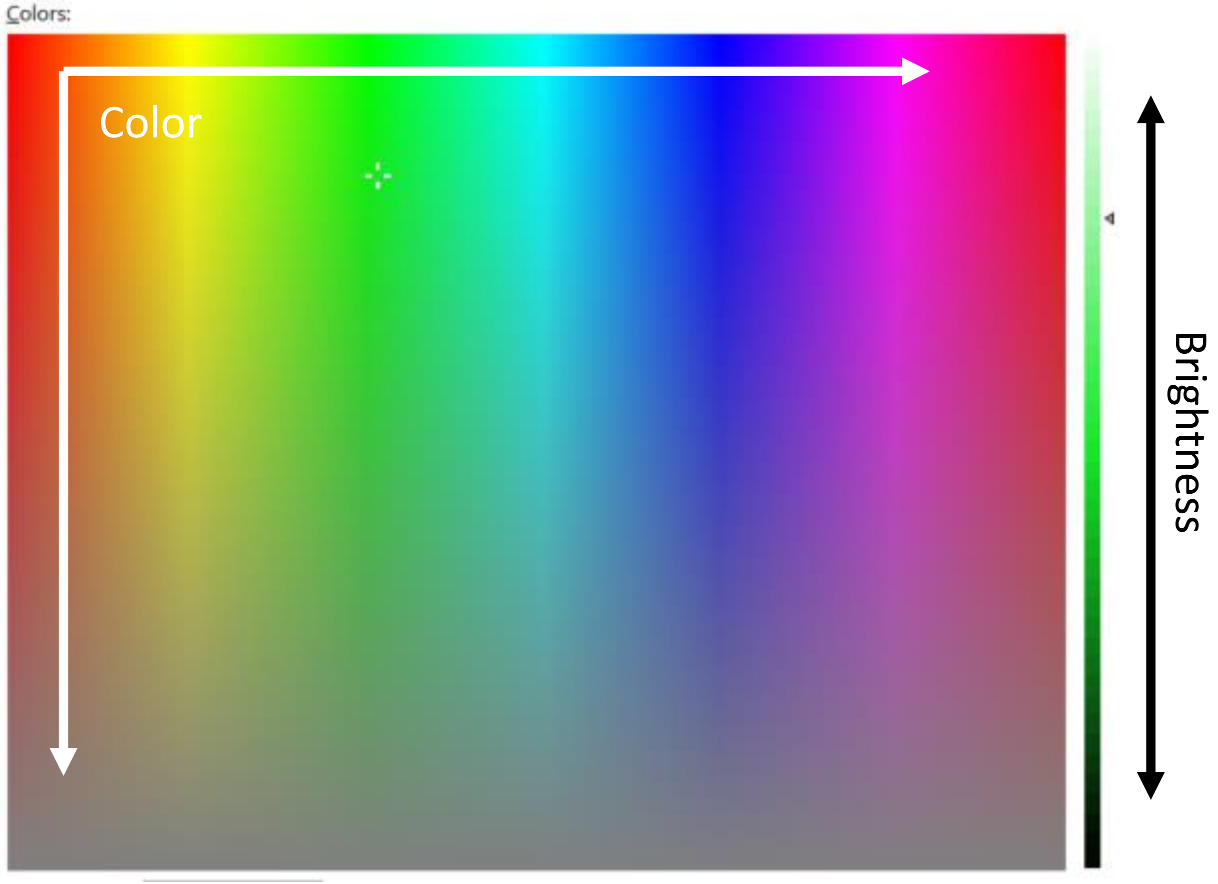
taichi

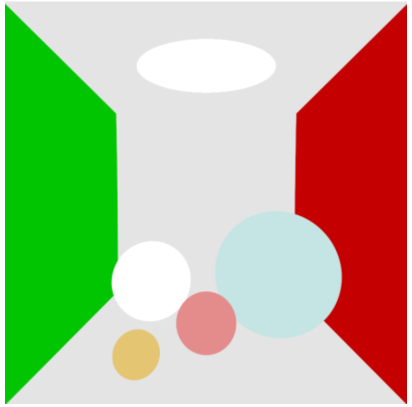
The codebase of ray tracers

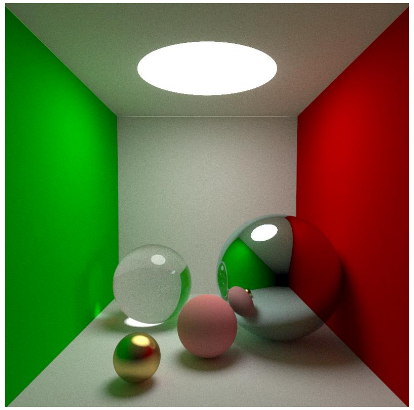
- https://github.com/taichiCourse01/taichi_ray_tracing
- Courtesy of Mingrui Zhang (@erizmr)
- Main reference:
  - Ray tracing in one weekend [Link]



Recap: what we see = color * brightness



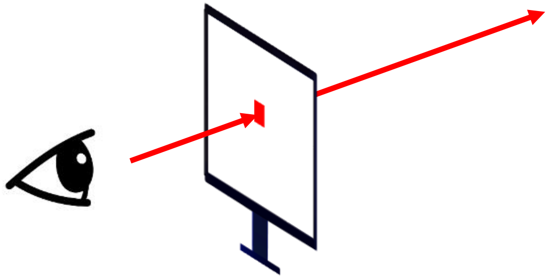


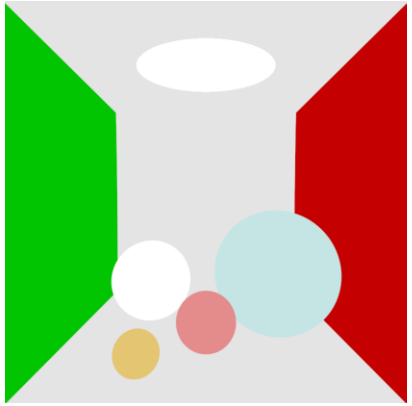


Recap: what we see = color * brightness

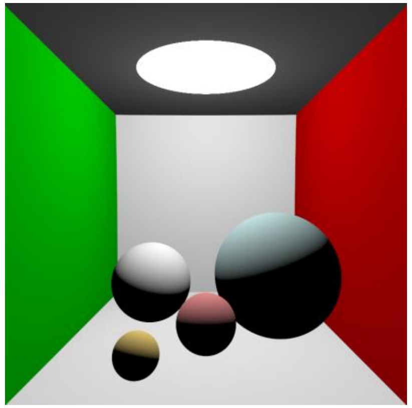
- Color:
  - RGB channels
  - Range $\in [0.0, 1.0]$
  - You can see it as a “filter”
- Brightness:
  - power per unit solid angle per unit projected area (Unit: $\frac{\mathrm{lm}}{sr \cdot m^2}$ or $\frac{W}{sr \cdot m^2}$)
  - Range $\in [0.0, +\infty)$
  - Is called ***Radiance*** in Radiometry
- What we see = color * brightness
- What we see after multiple bounces = color*color*color*…*brightness

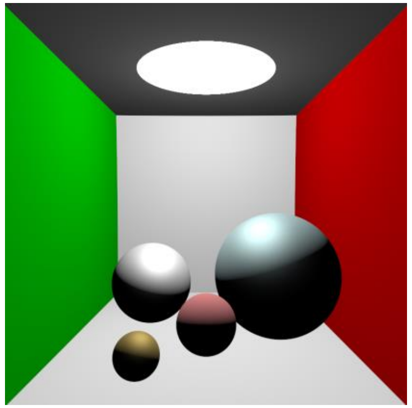
Recap: what color does the ray see?



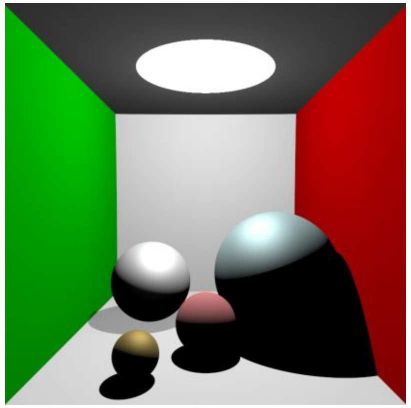


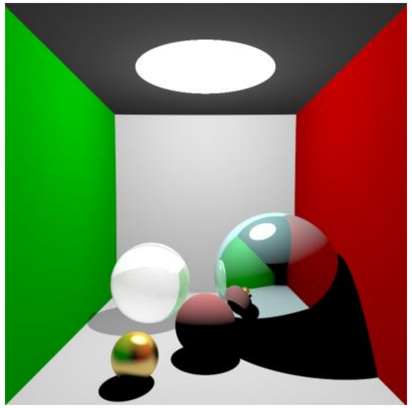
Color



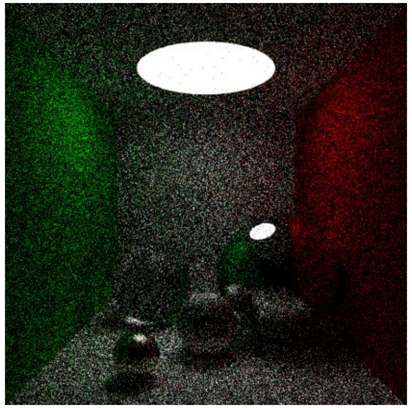


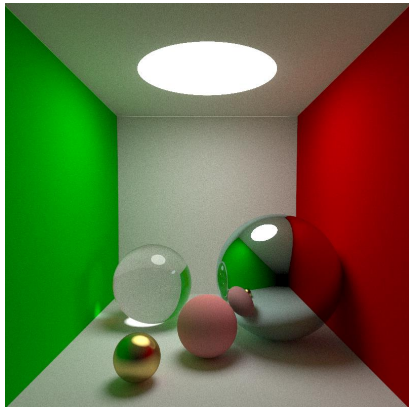
The Shading Models





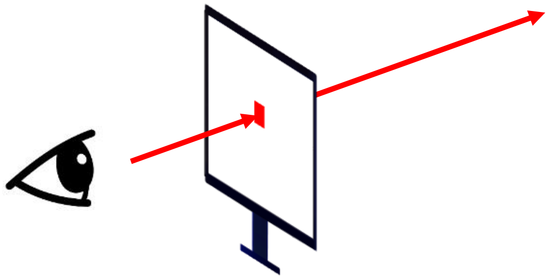
The Whitted-style Ray Tracer





The Path Tracer

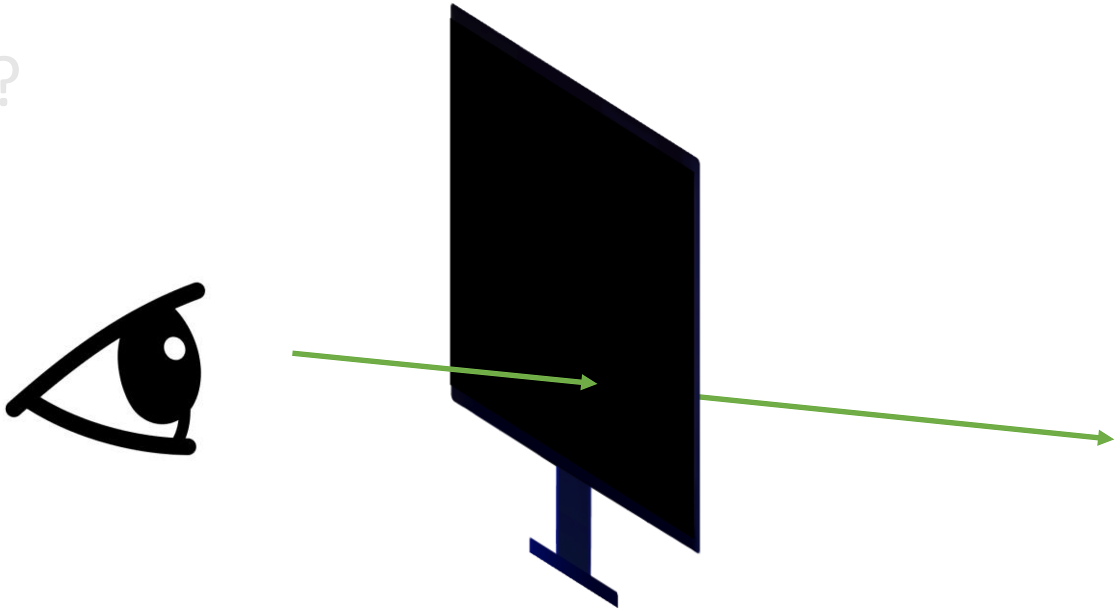
Recap: a path tracer



```
def what_color_does_this_ray_see(ray_o, ray_dir):
    if (random() > p_RR):
        return 0
    else:
        flag, P, N, material = first_hit(ray_o, ray_dir, scene)
        if flag == False:
            return 0
        if material.type == LIGHT_SOURCE:
            return 1 # could be more than 1
        else:
            ray2_o = P
            ray2_dir = scatter(ray_dir, P, N)
            # the cos(theta) in DIFFUSE is hidden in the scatter function
            L_i = what_color_does_this_ray_see(ray2_o, ray2_dir)
            L_o = material.color * L_i / p_RR
            return L_o
```

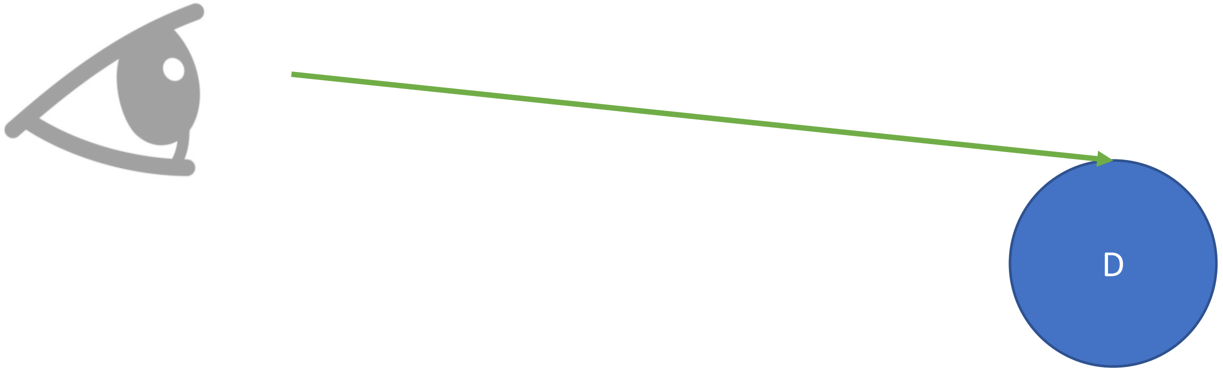
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



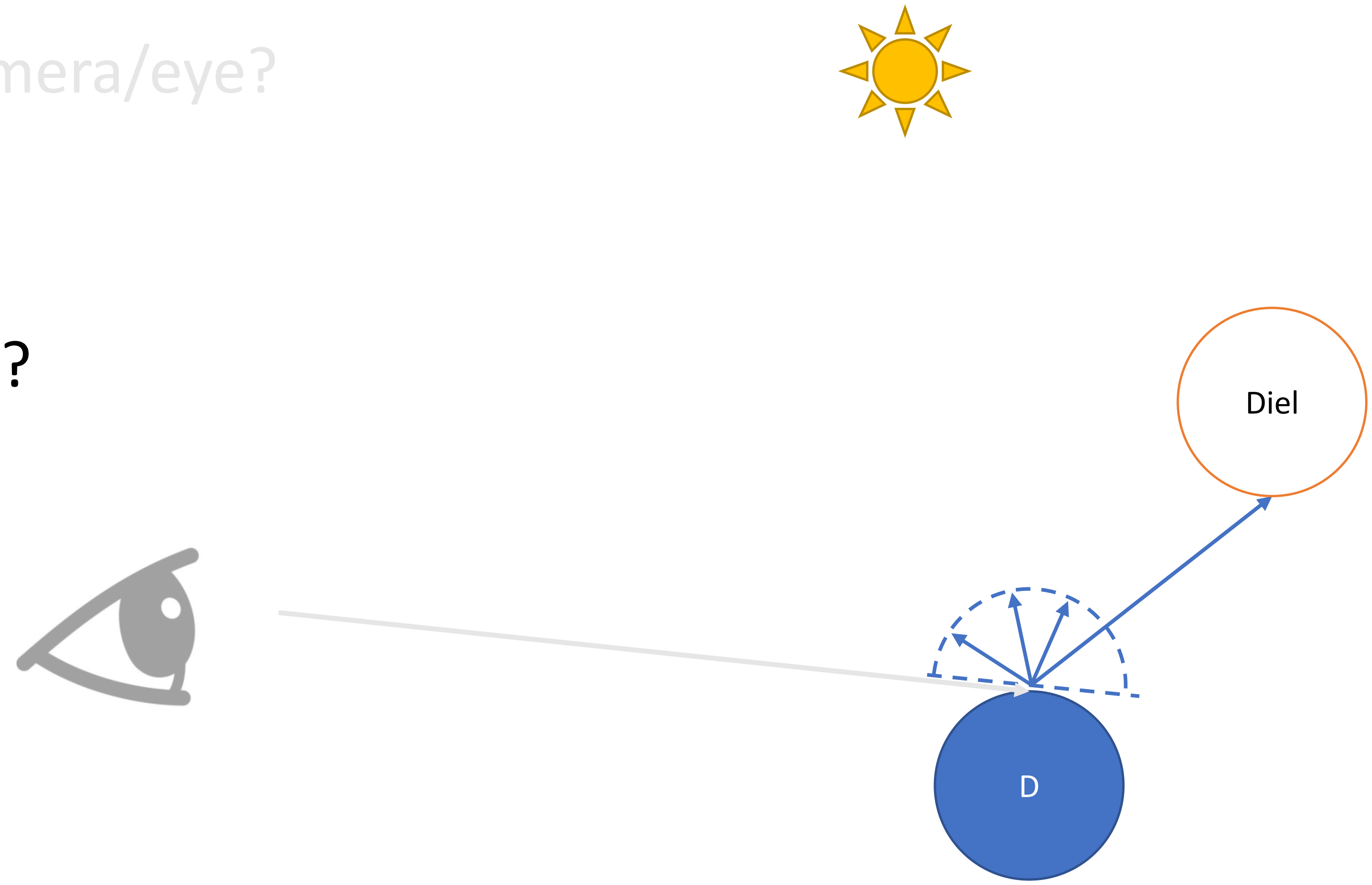
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



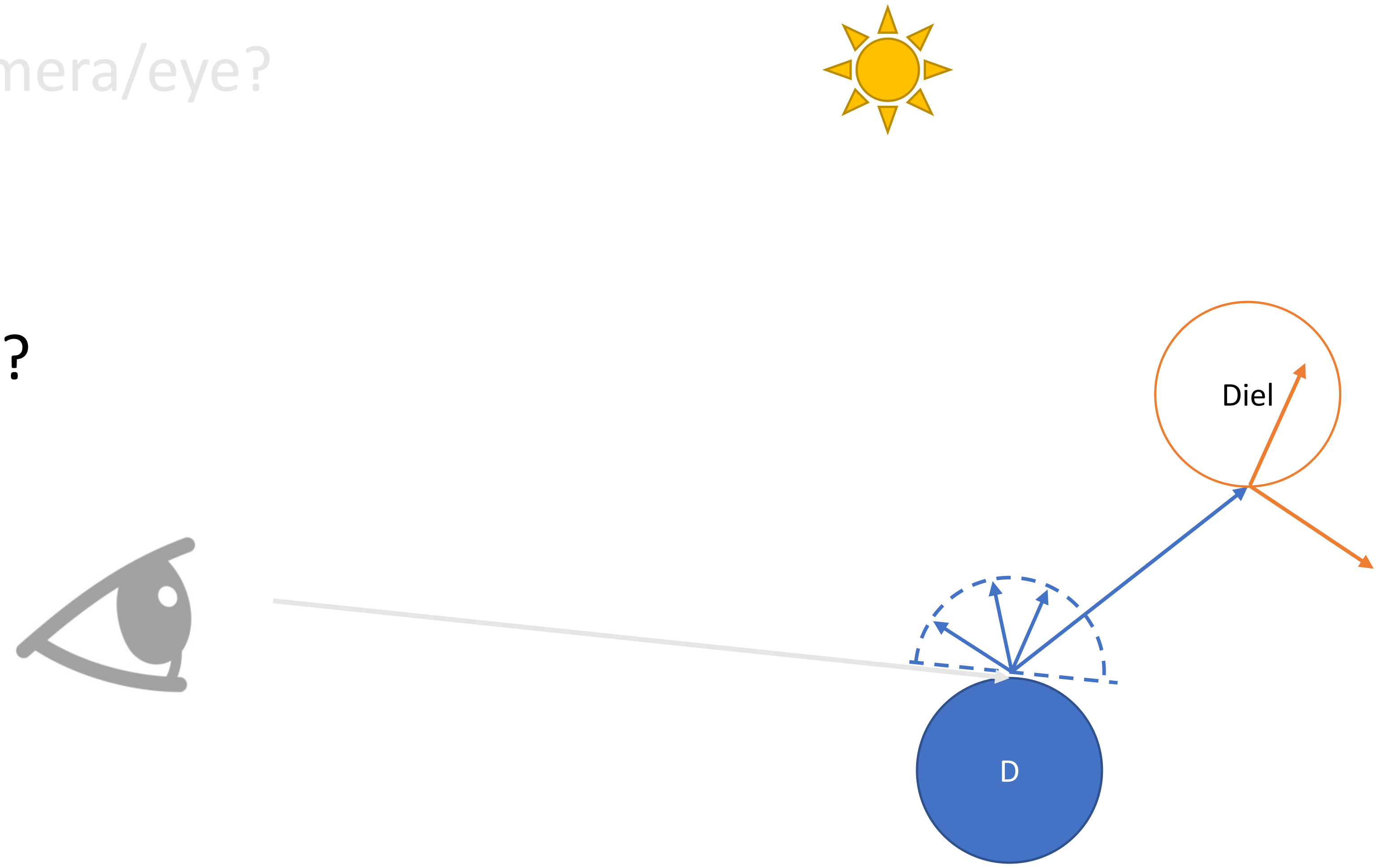
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



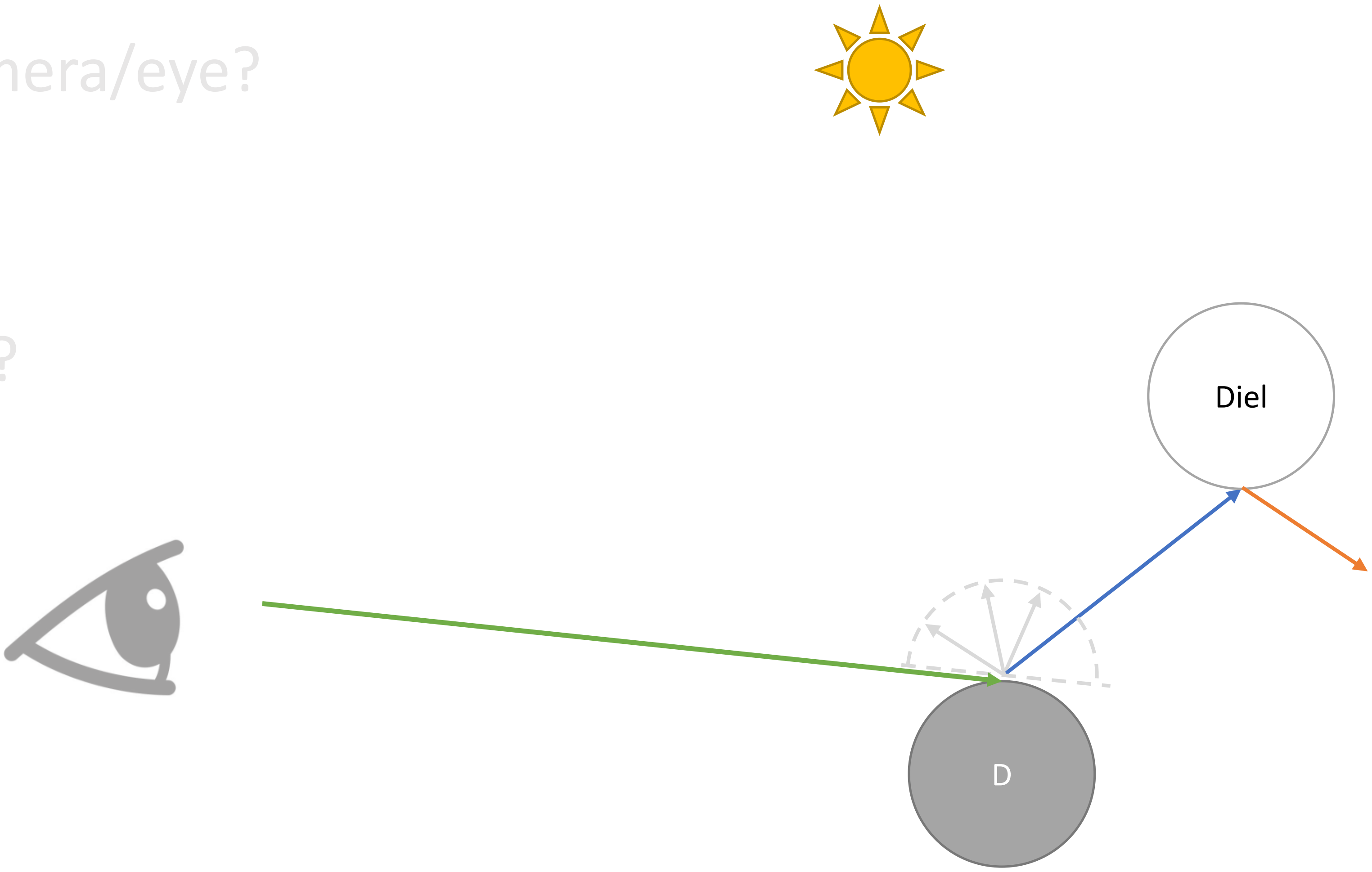
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



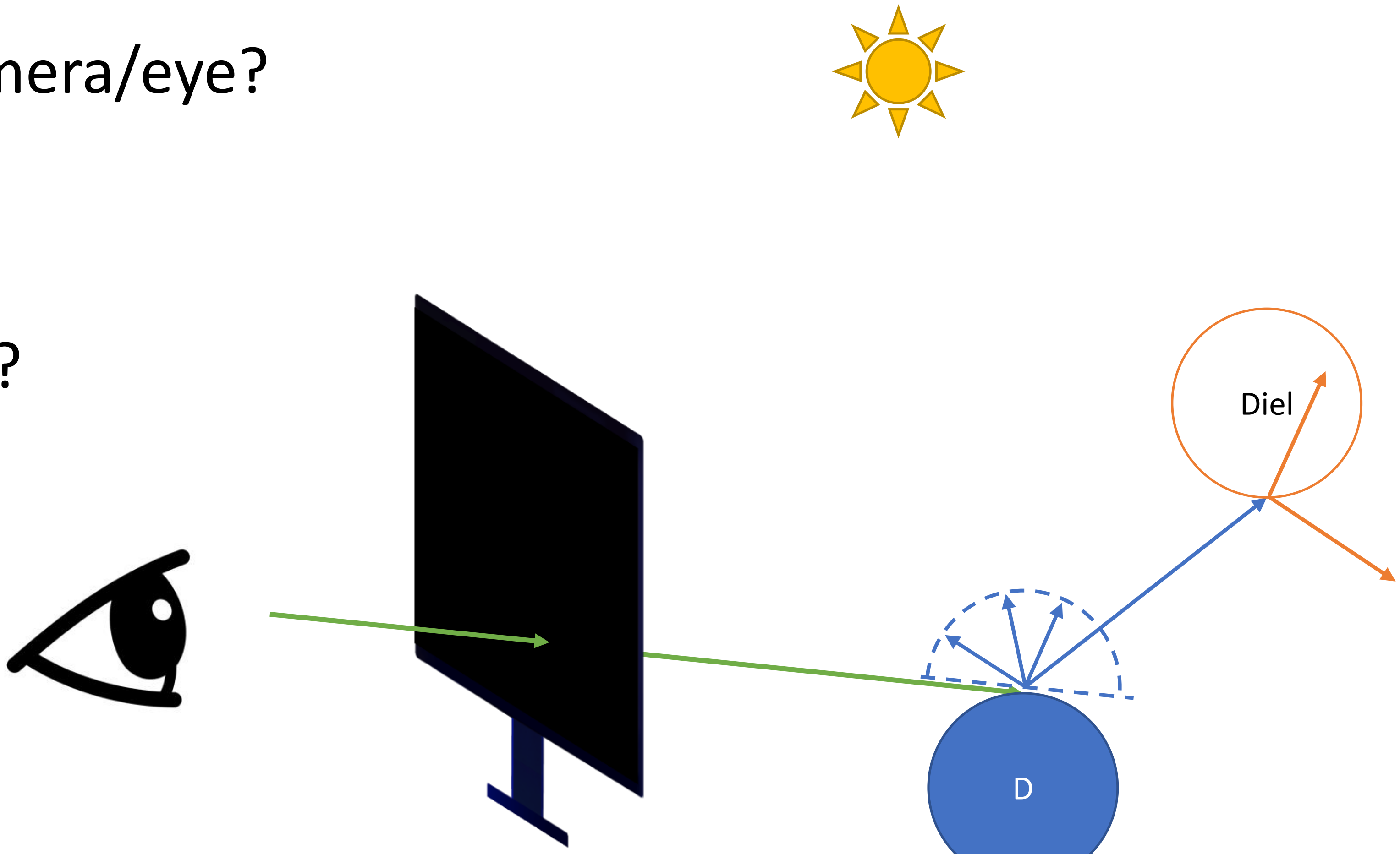
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



Outline Today

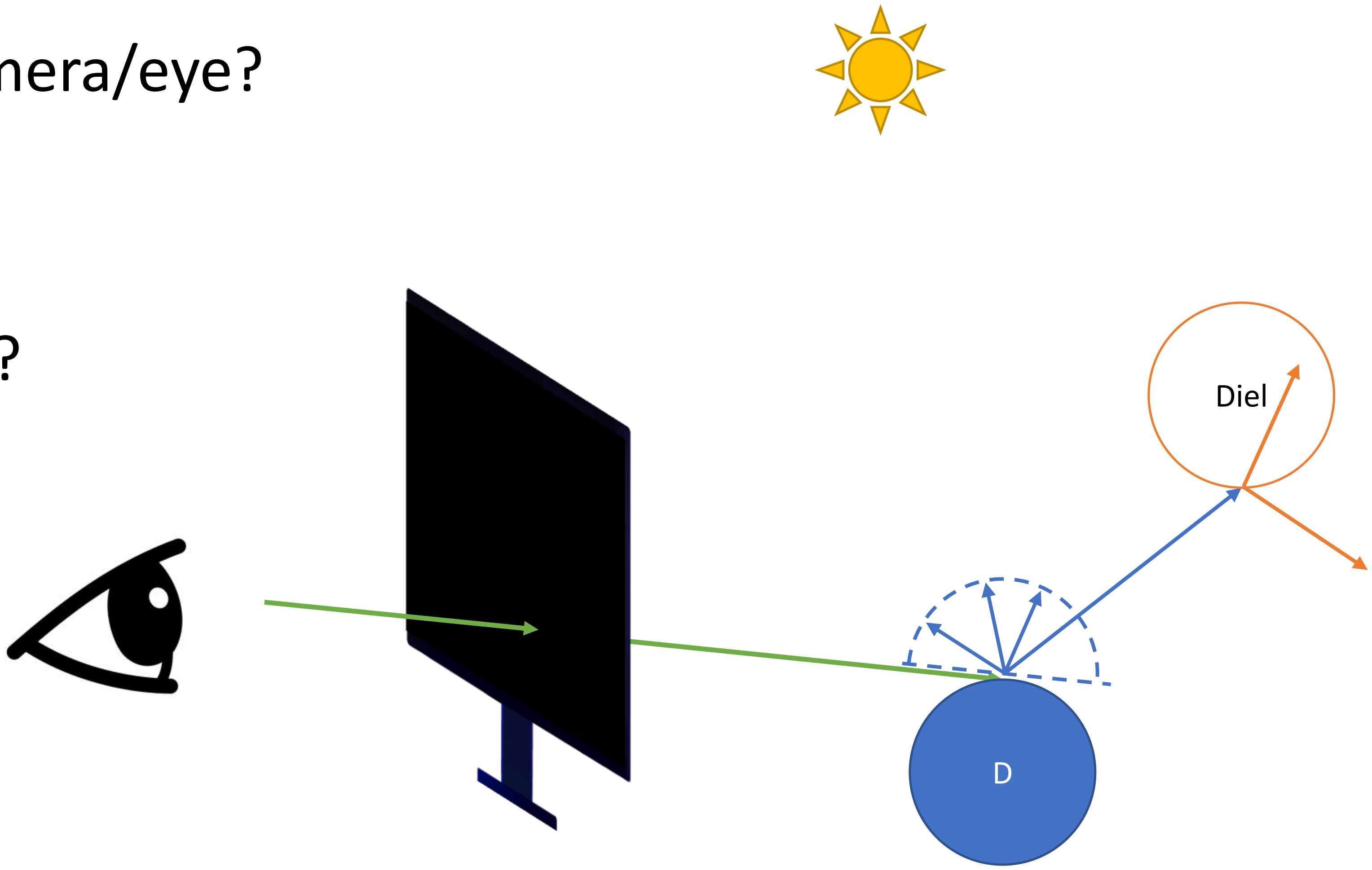
- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?



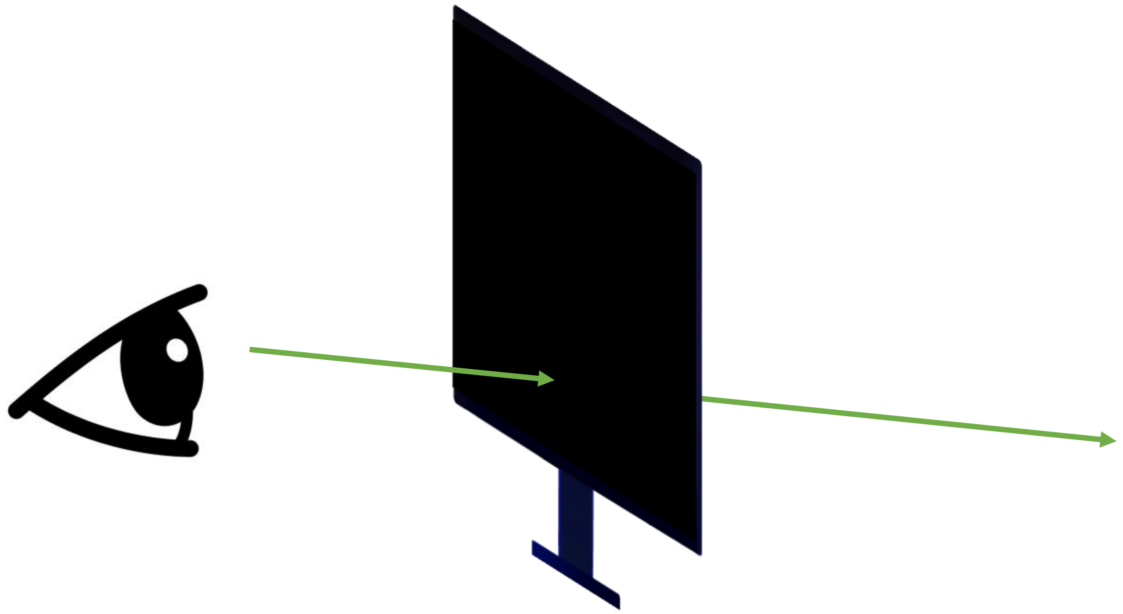
Outline Today

- Ray-casting from the camera/eye?
- Ray-object intersection?
- Sampling?
- Reflection v.s. refraction?
- Recursions in Taichi?

- Anti-aliasing?

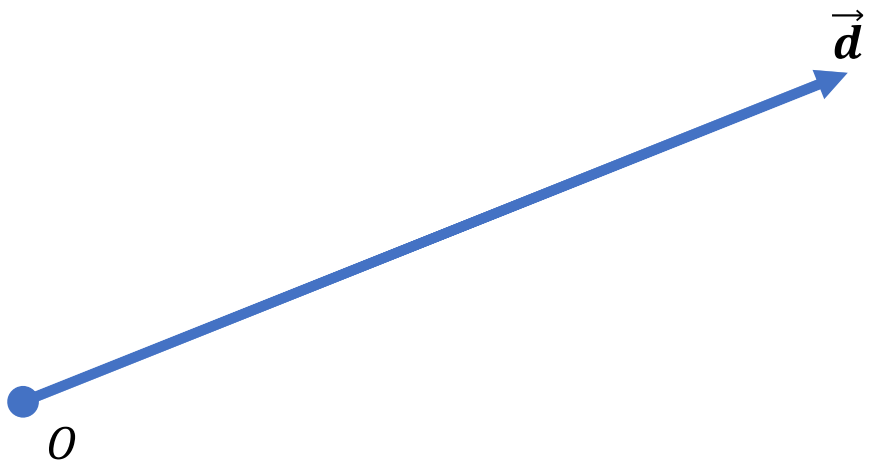


Ray-casting



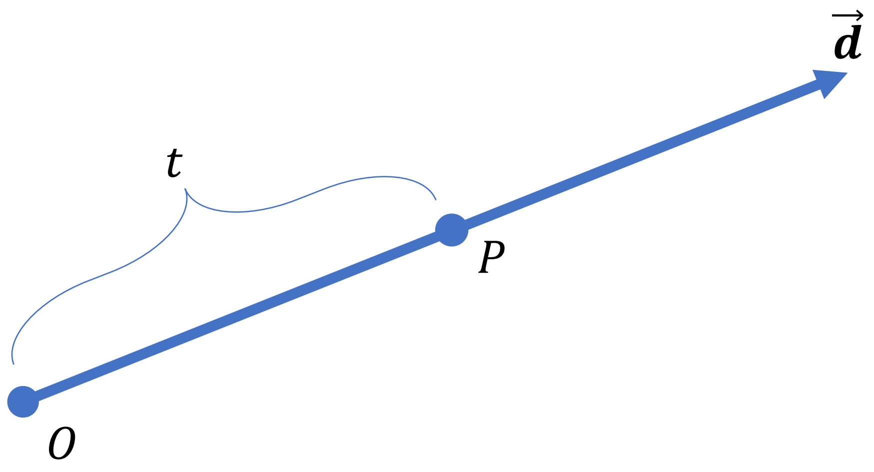
What is a ray?

- A ray is a line defined by its origin and direction



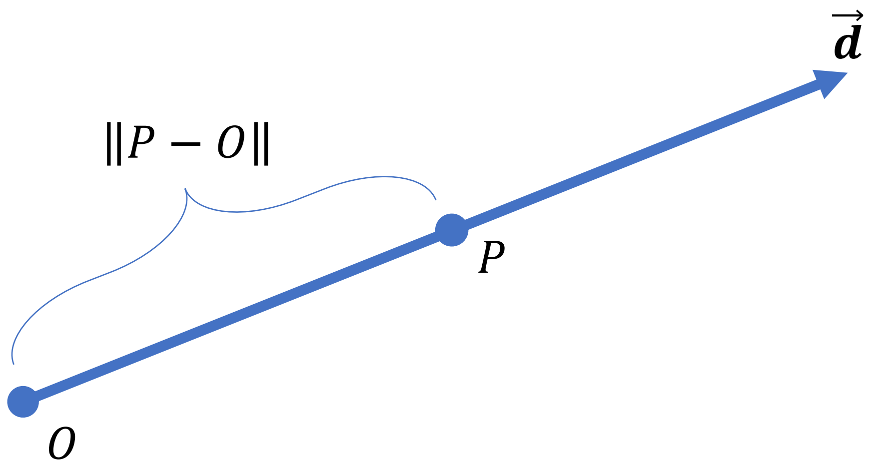
What is a ray?

- A ray is a line defined by its origin and direction
- Any point on a ray can be described using a single parameter t:
  - $P = O + t\vec{\boldsymbol{d}}$

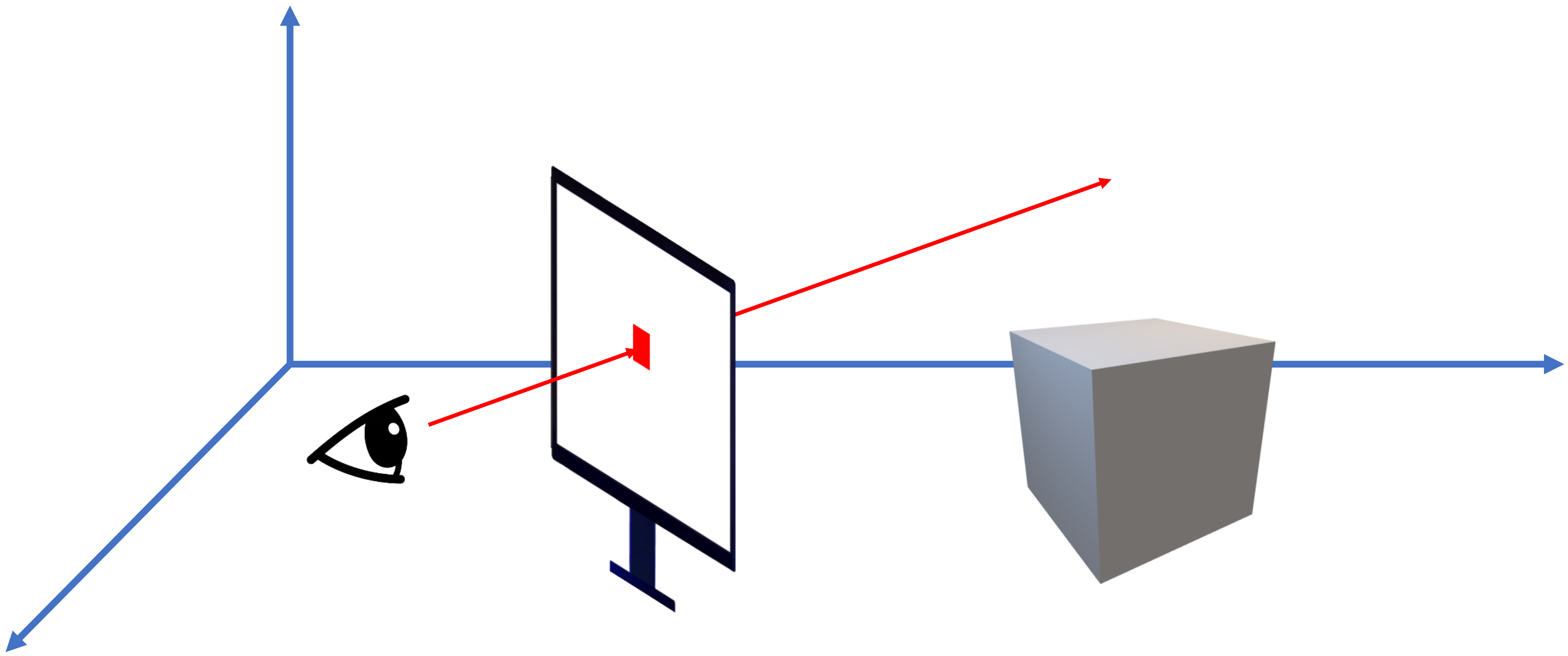


What is a ray?

- A ray is a line defined by its origin and direction
- A ray can be determined by its origin and another point on the ray:
  - $\vec{\boldsymbol{d}} = \frac{P-O}{\|P-O\|}$

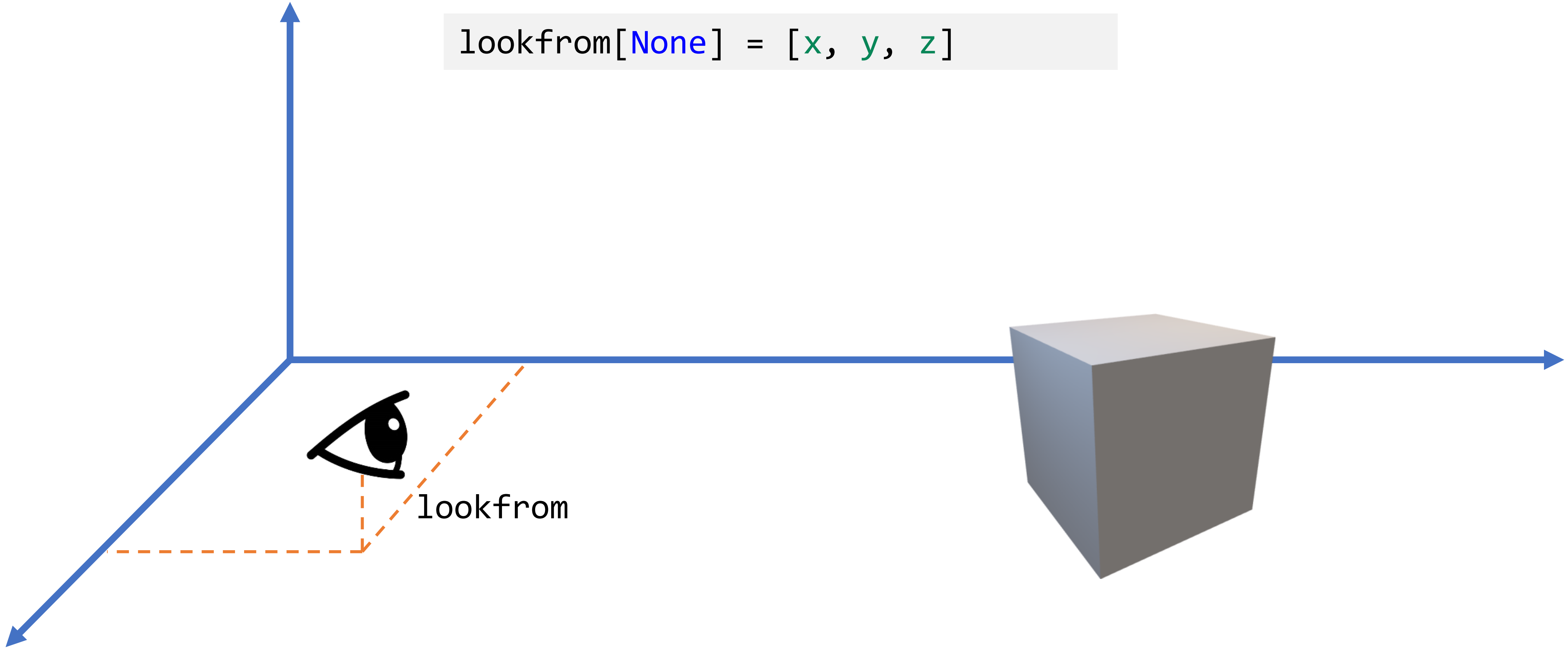


Setting up the camera



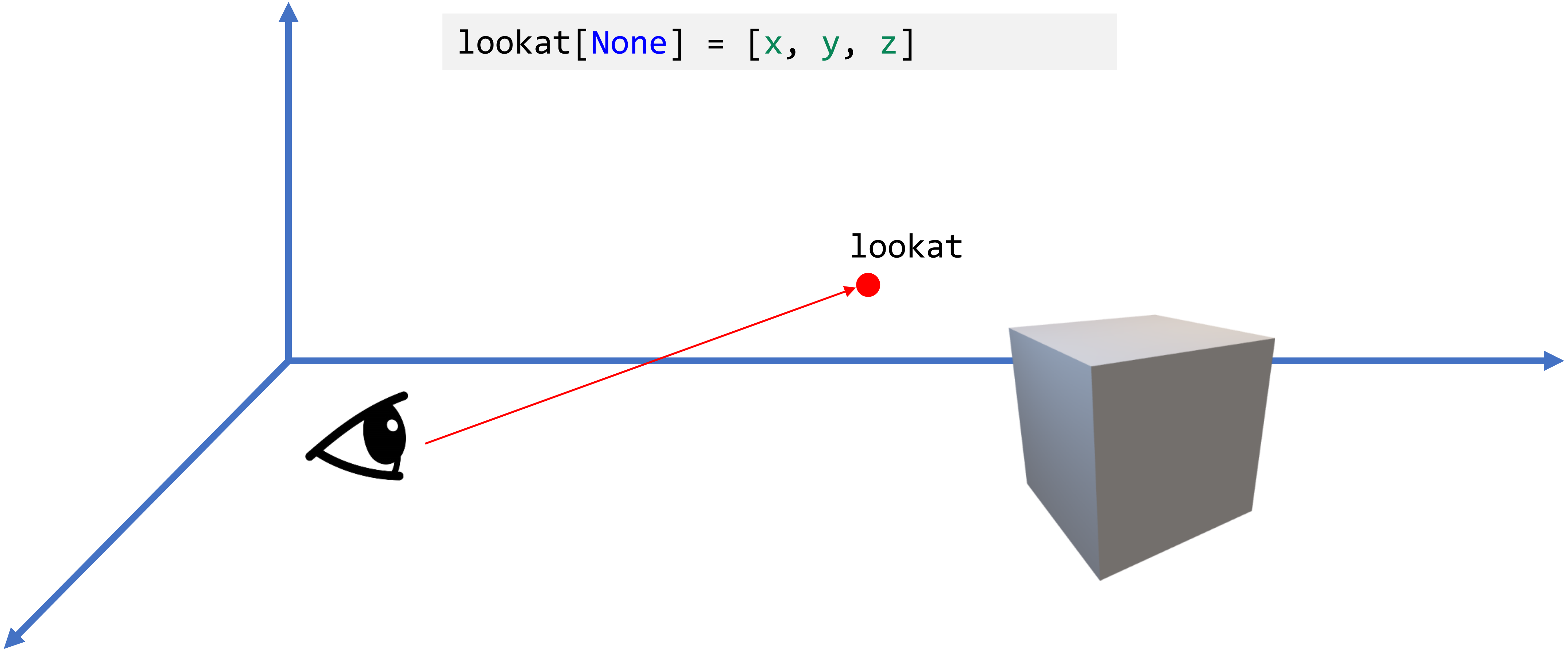
Positioning the camera/eye (lookfrom)

```
lookfrom[None] = [x, y, z]
```



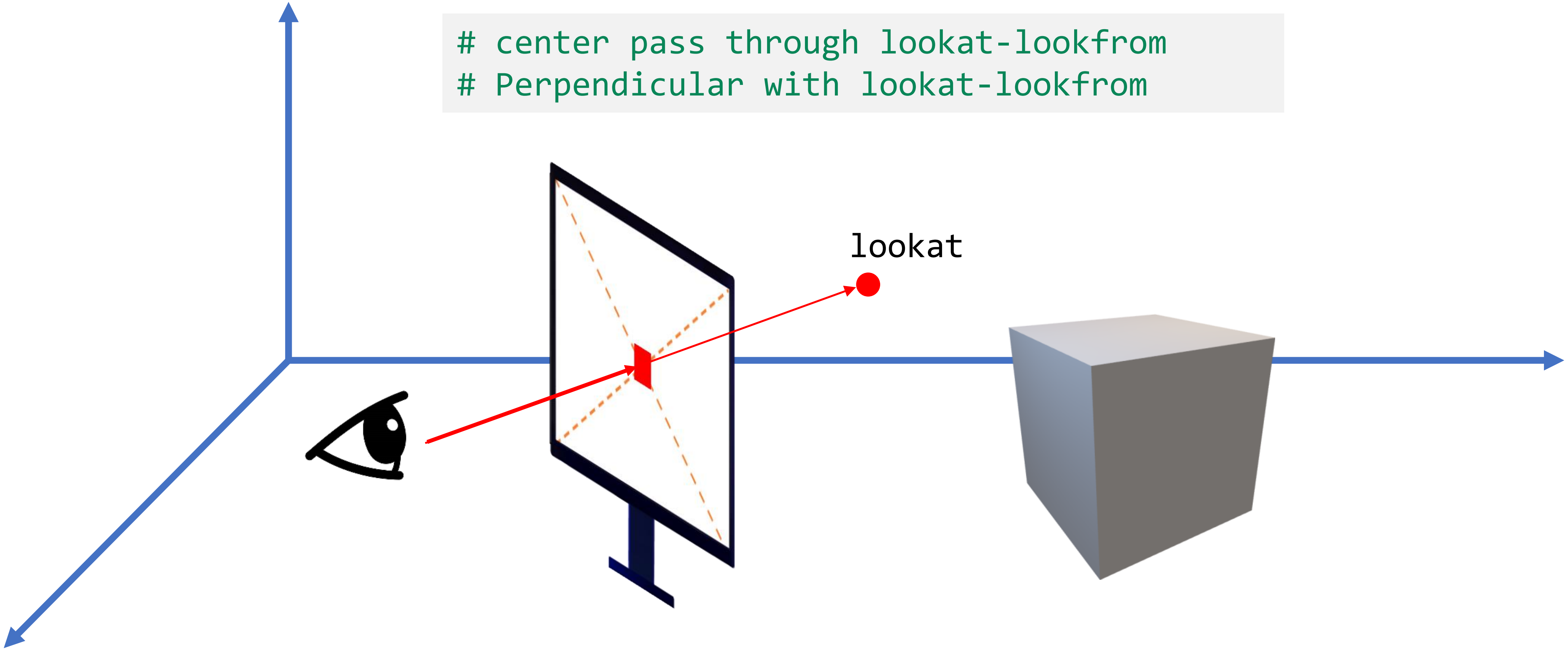
Orienting the camera/eye (lookat)

```
lookat[None] = [x, y, z]
```



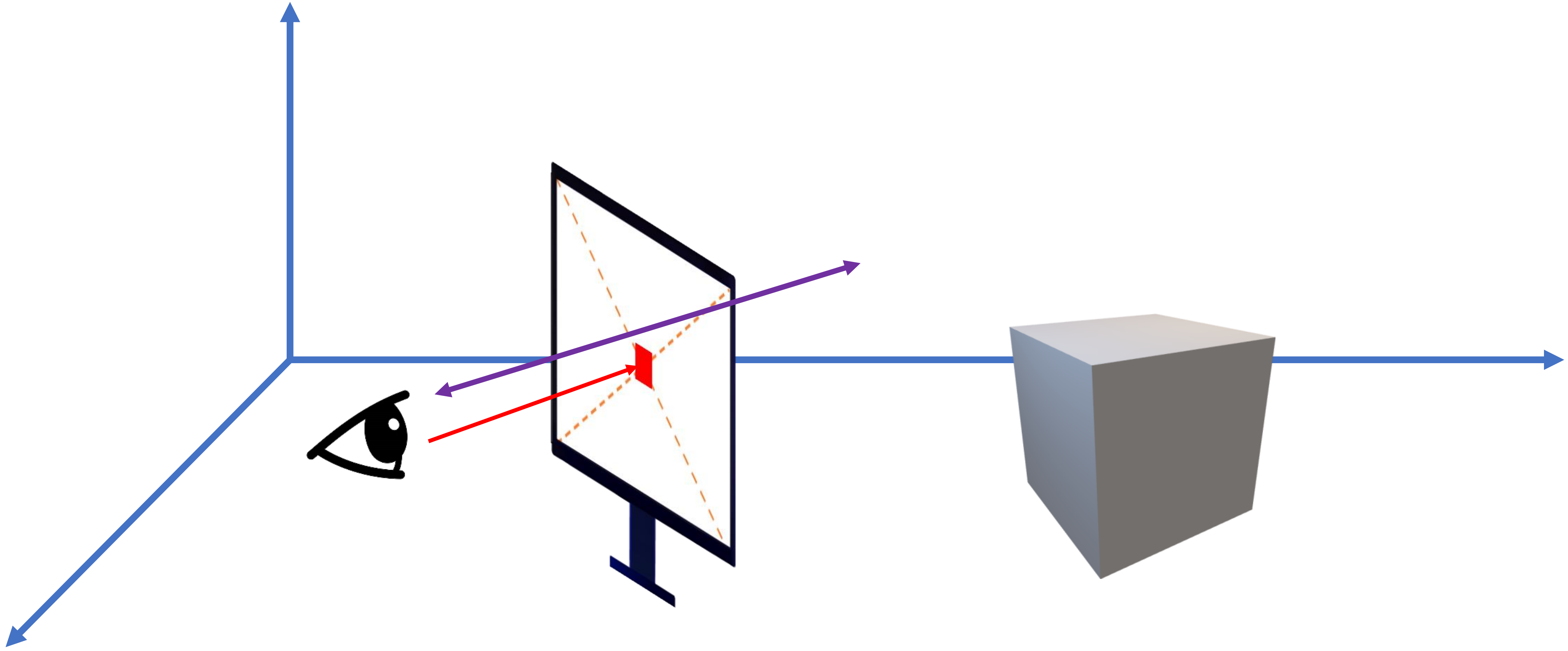
Placing the screen

```
# center pass through lookat-lookfrom
# Perpendicular with lookat-lookfrom
```



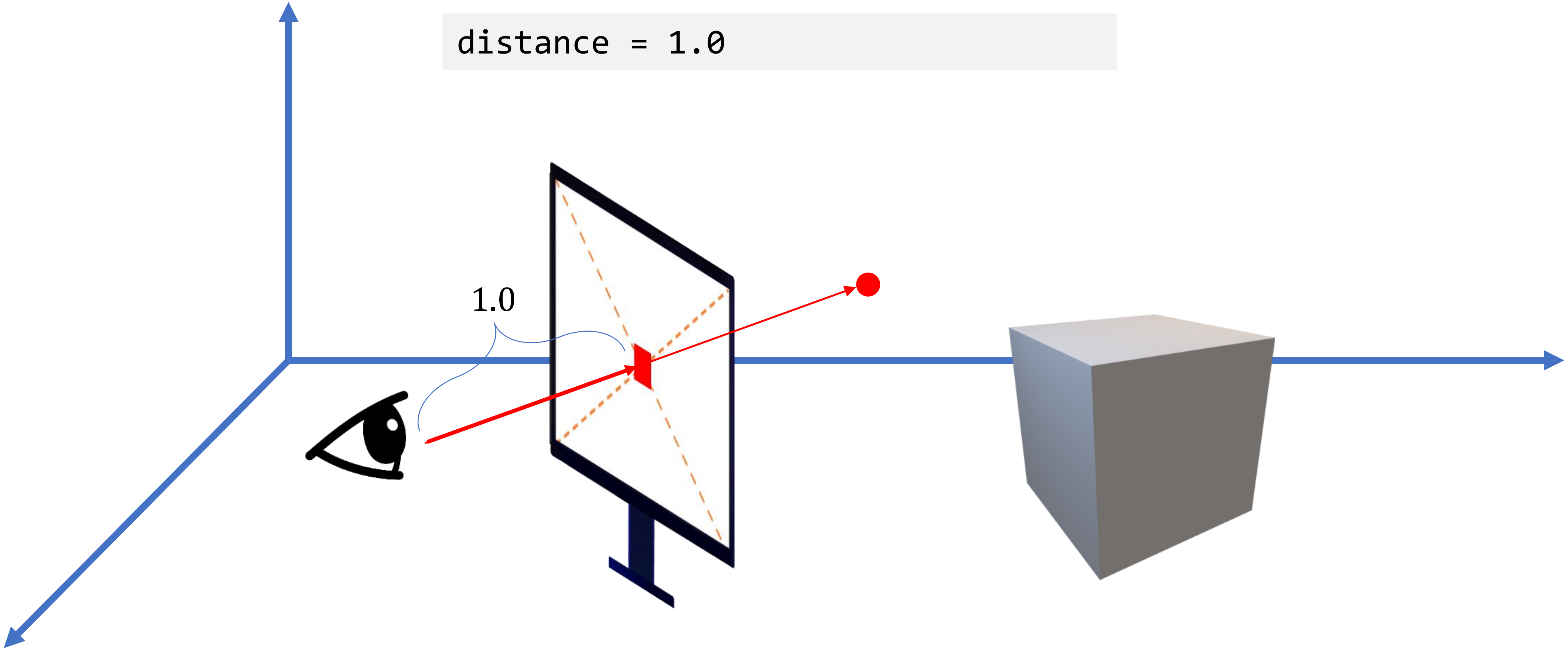
Problem:

1. the distance between the screen and the camera is not decided



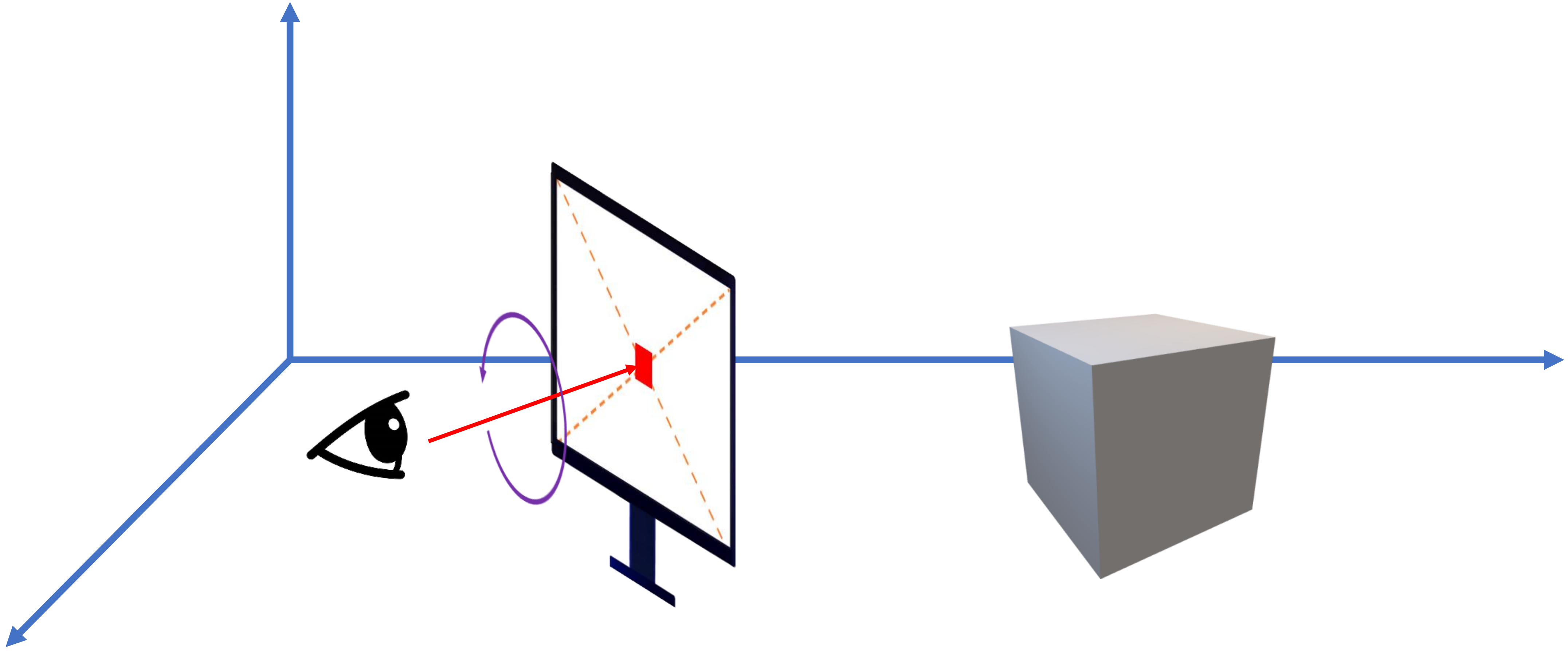
Placing the screen

```
distance = 1.0
```



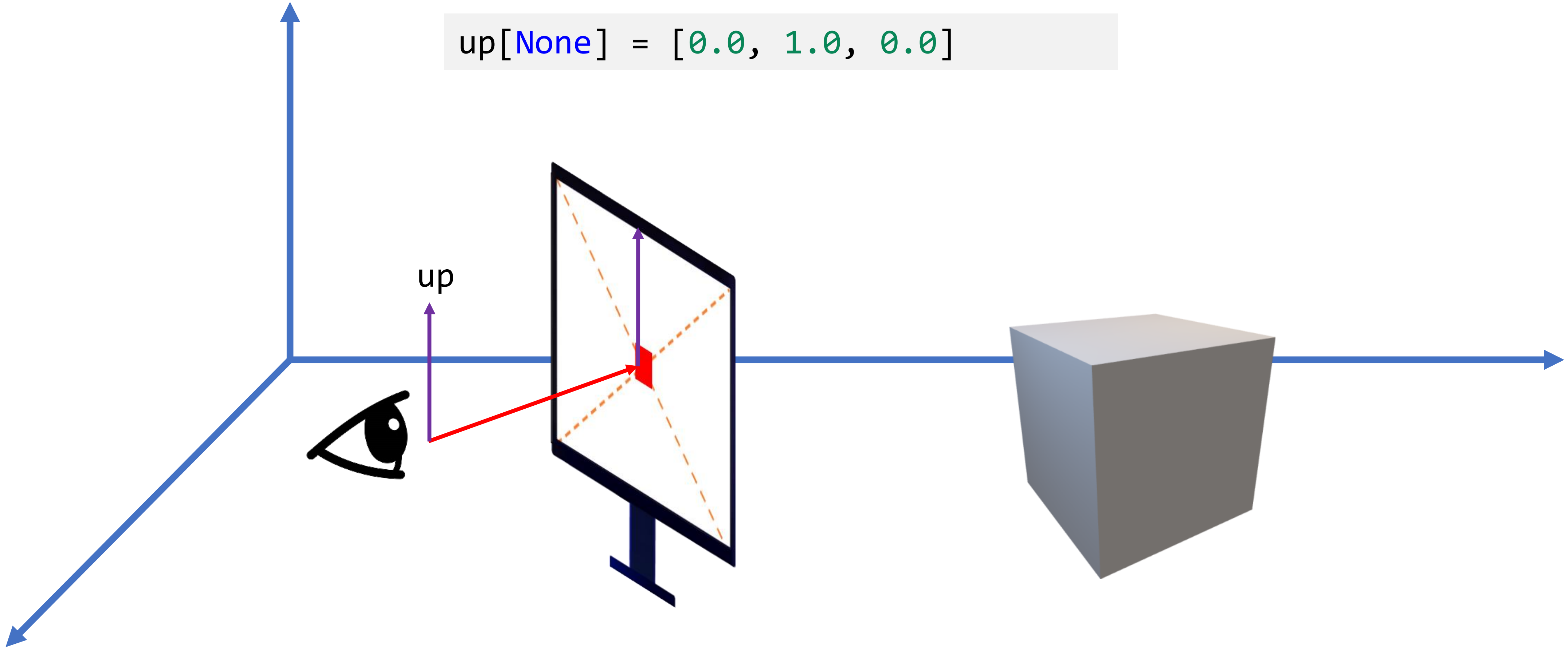
Problem:

2. the orientation of the screen is not decided



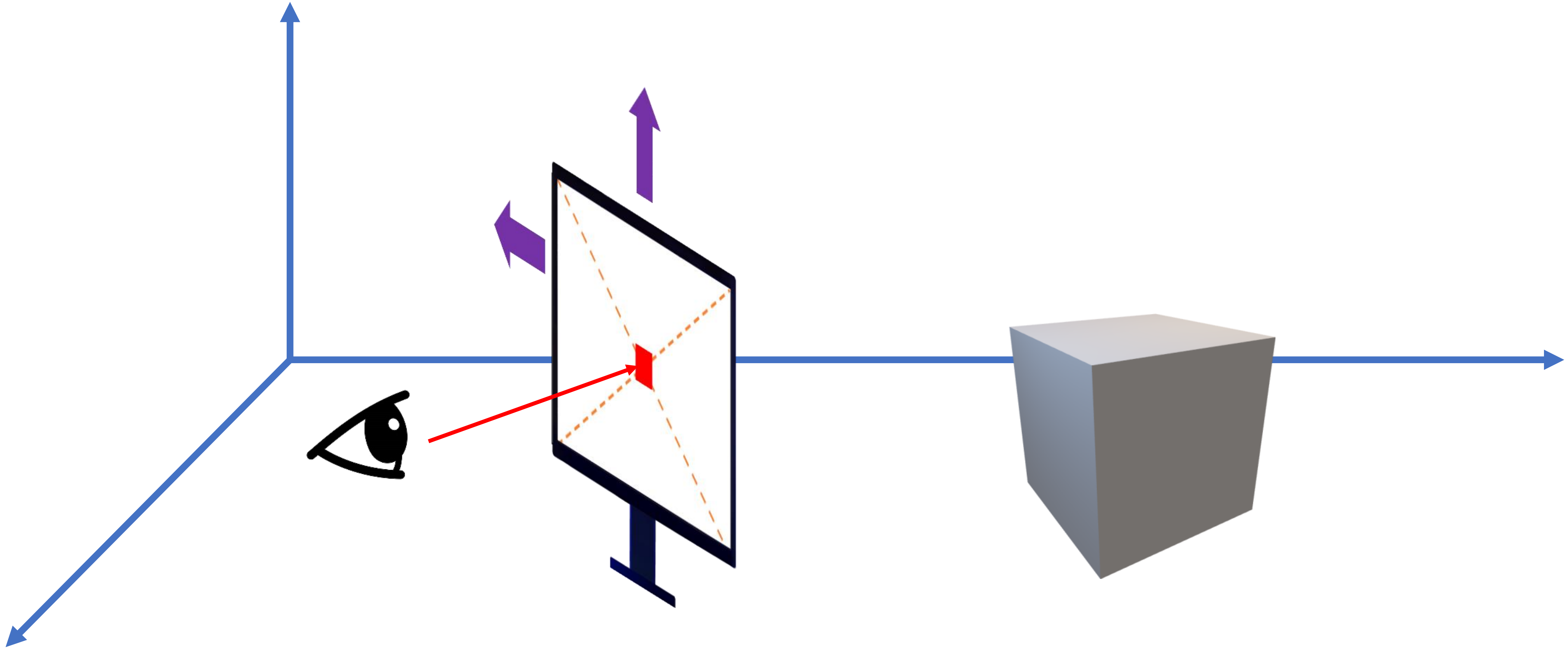
Orienting the screen (up vector)

```
up[None] = [0.0, 1.0, 0.0]
```



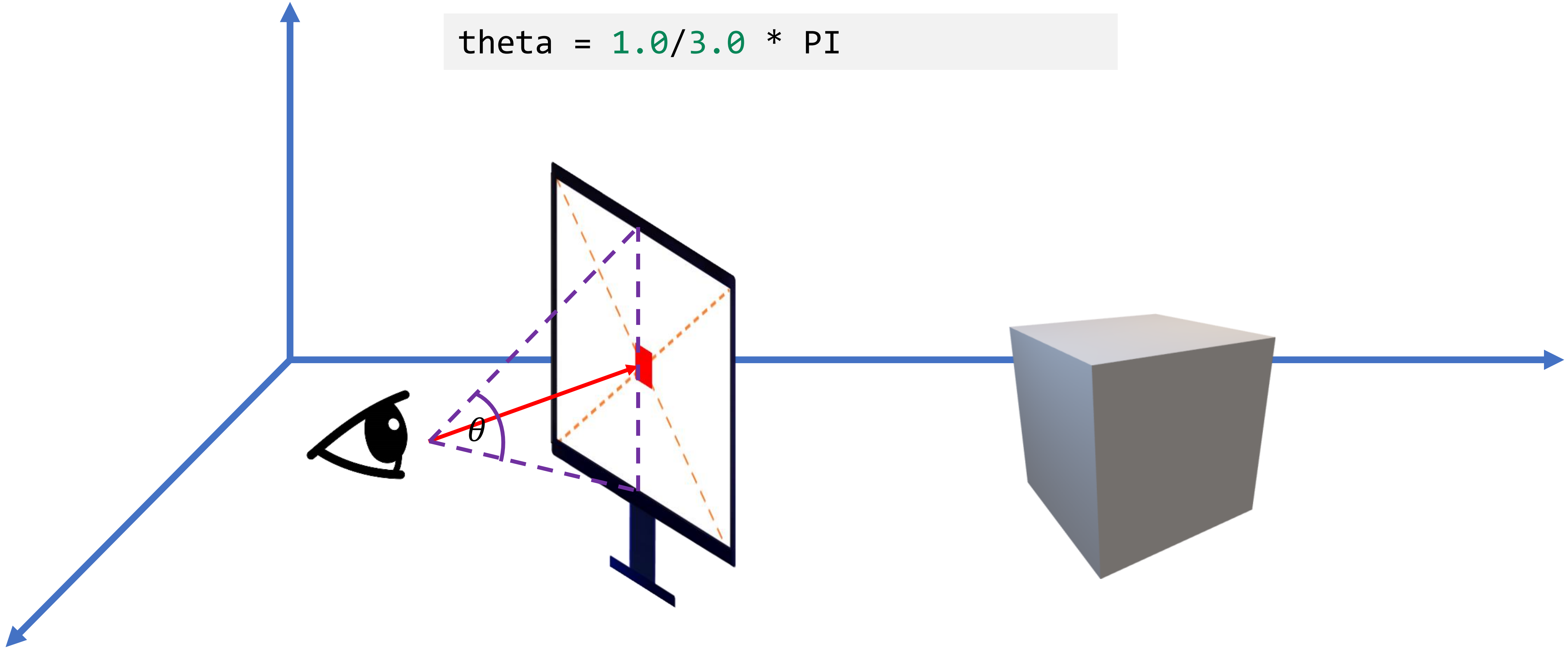
Problem:

3. the size of the screen is not decided

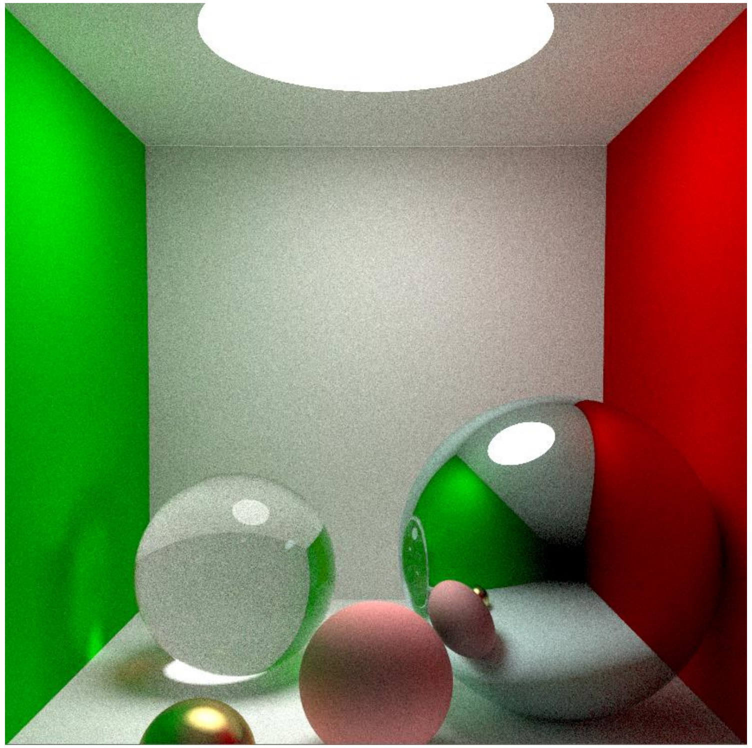


Sizing the screen (using field of view)

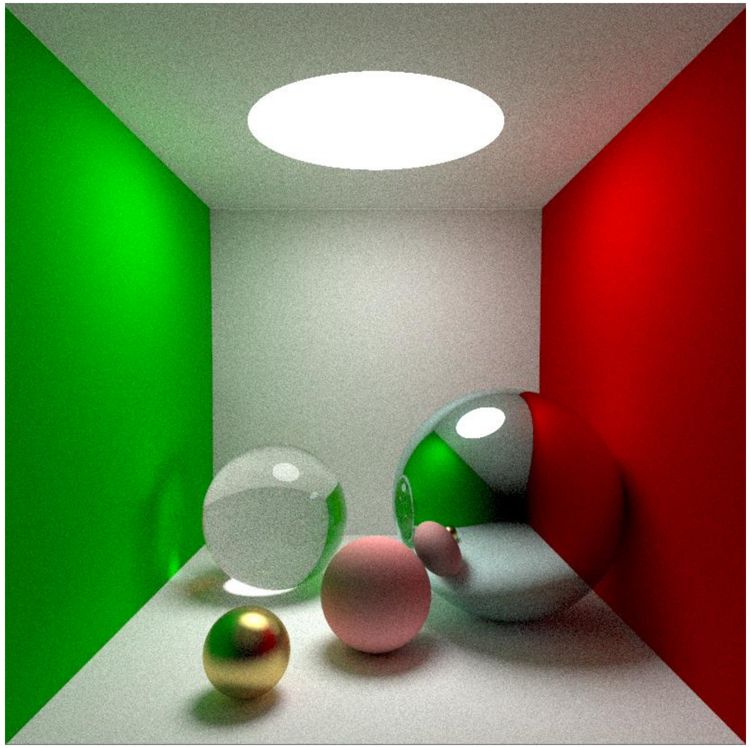
```
theta = 1.0/3.0 * PI
```



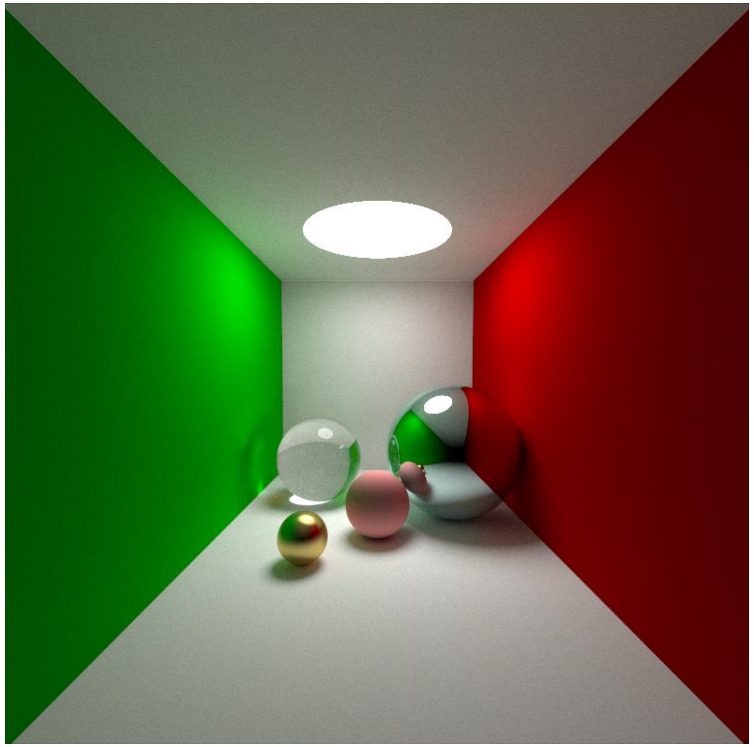
Field of view



fov=45°



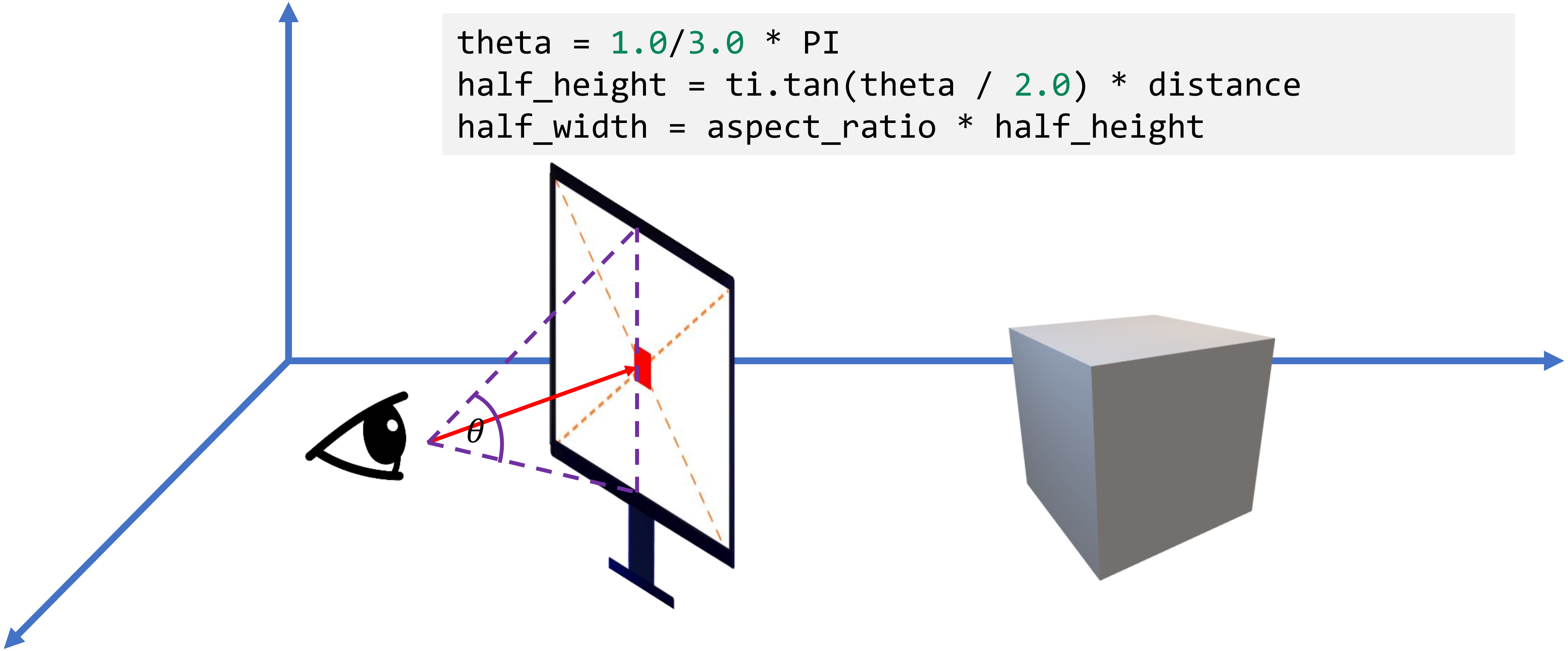
fov=60°



fov=90°

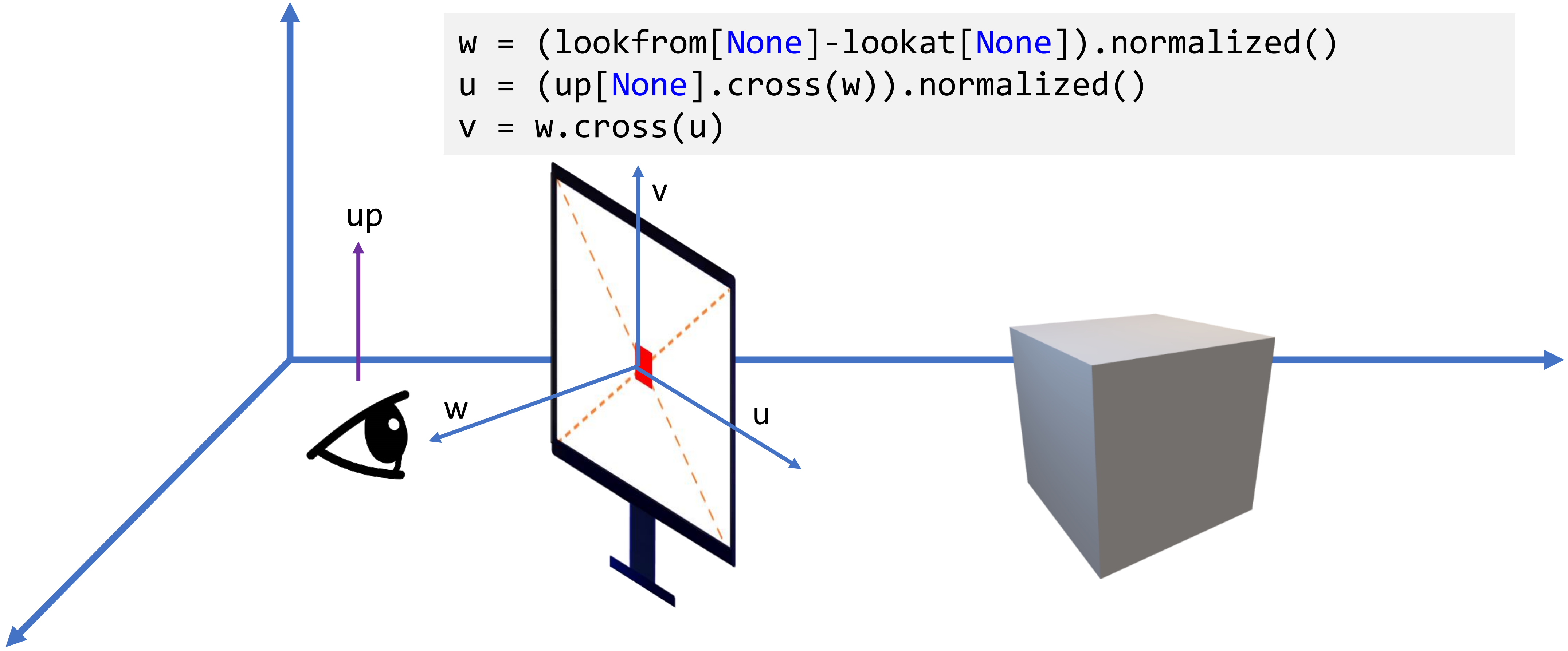
Sizing the screen (using field of view)

```
theta = 1.0/3.0 * PI
half_height = ti.tan(theta / 2.0) * distance
half_width = aspect_ratio * half_height
```



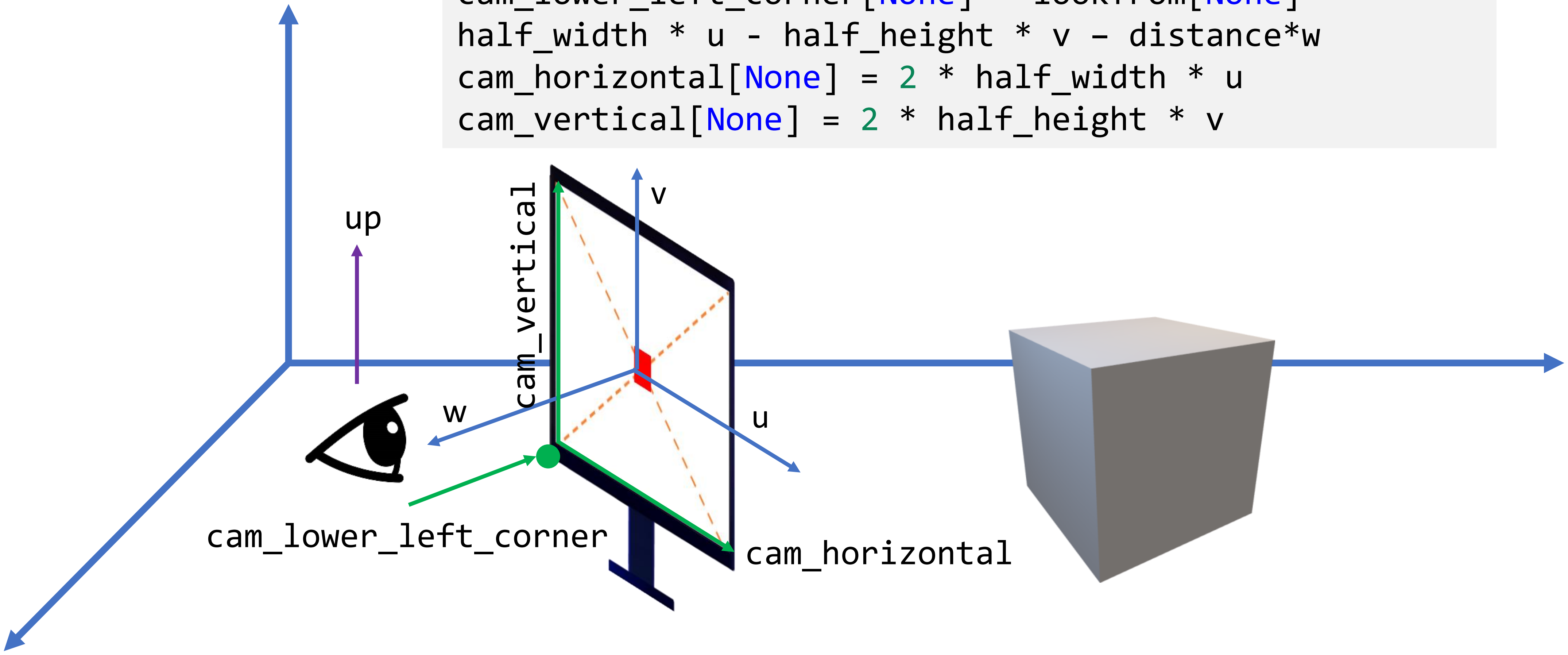
Sizing the screen (using field of view)

```
w = (lookfrom[None]-lookat[None]).normalized()
u = (up[None].cross(w)).normalized()
v = w.cross(u)
```

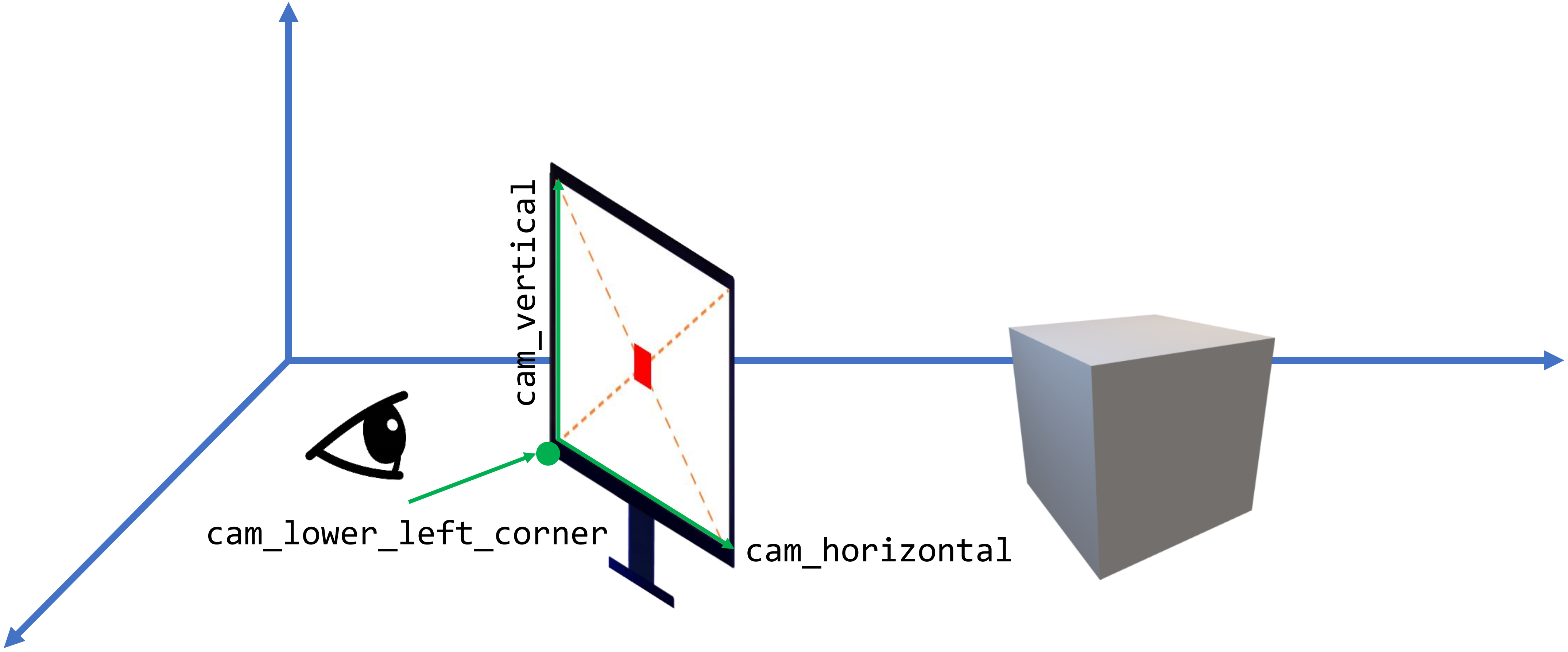


Sizing the screen (using field of view)

```
cam_lower_left_corner[None] = lookfrom[None] -
half_width * u - half_height * v – distance*w
cam_horizontal[None] = 2 * half_width * u
cam_vertical[None] = 2 * half_height * v
```

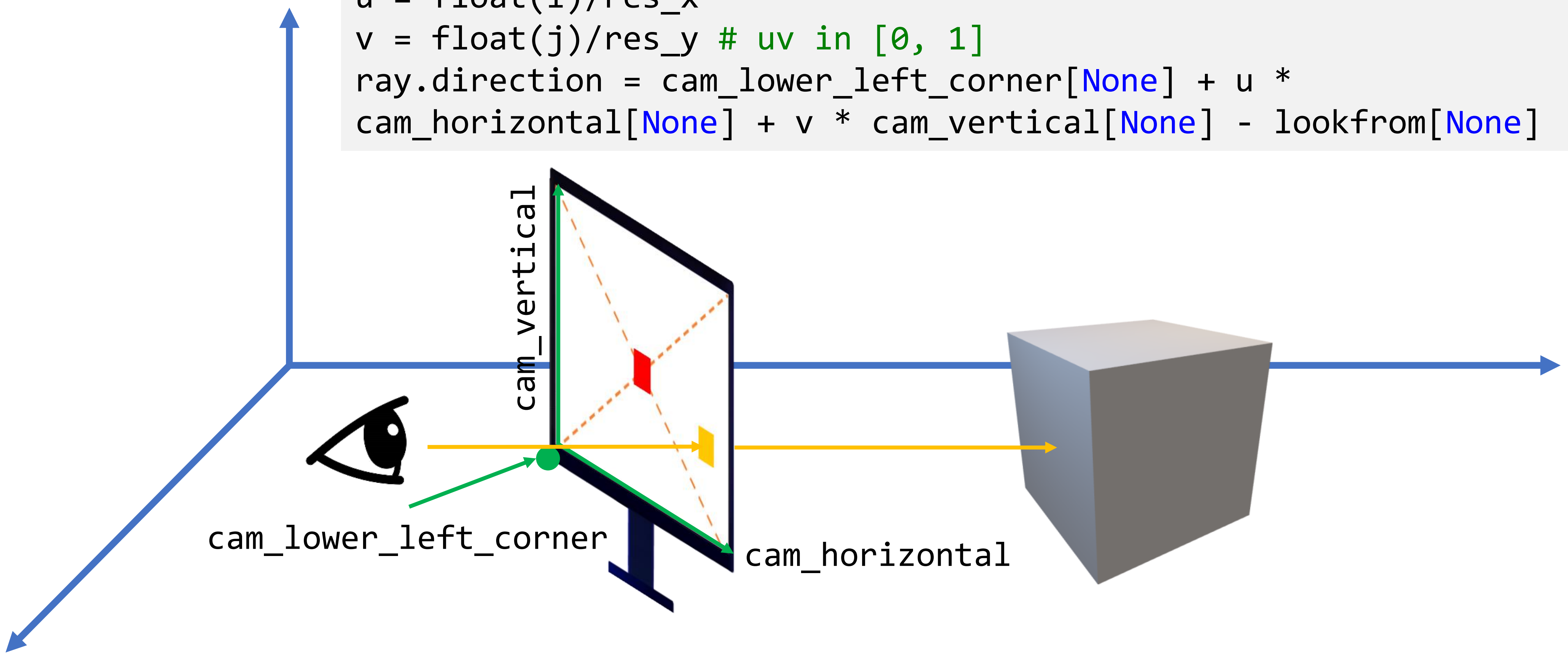


Getting ready to cast a ray!



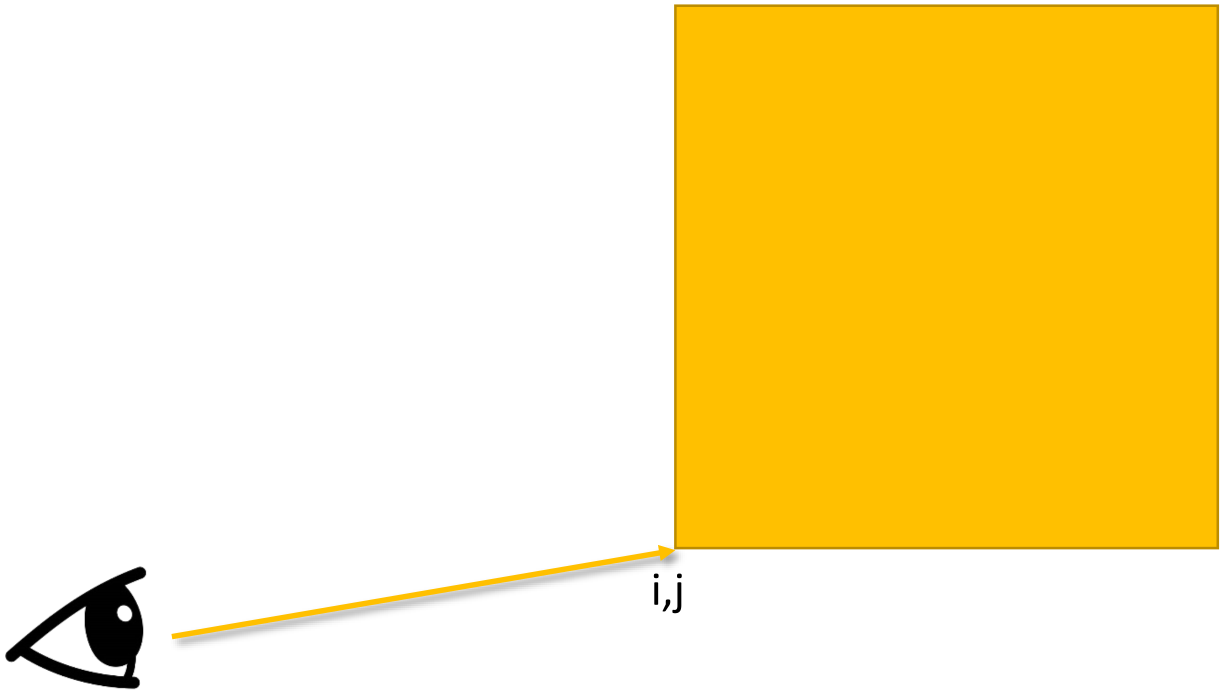
Ray-casting!

```
u = float(i)/res_x
v = float(j)/res_y # uv in [0, 1]
ray.direction = cam_lower_left_corner[None] + u *
cam_horizontal[None] + v * cam_vertical[None] - lookfrom[None]
```



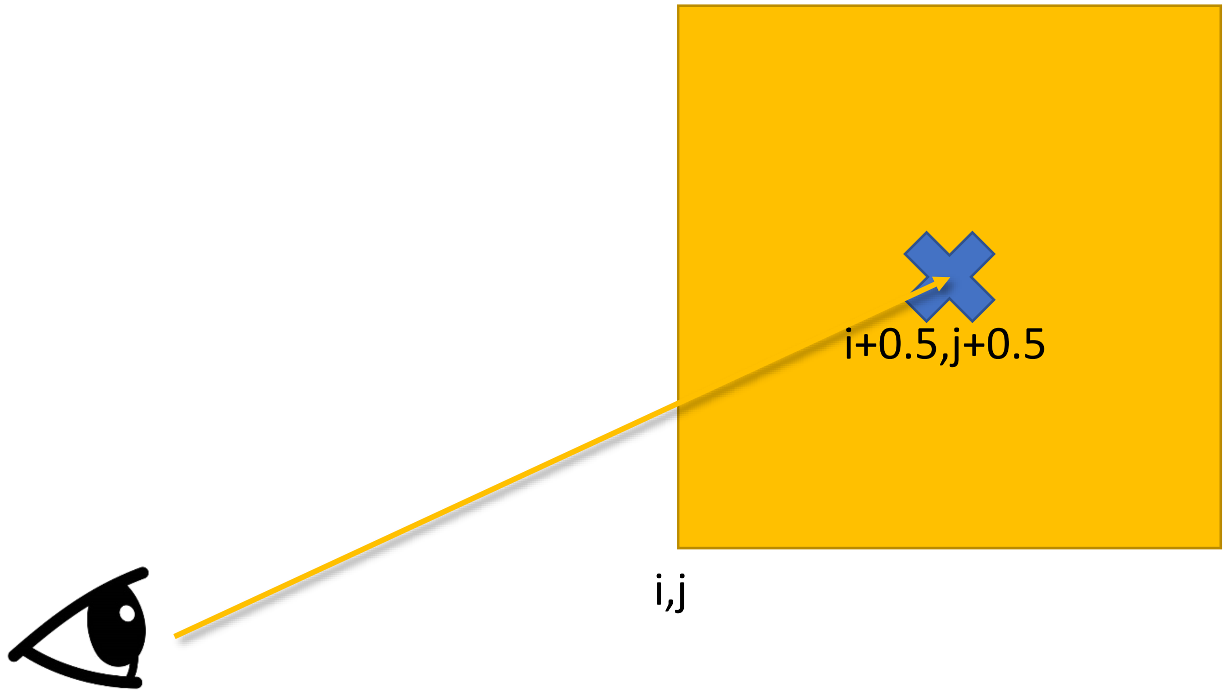
A pixel has its size as well

```
u = float(i     )/res_x
v = float(j     )/res_y # uv in [0, 1)
```

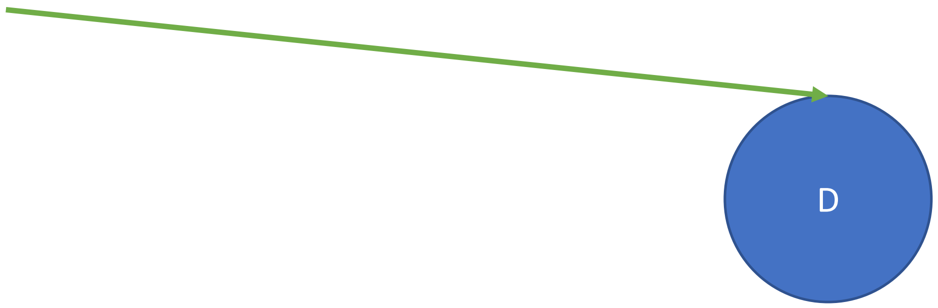


A pixel has its size as well

```
u = float(i+0.5)/res_x
v = float(j+0.5)/res_y # uv in (0, 1)
```

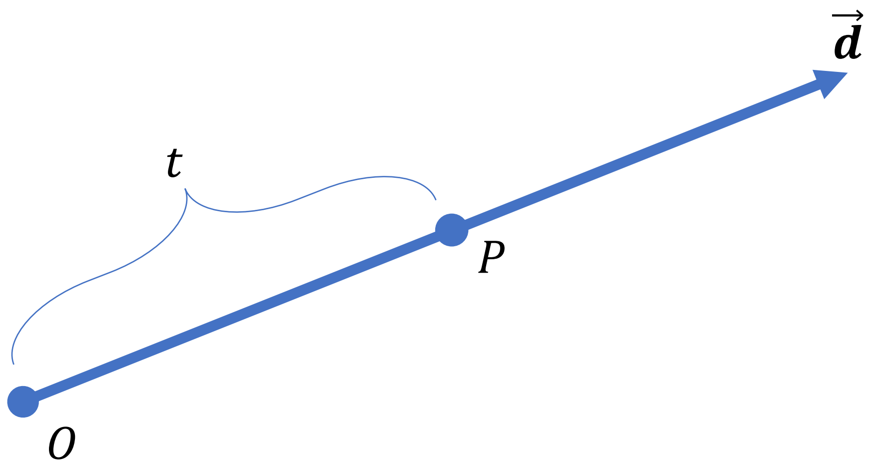


Ray-object intersection



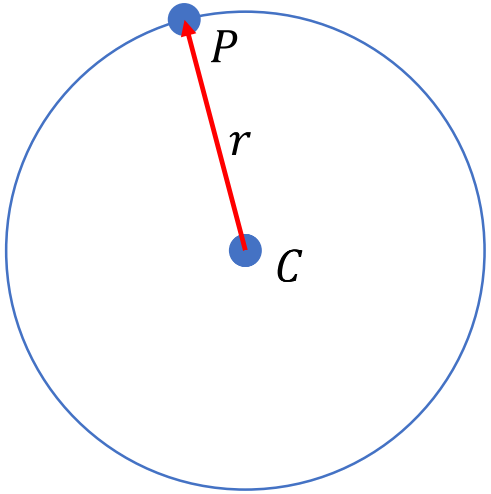
A point on a ray

- $P = O + t\vec{\boldsymbol{d}}$



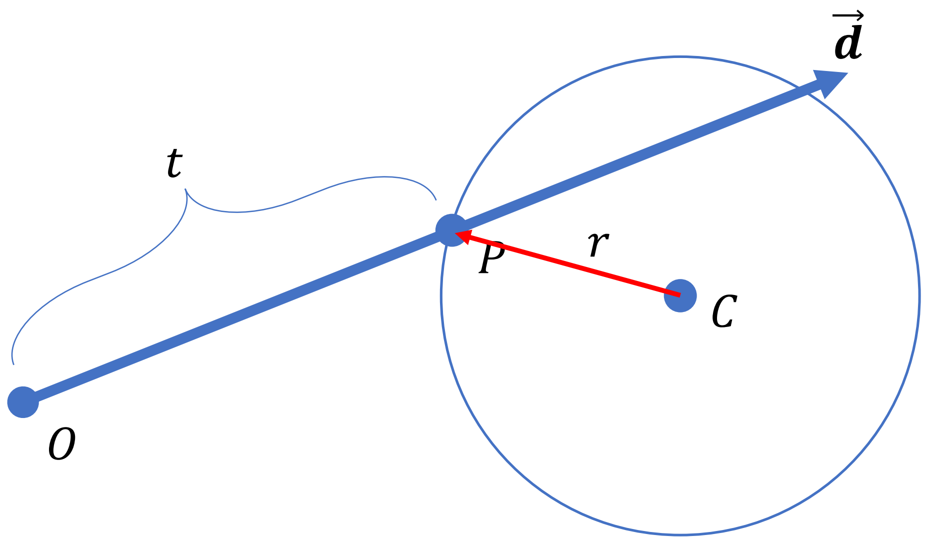
Definition of a sphere

- $\|P - C\|^2 - r^2 = 0$



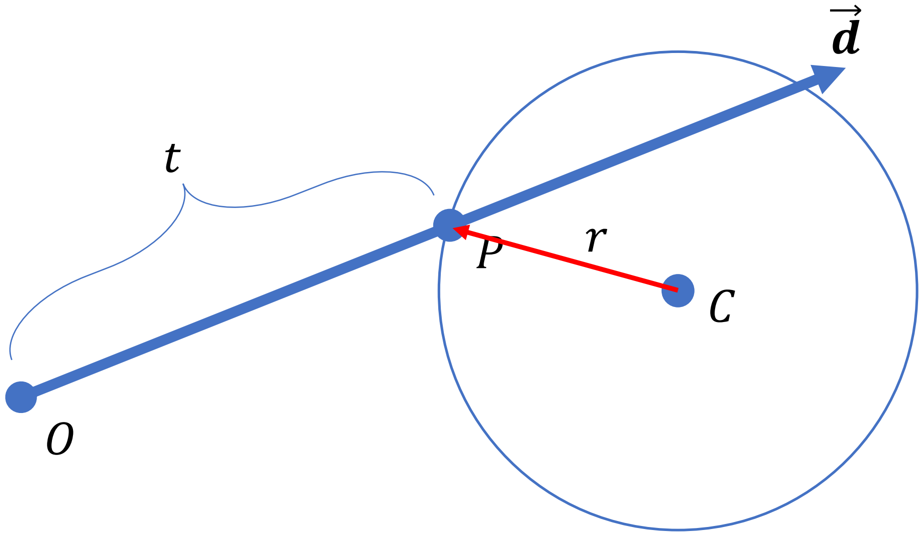
Let's find the intersection (if any)

- $\left\| O + t\vec{\boldsymbol{d}} - \mathrm{C} \right\|^2 - \mathrm{r}^2 = 0$



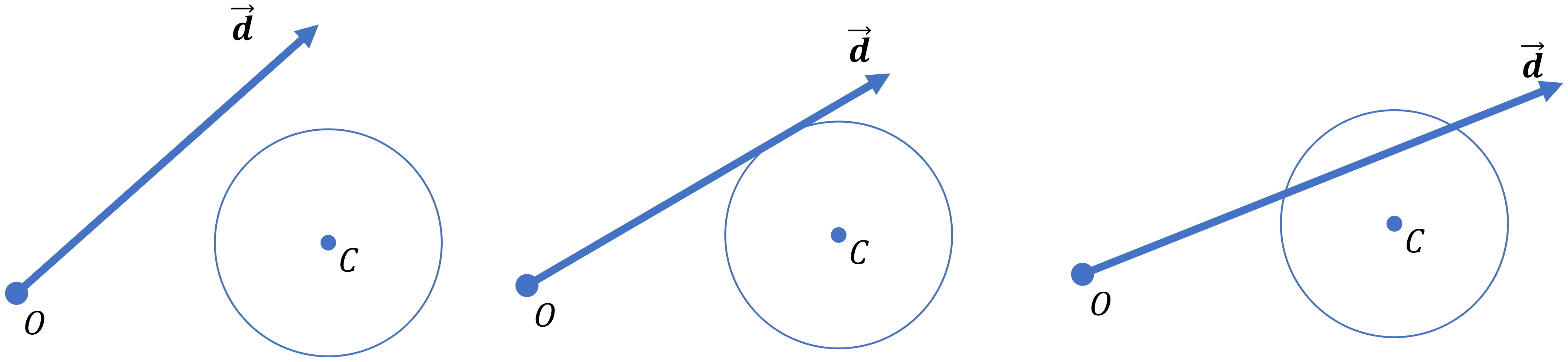
Let’s find the intersection (if any)

- $\left\|O + t\vec{\boldsymbol{d}} - \mathrm{C}\right\|^2 - \mathrm{r}^2 = 0$
- ==> $d^T d t^2 + 2d^T(O - C)t + (O - C)^{\mathrm{T}}(O - C) - \mathrm{r}^2 = 0$
- ==> $at^2 + bt + c = 0$



Let’s find the intersection (if any)

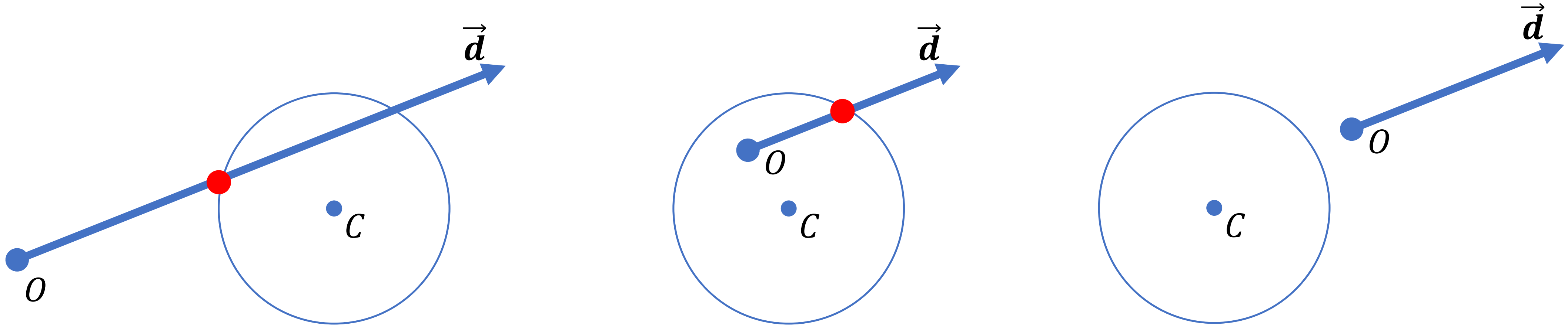
- $at^2 + bt + c = 0$ ➔ $t = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$



$b^2 - 4ac < 0$ $b^2 - 4ac = 0$ $b^2 - 4ac > 0$

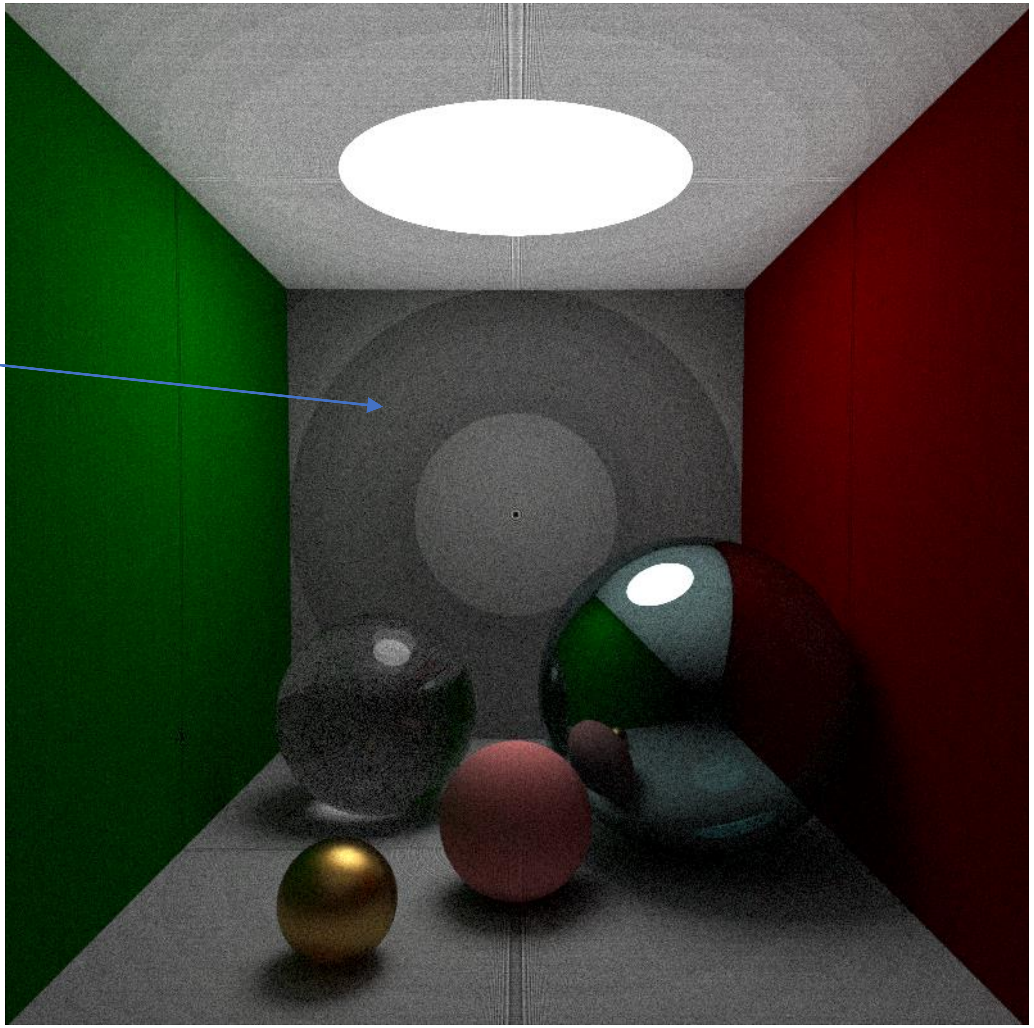
Find the smallest POSITIVE root

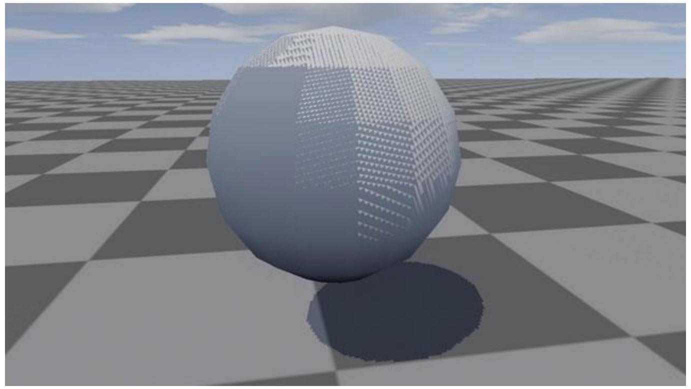
- $at^2 + bt + c = 0 \rightarrow t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, t > 0$



Taking the smallest positive root as a hit

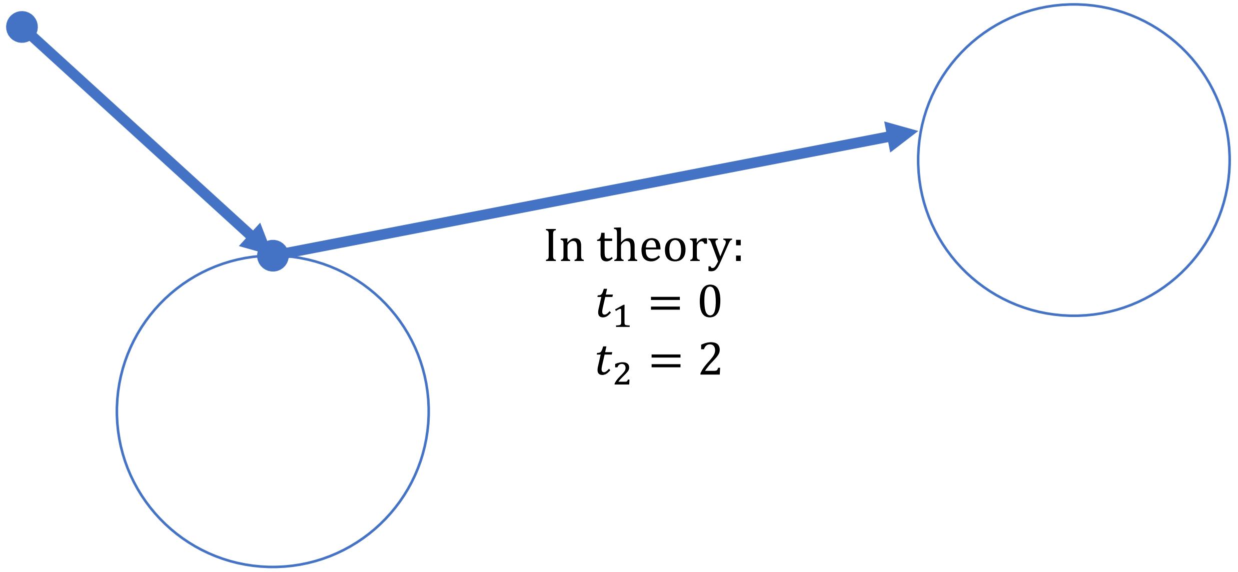
Shadow Acne





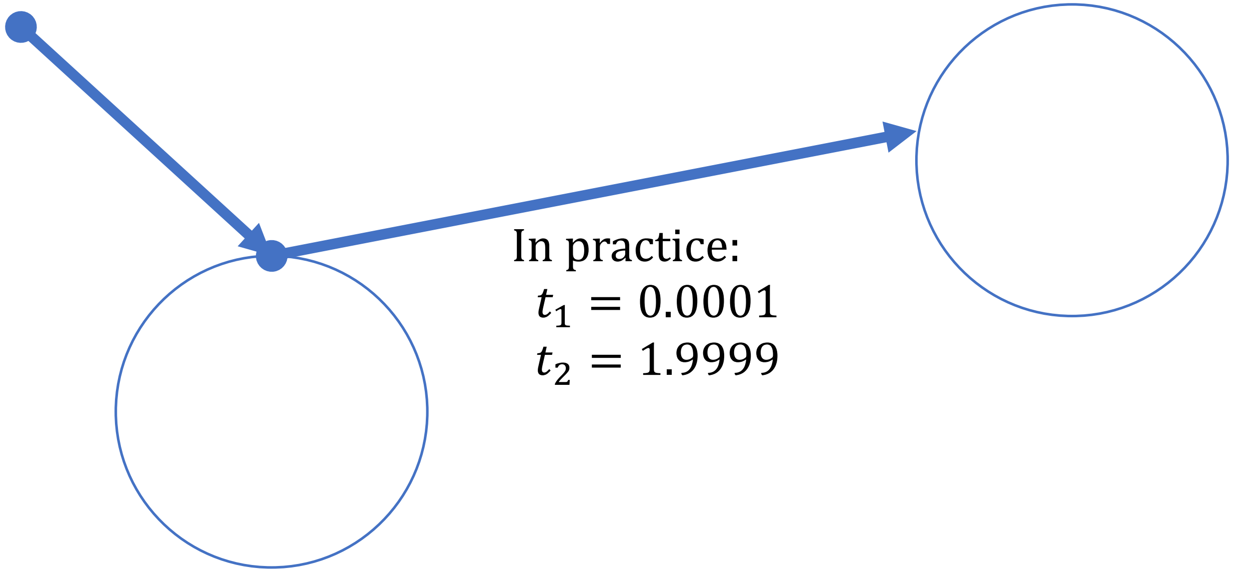
Find the smallest POSITIVE root

- You might want a slightly “more positive” number than zero
  - $at^2 + bt + c = 0 \Rightarrow t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, t > \epsilon$
  - For instance: $\epsilon = 0.001$



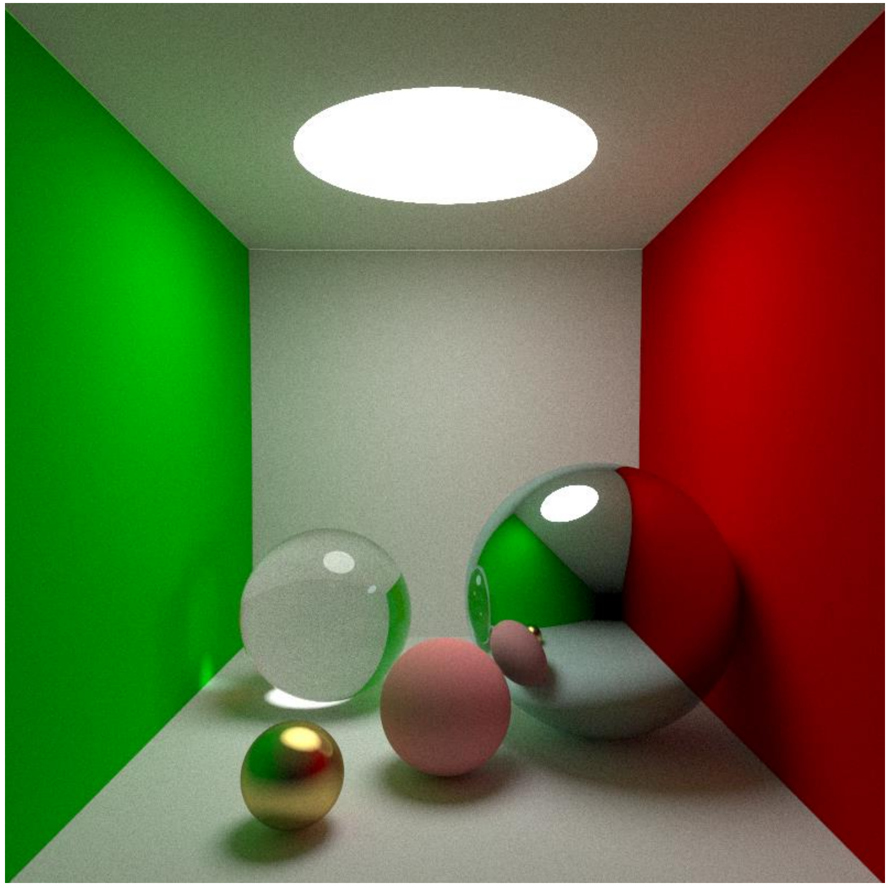
Find the smallest POSITIVE root

- You might want a slightly more positive number than zero
  - $at^2 + bt + c = 0 \rightarrow t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, t > \epsilon$
  - For instance: $\epsilon = 0.001$



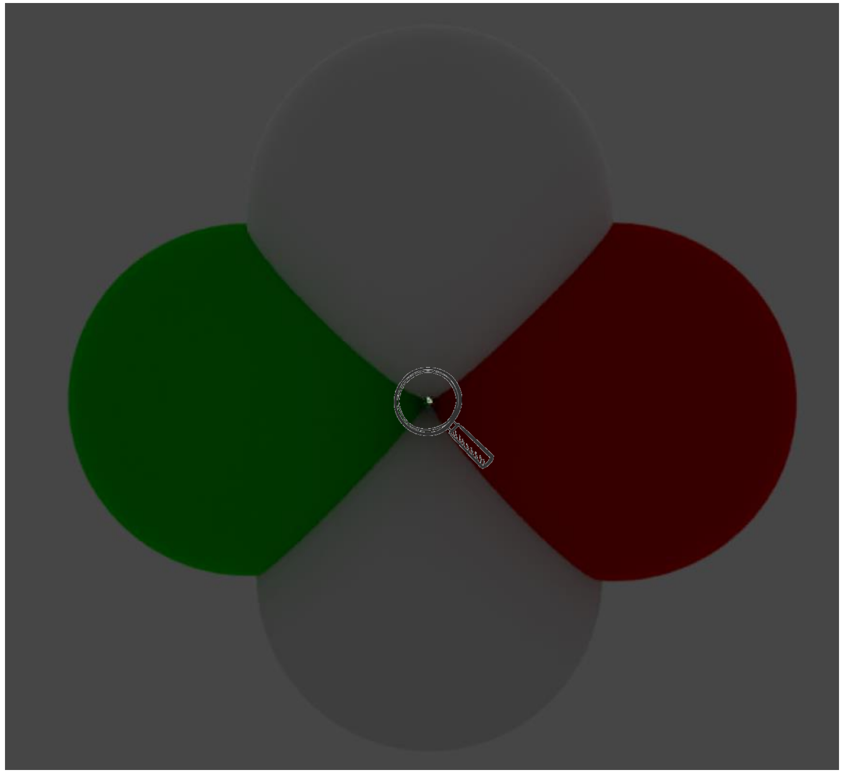
Build your Cornell box

- Wait, How do we find intersections between rays and planes?



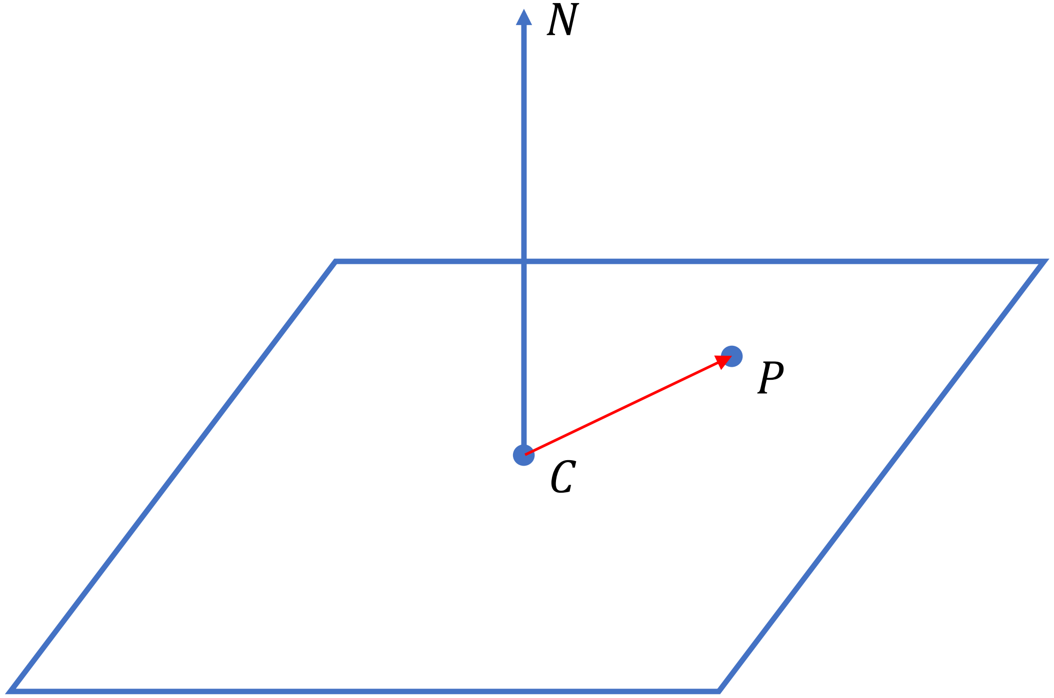
Build your Cornell box

- The Cornell box in our released repo is made of spheres :P



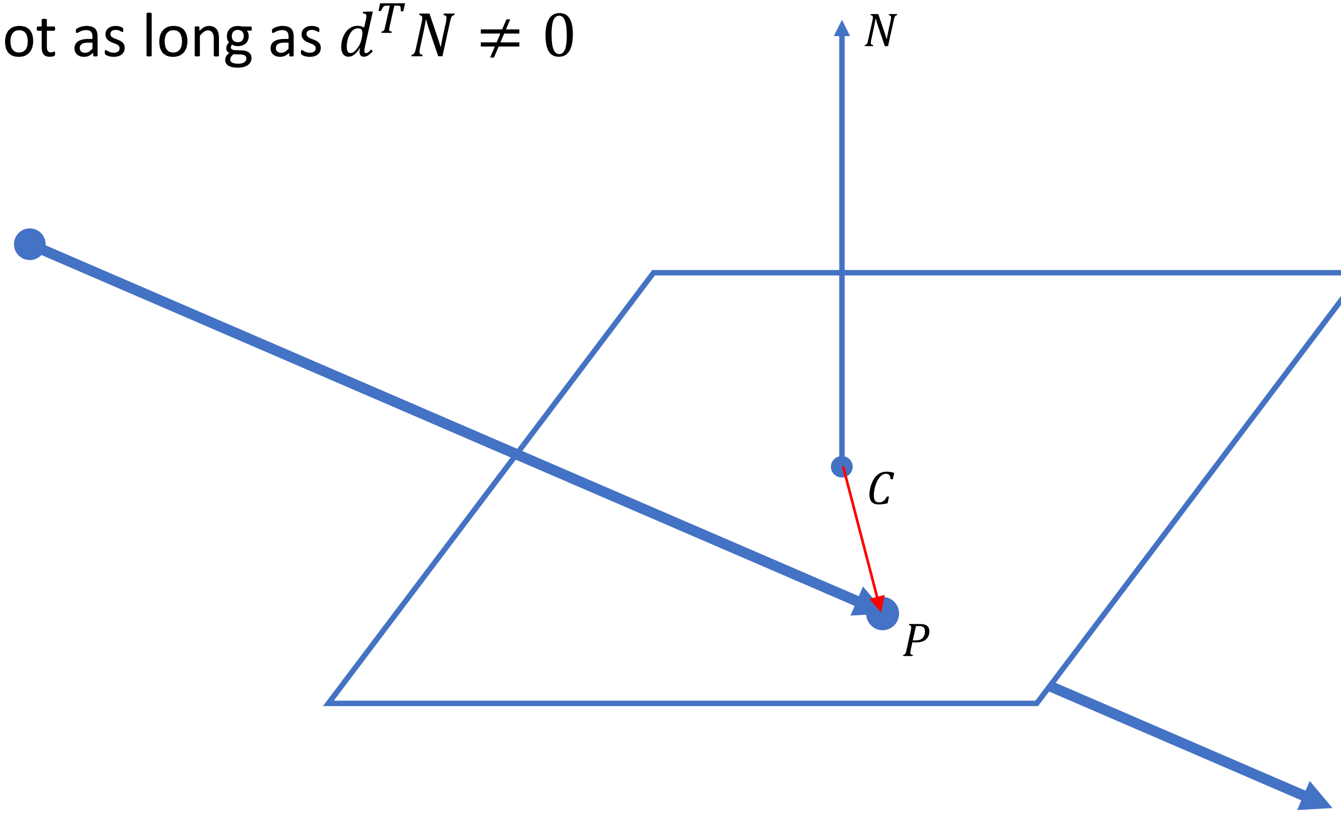
Well, if you still want a ray-plane intersection

- The definition of a plane:
  - $(P - C)^T N = 0$



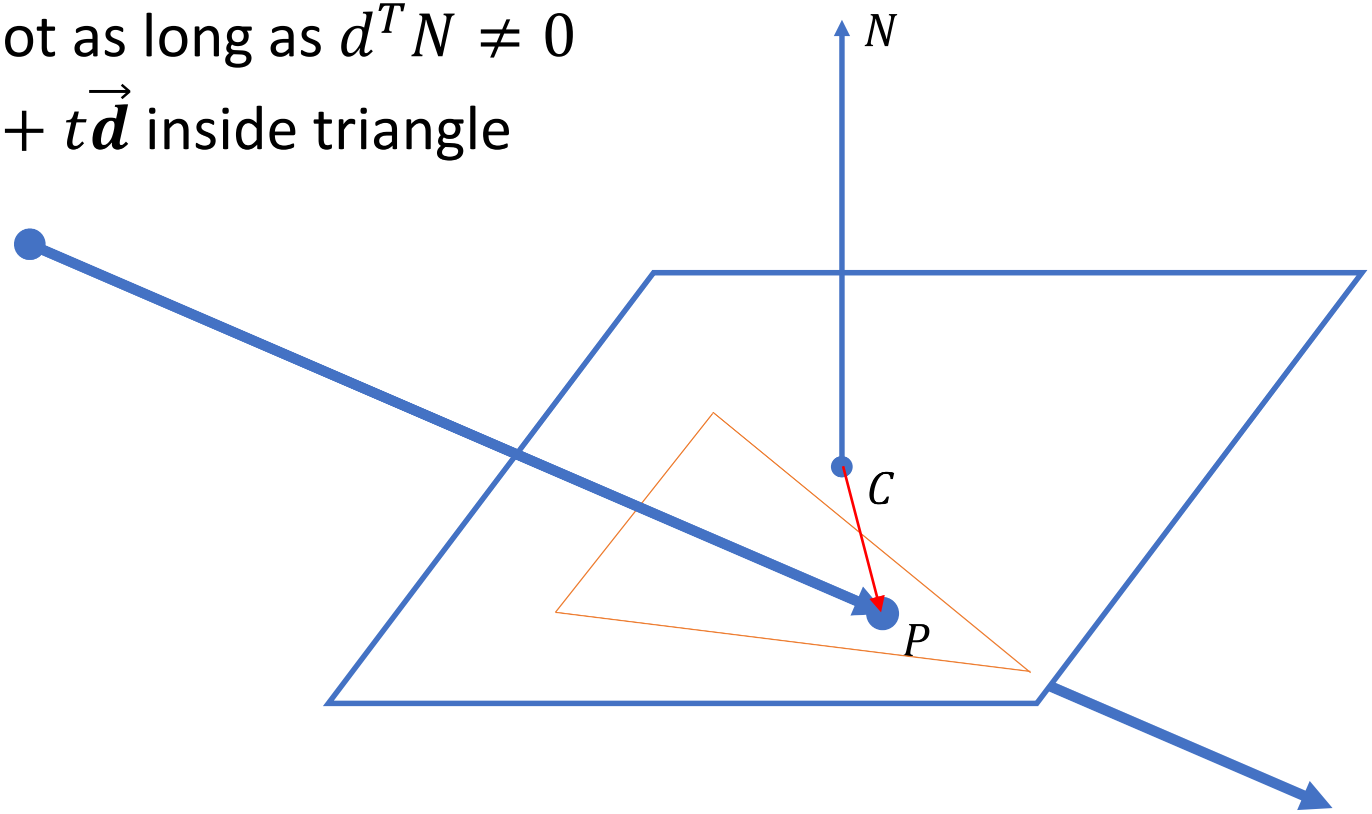
Let’s find the intersection (if any)

- $\left(O + t\vec{\boldsymbol{d}} - C\right)^T N = 0$
  - Has a root as long as $d^T N \neq 0$



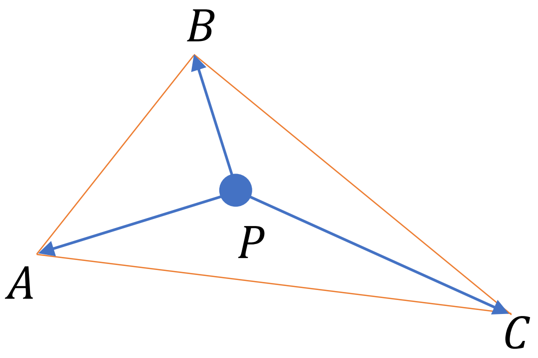
How about a ray-triangle intersection?

- $\left(O + t\vec{\boldsymbol{d}} - C\right)^T N = 0$
  - Has a root as long as $d^T N \neq 0$
  - AND: $O + t\vec{\boldsymbol{d}}$ inside triangle



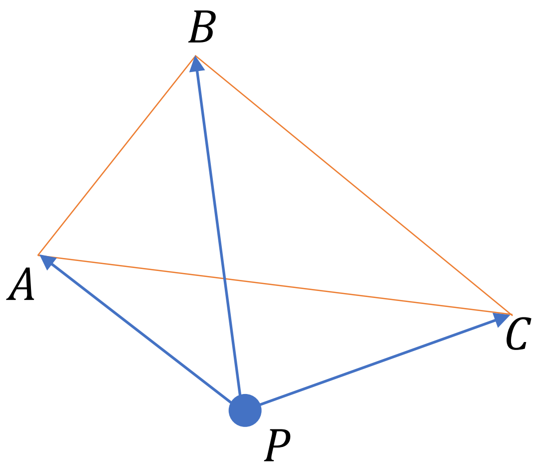
Am I inside or outside?

- $S_{\Delta PAB} + S_{\Delta PBC} + S_{\Delta PCA} = S_{\Delta ABC}$ iff P is inside ΔABC
  - $S_{\Delta PAB} = \frac{1}{2} \|\mathrm{PA} \times PB\|$
  - Actually $[\mathrm{a}, \mathrm{b}, \mathrm{c}] = \left[\frac{S_{\Delta PBC}}{S_{\Delta ABC}}, \frac{S_{\Delta PCA}}{S_{\Delta ABC}}, \frac{S_{\Delta PAB}}{S_{\Delta ABC}}\right]$ is the Barycentric coordinate of P in ΔABC: $P = aA + bB + cC$



Inside

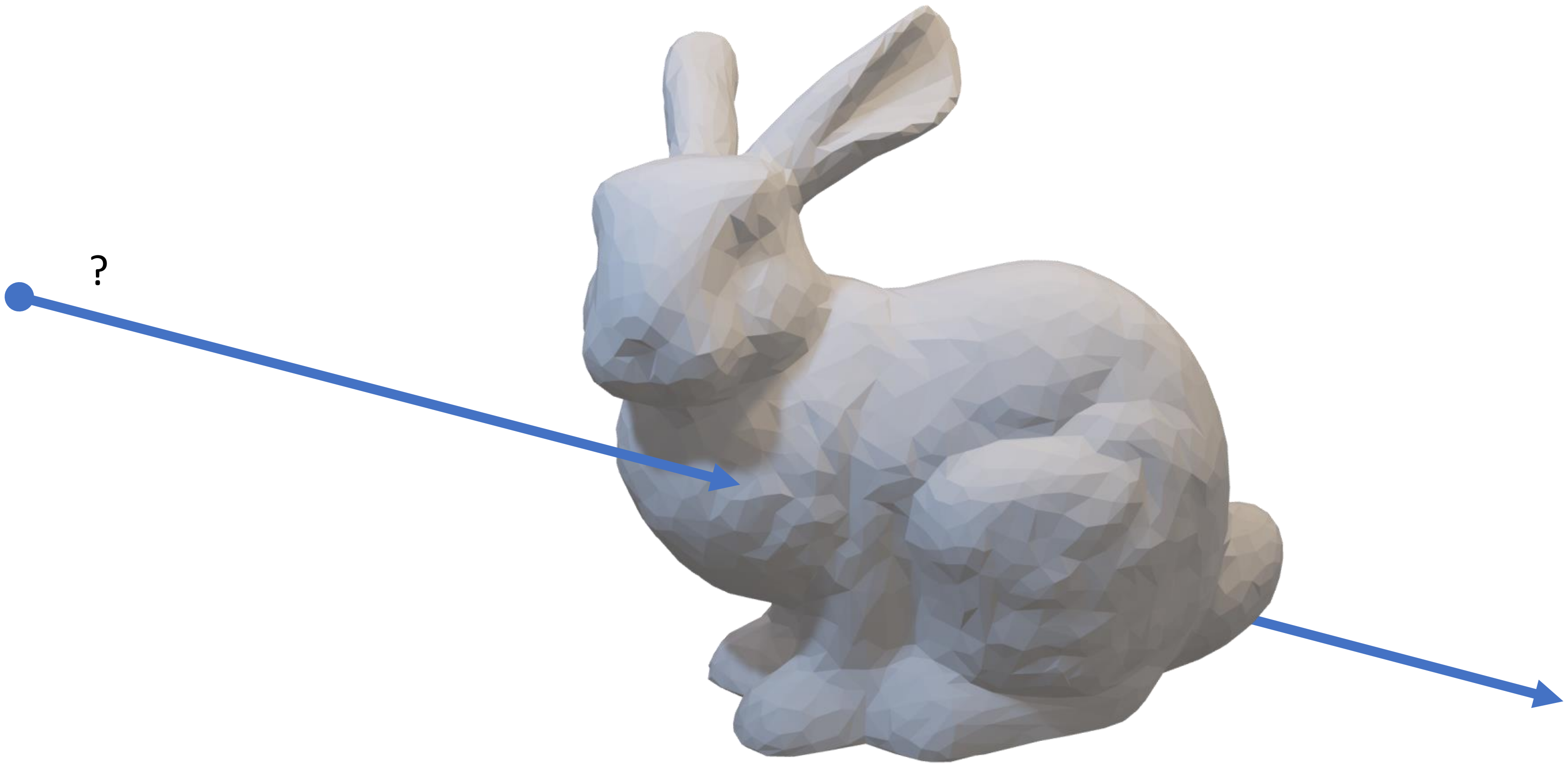
$S_{\Delta PAB} + S_{\Delta PBC} + S_{\Delta PCA} = S_{\Delta ABC}$



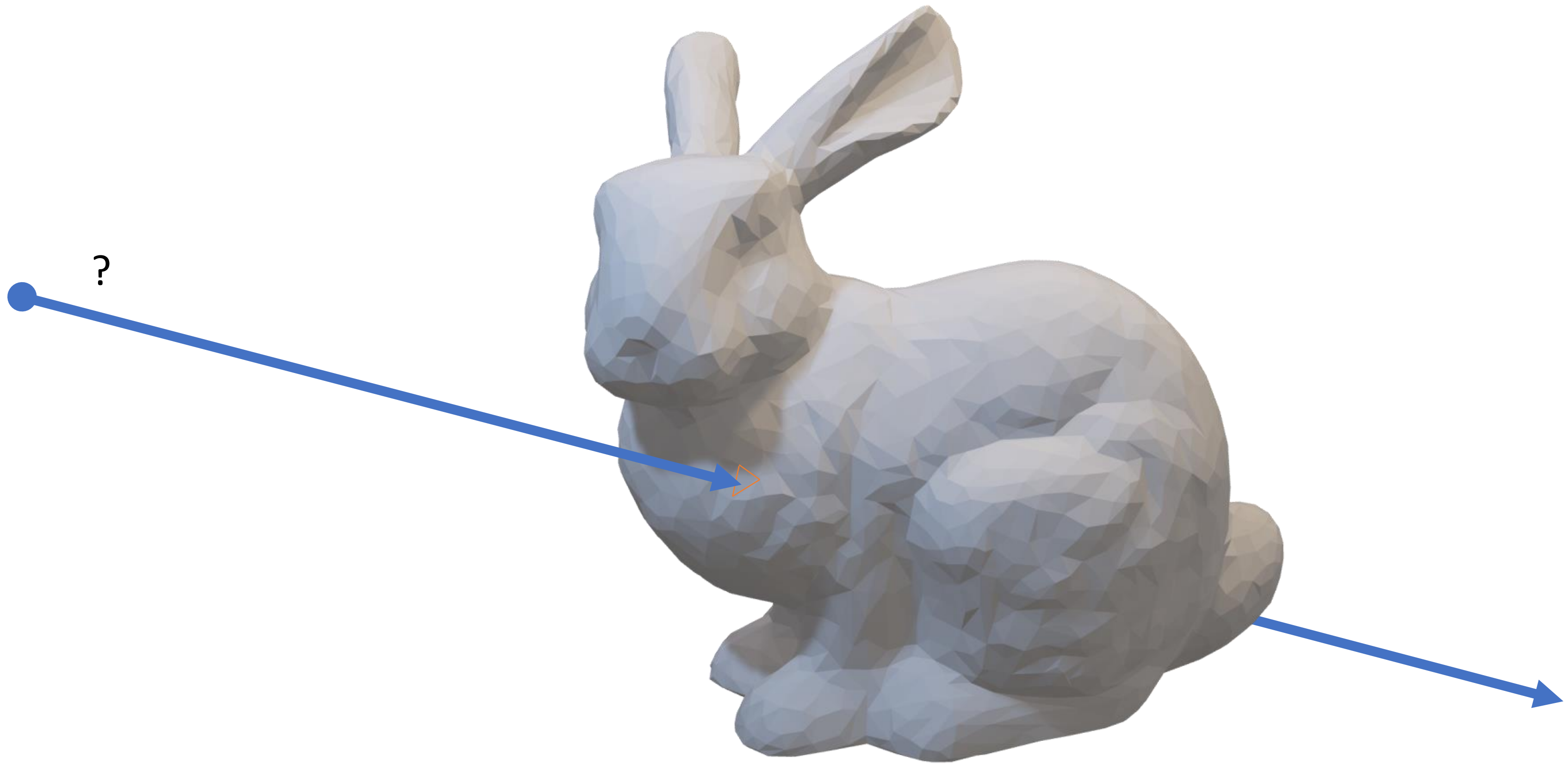
Outside

$S_{\Delta PAB} + S_{\Delta PBC} + S_{\Delta PCA} > S_{\Delta ABC}$

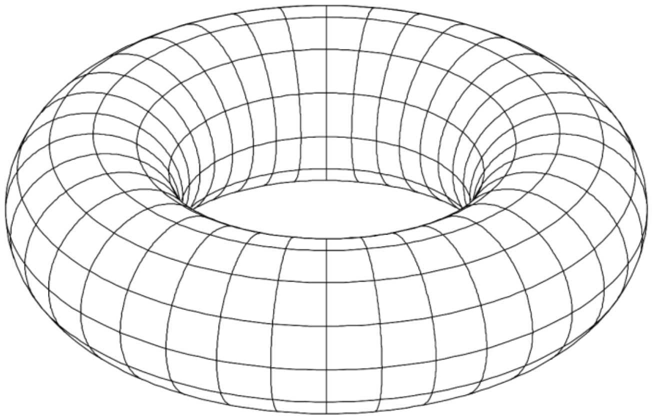
Why do we care about ray-triangle intersections?

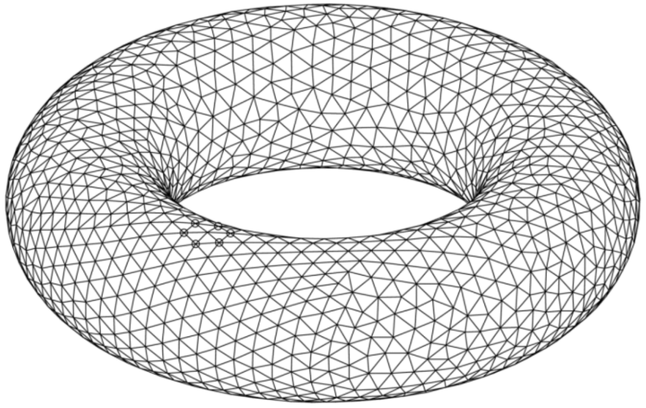


Polygon meshes are (usually) made of triangles



Ray-object intersection





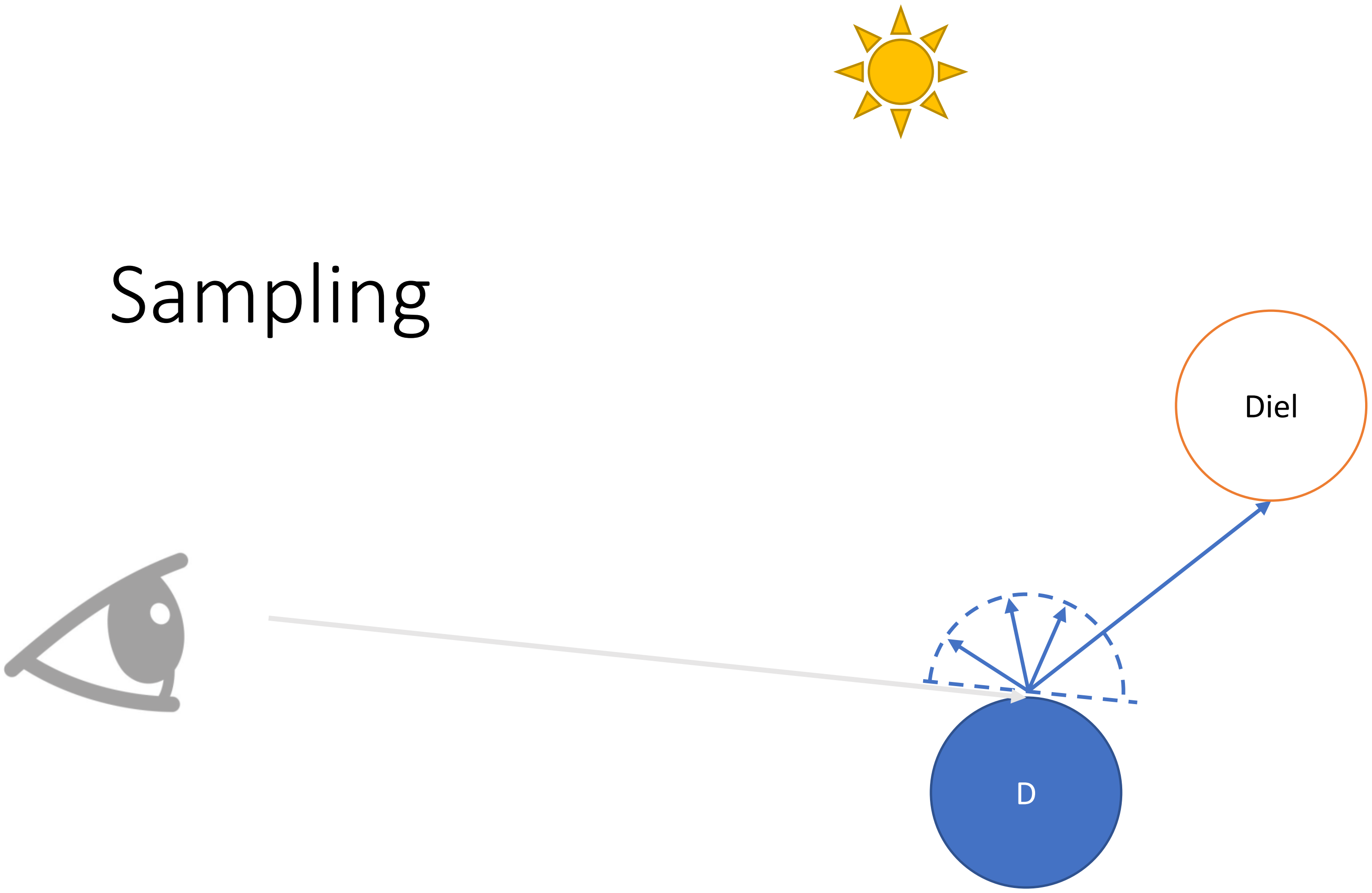
Implicit surfaces:

1. Find its surface definition
2. Plug the ray equation into the surface definition
3. Look for the smallest positive t

Polygonal surfaces:

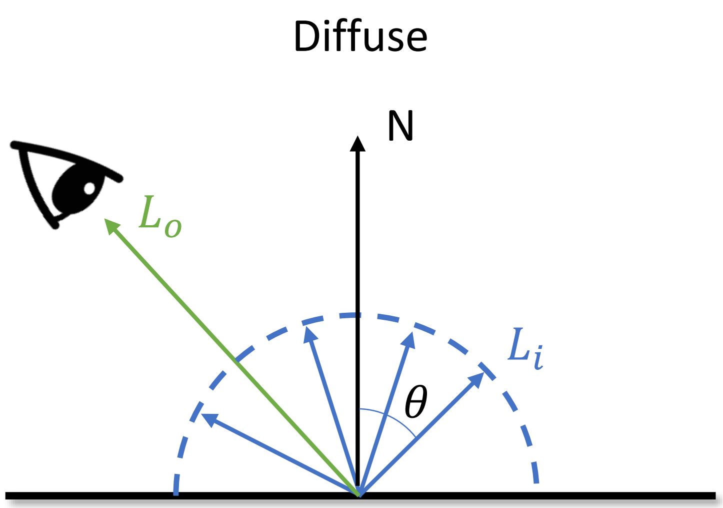
1. Loop over all its polygons (usually triangles)
2. Find the ray-polygon(triangle) intersection with the smallest positive t

Sampling



Sample the hemisphere uniformly

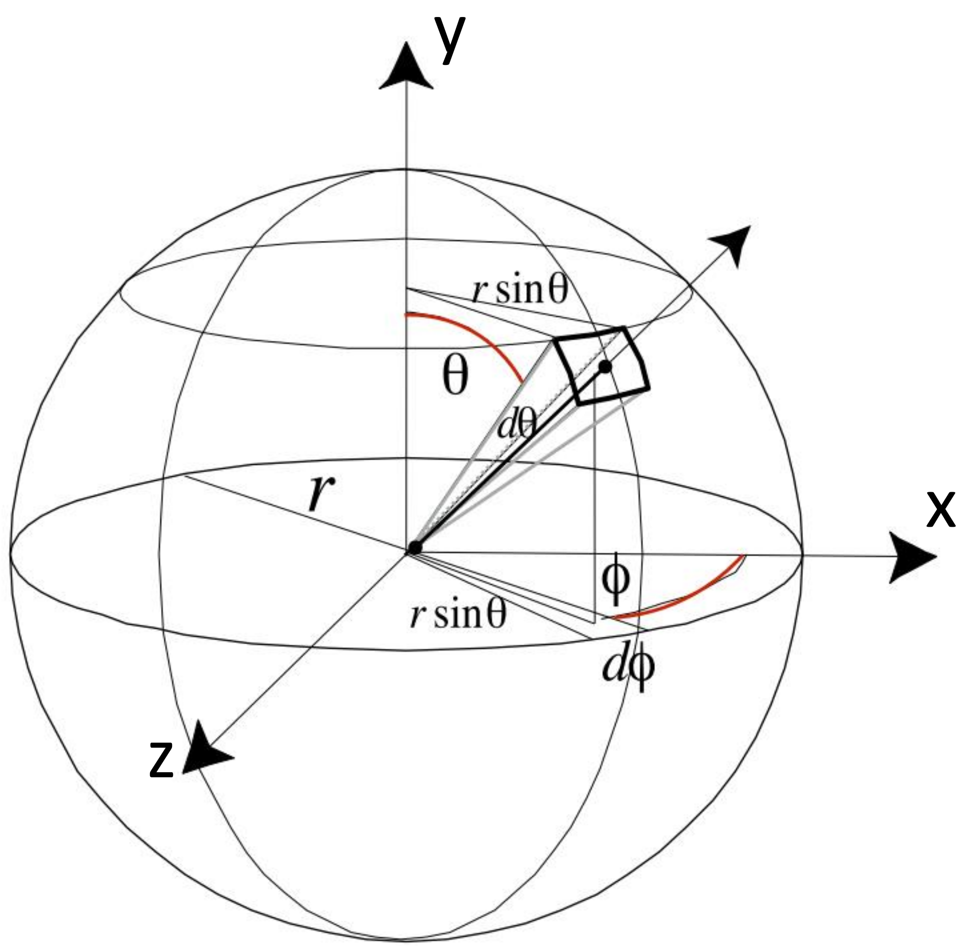
- What we want:
  - Sample the directions of rays uniformly
    - Find a uniform sampling on a sphere
    - Negate the direction if against the normal



$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

Cartesian coordinates v.s. polar coordinates

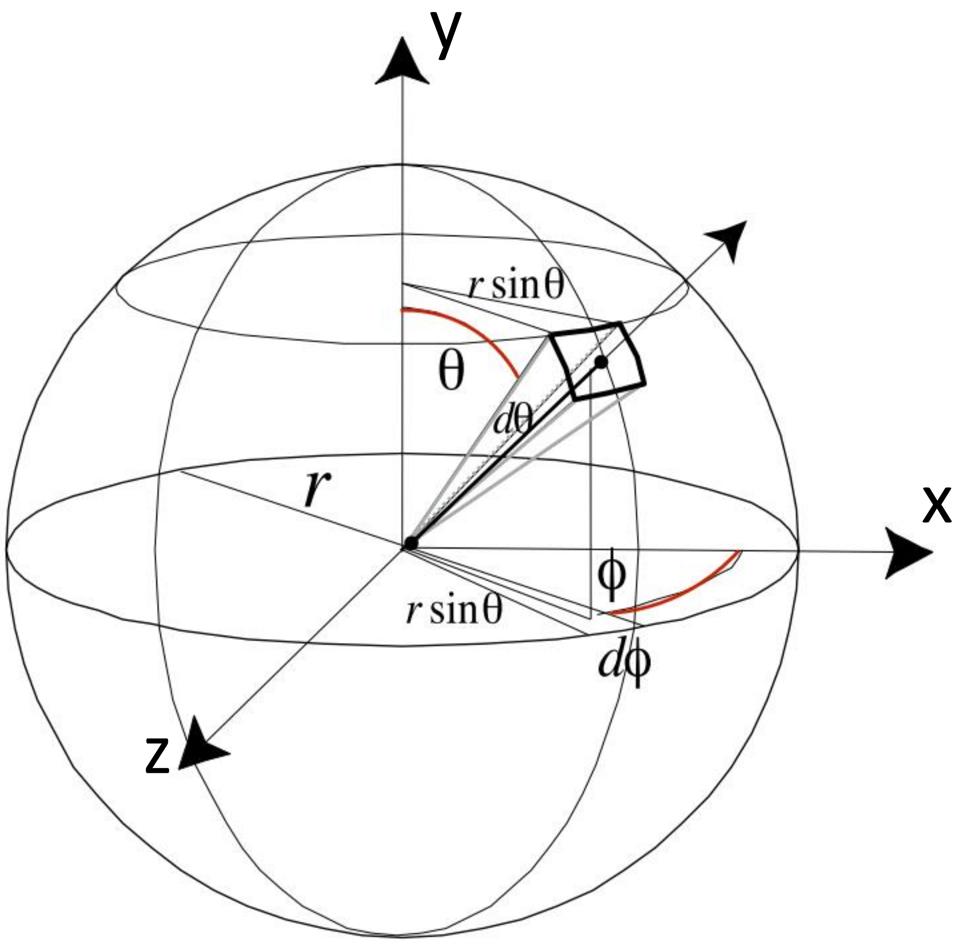
- Cartesian coordinates: $[x, y, z]$
- Polar coordinates: $[r, \phi, \theta]$



Cartesian coordinates v.s. polar coordinates

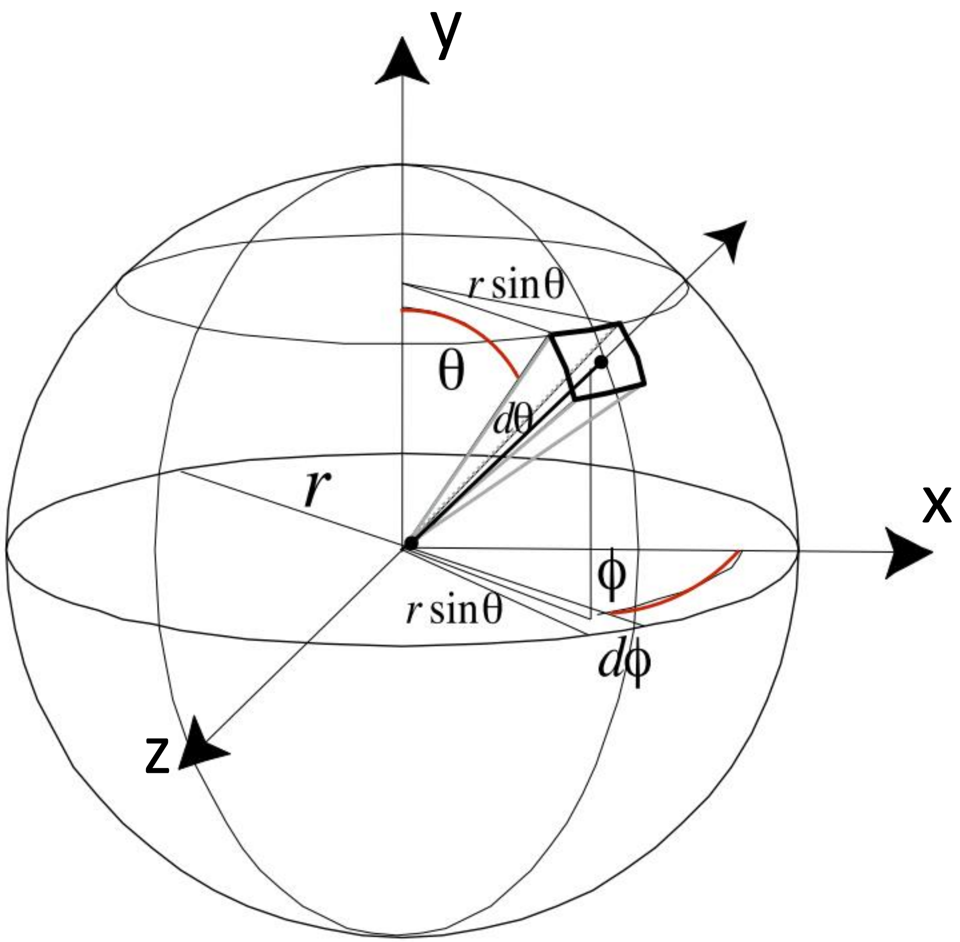
- Cartesian coordinates: $[x, y, z]$
- Polar coordinates: $[r, \phi, \theta]$

- $x = r * \cos(\phi) * \sin(\theta)$
- $z = r * \sin(\phi) * \sin(\theta)$
- $y = r * \cos(\theta)$



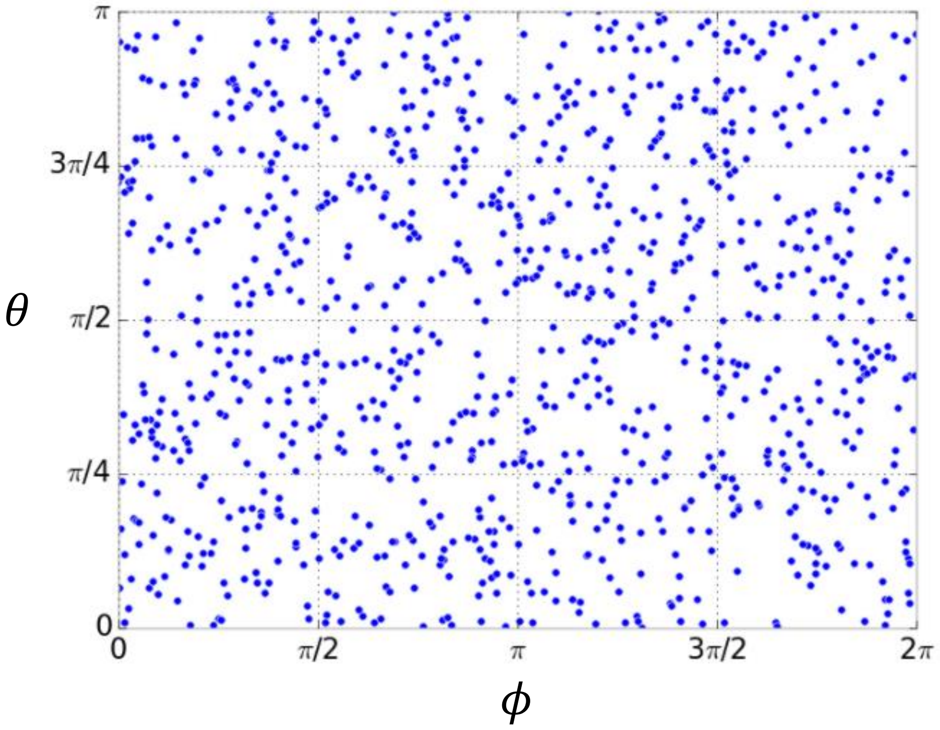
The first attempt:

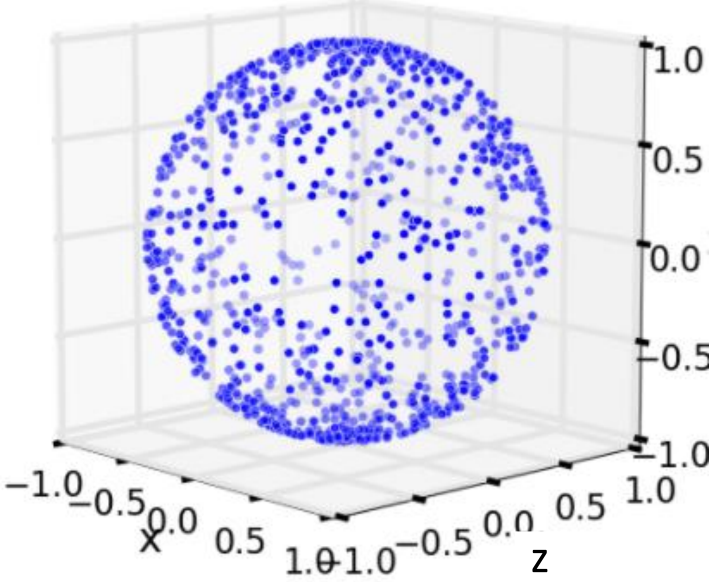
- $\phi = rand(0, 2\pi), \theta = rand(0, \pi), r = 1$



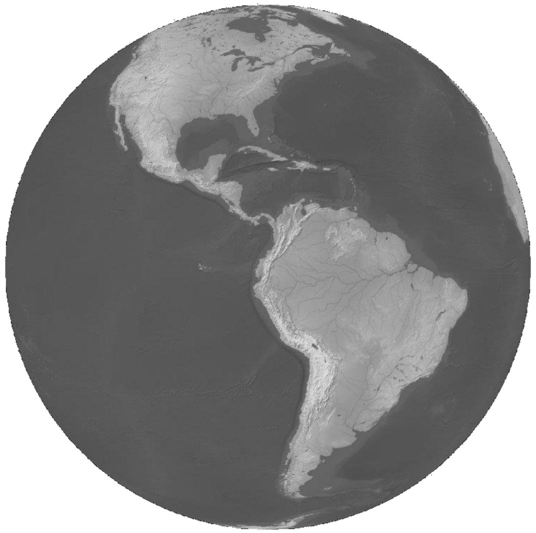
The first attempt:

- $\phi = rand(0, 2\pi), \theta = rand(0, \pi), r = 1$



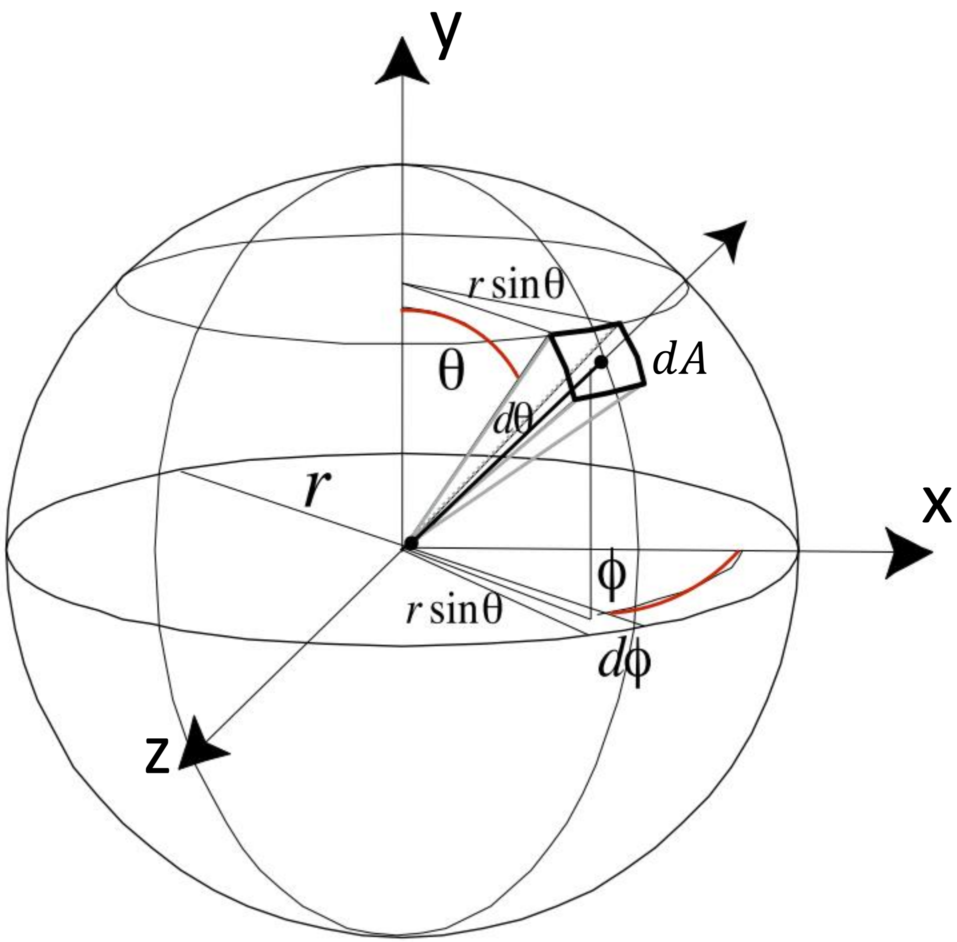


Why?



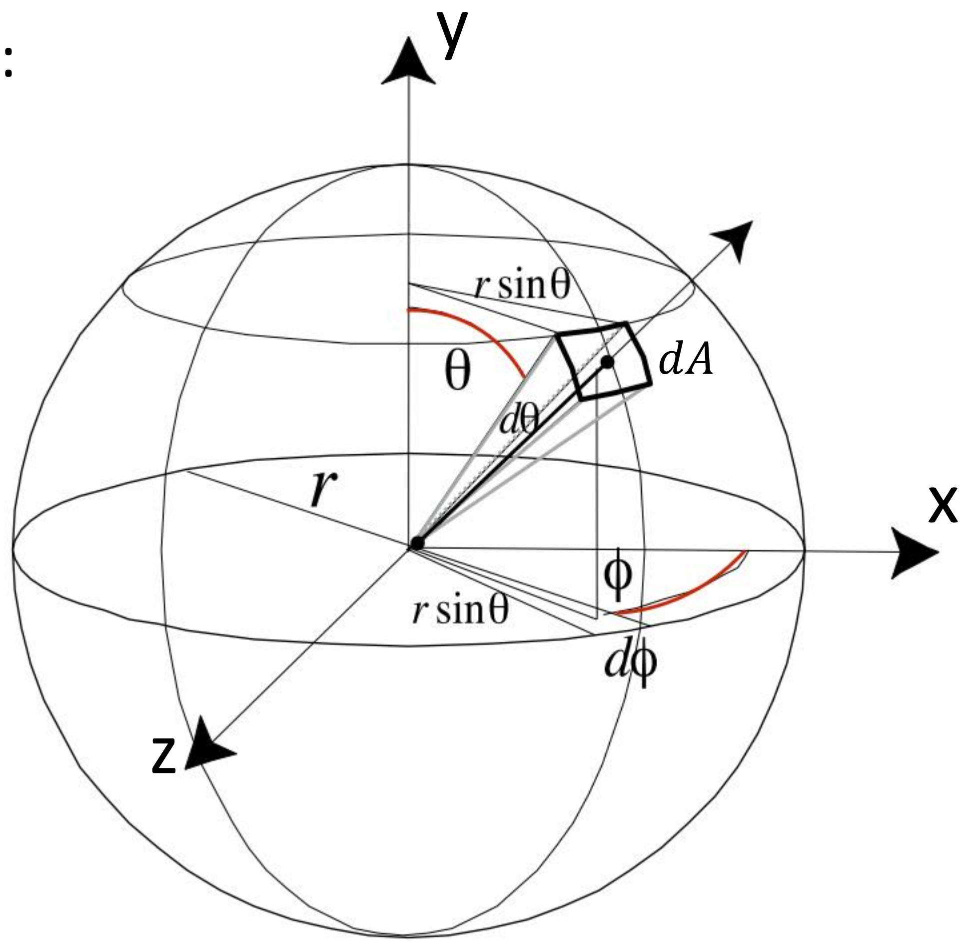
Rethink of the word "Uniform"

- Uniform:
  - The probability of sampling is proportional to the surface area
- The differential surface element:
  - $dA = r^2 \sin(\theta)\, d\theta d\phi$



Rethink of the word "Uniform"

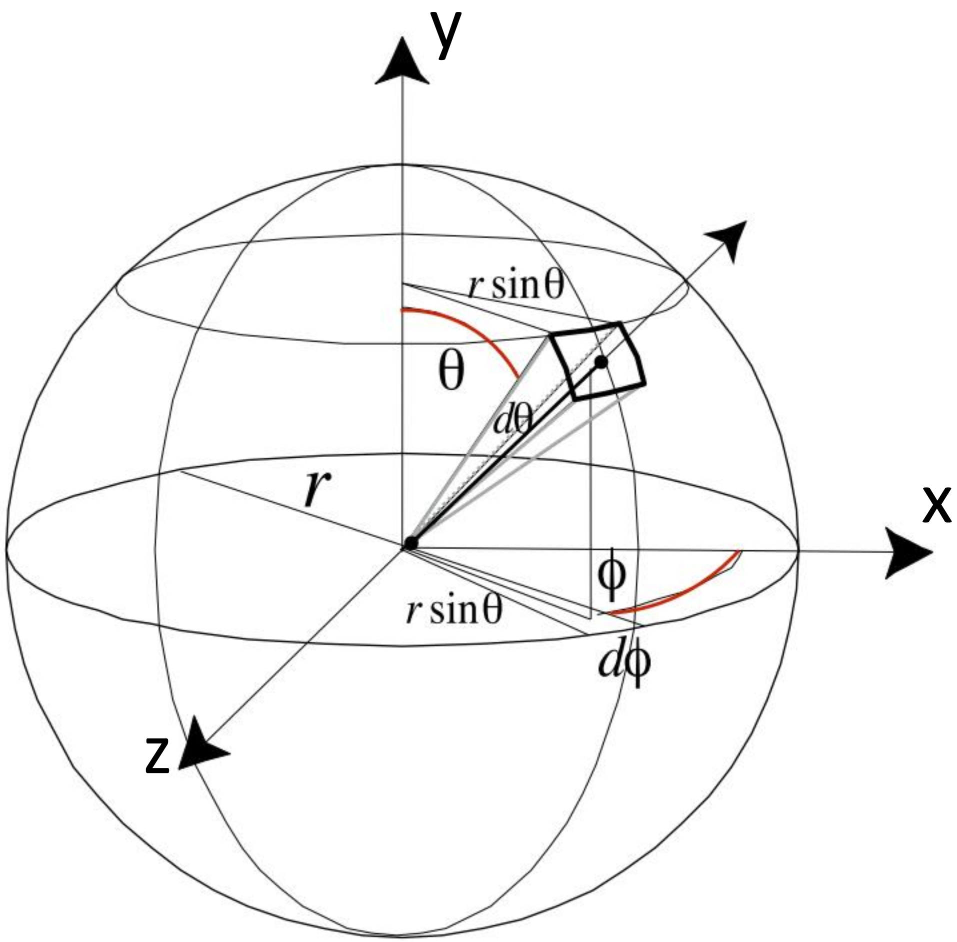
- If we have a uniform probability density function:
  - $f(v)$
  - since $\iint_S f(v)dA = 1$, and $\iint_S dA = 4\pi$
  - We have $f(v) = \frac{1}{4\pi}$
- Let's denote the p.d.f. on polar coordinates:
  - $f(\phi, \theta)$
  - Since $f(v)dA = f(\phi, \theta)\mathrm{d}\phi d\theta$
  - And $dA = \sin(\theta)\, d\phi d\theta$
  - We have $f(\phi, \theta) = \frac{\sin(\theta)}{4\pi}$
  - So: $f(\phi) = \int_0^{\pi} f(\phi, \theta) d\theta = \frac{1}{2\pi}$
  - So: $f(\theta) = \int_0^{2\pi} f(\phi, \theta) d\phi = \frac{\sin(\theta)}{2}$



The correct attempt:

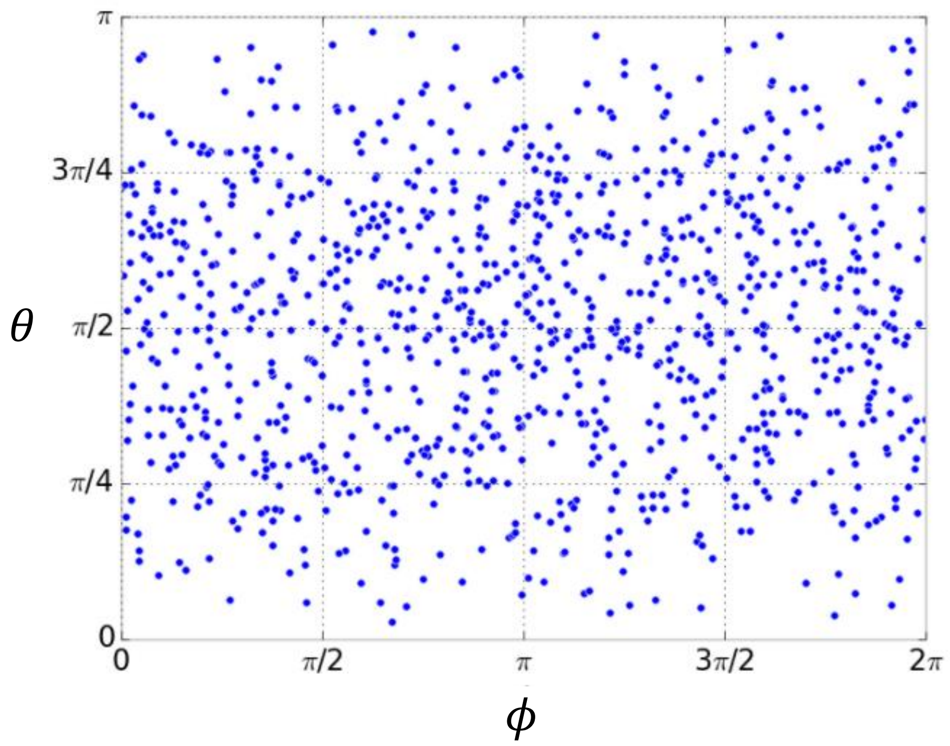
- $\phi = rand(0, 2\pi)$
- $\theta = \arccos\big(rand(-1,1)\big)$
- $r = 1$

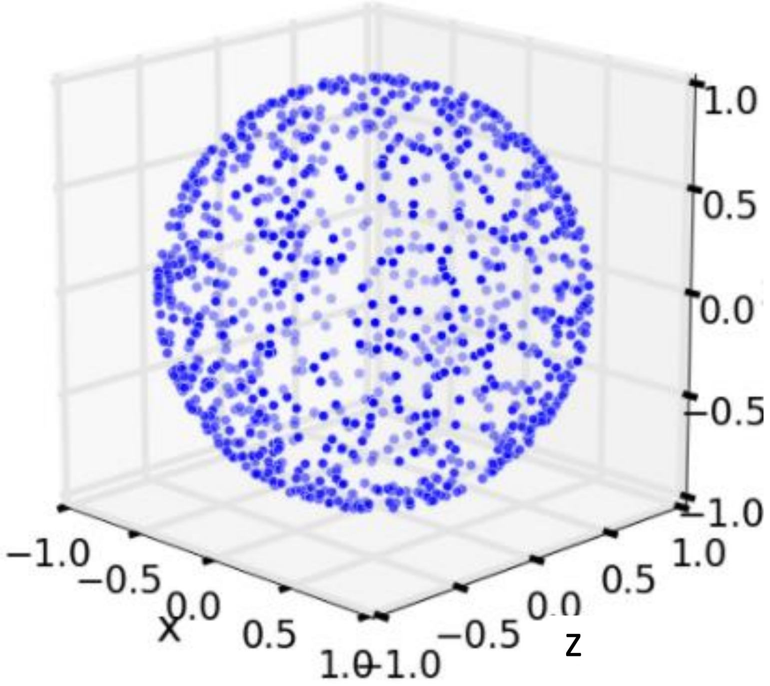
- Further Reading:
  - Inverse Transform Sampling [Link]



The correct attempt:

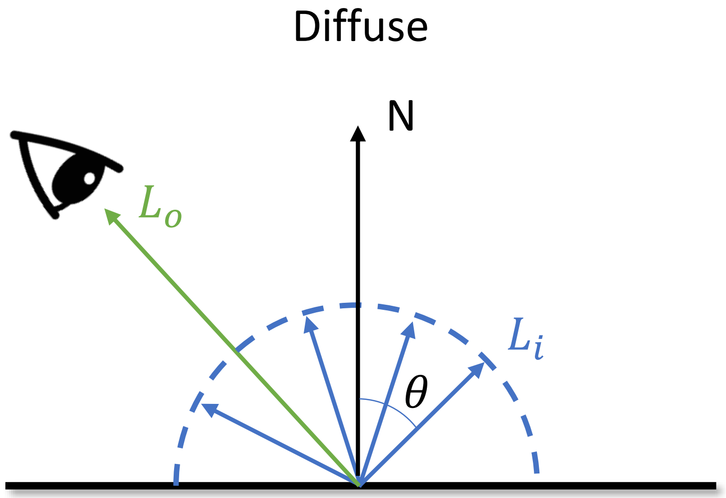
- $\phi = rand(0, 2\pi), \theta = \arccos\big(rand(-1,1)\big), r = 1$





Sample the hemisphere uniformly

- What we want:
  - Sample the directions of rays uniformly
    - Find a uniform sampling on a sphere
      - $\phi = rand(0, 2\pi), \theta = \arccos\big(rand(-1,1)\big), r = 1$
      - $x = r * \cos(\phi) * \sin(\theta)$
      - $z = r * \sin(\phi) * \sin(\theta)$
      - $y = r * \cos(\theta)$
    - Negate the direction if against the normal



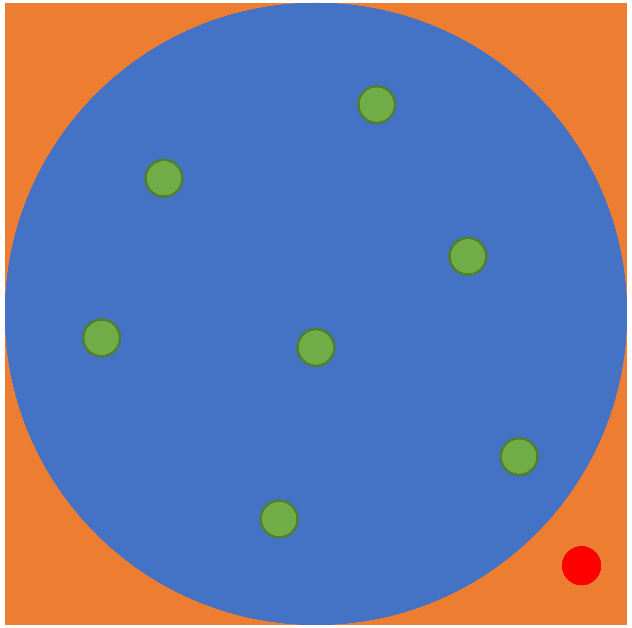
$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

Similarly, if we want to sample in a sphere:

- $\phi = rand(0, 2\pi)$
- $\theta = \arccos\left(rand(-1,1)\right)$
- $r = \sqrt[3]{rand(0,1)}$

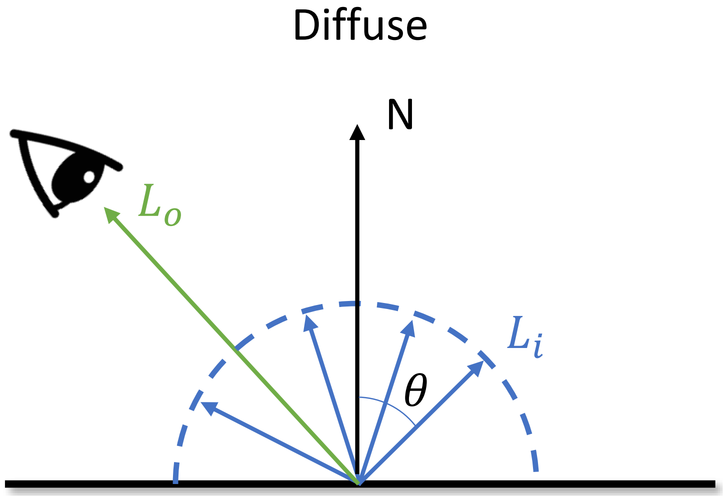
One alternative: the rejection method

- Sample inside a uniform sphere:
  - $x = rand(-1,1), y = rand(-1,1), z = rand(-1,1)$
  - Reject if $x^2 + y^2 + z^2 > 1$, and resample
- Sample on a uniform sphere:
  - Sample inside a uniform sphere and project



Are we done?

- Yes, all the components of a required sampling are there
- No, since we can do better

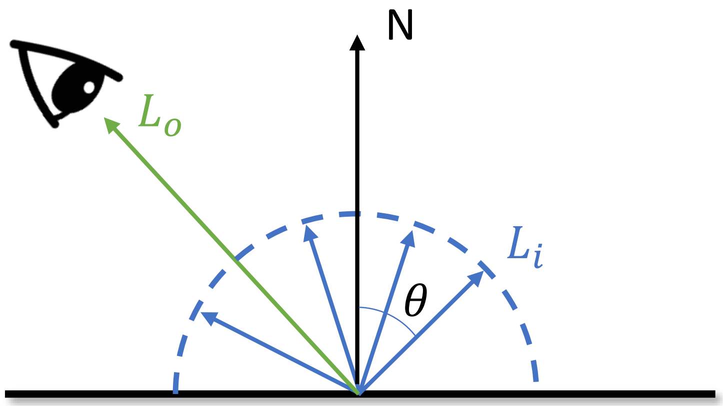


$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

The key idea of Monte Carlo is to use the **expectation** of random samples to obtain numerical results

- $L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$

Diffuse

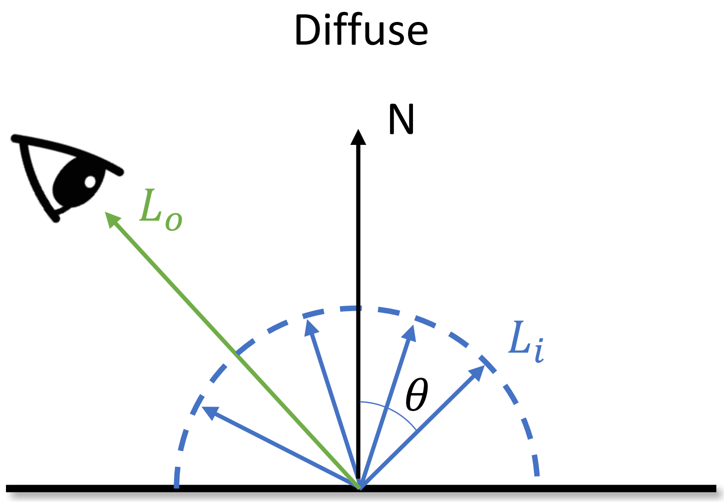


$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

Expectation of what?

- $L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k) \times 1$
  - Sample the hemisphere uniformly
  - Each sample contribute differently to the outcome

- $L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \times \cos(\theta_k)$
  - Sample the hemisphere with cosine-importance sampling
  - Each sample contribute the same to the outcome



$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

Think of your salary plan…

- Plan 1:
  - Salary = top_percent * 50K

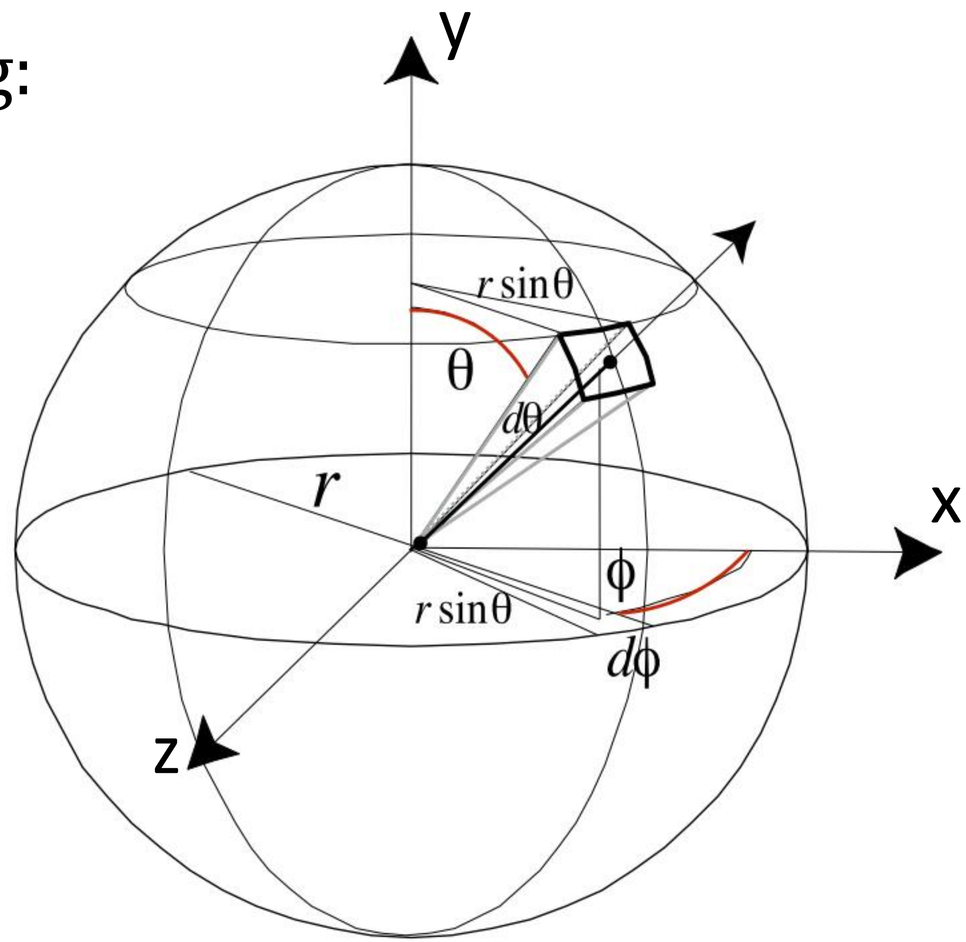
- Plan 2:
  - Salary = 50K, with top_percent probability to happen

- Which one do you prefer?

- *top_percent = the percentage of people you outperformed in your company

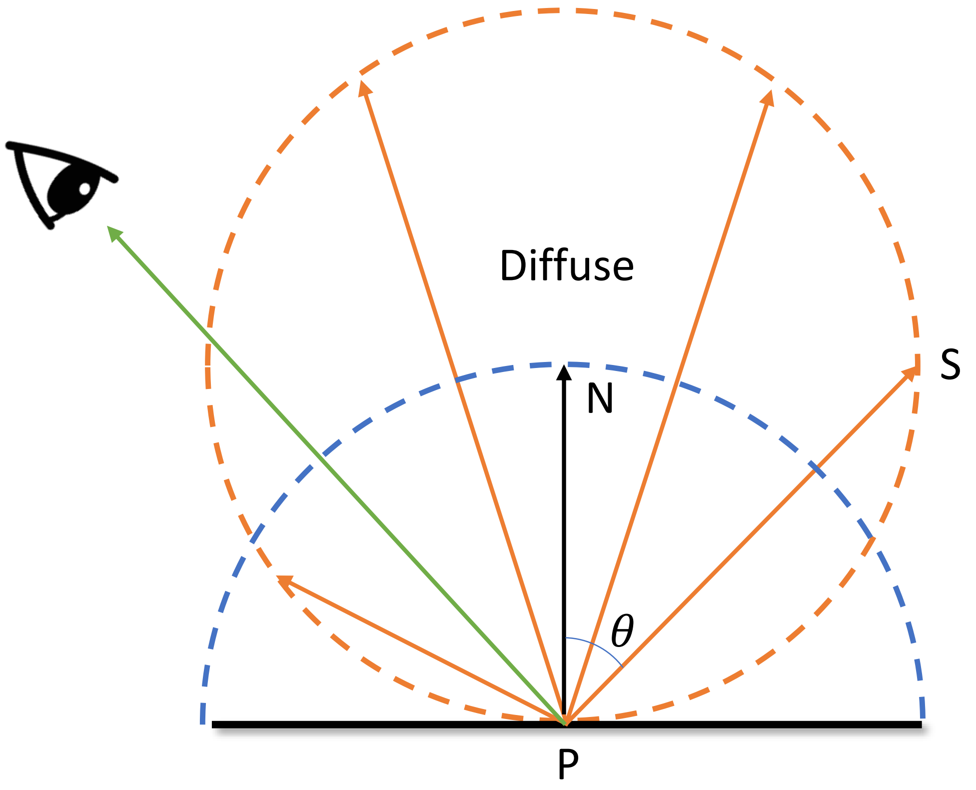
Importance sampling, great!

- A $\cos(\theta)$ weighted importance sampling:
  - $\phi = rand(0, 2\pi)$
  - $\theta = \arccos\left(sqrt\left(rand(0,1)\right)\right)$
  - $r = 1$



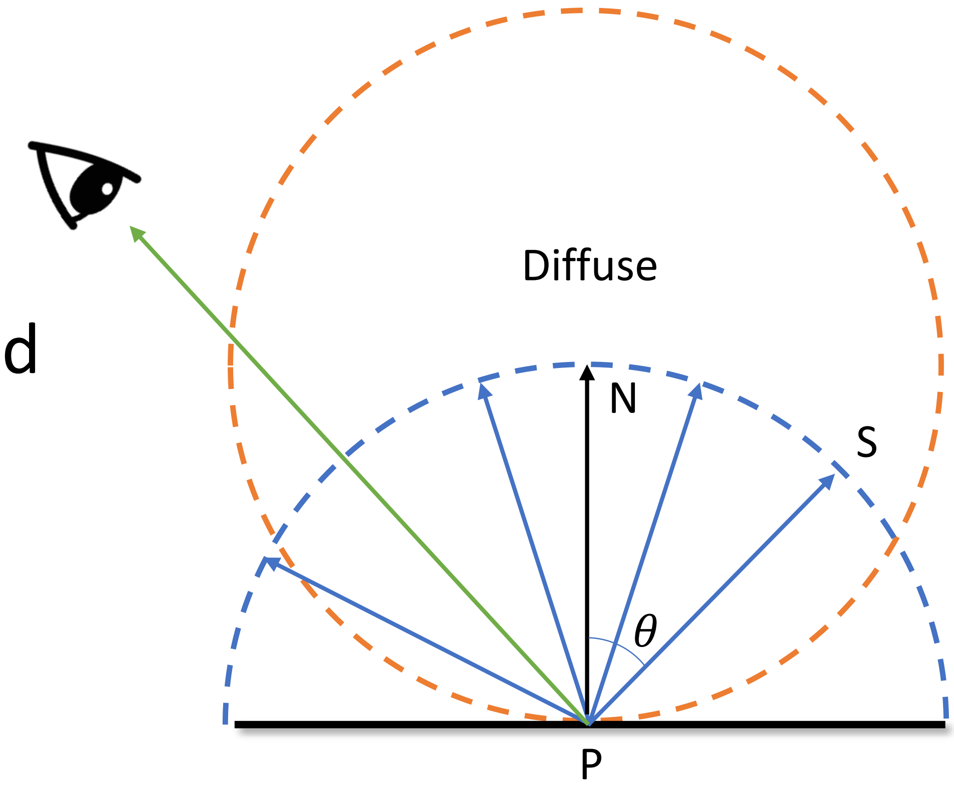
Importance sampling, an alternative

- Uniformly sample a point on
  a uniform sphere centered at P+N,
  say S

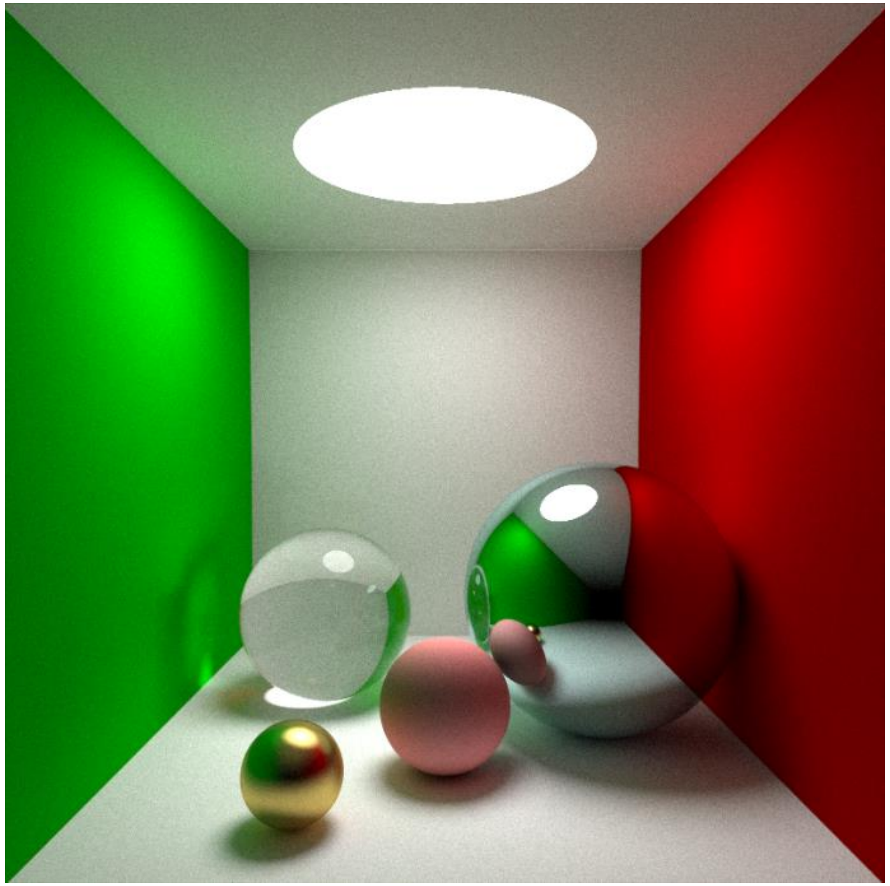


Importance sampling, an alternative

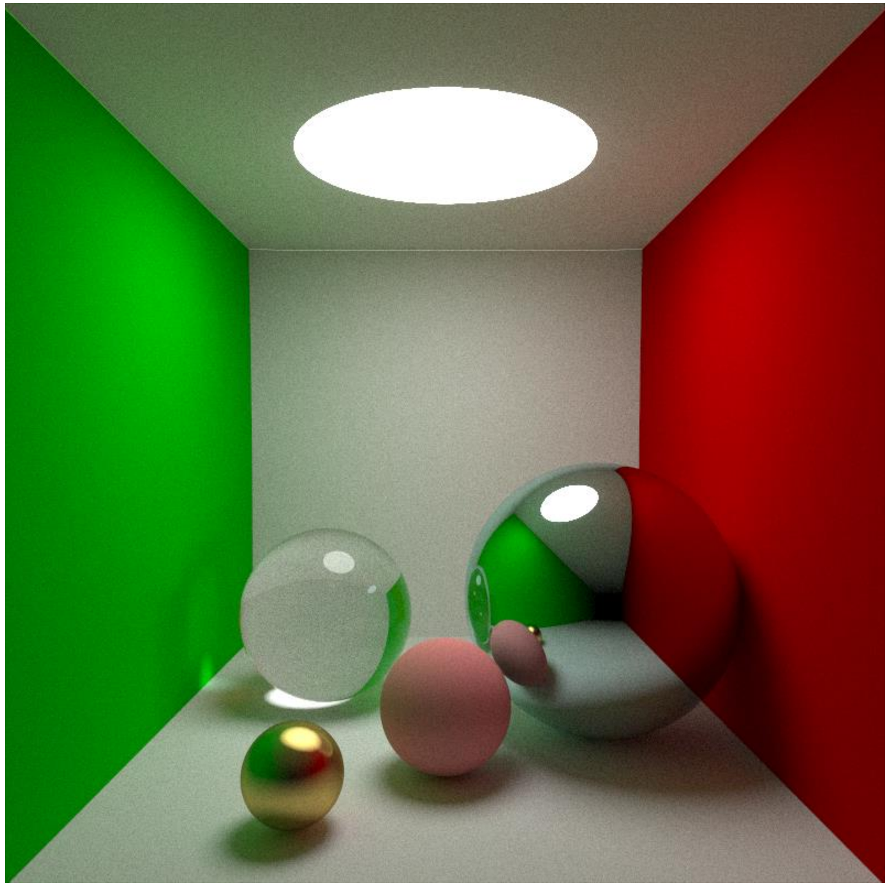
- Uniformly sample a point on a uniform sphere centered at P+N, say S.
- Normalize S-P, as the cosine-weighted sampled direction.



Note: the implementation in our repo was wrong

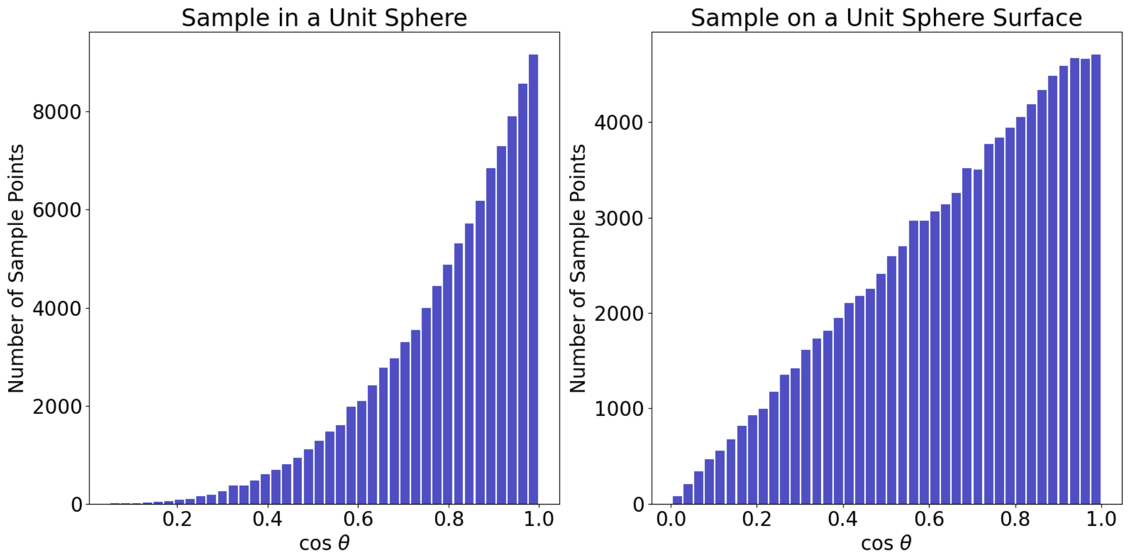


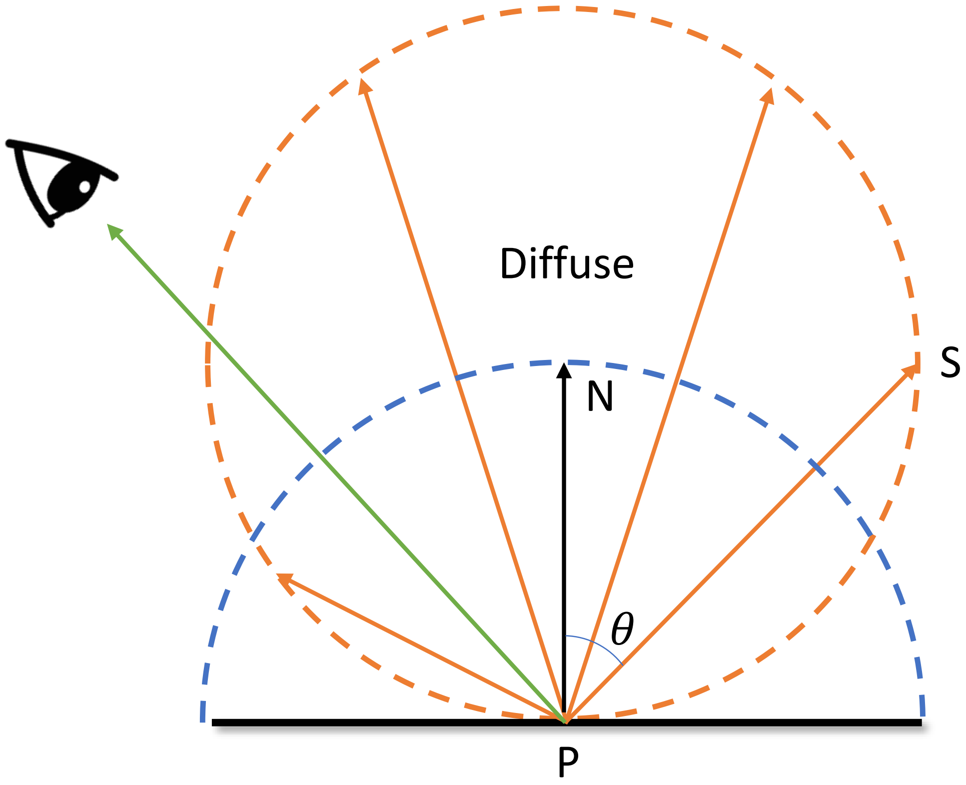
Incorrect



Correct

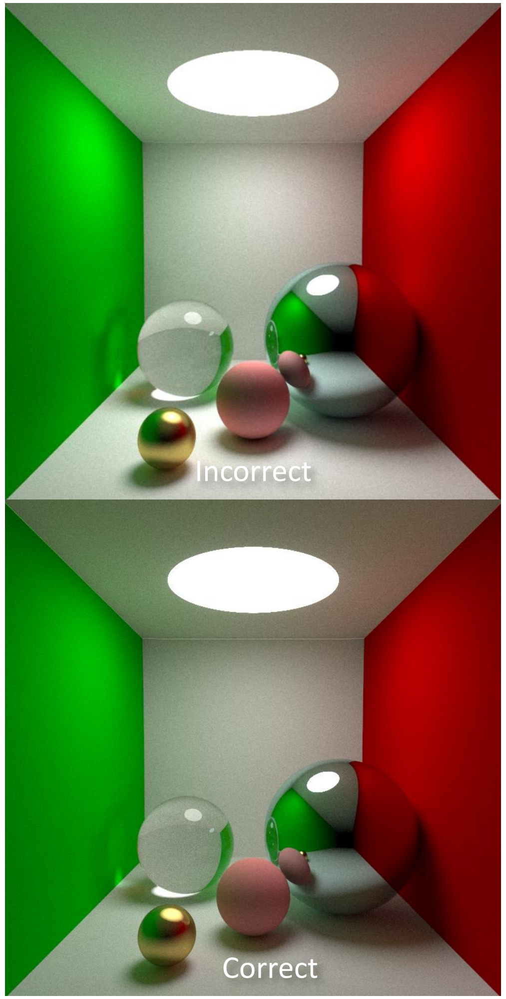
Uniform sampling inside/on the unit sphere

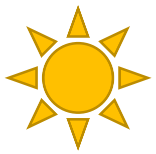




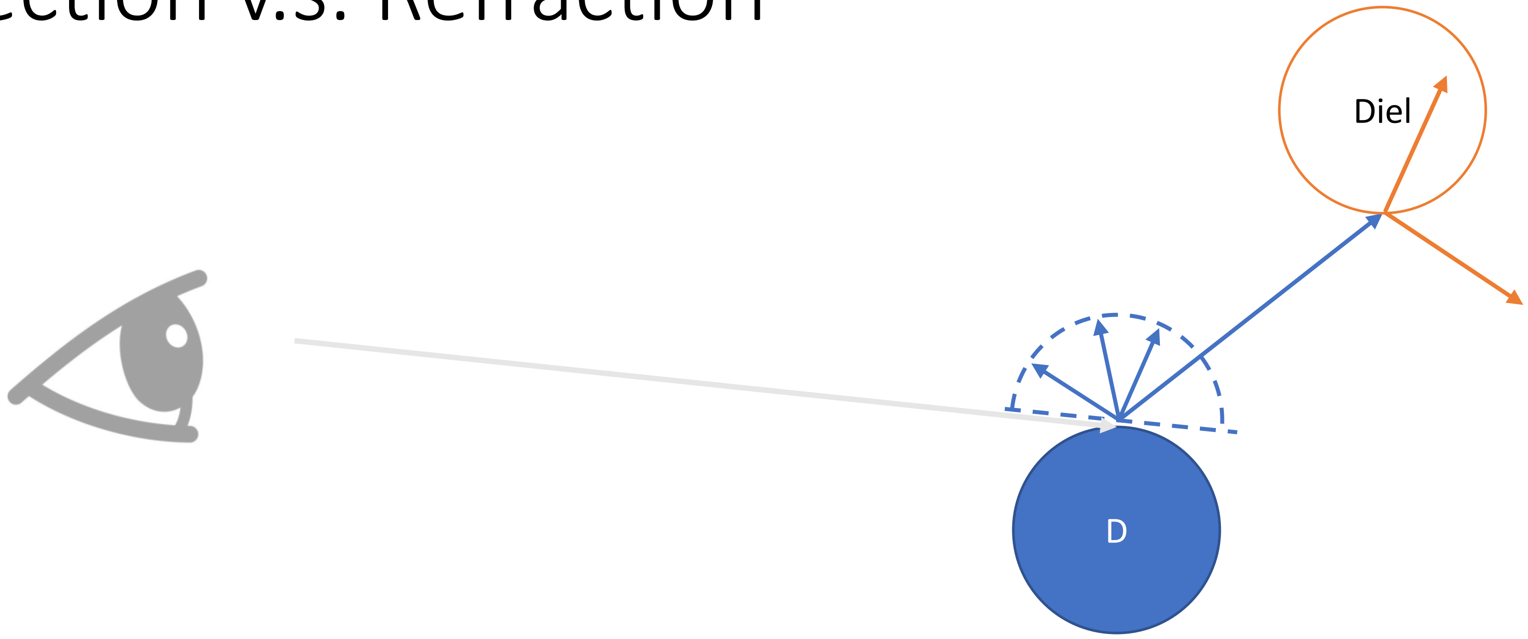
Take-aways

- It is extremely hard to notice an incorrect sampling
  - due to the lack of intuition how $\cos(\theta)$ looks like
- Try:
  - deriving the probably density function theoretically
  - ploting the distribution histogram to verify your samples
- Further Reading:
  - The PBR Book Chapter 13 [Link]



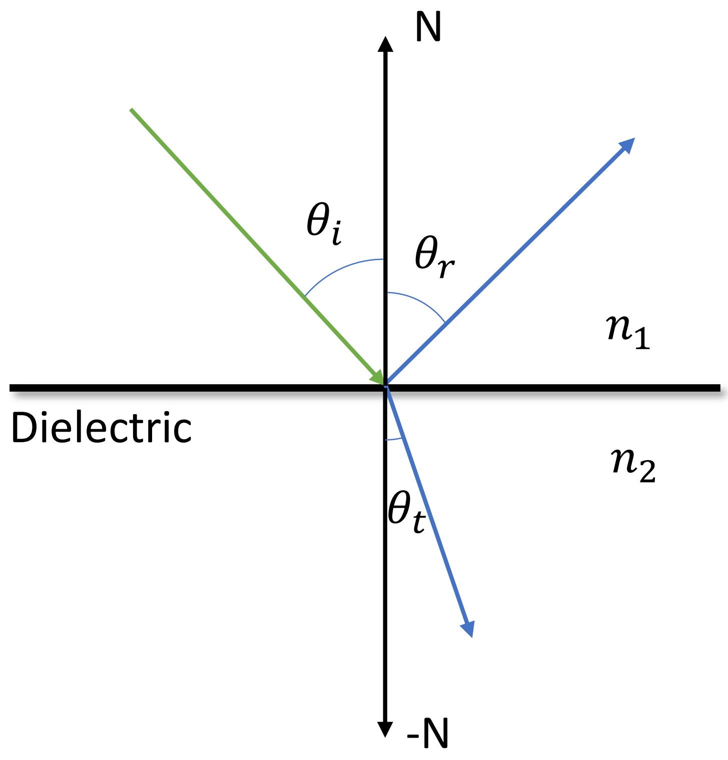


Reflection v.s. Refraction



Reflection and refraction

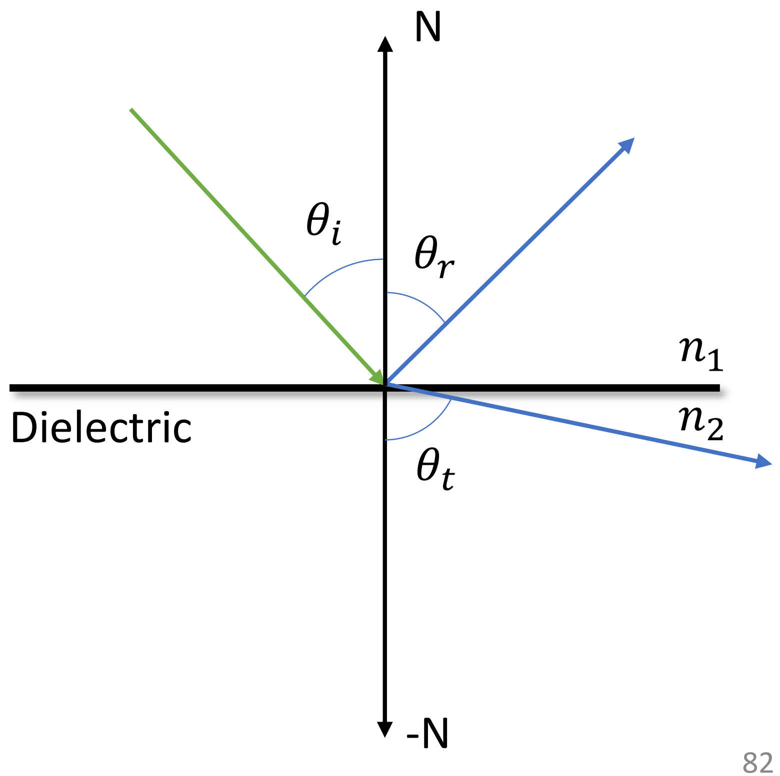
- Law of reflection:
  - $\theta_i = \theta_r$
- Snell's law (for refraction)
  - $n_1 \sin(\theta_i) = n_2 \sin(\theta_t)$



Total reflection

- Happens when $n_1 > n_2$
  - For example from glass to air

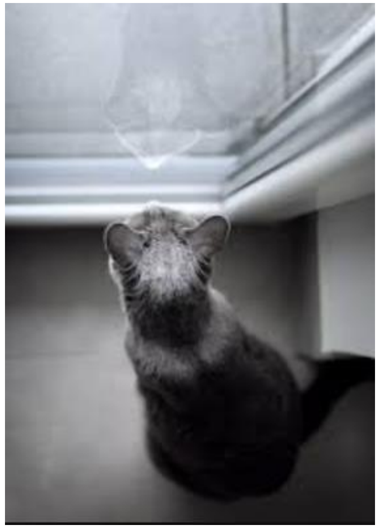
- Snell's law may fail to give you a solution
  - $\sin(\theta_t) = \frac{n_1}{n_2}\sin(\theta_i) > 1$

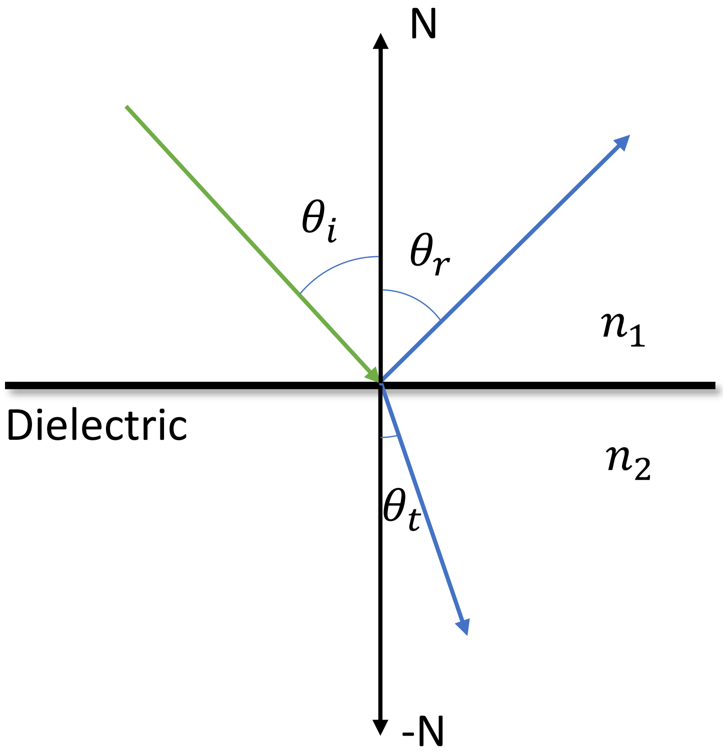


Reflection and refraction

- How much light should be refracted and how much should be reflected?
  - Whitted-style ray tracer: set it by yourself

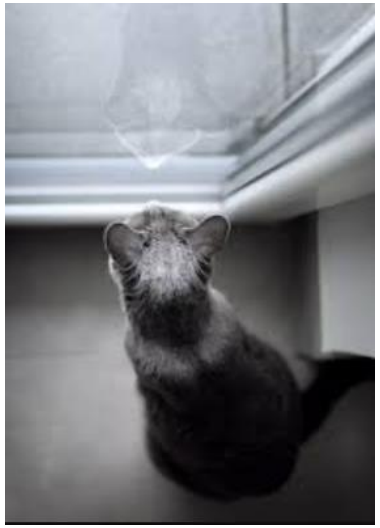
  - Have you looked at a window at a steep angle?





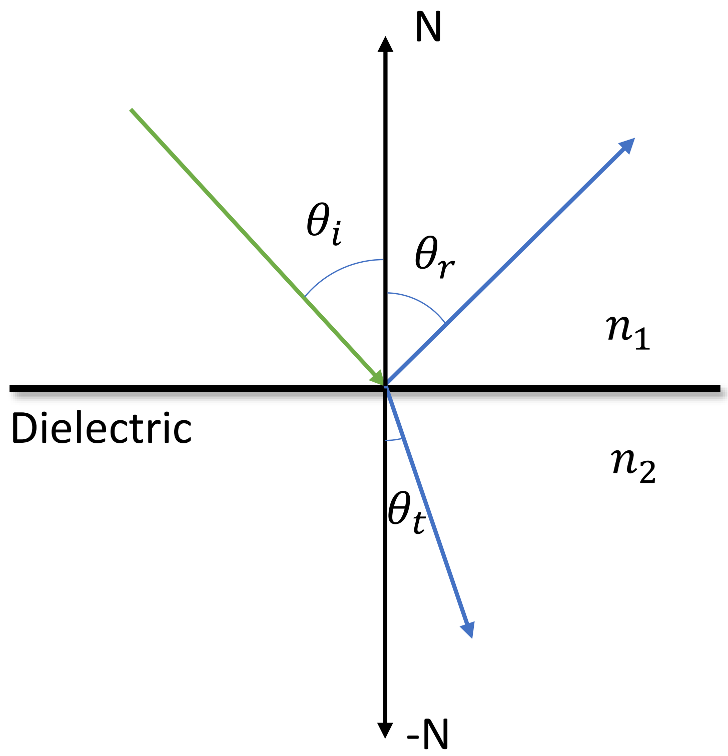
The reflection coefficient: R

- R: how much of a wave is reflected by an impedance discontinuity in the transmission medium
- R should be material dependent (function of n_1 and n_2)
- R should be view point dependent (function of θ)
- The refraction coefficient: $T = 1 - R$



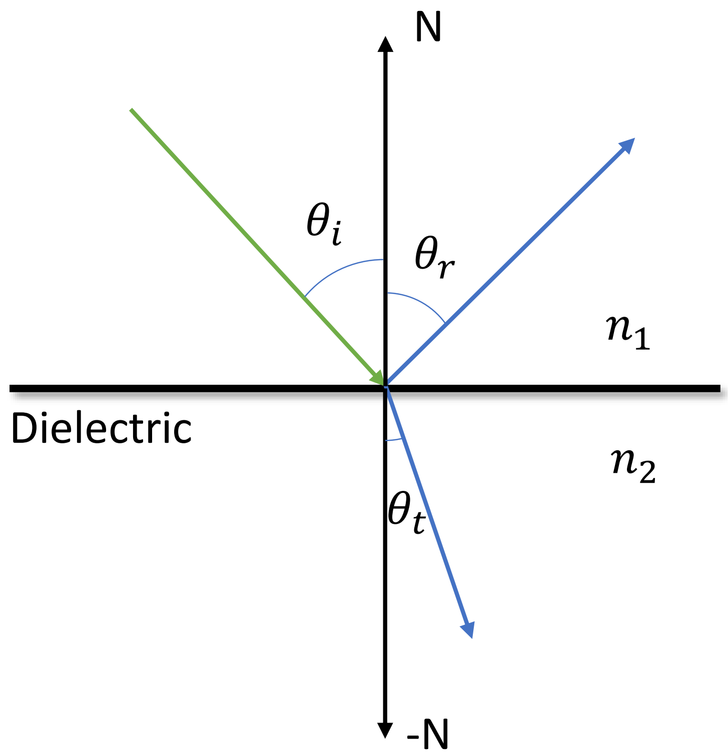
The reflection coefficient: R

- Fresnel's equations
  - S-polarization:
    - $R_s = \left(\frac{n_1 \cos(\theta_i) - n_2 \cos(\theta_t)}{n_1 \cos(\theta_i) + n_2 \cos(\theta_t)}\right)^2$
  - P-polarization:
    - $R_P = \left(\frac{n_1 \cos(\theta_t) - n_2 \cos(\theta_i)}{n_1 \cos(\theta_t) + n_2 \cos(\theta_i)}\right)^2$
  - For "natural light"
    - $R = \frac{1}{2}(R_S + R_P)$



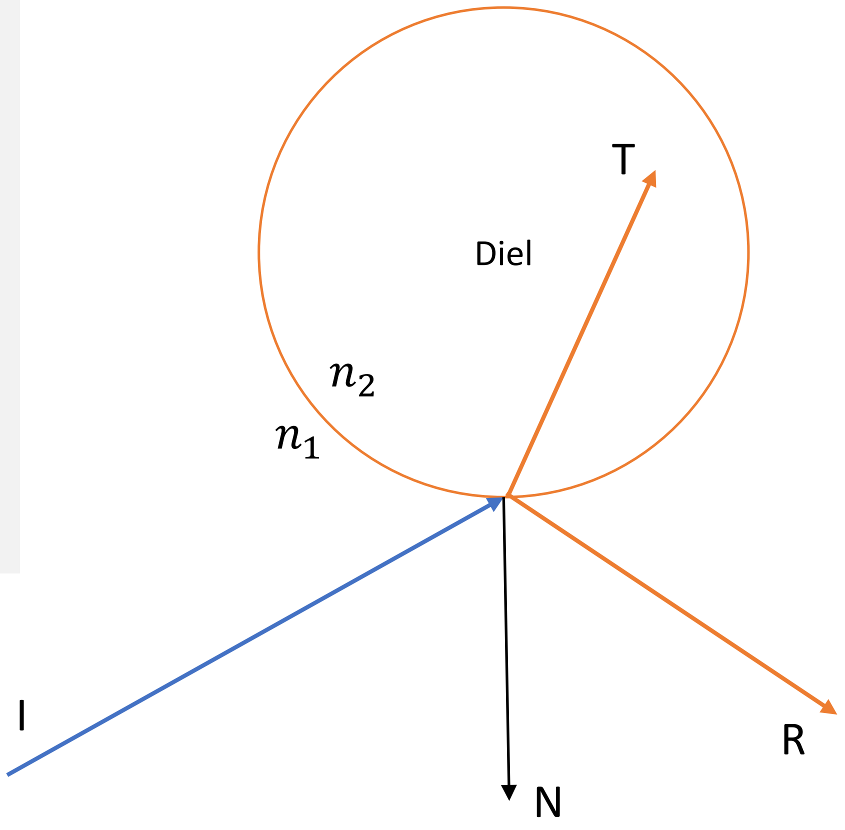
The reflection coefficient: R

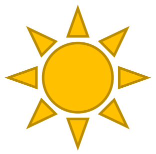
- Schlick's approximation
  - $R(\theta_i) = R_0 + (1 - R_0)(1 - \cos(\theta_i))^5$
  - $R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$



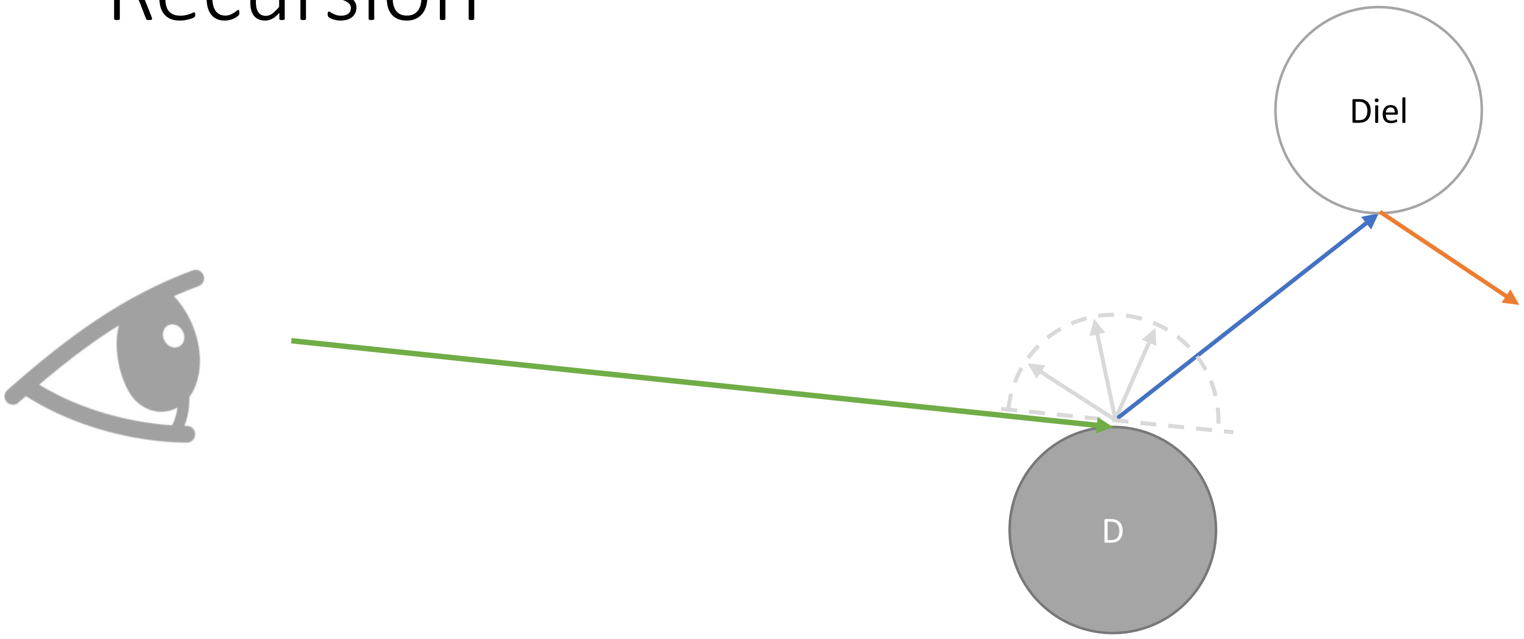
Path tracing with reflection coefficient R_c

```
def scatter_on_a_dielectric_surface(I):
    sin_theta_i = -I.cross(N)
    theta_i = arcsin(sin_theta_i)
    if n1/n2*sin_theta_i > 1.0:
        return R # total internal reflection
    else:
        R_c = reflectance(theta_i, n1, n2)
        if random() <= R_c:
            return R # reflection
        else:
            return T # refraction
```





Recursion



The path tracer

```
def what_color_does_this_ray_see(ray_o, ray_dir):
    if (random() > p_RR):
        return 0
    else:
        flag, P, N, material = first_hit(ray_o, ray_dir, scene)
        if flag == False:
            return 0
        if material.type == LIGHT_SOURCE:
            return 1 # could be more than 1
        else:
            ray2_o = P
            ray2_dir = scatter(ray_dir, P, N)
            # the cos(theta) in DIFFUSE is hidden in the scatter function
            L_i = what_color_does_this_ray_see(ray2_o, ray2_dir)
            L_o = material.color * L_i / p_RR
            return L_o
```

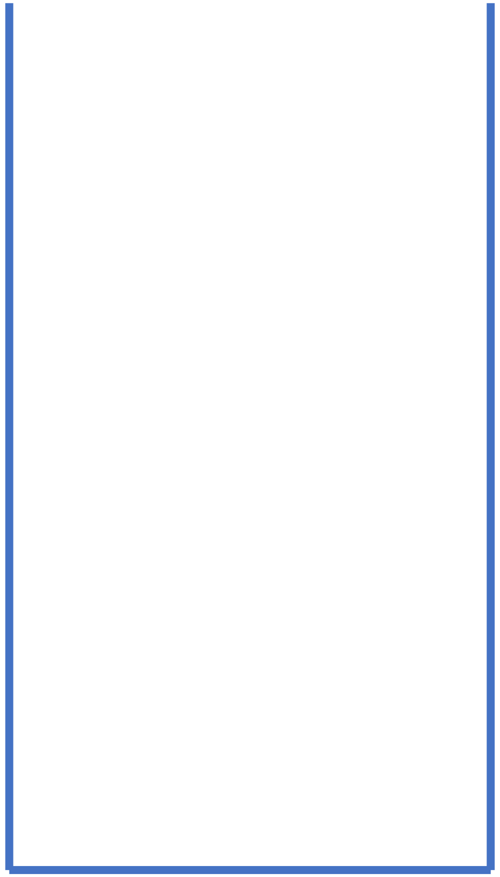
How are functions being called?

```python
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```



Stack

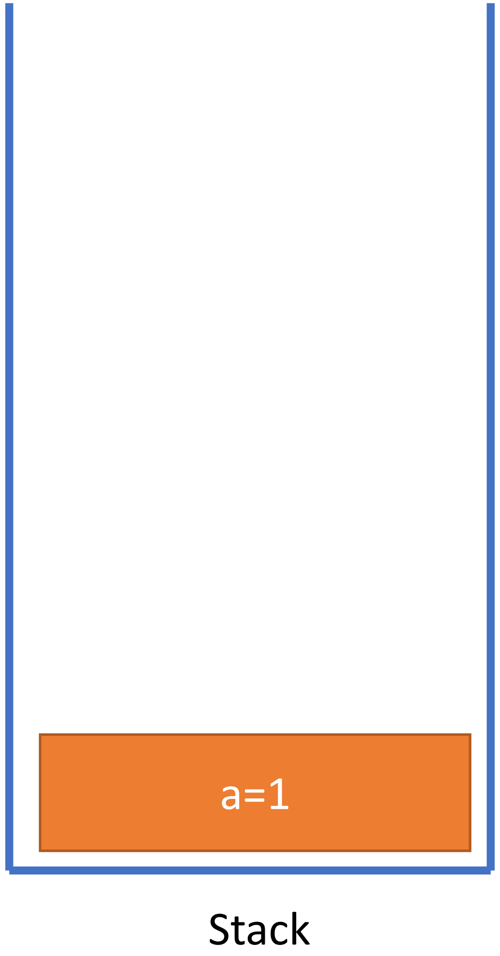
How are functions being called?

```
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```



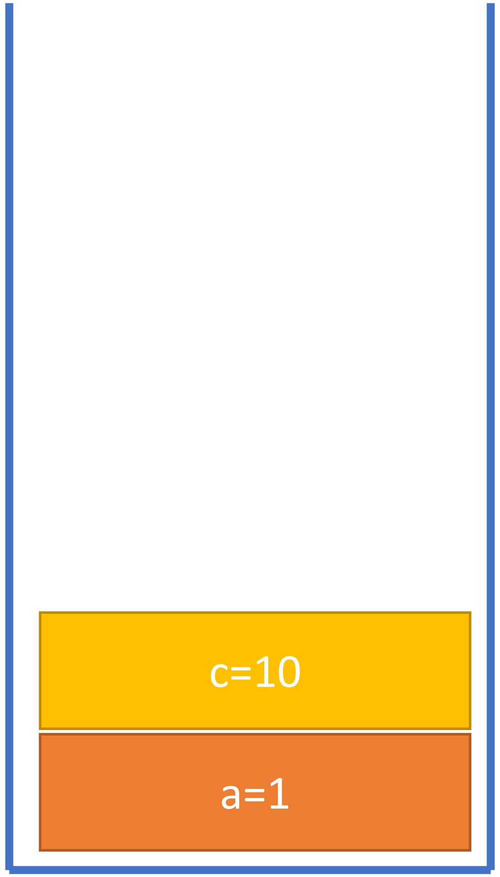
How are functions being called?

```python
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```



Stack

How are functions being called?

```
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```

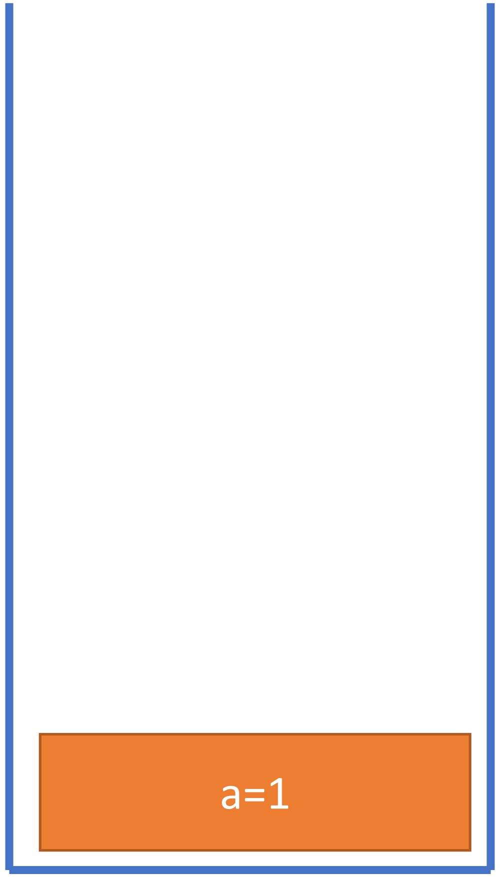
How are functions being called?

```
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```



Stack

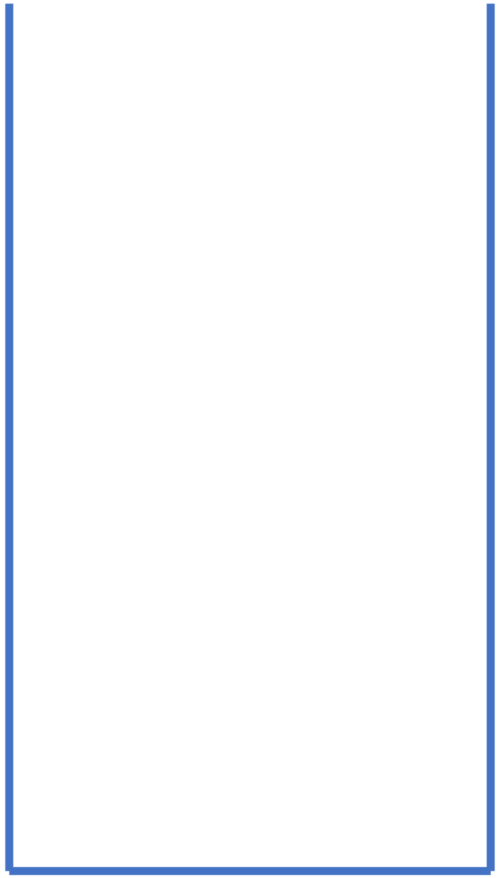
How are functions being called?

```python
def foo():
    a = 1
    b = bar()
    return a+b

def bar():
    c = 10
    d = baz()
    return c*d

def baz():
    return 100

foo()
```



Stack

Revisit the path tracer

```
def what_color_does_this_ray_see(ray_o, ray_dir):
    if (random() > p_RR):
        return 0
    else:
        flag, P, N, material = first_hit(ray_o, ray_dir, scene)
        if flag == False:
            return 0
        if material.type == LIGHT_SOURCE:
            return 1 # could be more than 1
        else:
            ray2_o = P
            ray2_dir = scatter(ray_dir, P, N)
            # the cos(theta) in DIFFUSE is hidden in the scatter function
            L_i = what_color_does_this_ray_see(ray2_o, ray2_dir)
            L_o = material.color * L_i / p_RR
            return L_o
```

Considering a similar recursive function

- fact computes the factorial of a number:
  - fact(n) = n!

- When called fact(5):

fact(5)
{5 * fact(4)}
{5 * {4 * fact(3)}}
{5 * {4 * {3 * fact(2)}}}
{5 * {4 * {3 * {2 * fact(1)}}}}
{5 * {4 * {3 * {2 * 1}}}}
{5 * {4 * {3 * 2}}}
{5 * {4 * 6}}
{5 * 24}
120

```
def fact(n):

    if n == 1:
        return 1

    temp = fact(n-1)
    ret = n * temp
    return ret
```

A better solution

- The previous recursion can be optimized using a tail-recursion
- …which can be further optimized using a loop (a stack-less version)

- When called fact(5):

  5 * 4 * 3 * 2 * 1

```
def fact(n):
    ret = 1

    while True:
        if n == 1:
            break
        ret *= n
        n = n-1

    return ret
```

A recursion-less path tracer

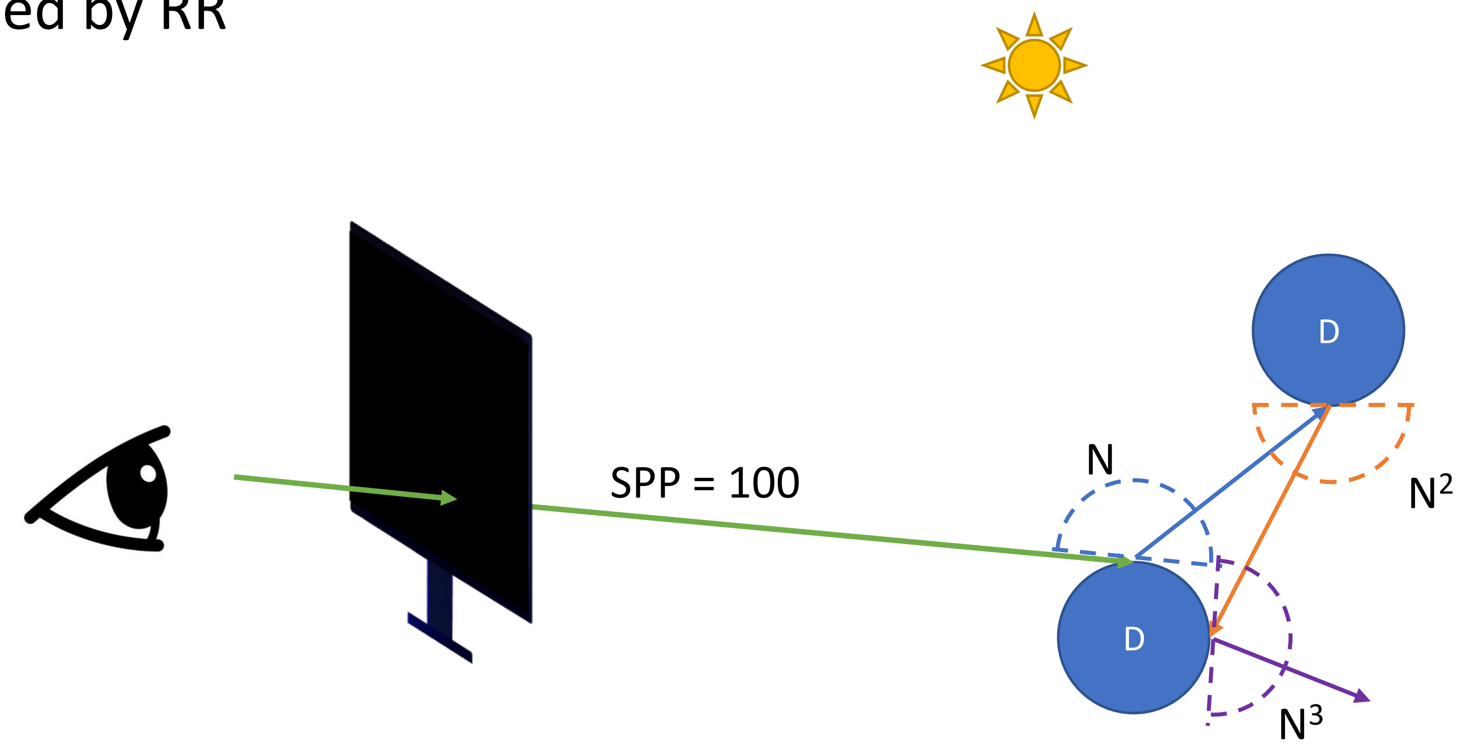
```
def what_color_does_this_ray_see(ray_o, ray_dir):
    color = 0.0
    brightness = 1.0
    for n in range(depth_cap): # could use a while True:
        if (random() > p_RR):
            break
        else:
            flag, P, N, material = first_hit(ray_o, ray_dir, scene)
            if flag == False:
                break
            if material.type == LIGHT_SOURCE:
                color = 1.0 * brightness # could be more than 1.0
                break
            else:
                brightness *= material.color / p_RR
                ray_o = P
                ray_dir = scatter(ray_dir, P, N)
                # the cos(theta) in DIFFUSE is hidden in the scatter function
    return color
```

A recursion-less path tracer

- What we see after multiple bounces

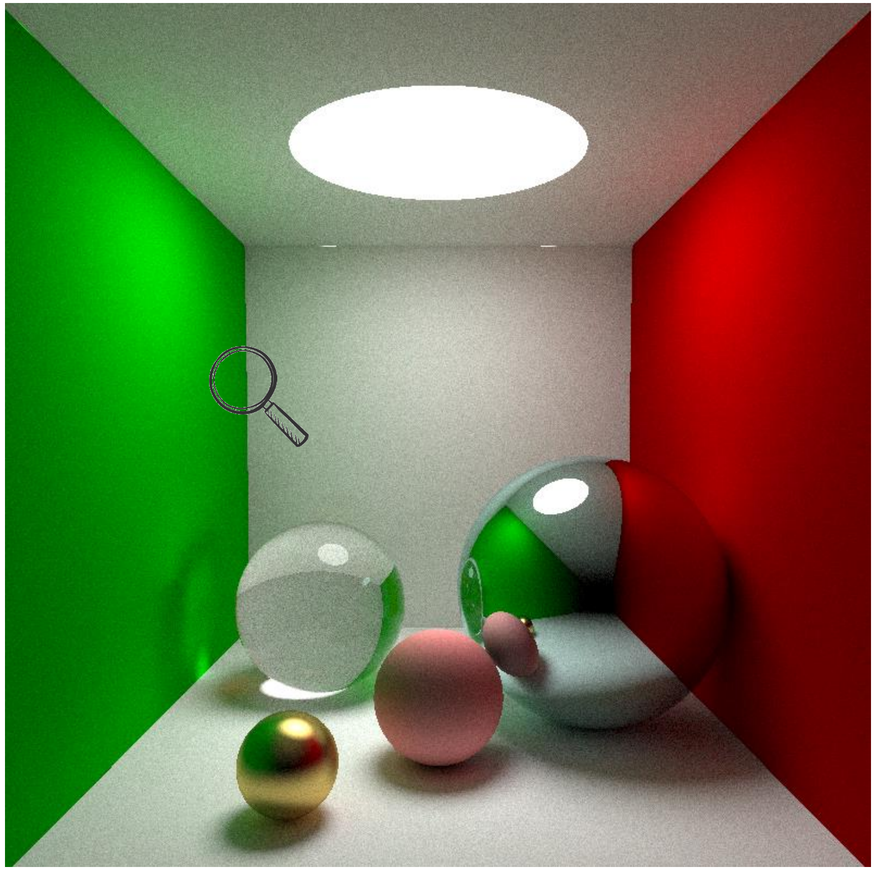
  = color*color*color*…*brightness_of_light_source, hit_light

  = black, hit_void or killed by RR



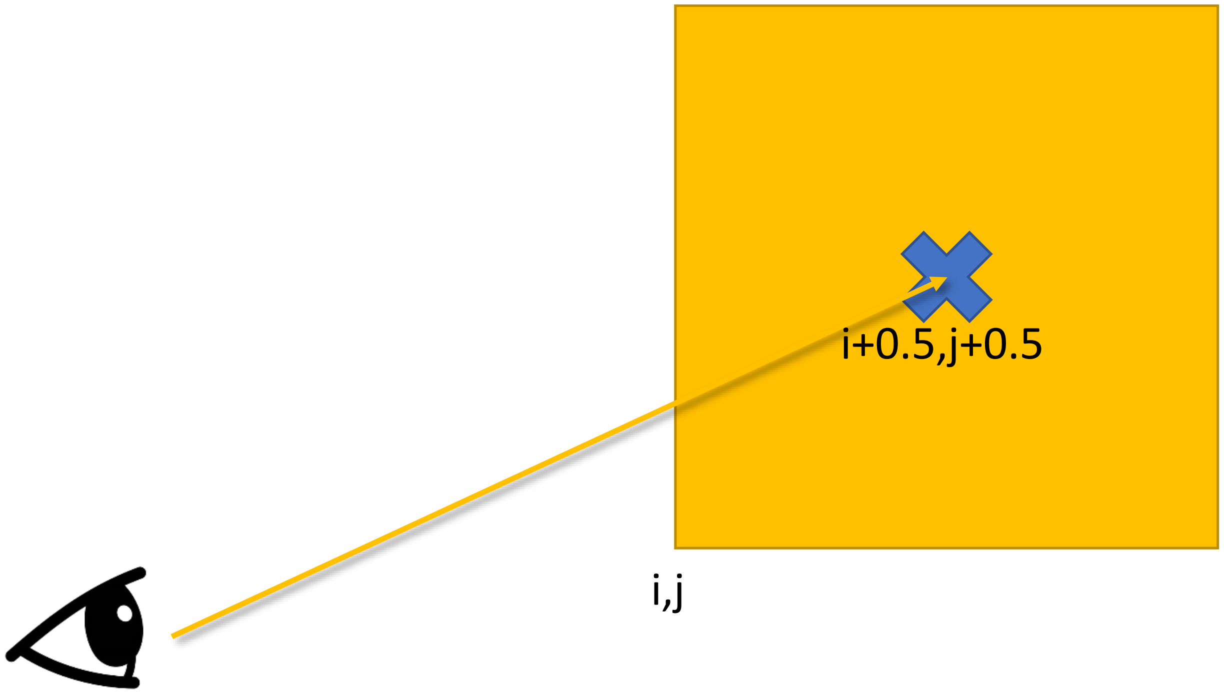
Anti-aliasing

The problem of aliasing

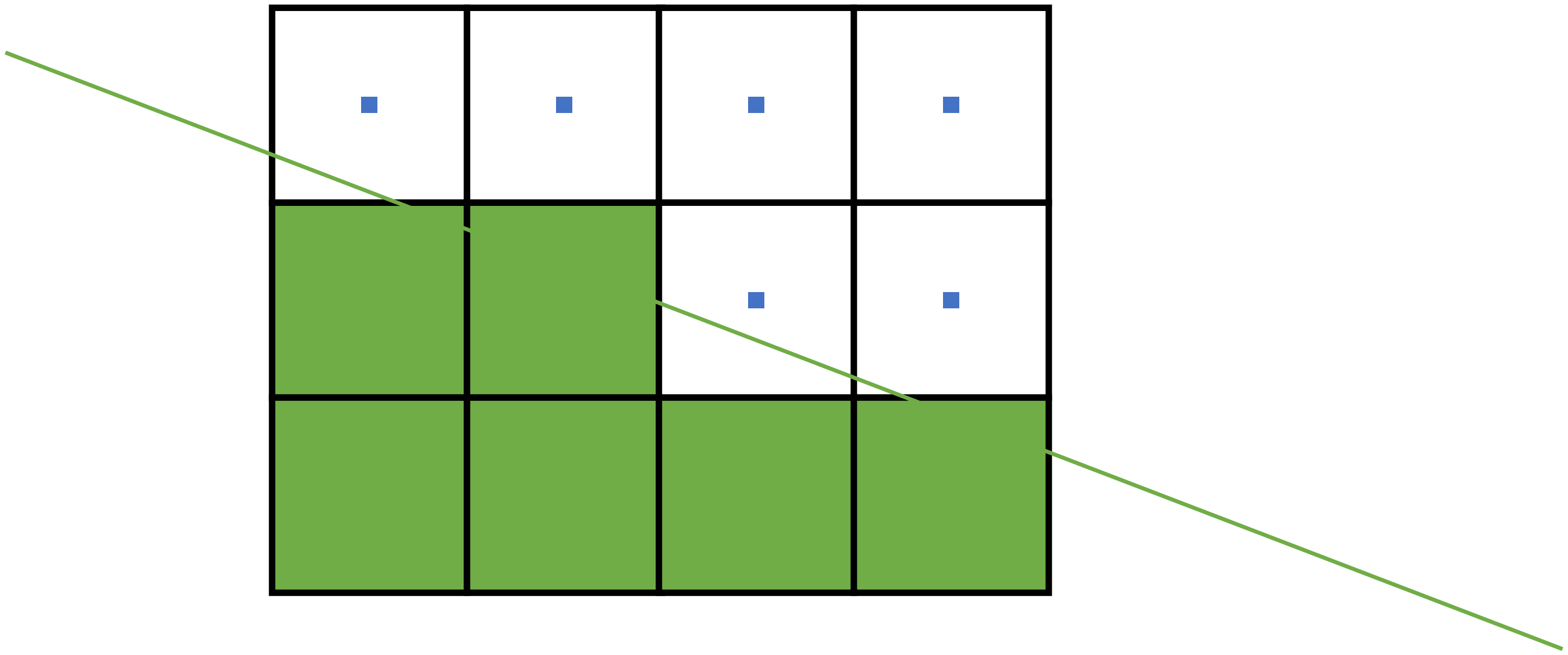


Rays are always casted through the center

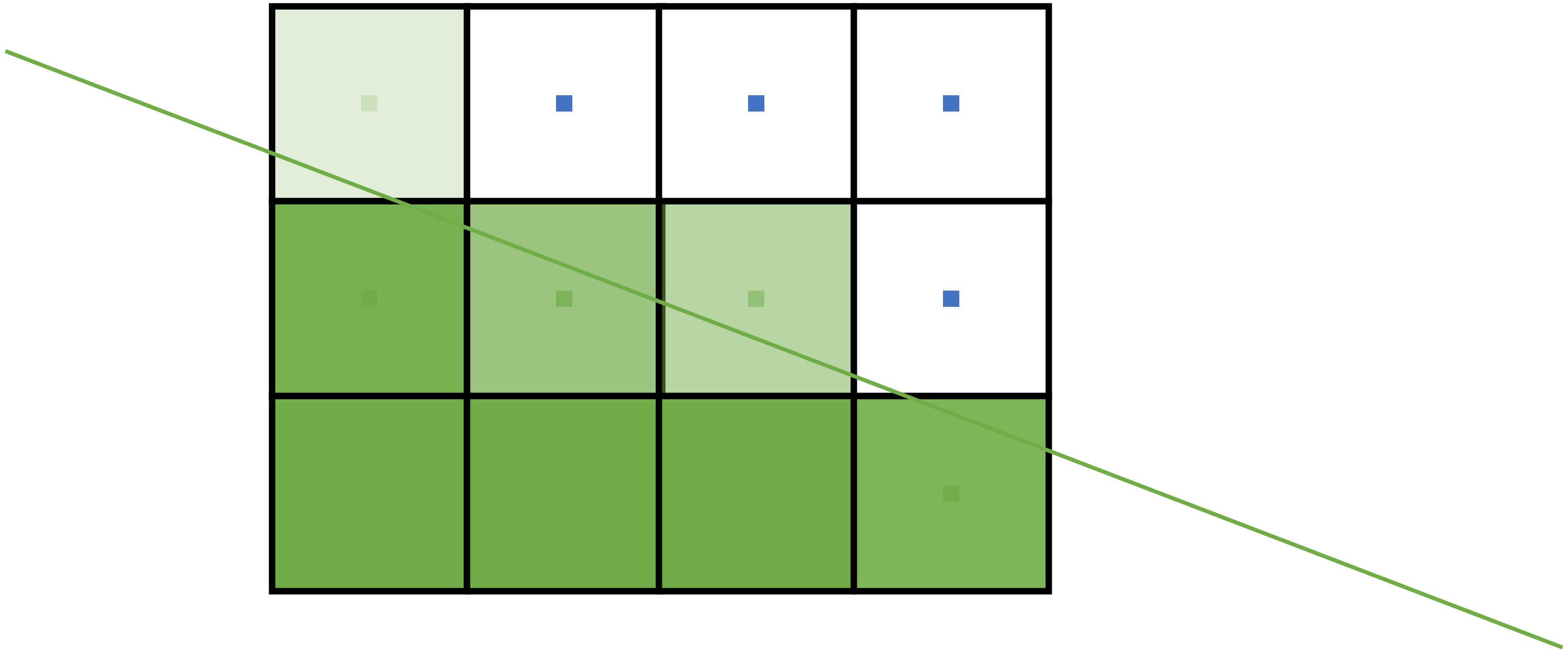
```
u = float(i+0.5)/res_x
v = float(j+0.5)/res_y # uv in (0, 1)
```



A zig-zag looking of the edges

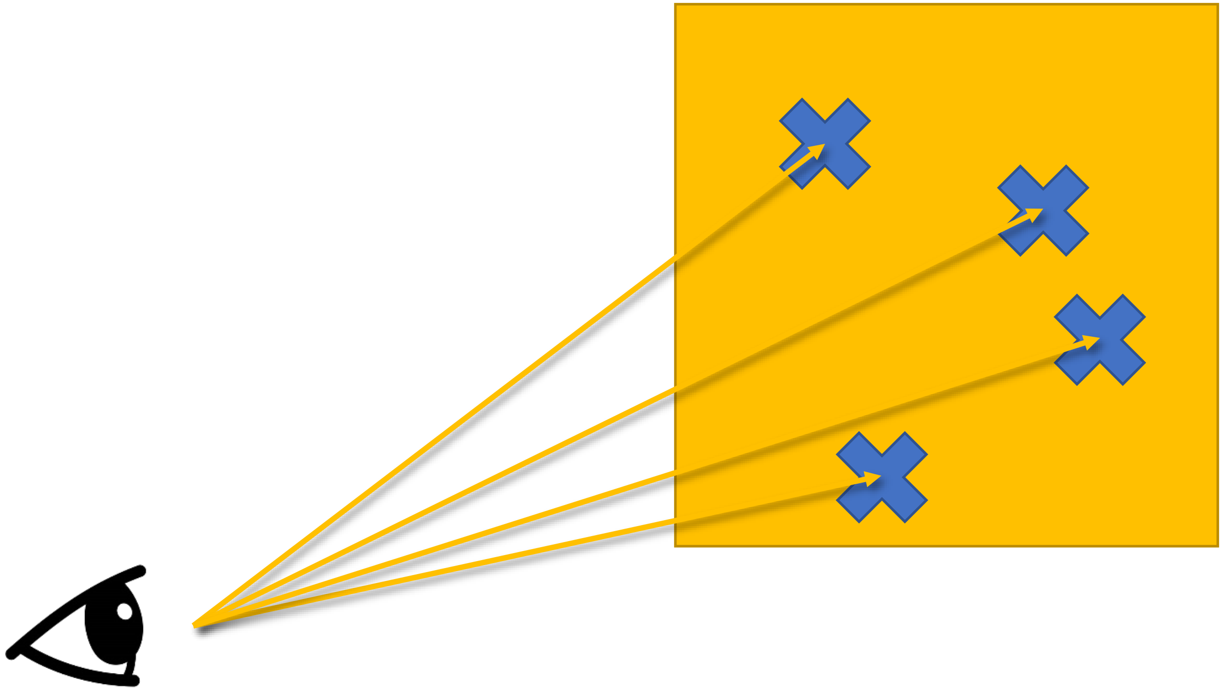


Softening the edges

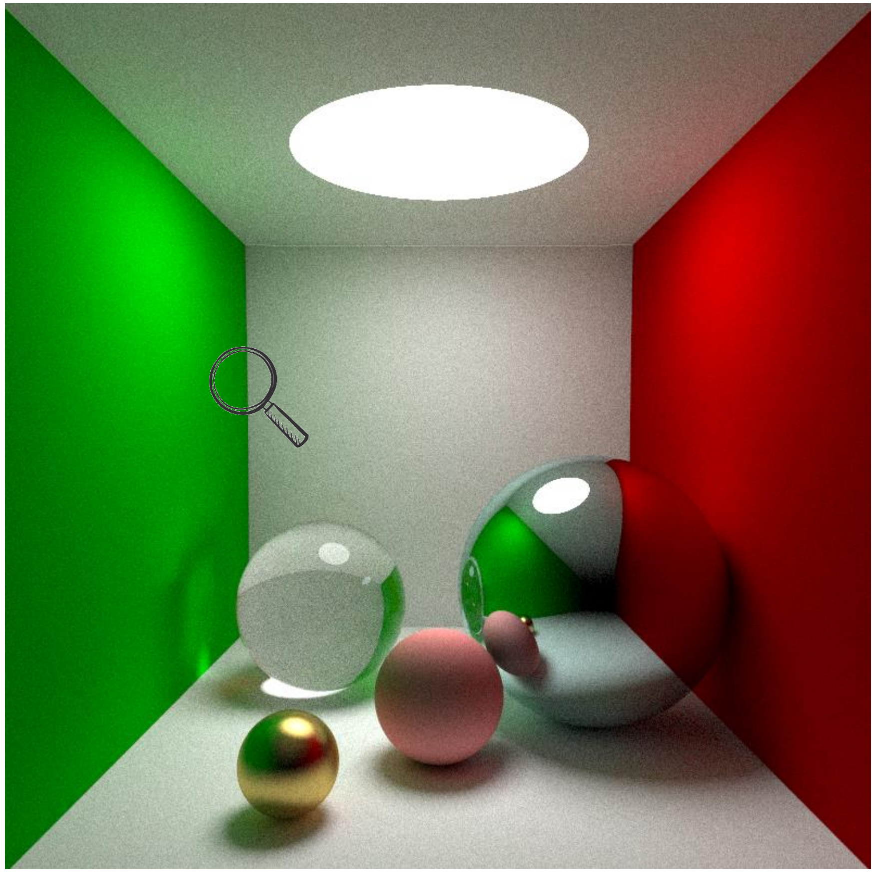


Anti-aliasing

```
u = float(i+ti.random())/res_x
v = float(j+ti.random())/res_y # uv in [0, 1]
```



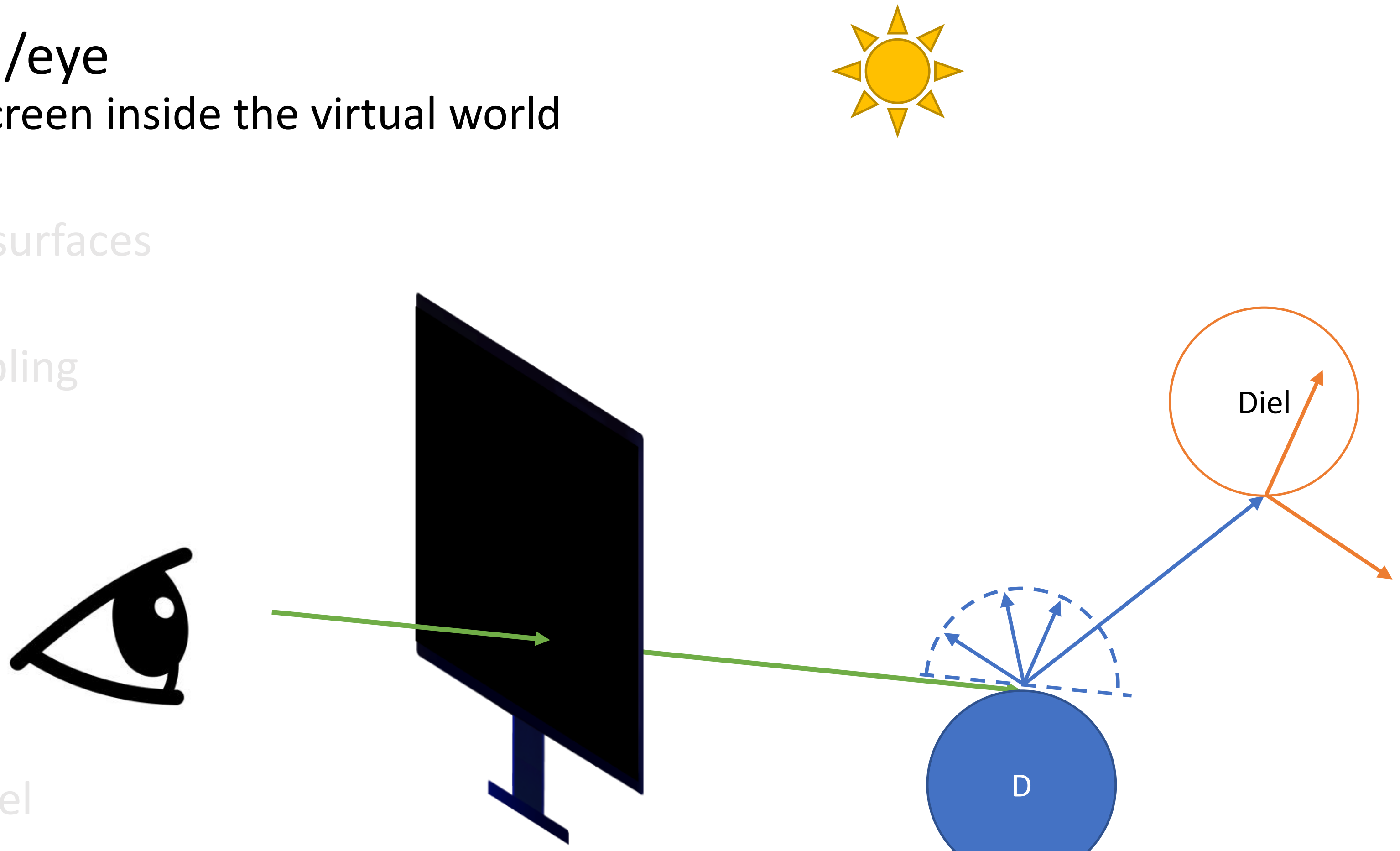
Anti-aliasing



Remark

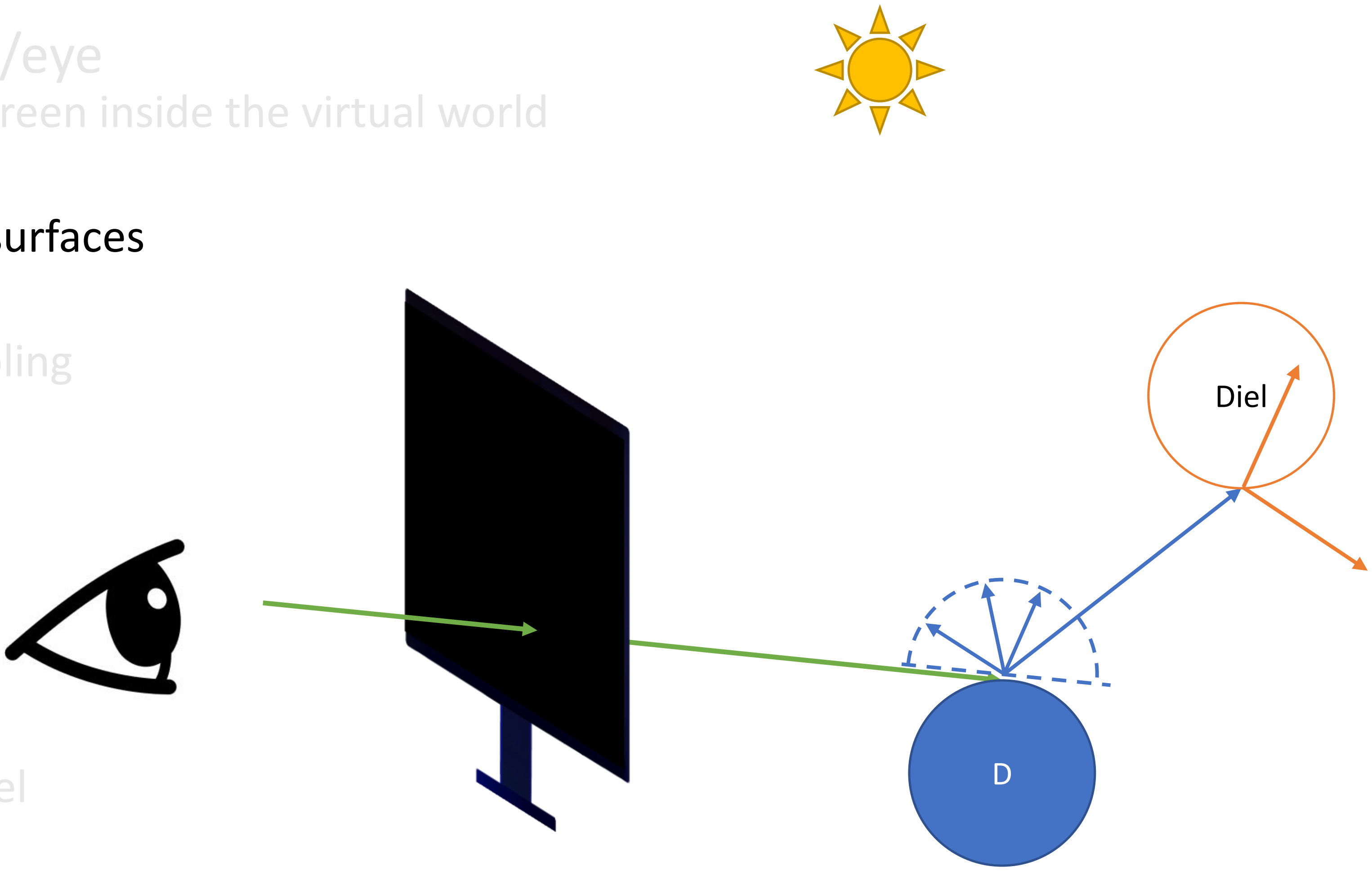
Remark

- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel



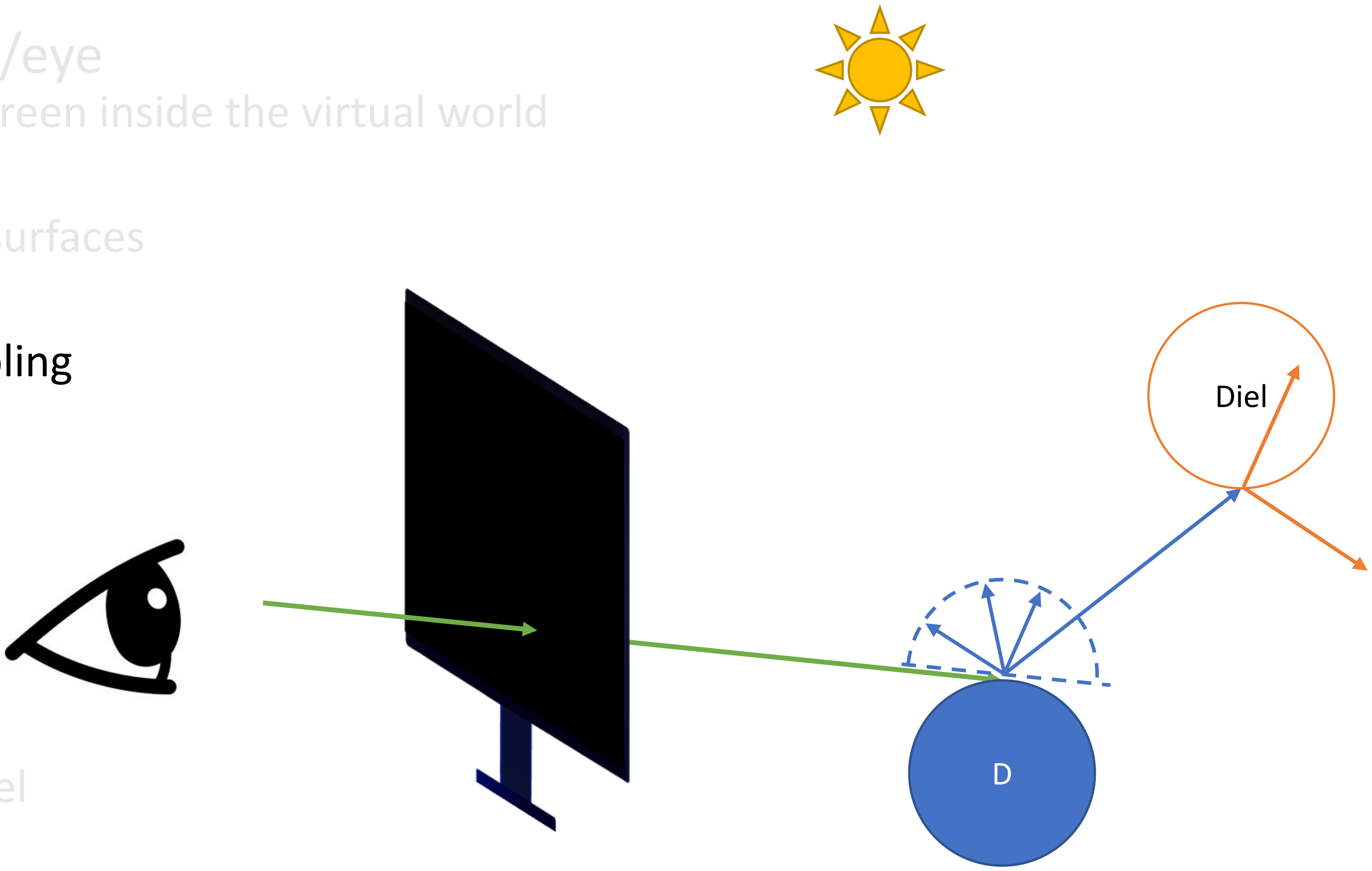
Remark

- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel



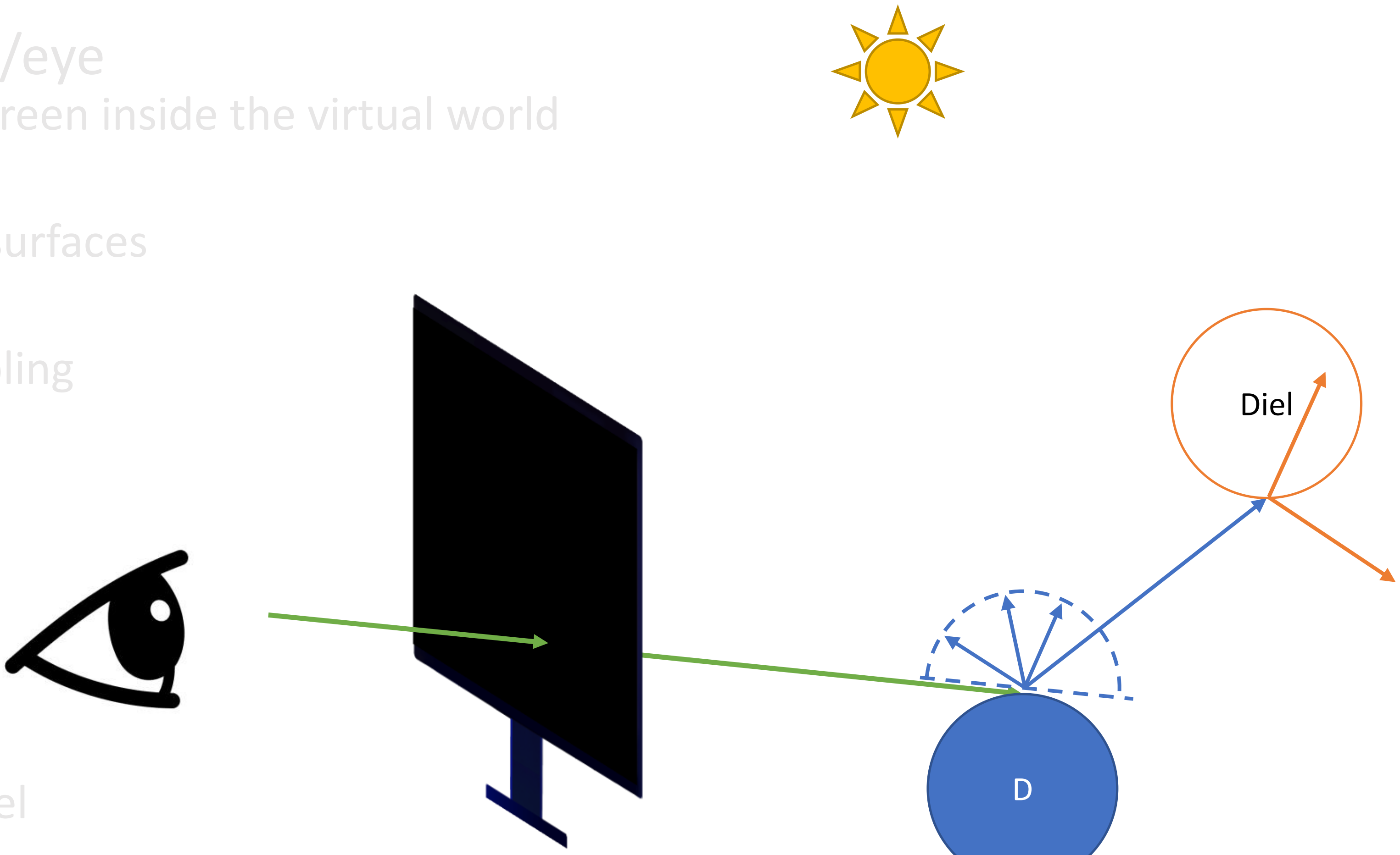
Remark

- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel



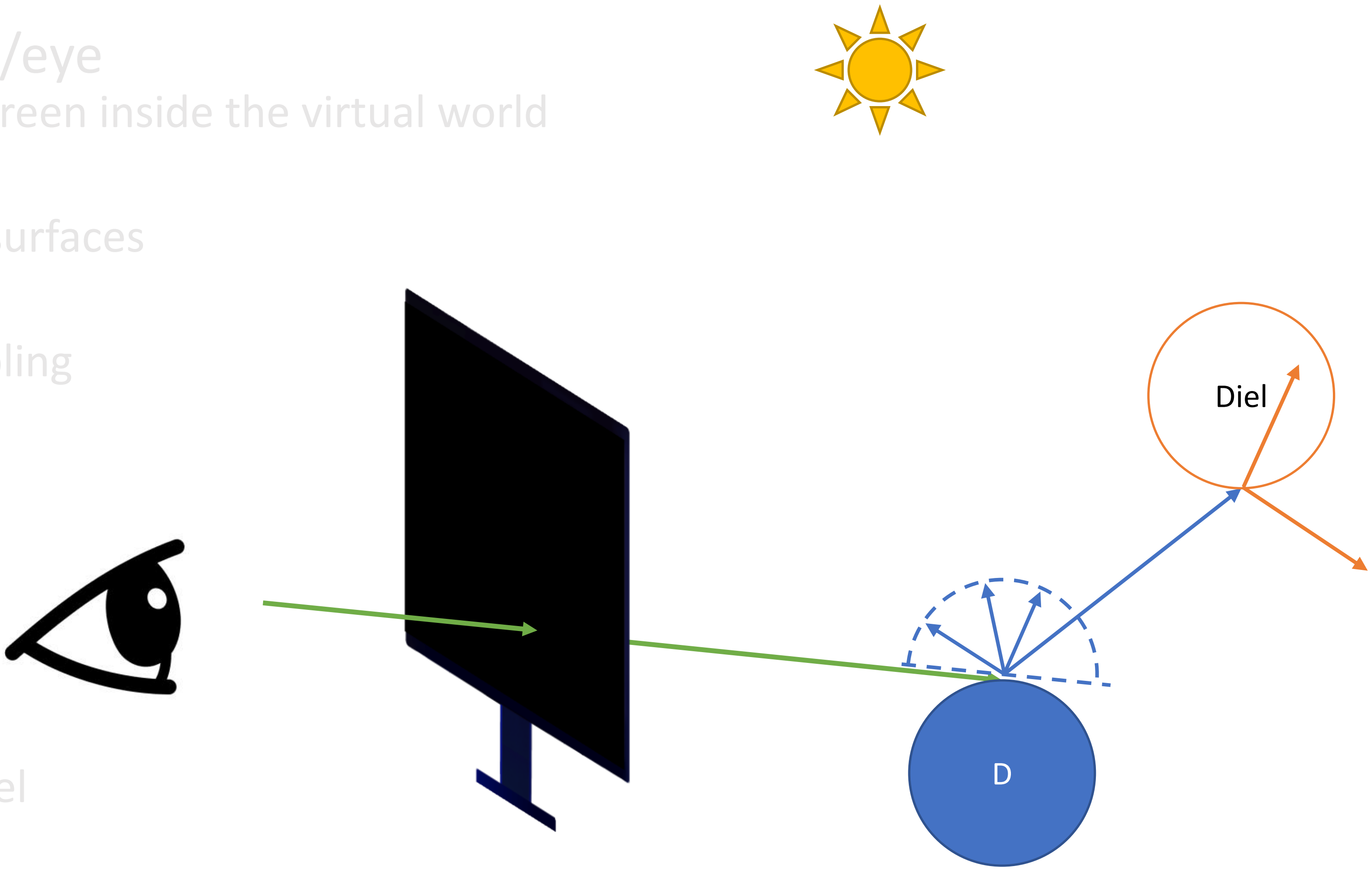
Remark

- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel



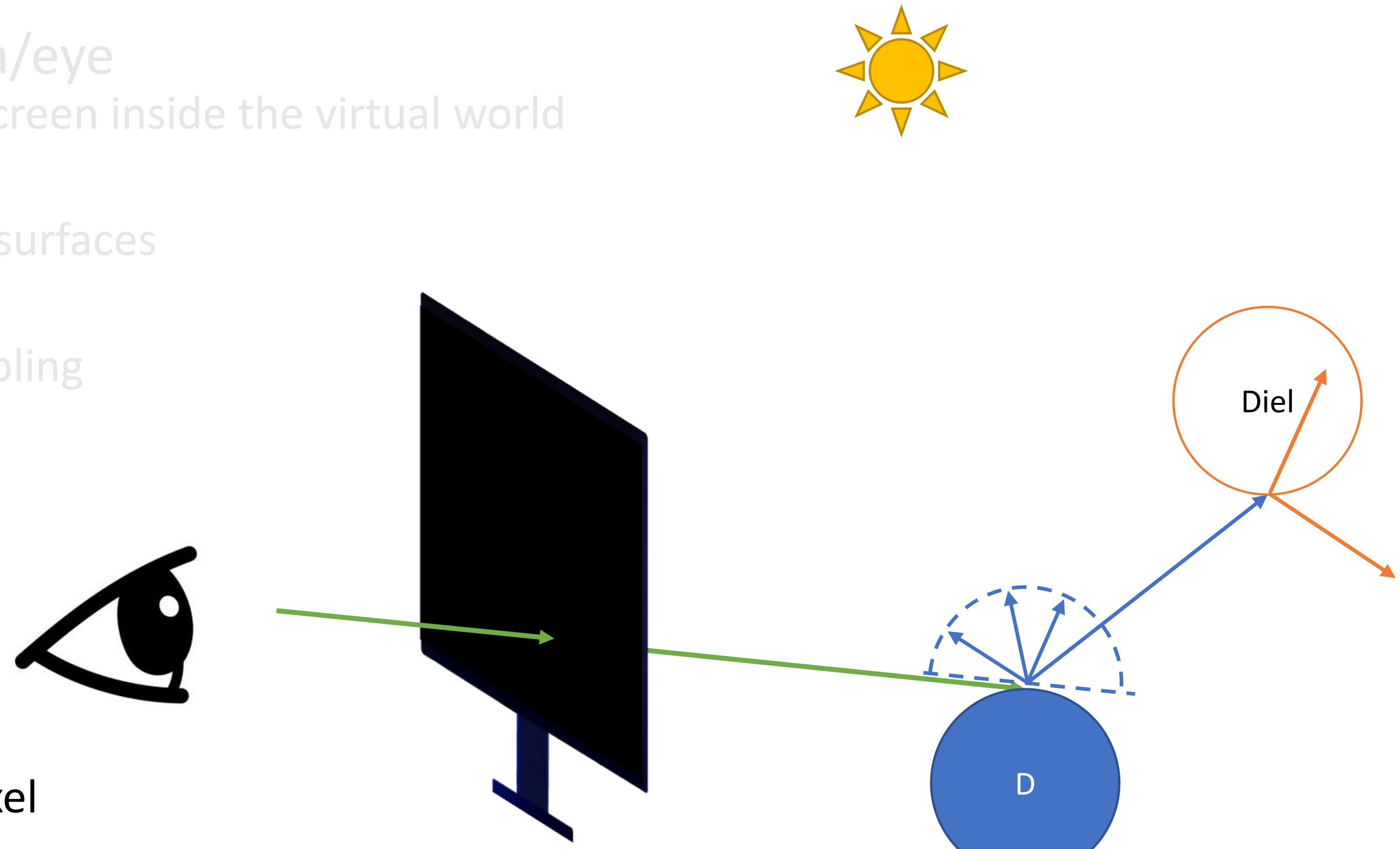
Remark

- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel



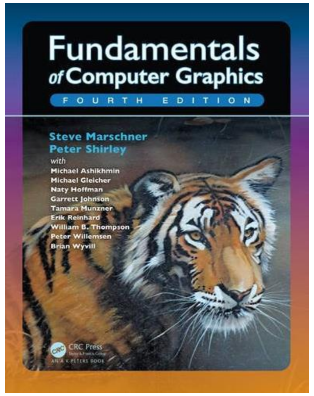
Remark

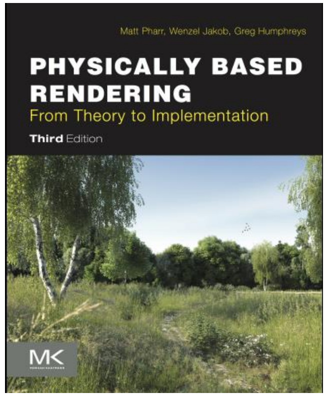
- Ray-casting from the camera/eye
  - Placing the camera and the screen inside the virtual world
- Ray-object intersection
  - Implicit surfaces v.s. polygon surfaces
- Sampling
  - Uniform v.s. importance sampling
- Reflection v.s. refraction
  - Snell's law
  - Fresnel's equations
- Recursions in Taichi
  - Converting them to loops
- Anti-aliasing?
  - Randomly sample inside a pixel

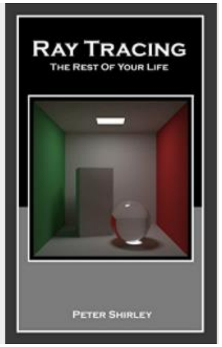


Further readings

- *Fundamentals of Computer Graphics* [Chapter 3, 4, 10.1 and 10.2]
- *Physically Based Rendering: From Theory To Implementation* [Link]
- *Ray Tracing…*
  - *In One Weekend* [Link]
  - *The Next Week* [Link]
  - *The Rest of Your Life* [Link]
- GAMES 101 [Lesson 13-16] [Link]
- GAMES 202 [Link]







Homework

Homework Today

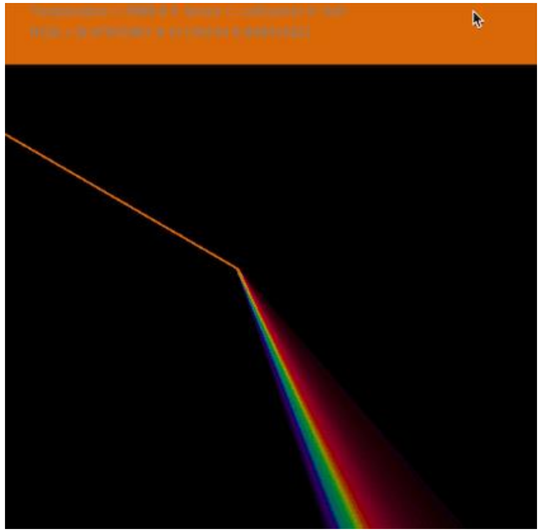
- Download the ray tracing examples in Taichi
  - https://github.com/taichiCourse01/taichi_ray_tracing
- Add a controllable camera by changing the camera settings
- Add a plane/torus/box/cylinder into the scene

Start your final project if you are into rendering

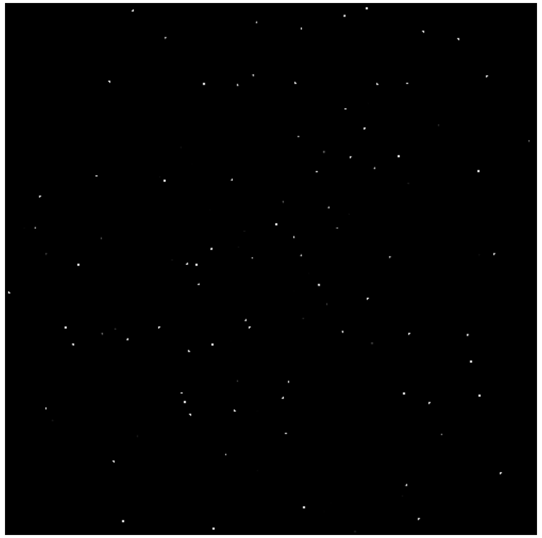
- Candidate topics:
  - Add one acceleration data structure for ray-object intersection tests.
    - Spatial hashing, Octree, kd-tree, etc.
  - Load an triangle mesh (such as .obj file) with texture
  - Bidirectional path tracing [Link]
  - Participating media [Link][Link]
  - Support BRDF/BTDF/BSSRDF etc. [Link]
  - Use your ray tracer to render your simulations (deformables/water/smoke etc.)

- Make sure your pictures look great ☺

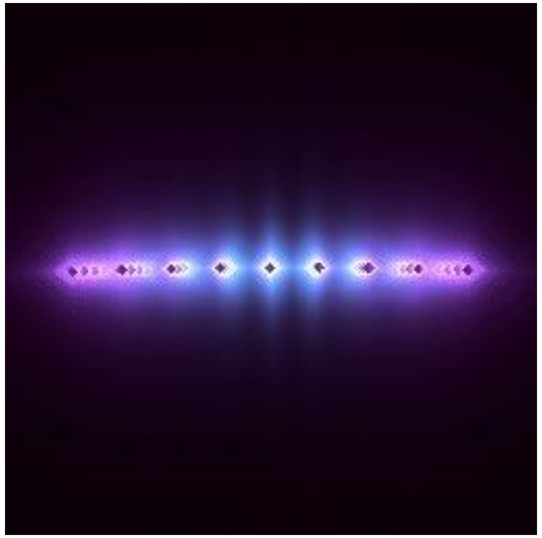
Excellent homework assignments



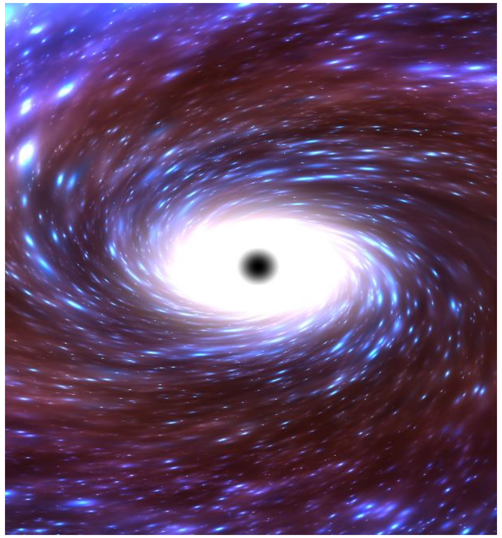
[@kphmd]



[@rockeyshao]



[@zhanlonqi]



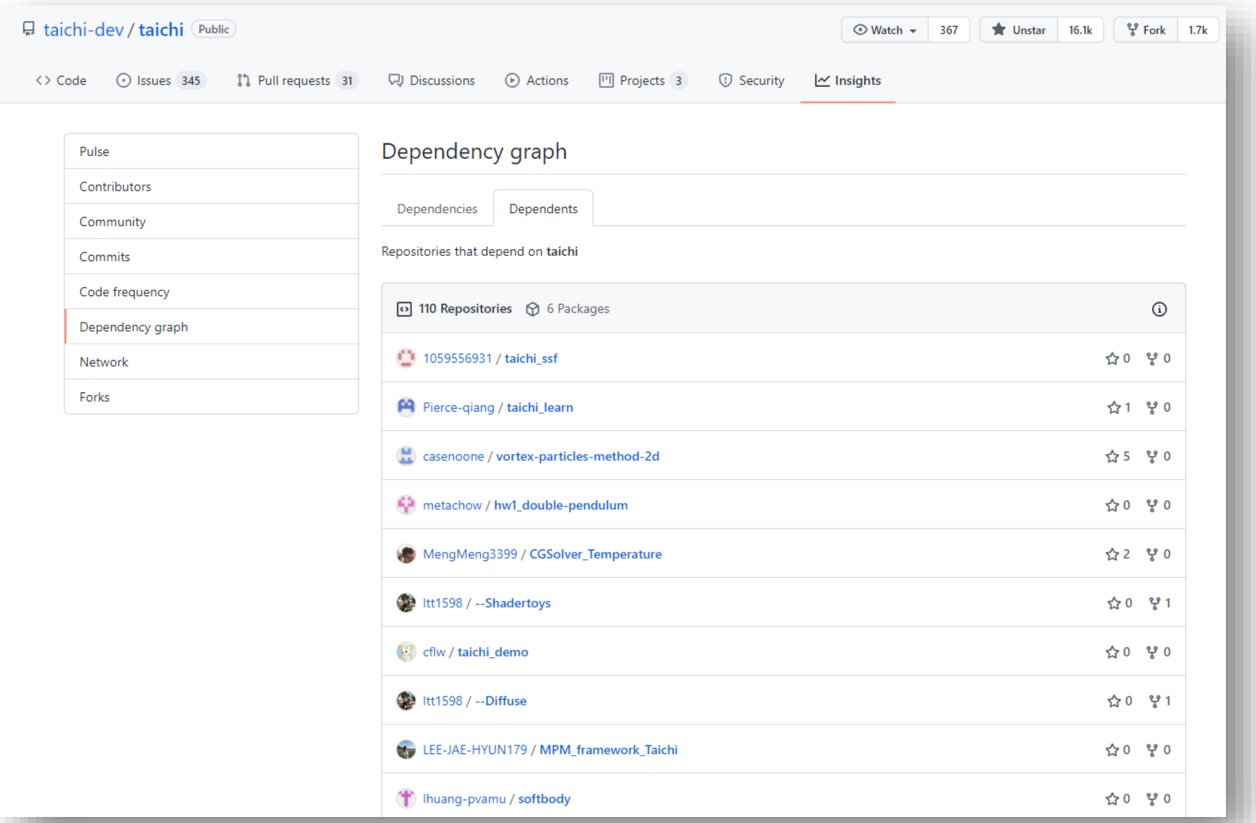
[@Huanghongru]

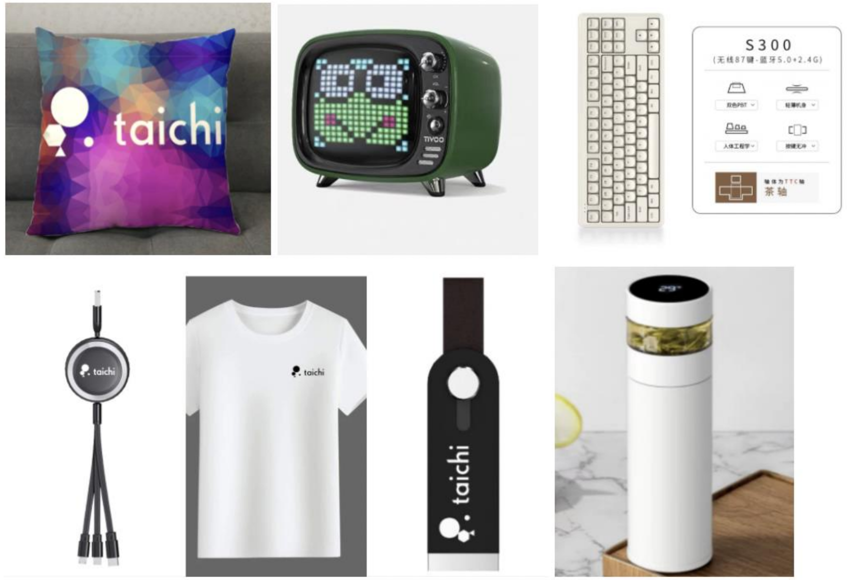
Excellent homework assignments

[@wangfeng70117]

Gifts for the gifted

- Use Template for your homework
- Lucky draw today ☺





Questions?

本次答疑：11/11 🡐作业分享也在这里

下次直播：11/16

直播回放：Bilibili 搜索「太极图形」

主页&课件：https://github.com/taichiCourse01

主页&课件(backup)：https://docs.taichi.graphics/tgc01