太极图形课

第06讲 Rendering: Basics of Ray Tracing

taichi

太极图形课

第06讲 Rendering: Basics of Ray Tracing

taichi

Previously in this Taichi Graphics Course…

Procedural Animation

Rendering

Deformable Simulation

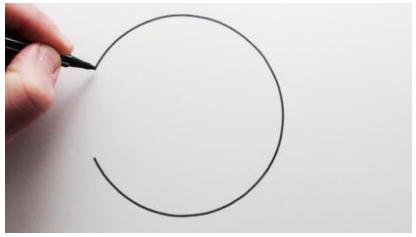
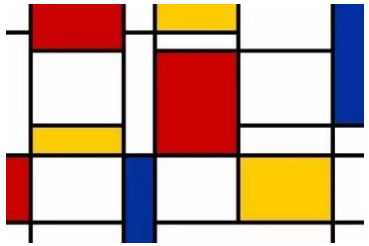
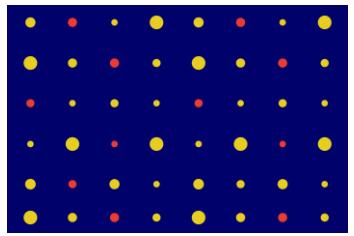
Fluid Simulation

Recap: procedural animation

- What's the color of this pixel?
  1. Setup your canvas
  2. Put some colors on your canvas
  3. Draw a basic unit
  4. Repeat the basic units: tiles and fractals
  5. Animate your pictures
  6. Introduce some randomness (Chaos!)

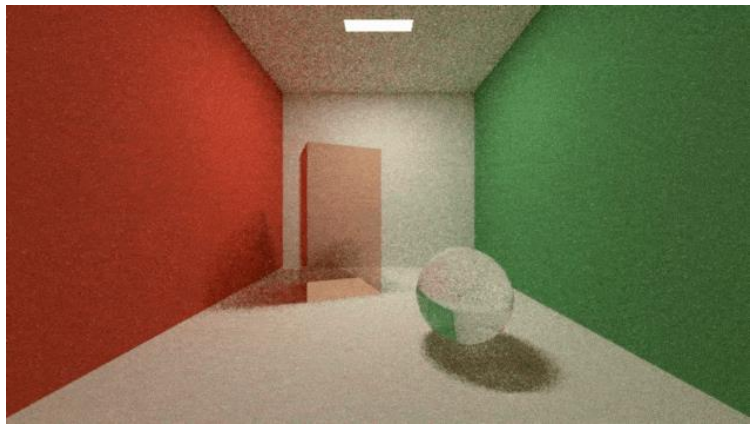
```
@ti.kernel
def render():
    for i,j in pixels:
        color = # pick a color
        pixels[i,j] = color
```




In the two following classes…

Procedural Animation



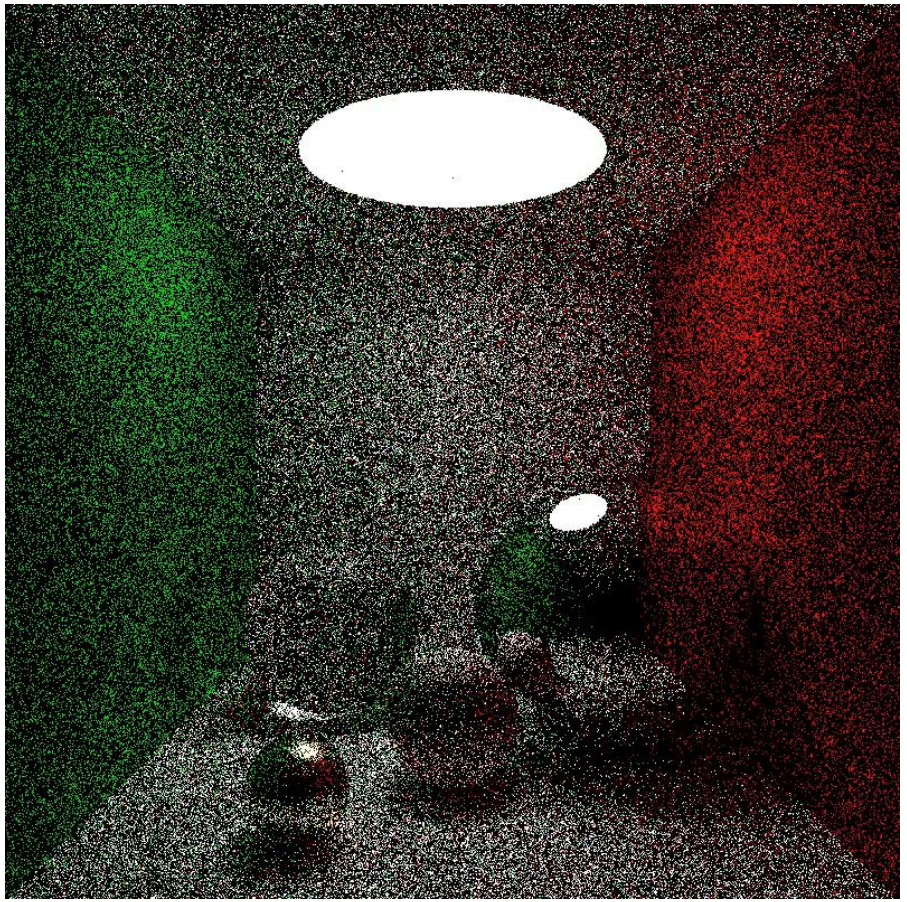
Rendering

Deformable Simulation

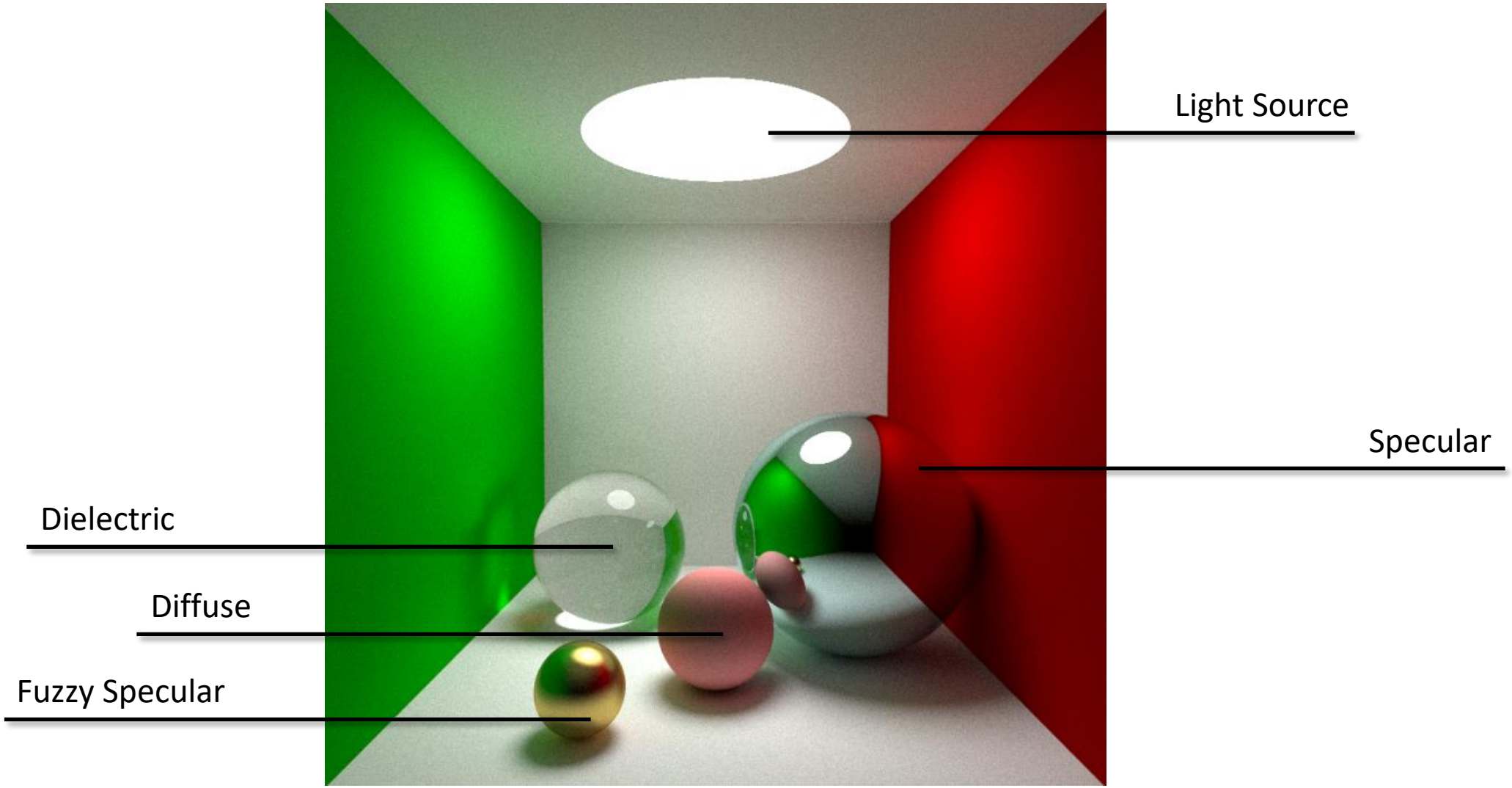
Fluid Simulation

The codebase in the following two classes

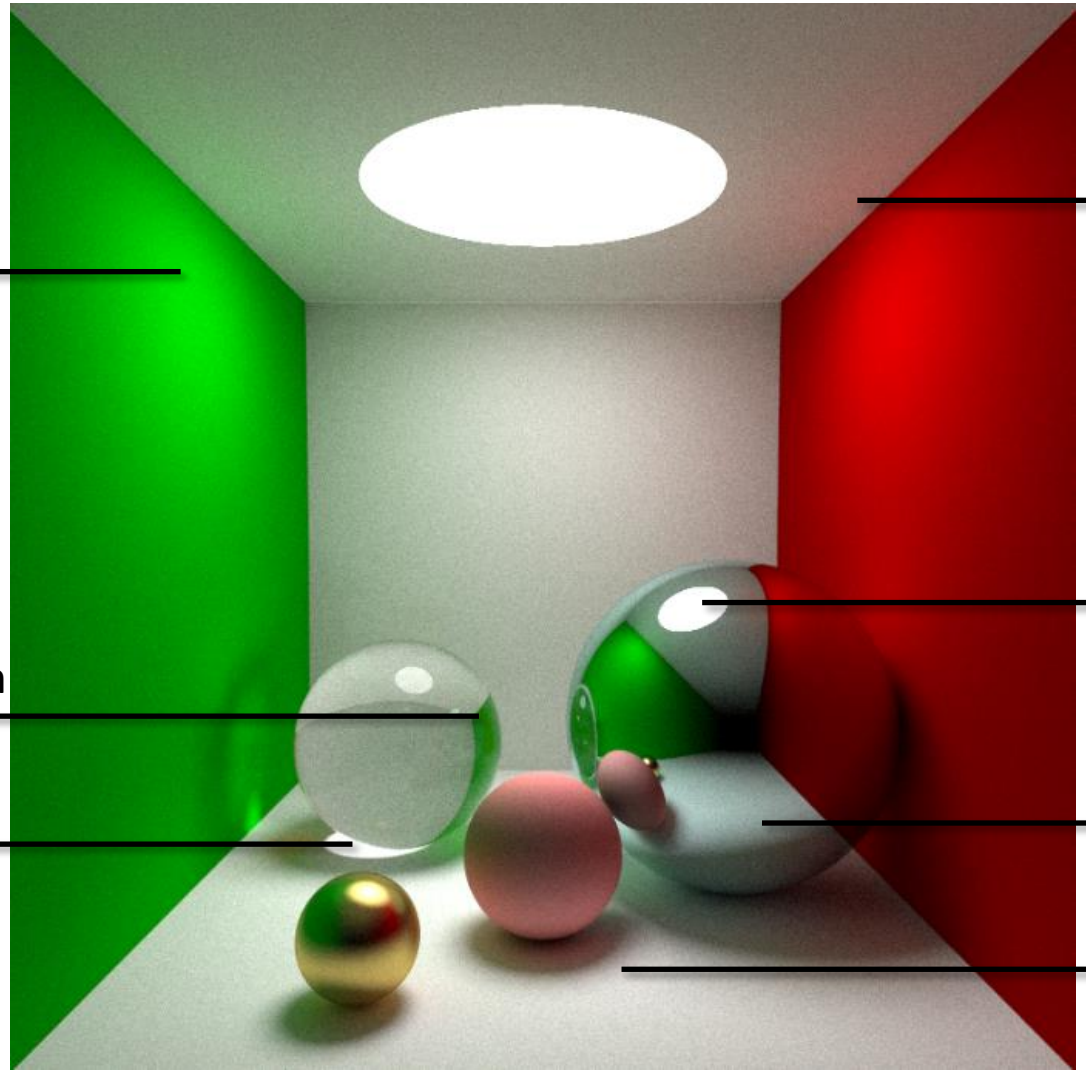
- https://github.com/taichiCourse01/taichi_ray_tracing
- Main reference:
  - Ray tracing in one weekend [Link]



What can we get from a ray tracer



What can we get from a ray tracer



Ray tracing in a nutshell

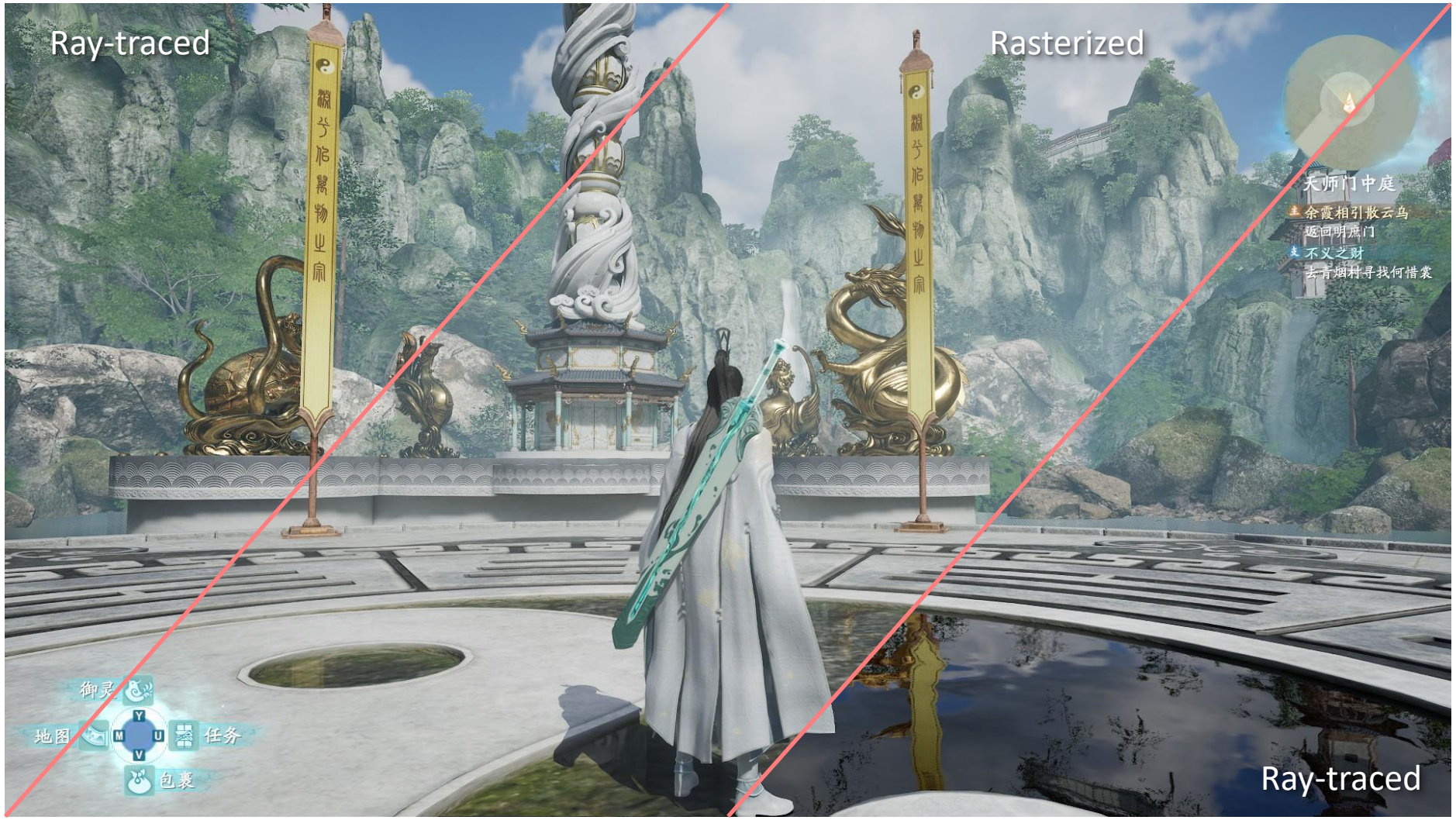
Rendering techniques

- Real-time rendering
  - Rasterization based methods
- Offline rendering
  - Ray-tracing based methods

[Park and Baek, 2021]

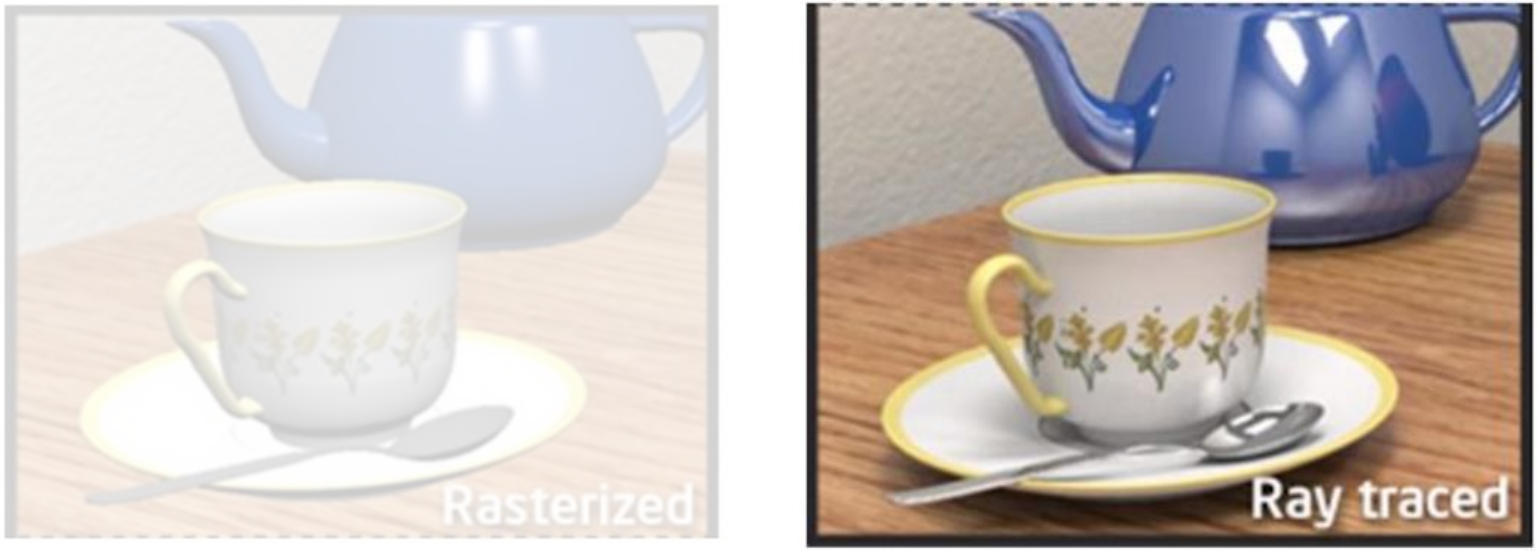
What I’ve been playing recently:

What I’ve been playing recently:



What we are going to cover today:

- Real-time rendering
  - Rasterization based methods
- Offline rendering
  - Ray-tracing based methods

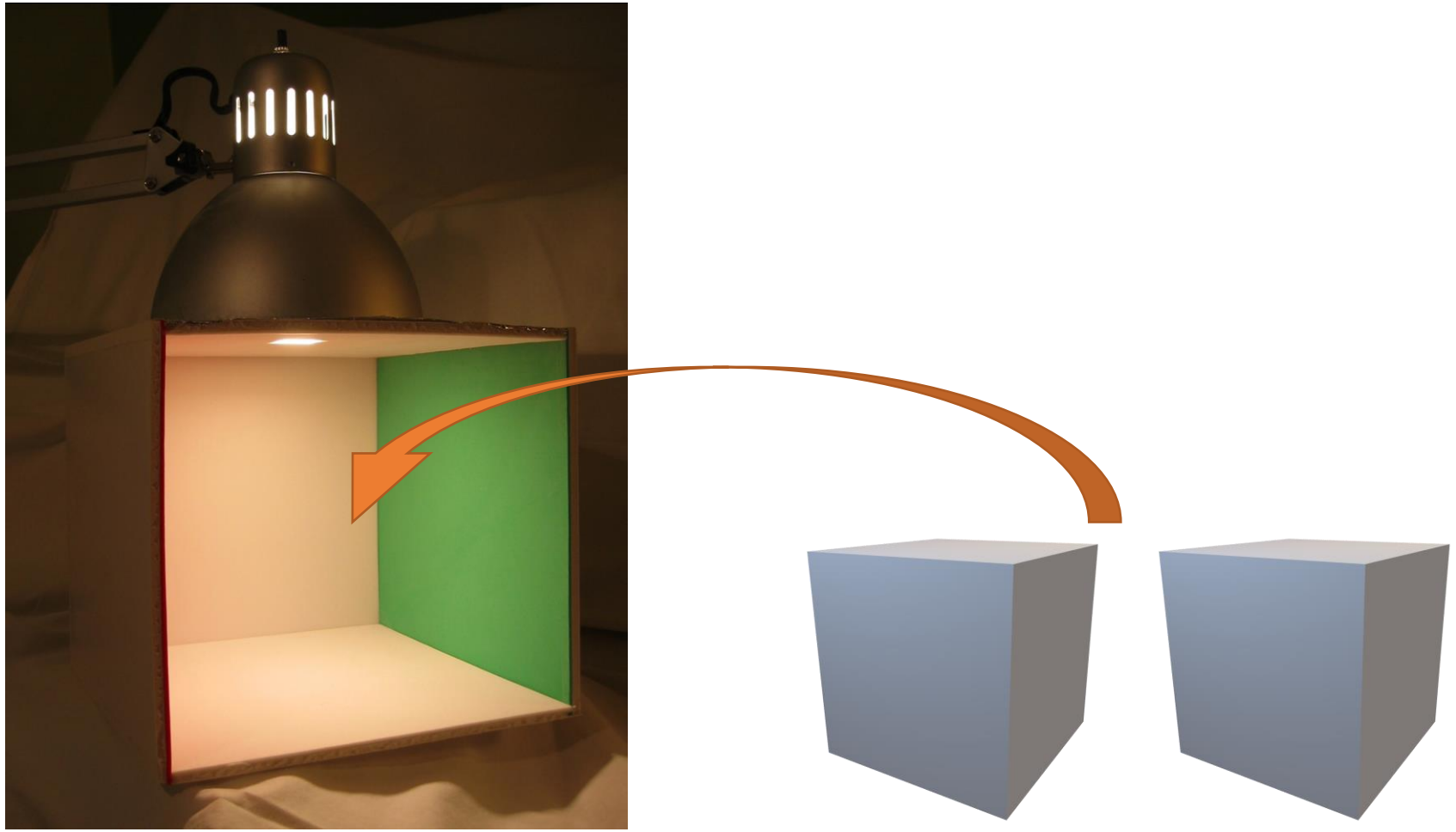


[Park and Baek, 2021]

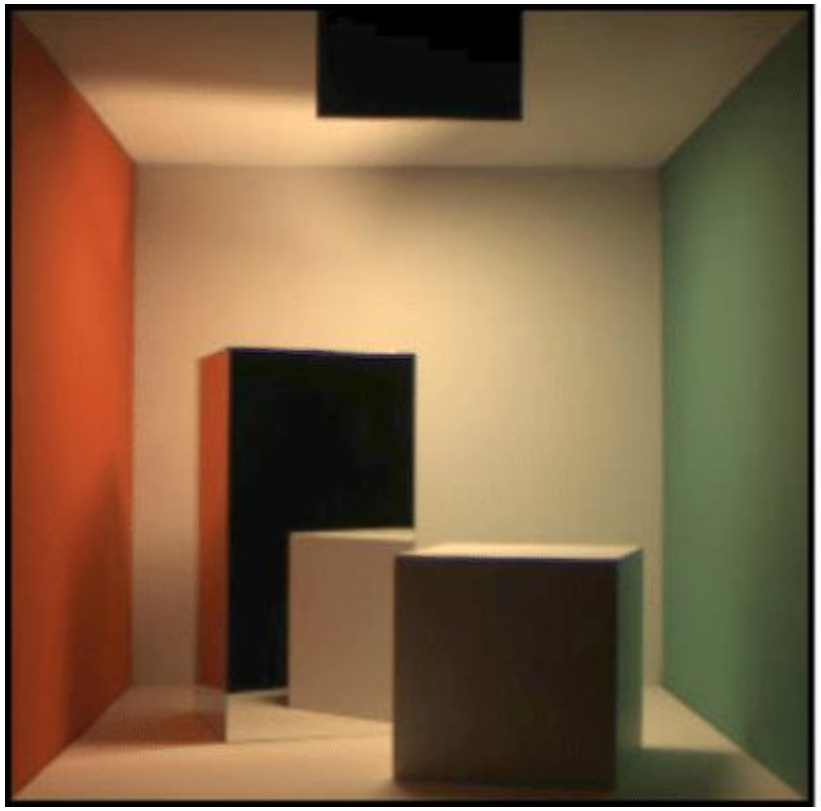
Why ray tracing?

- It provides us nicely-looking pictures
- It is conceptually easier to understand

The Cornell Box [Link]



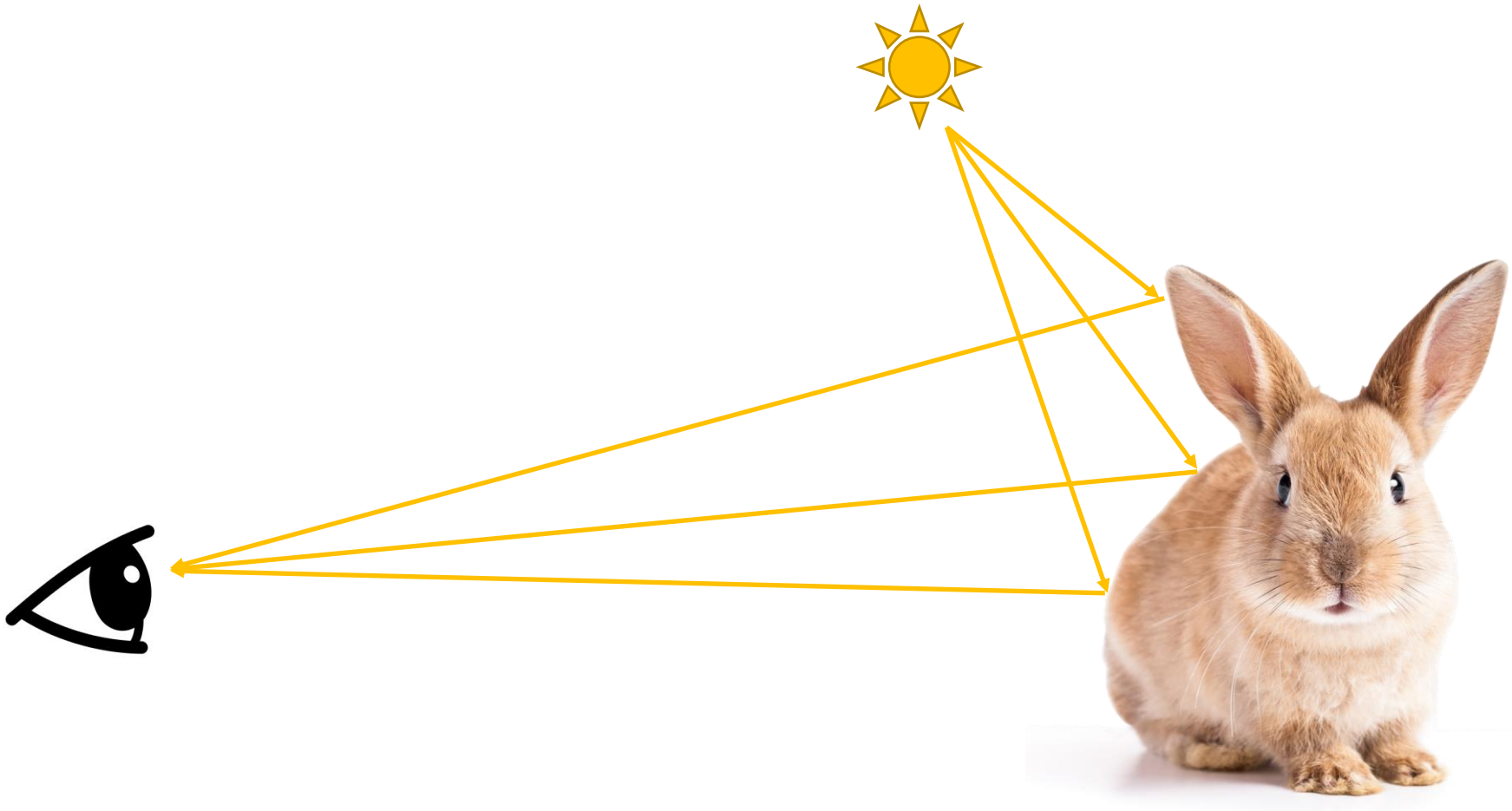
Which one is ray-traced?



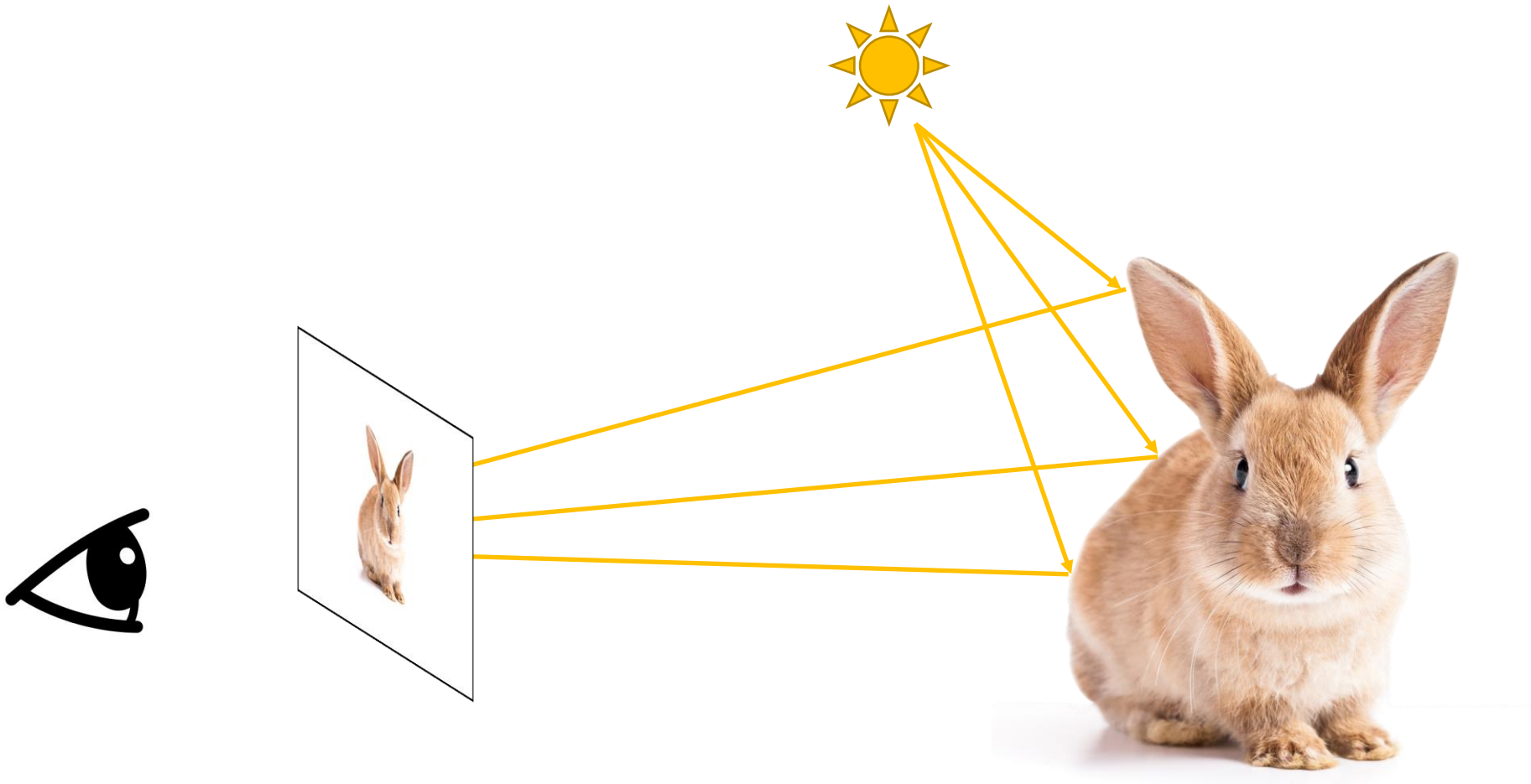
Real

Rendered

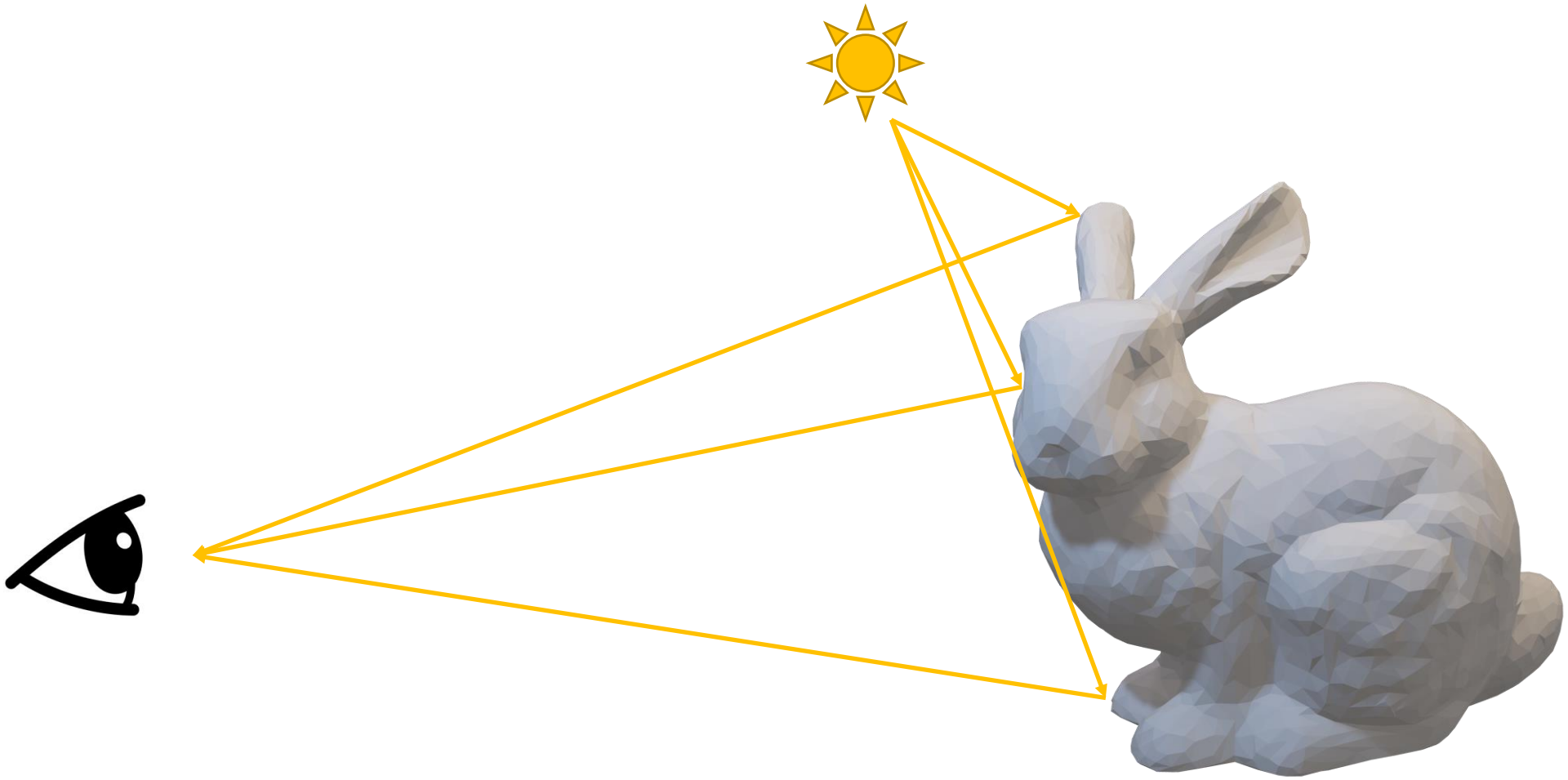
How do we see the world



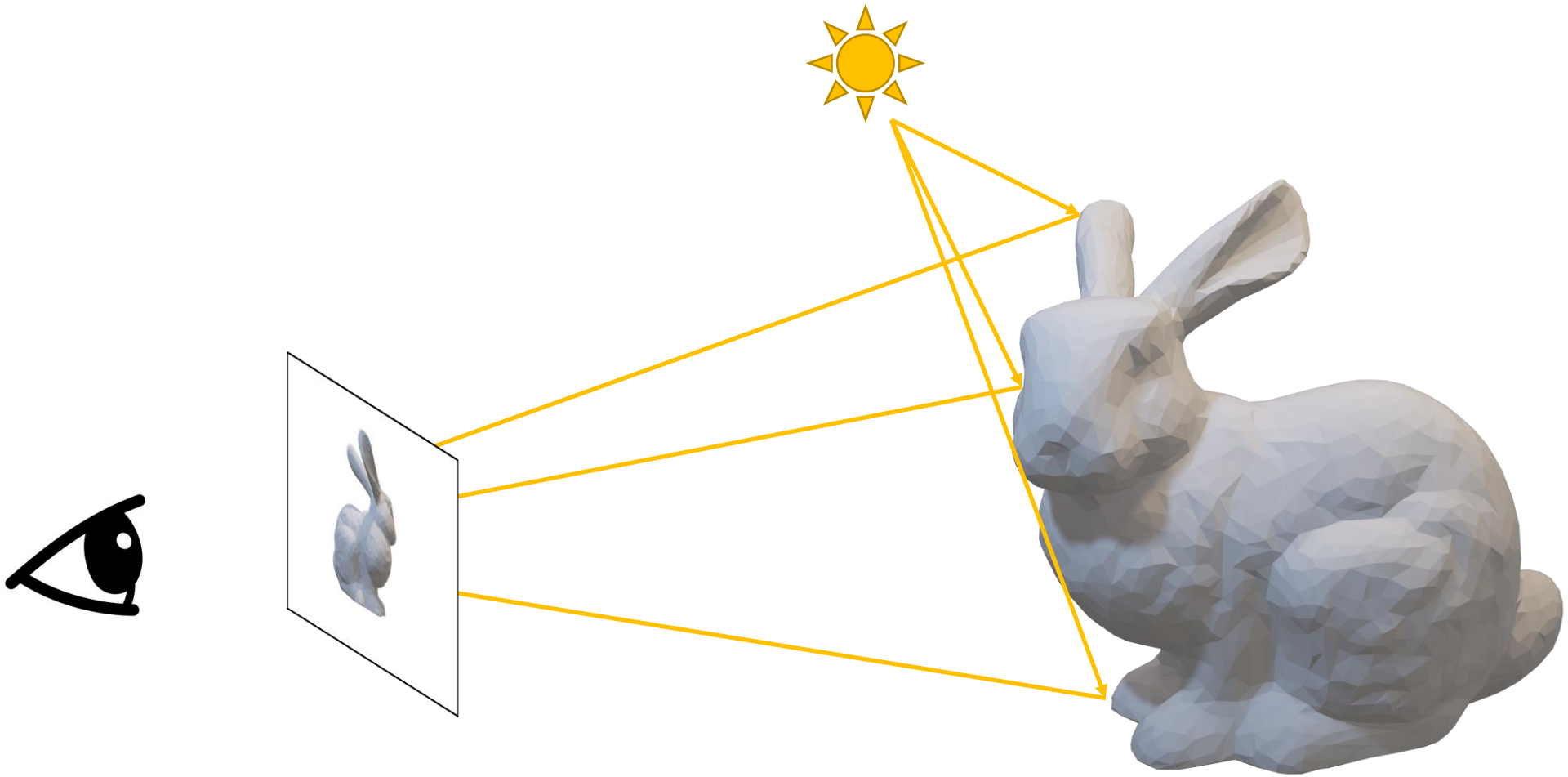
How do we see the world



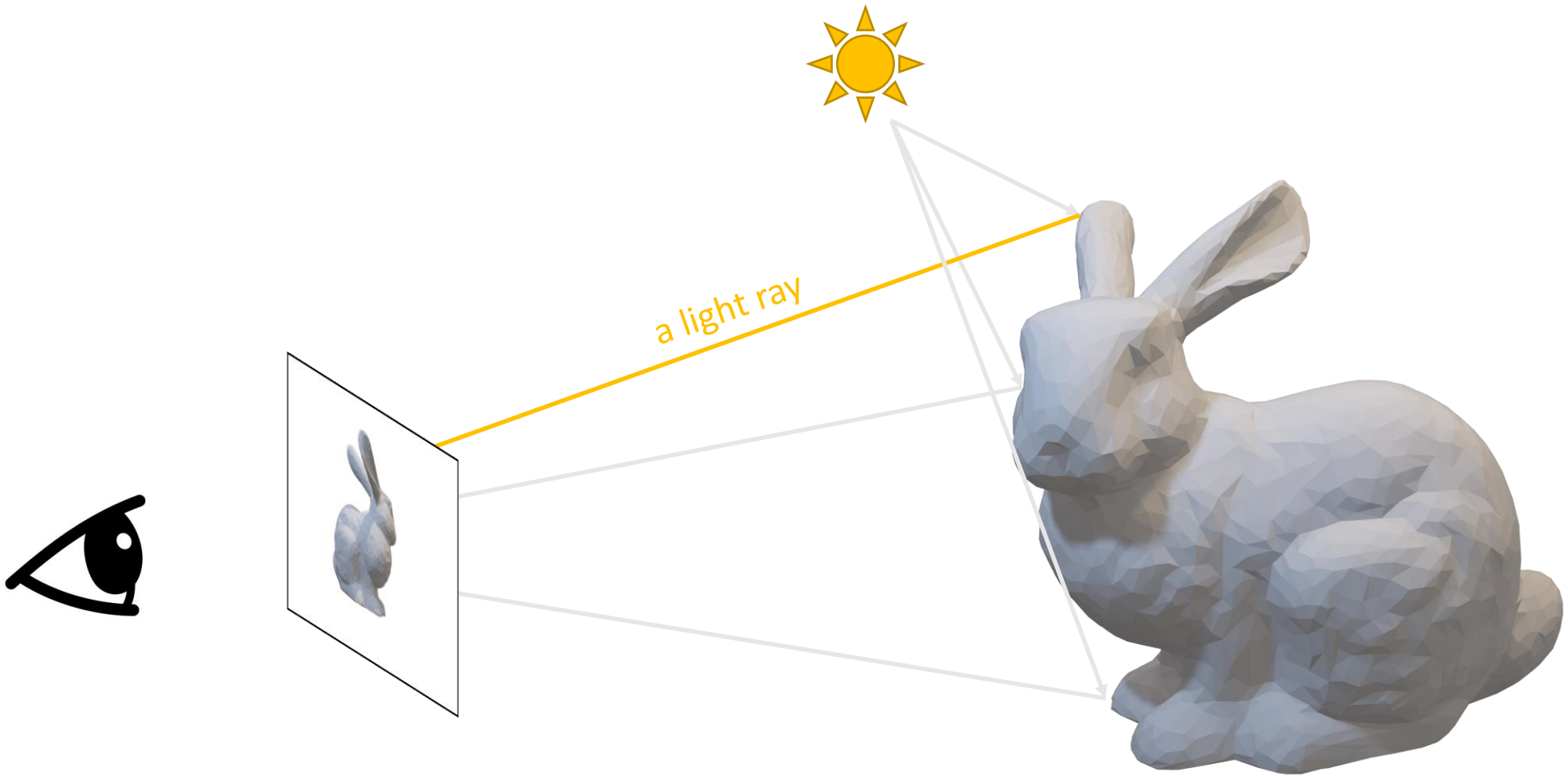
How do we see the (virtual) world



How do we see the (virtual) world

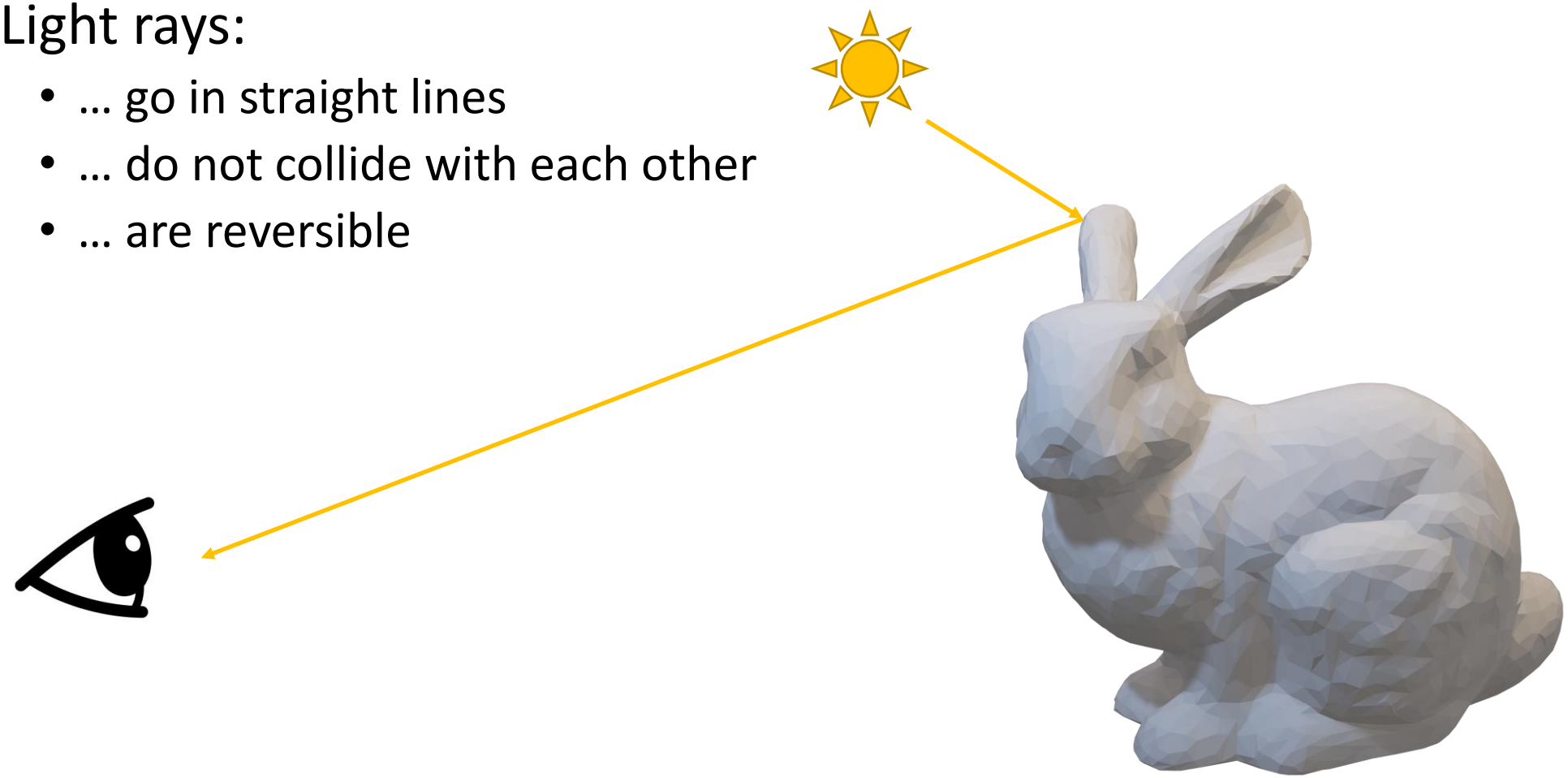


How do we see the (virtual) world



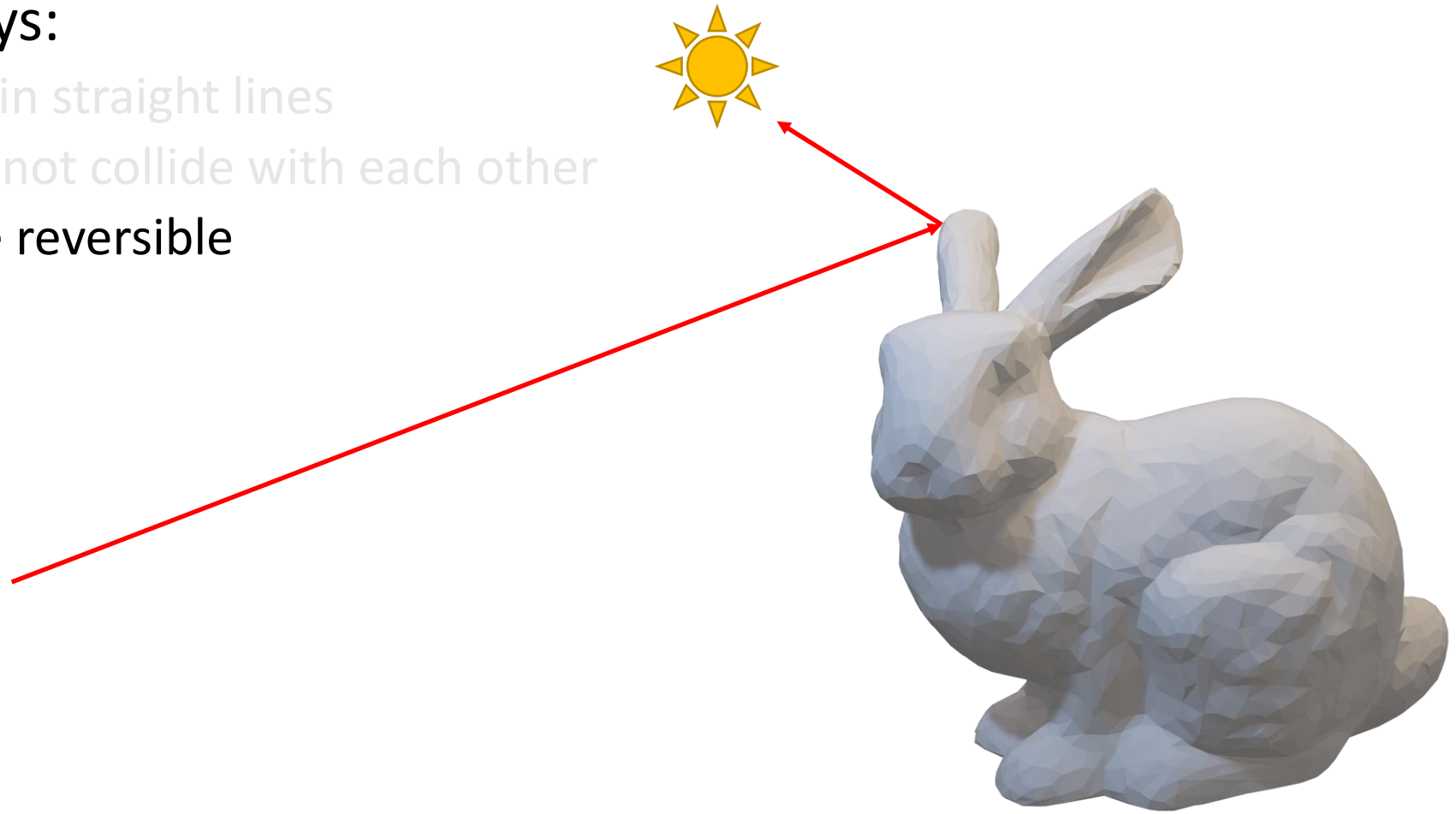
Assumptions of the light rays

- Light rays:
  - … go in straight lines
  - … do not collide with each other
  - … are reversible



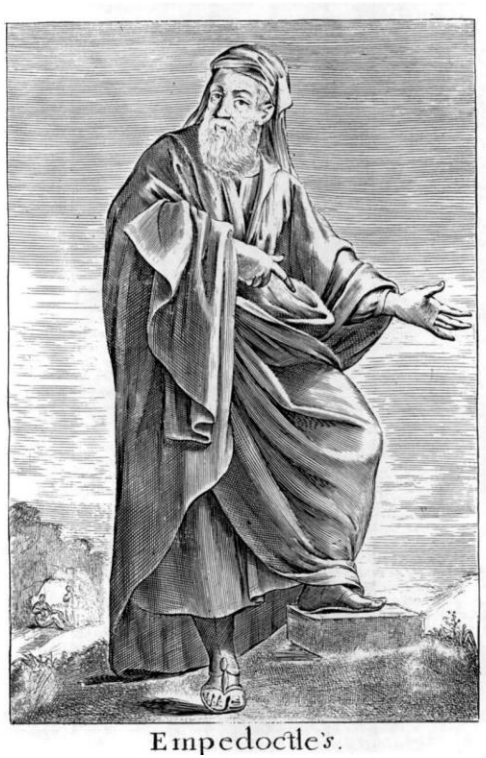
Assumptions of the light rays

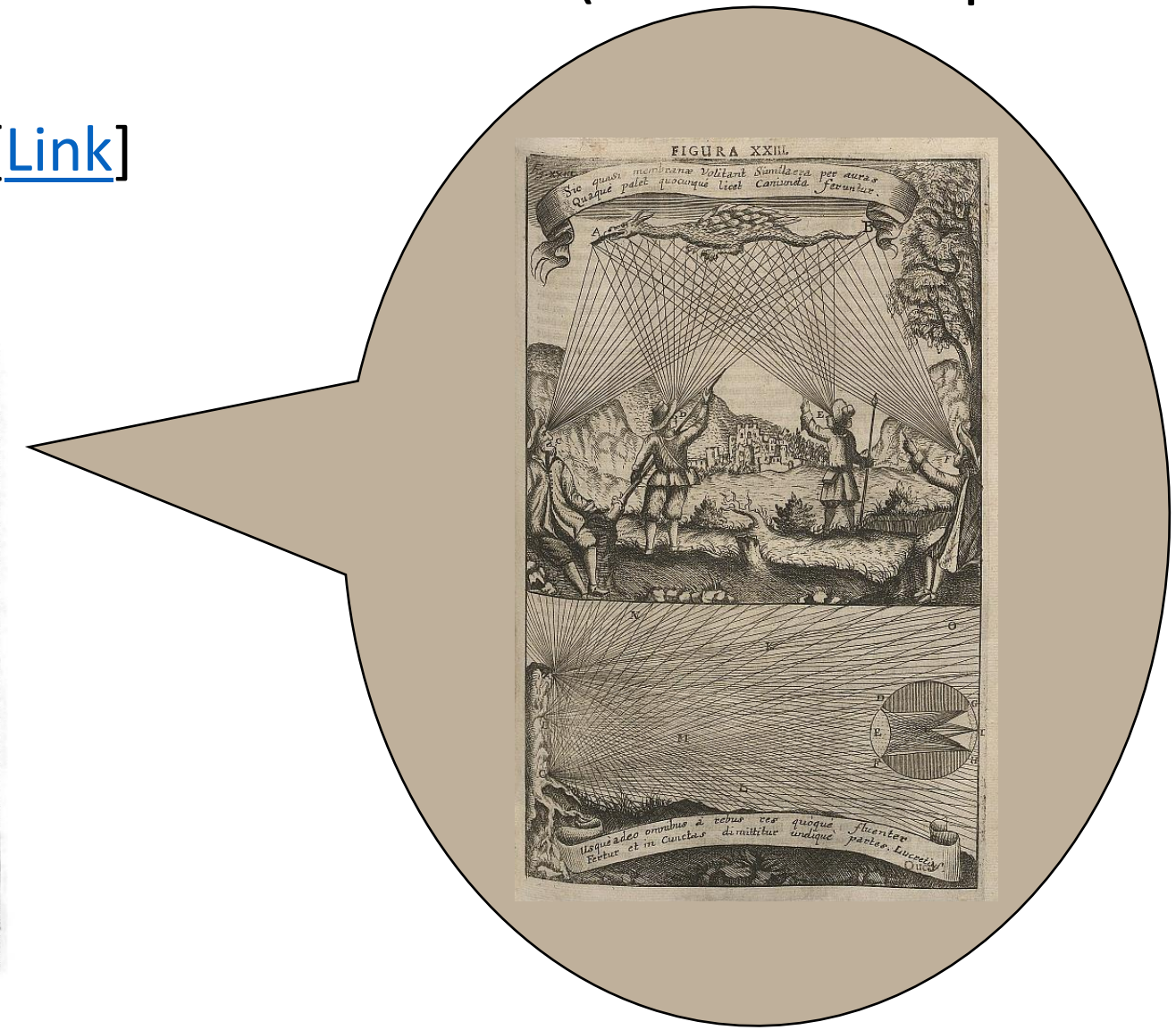
- Light rays:
  - … go in straight lines
  - … do not collide with each other
  - … are reversible



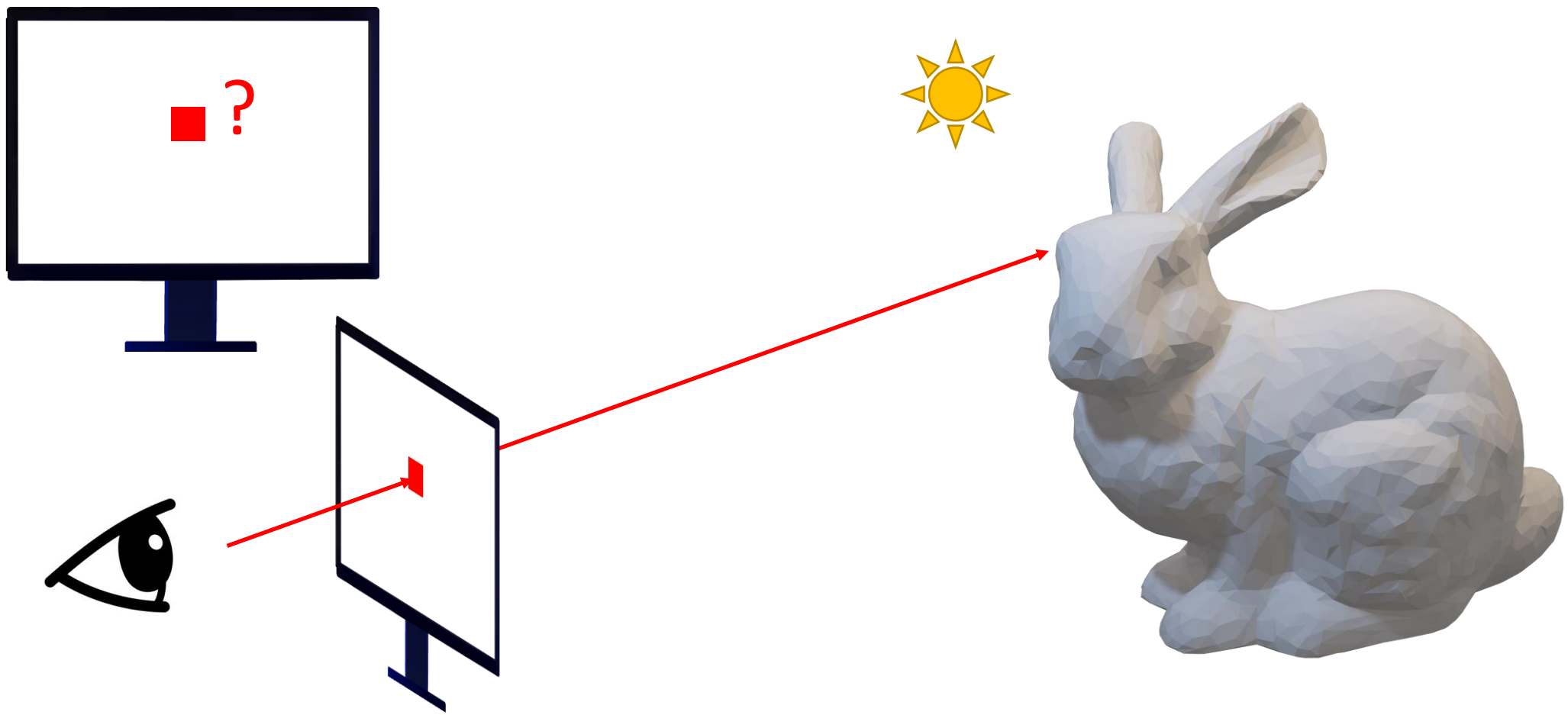
How to we see the world (from Empedocles)

- Emission theory [Link]



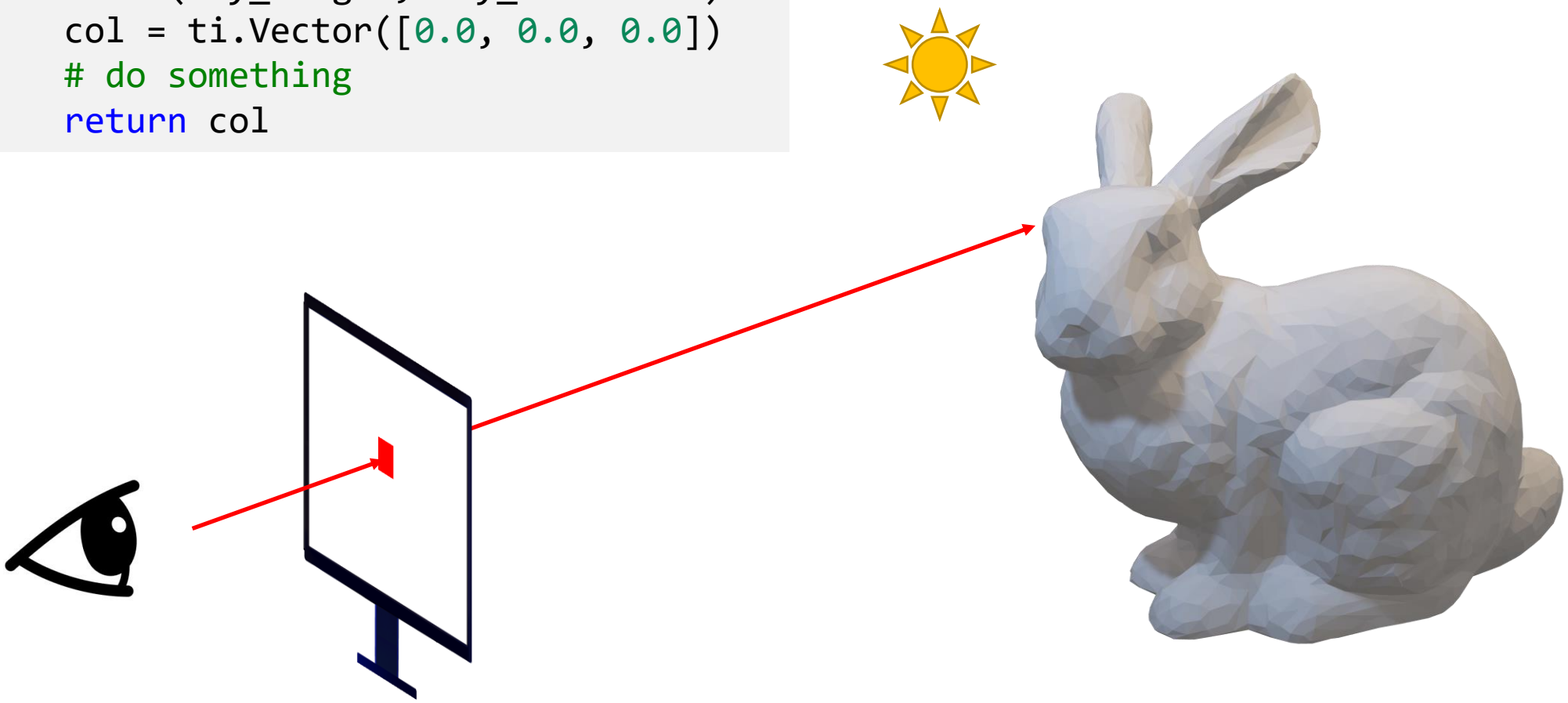


The ultimate question in rendering:
What is the color of this pixel?

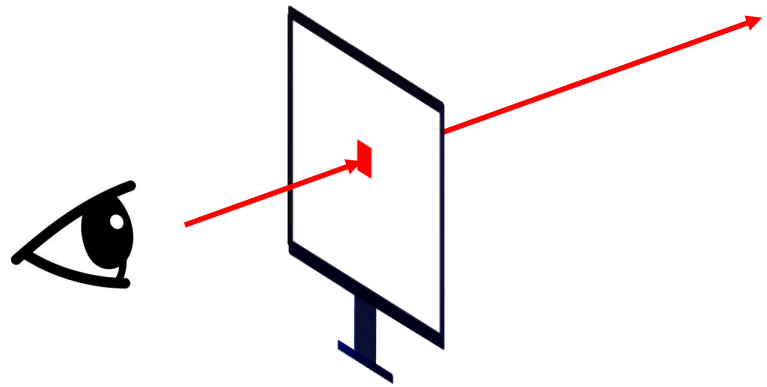


The ultimate question in ray tracing:
What color does the ray see?

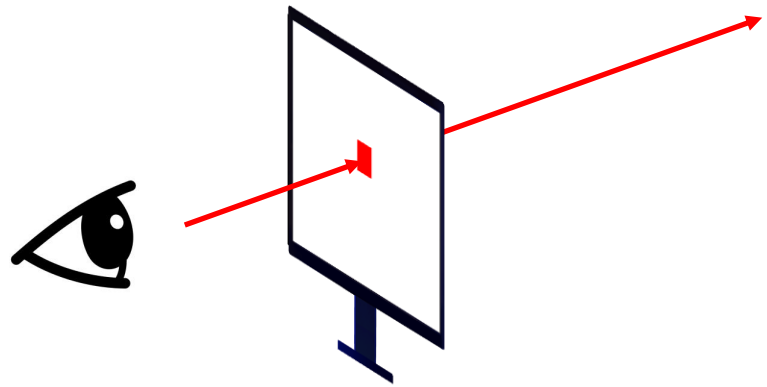
```
def color(ray_origin, ray_direction):
    col = ti.Vector([0.0, 0.0, 0.0])
    # do something
    return col
```



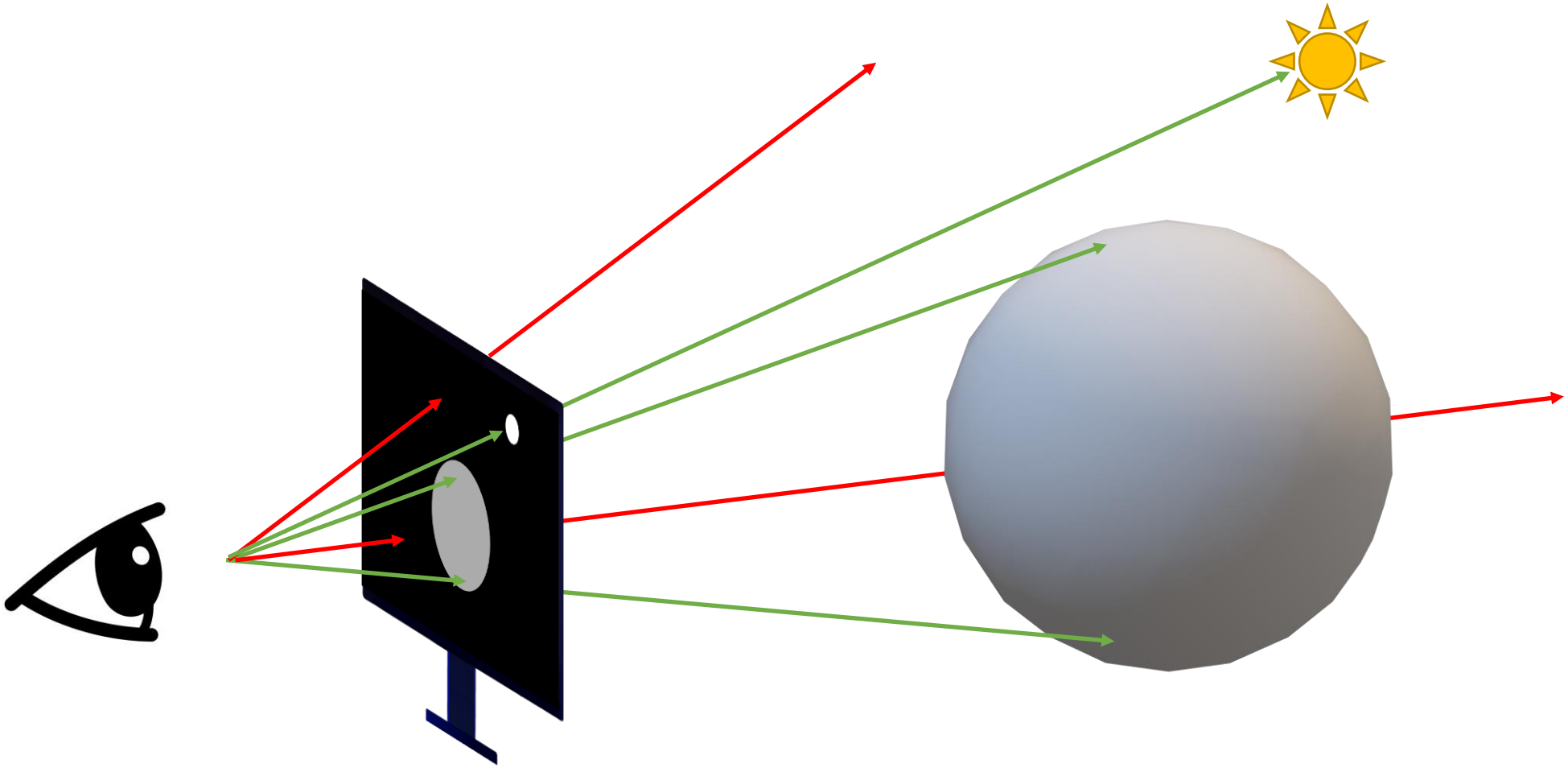
What color does the ray see?



Option I: color

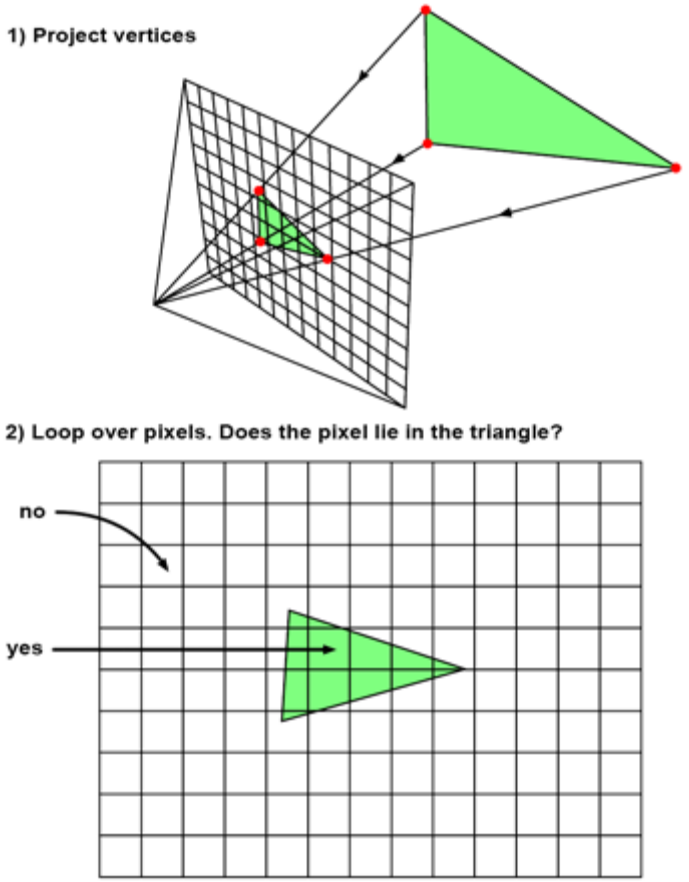


Fill the color of the first hit object



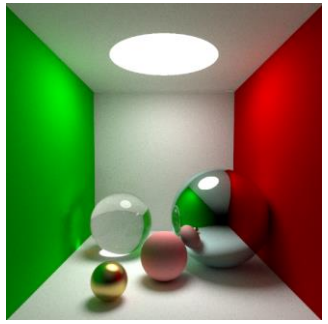
Usually done with (perspective) projections

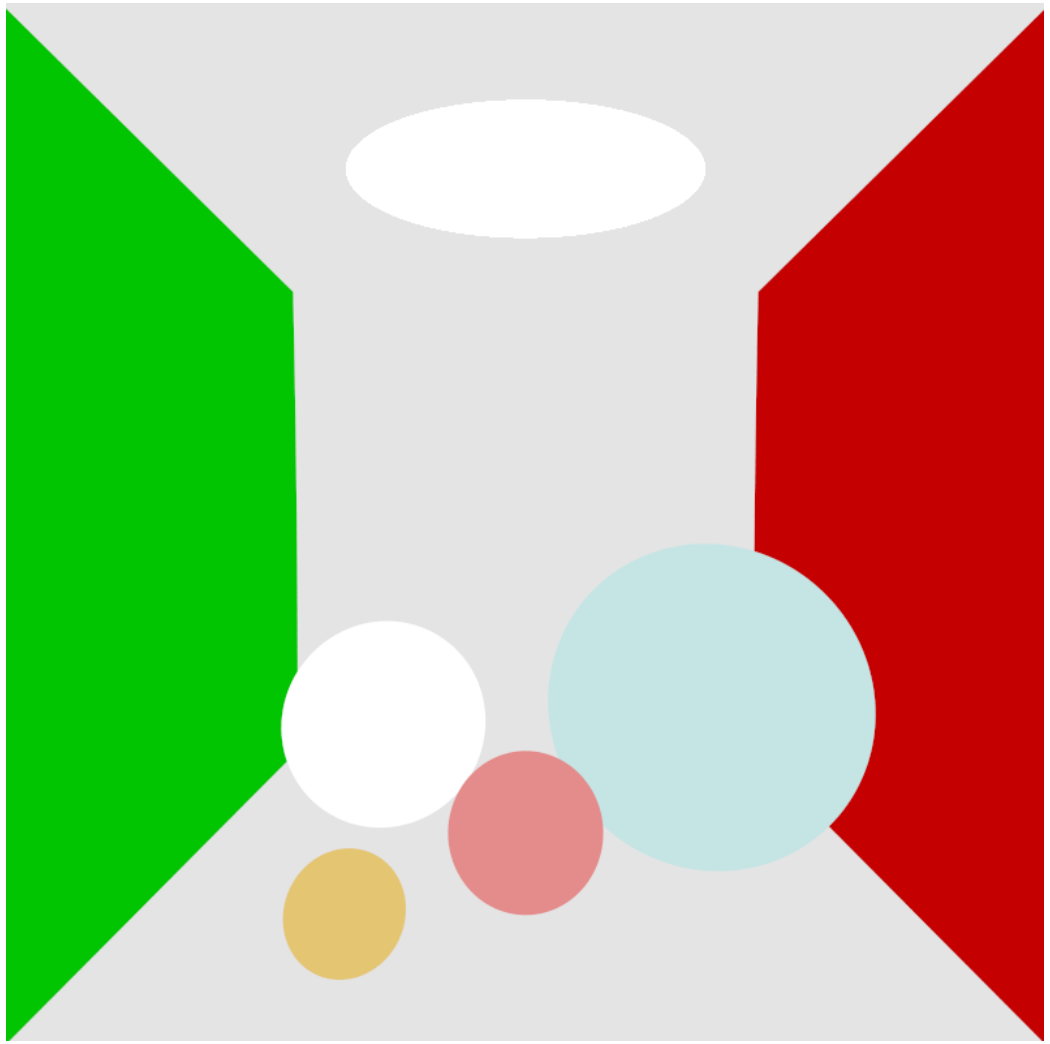
- Considering an 256x128 image
- Ray-tracing style:
  - Generate 256x128 rays in 3D
  - Check Ray-triangle intersection 256x128 times in 3D
- Rasterization style:
  - Project 3 points into the 2D plane
  - Check if a point is inside the triangle 256x128 times in 2D



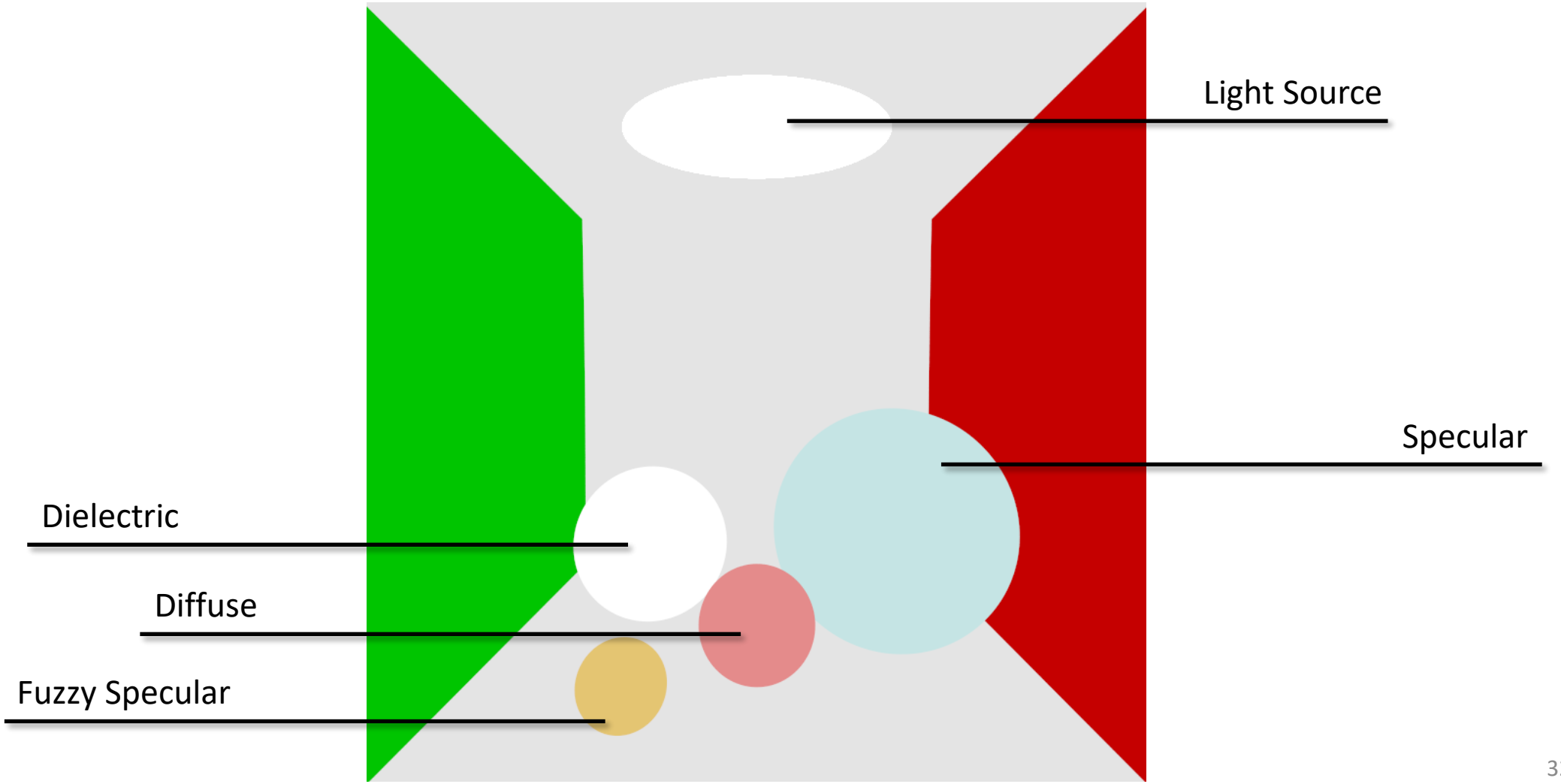
[Image courtesy of scratchapixel.com]

Returns flat-looking results

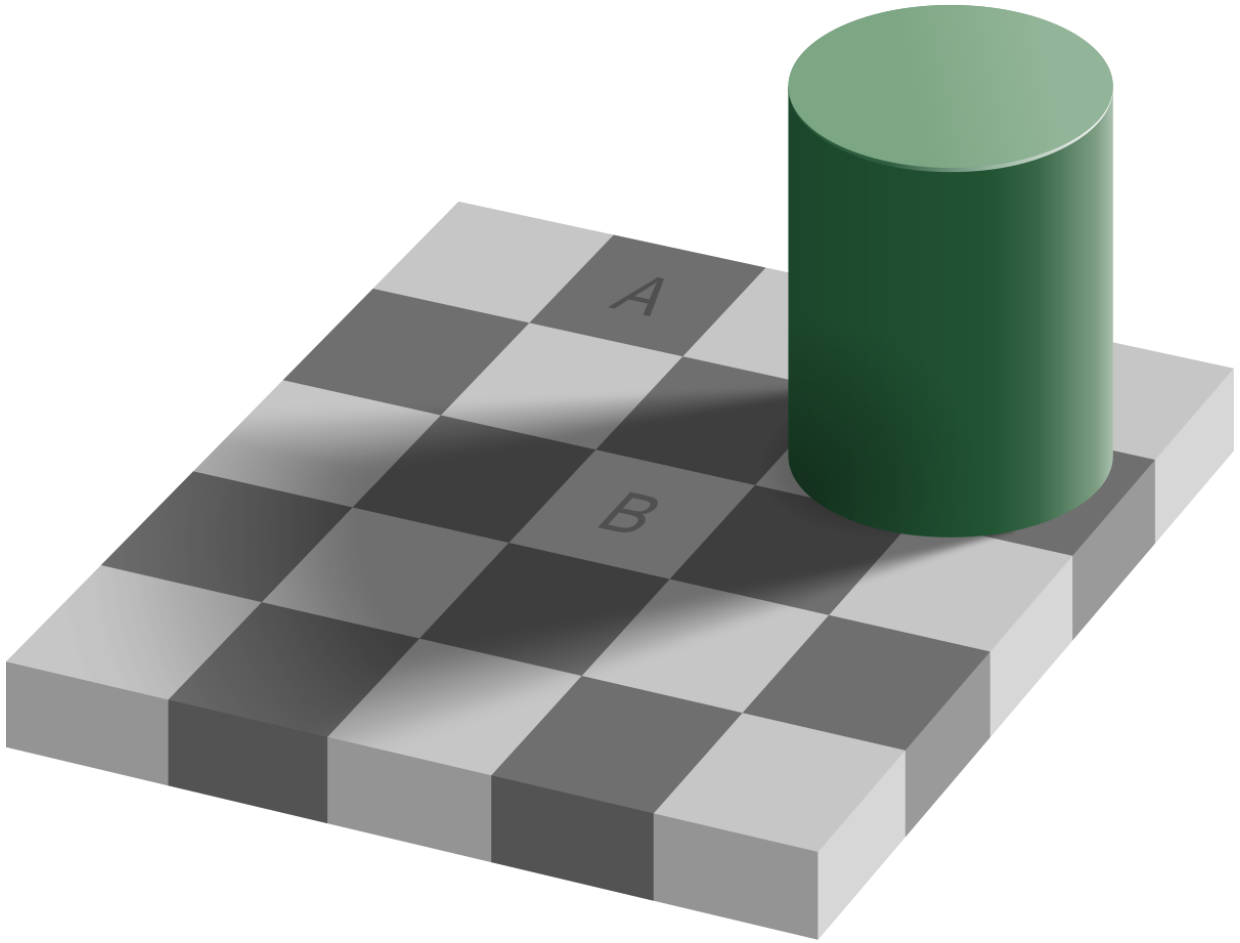




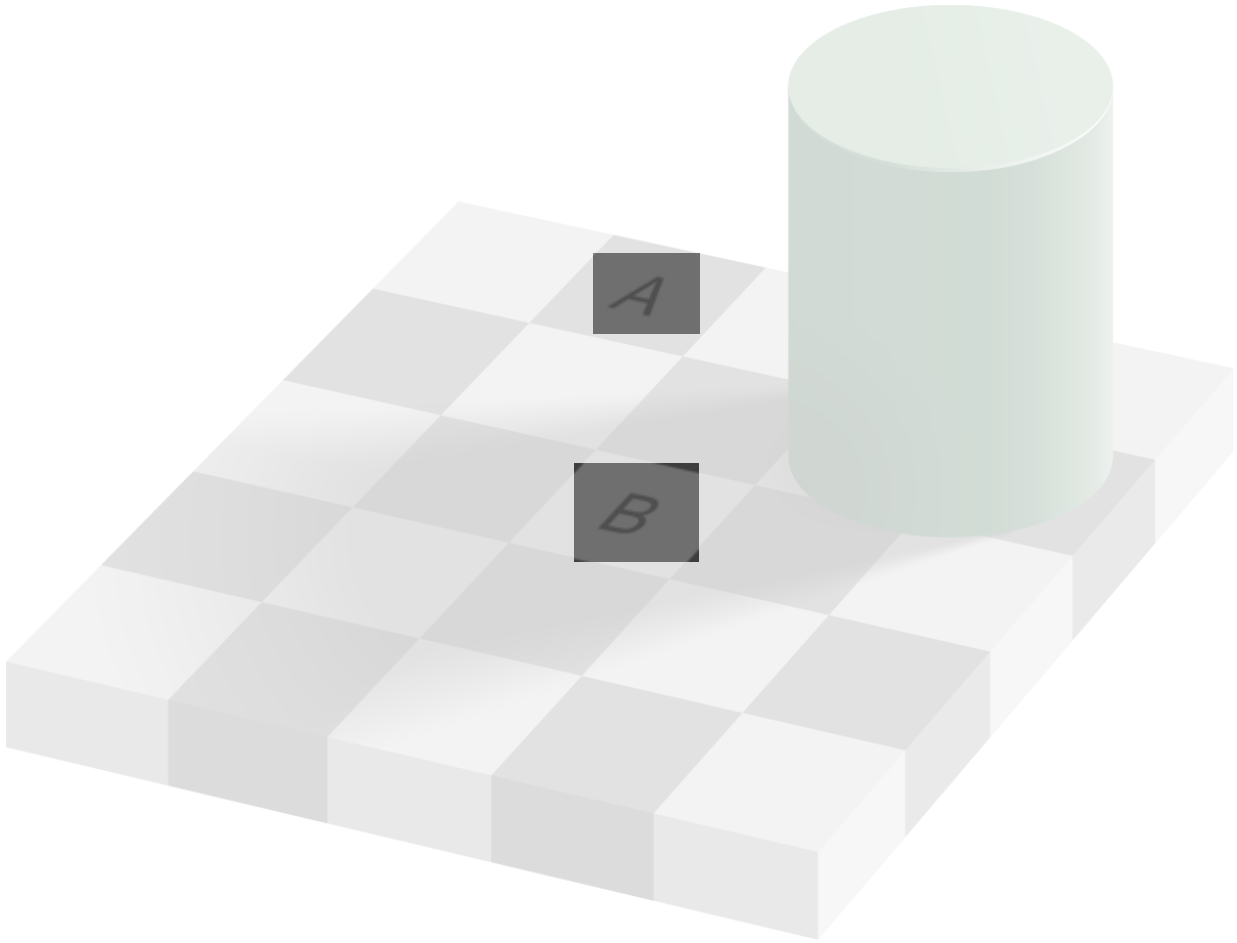
We cannot tell the materials using their colors



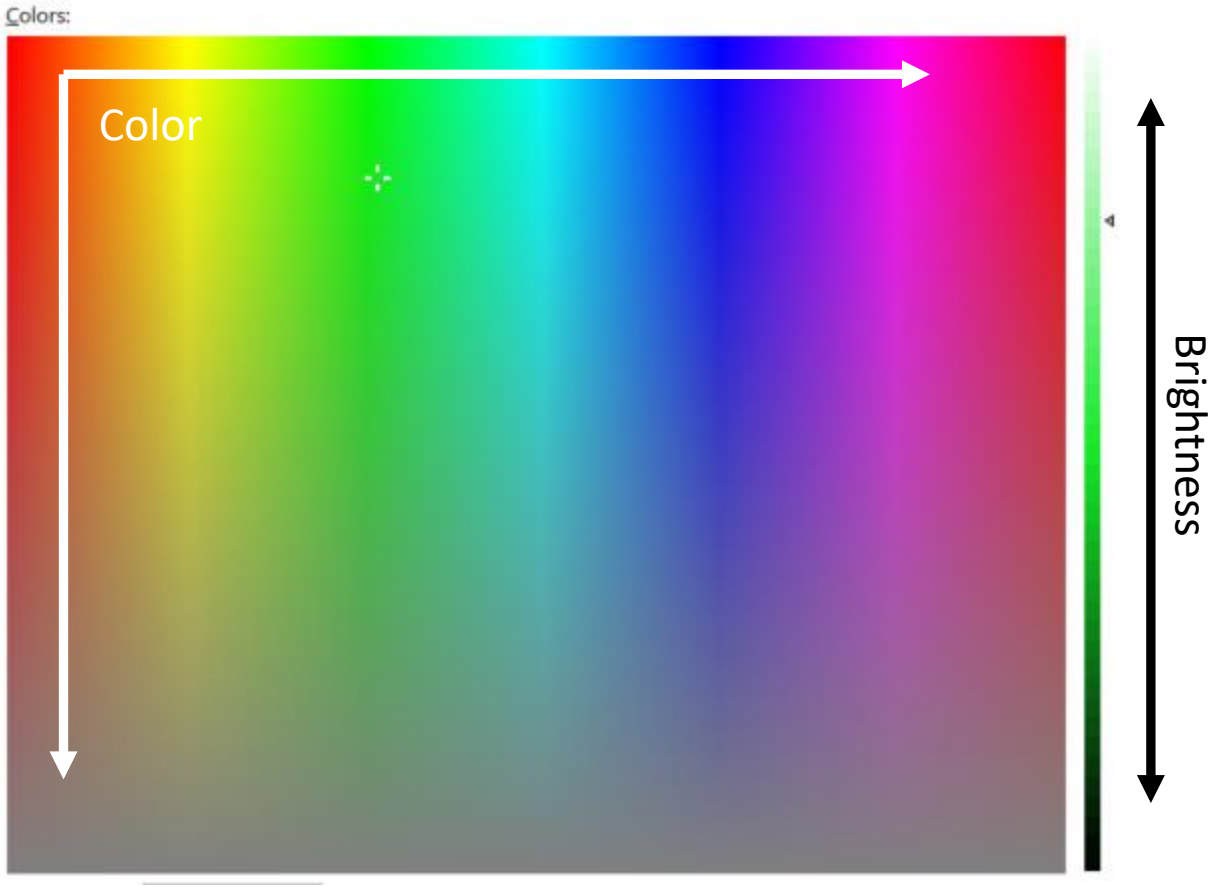
A and B, which one is “whiter”?



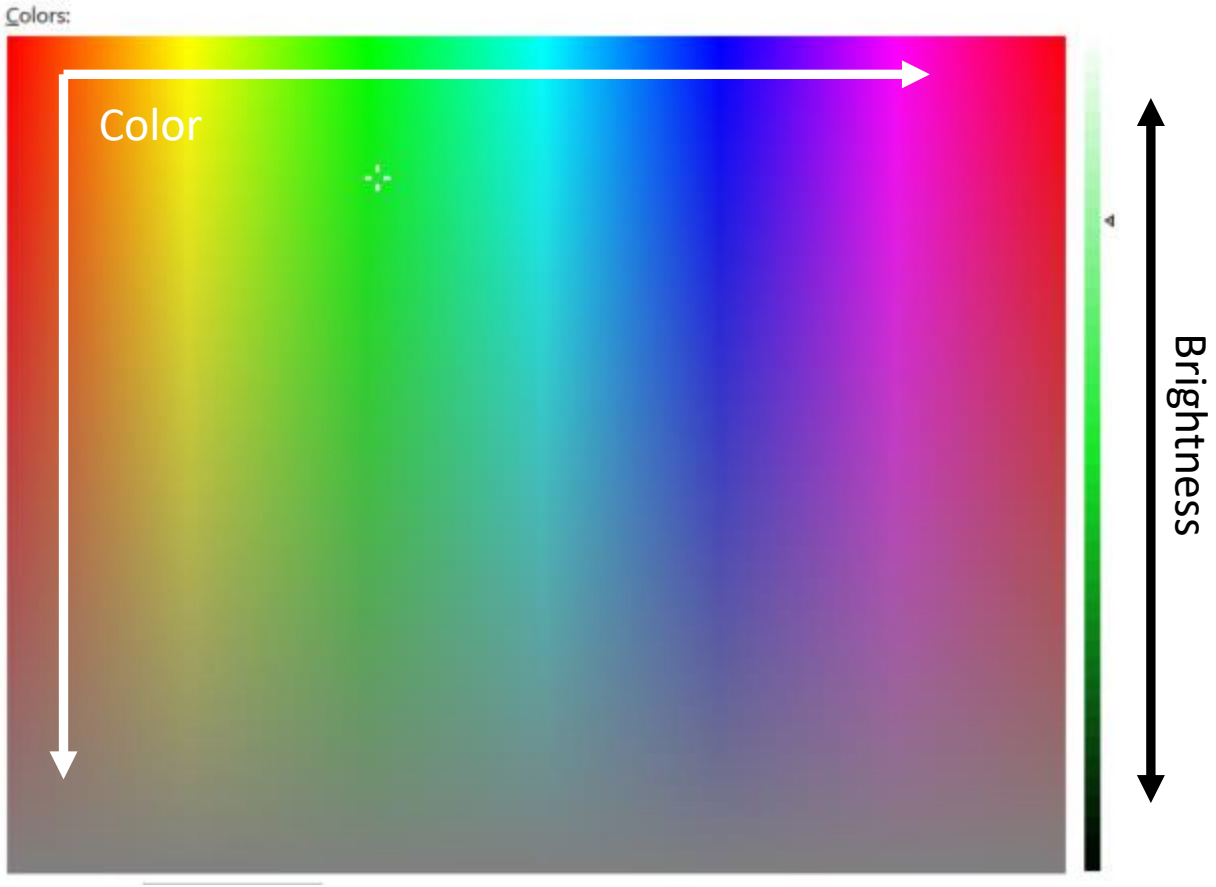
A and B, which one is “whiter”?

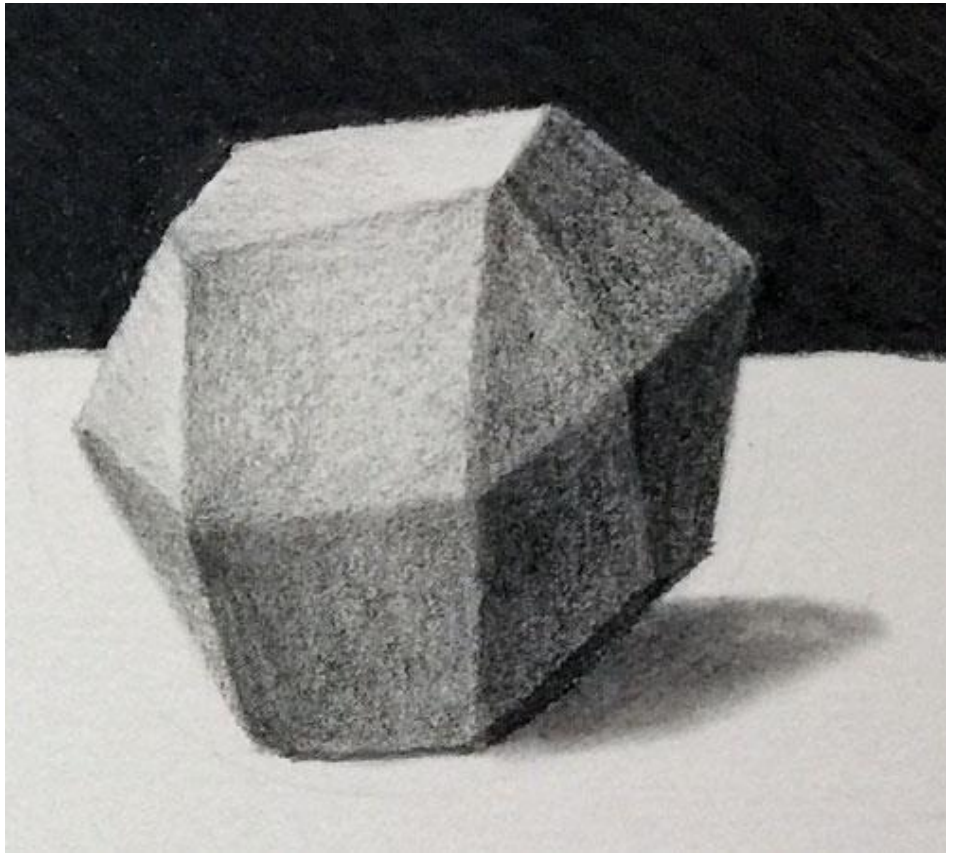


what you see = color * brightness

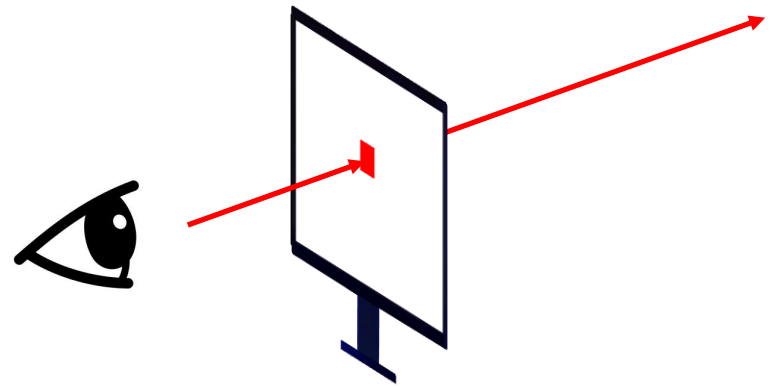


“Let there be ~~light~~ shading”

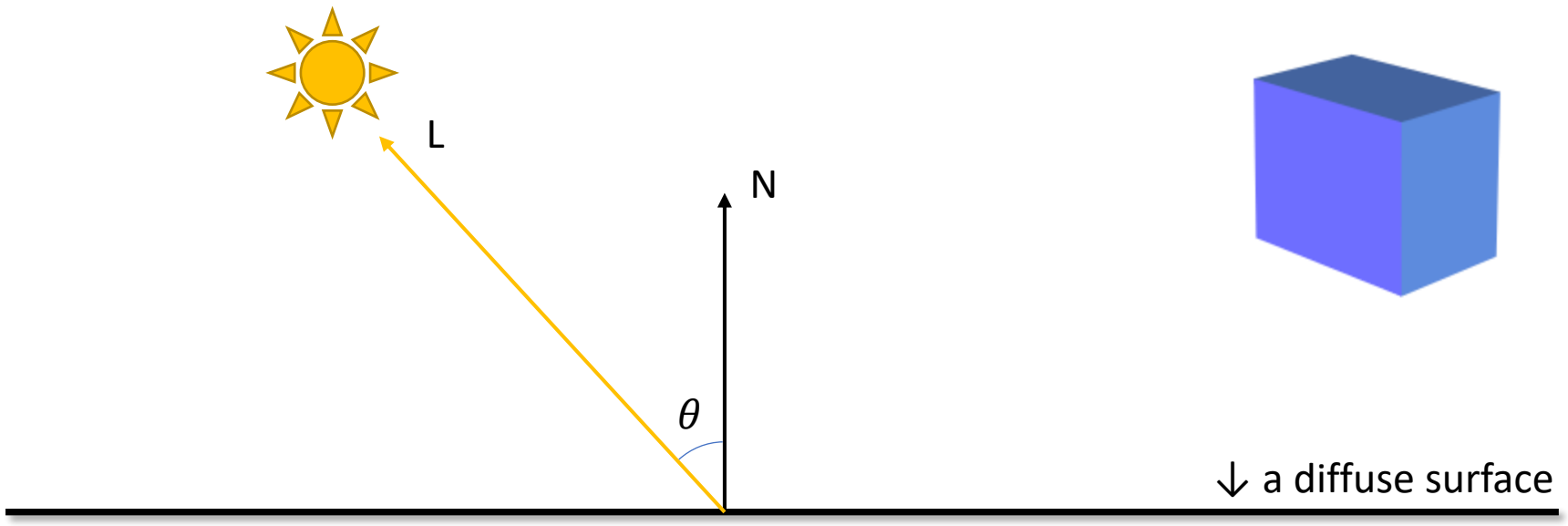




Option II: color + shading

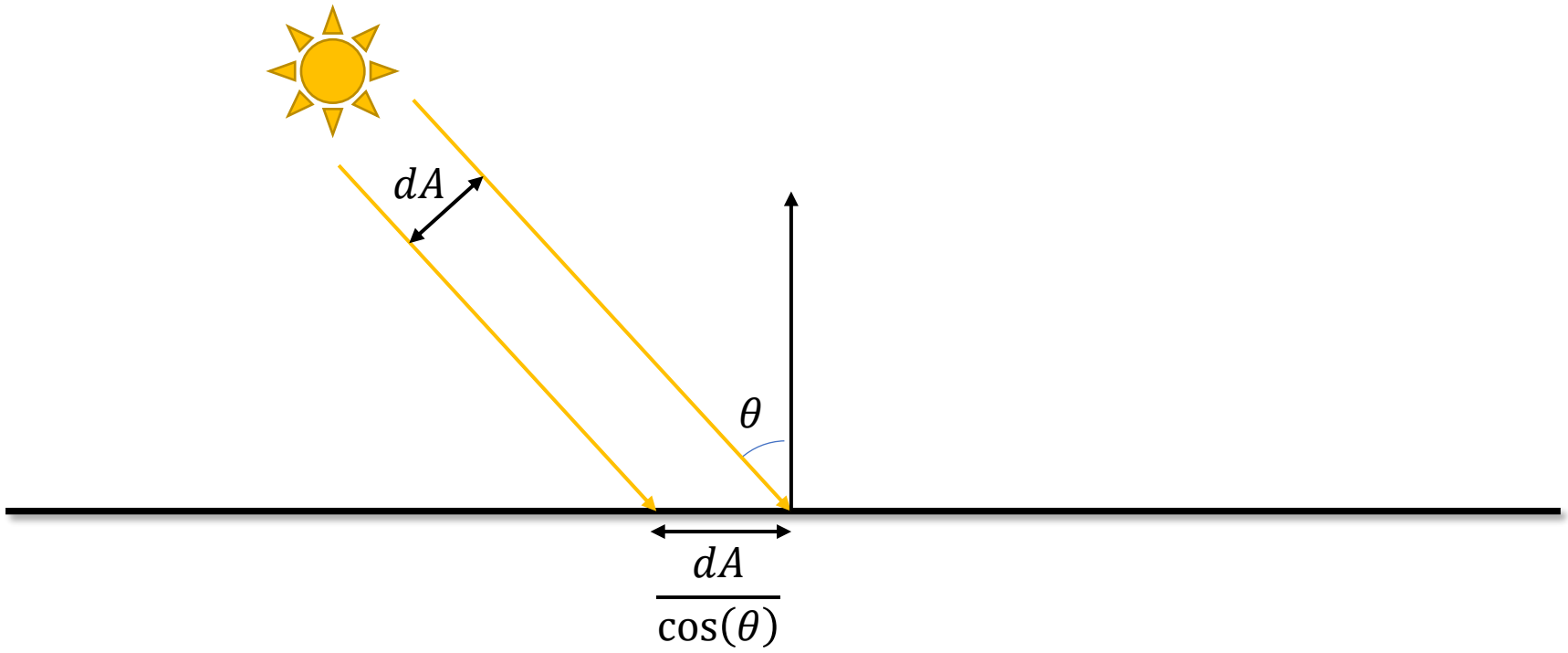


The Lambertian reflectance model

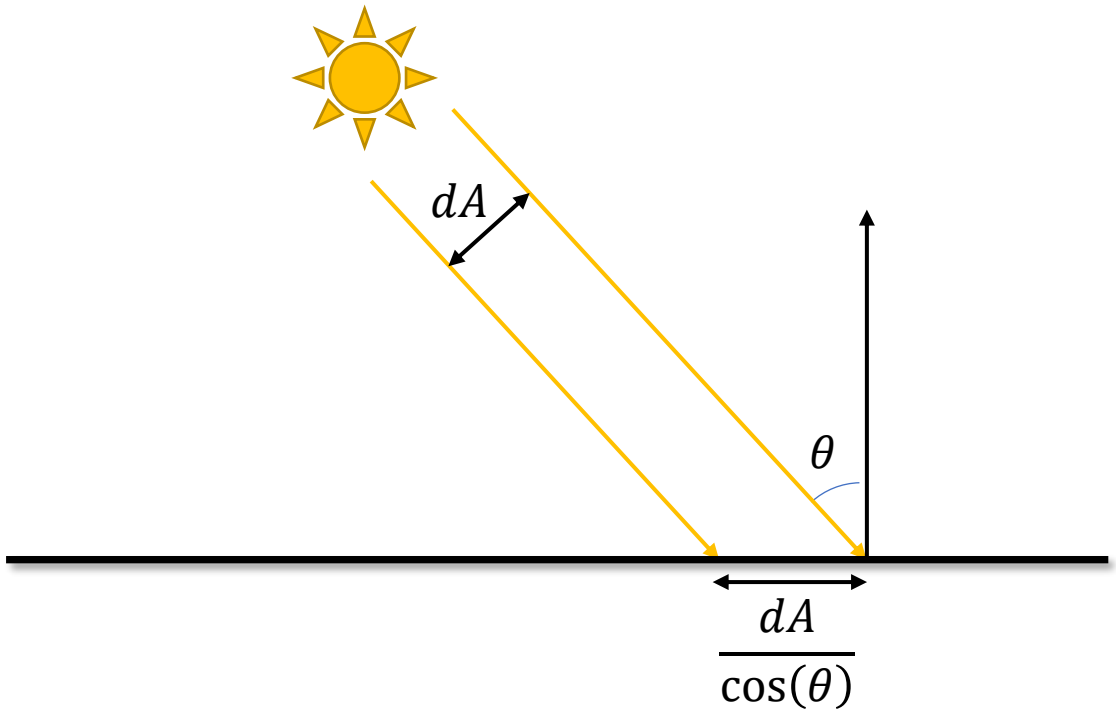


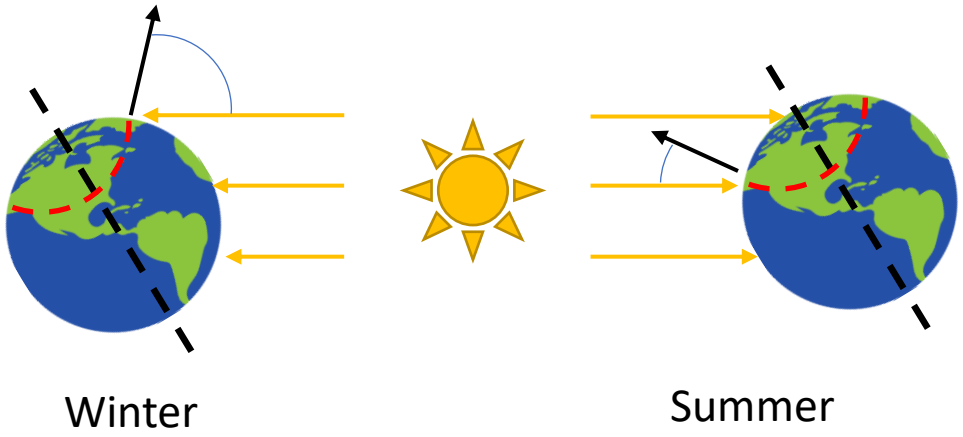
Brightness = $\cos(\theta)$

Why the cos(θ)?

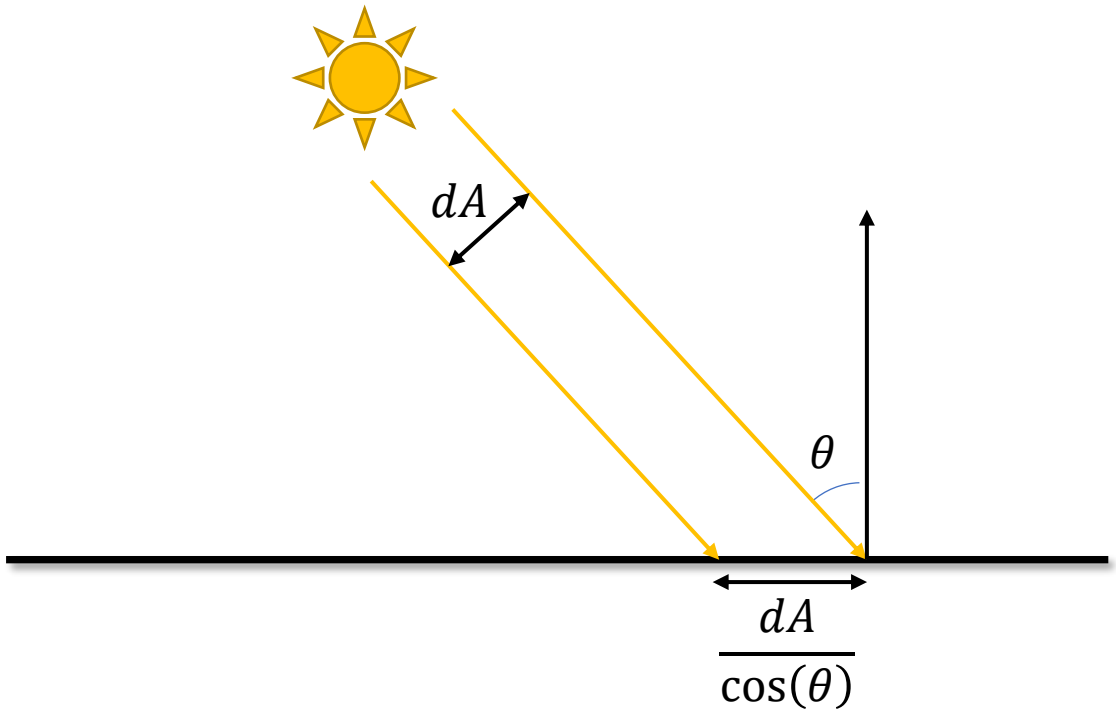


The larger cos(θ), the higher energy/area



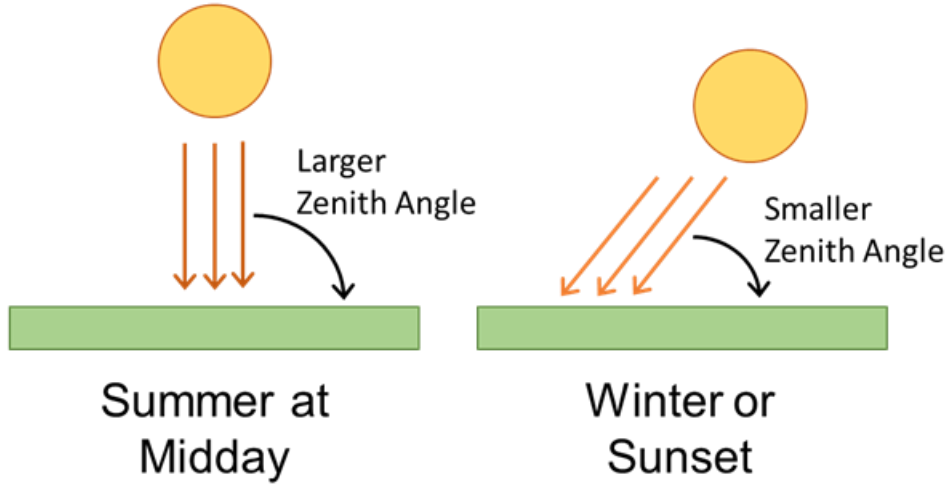


The larger cos(θ), the higher energy/area

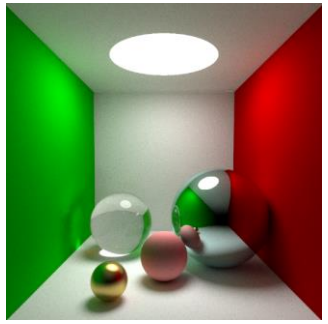


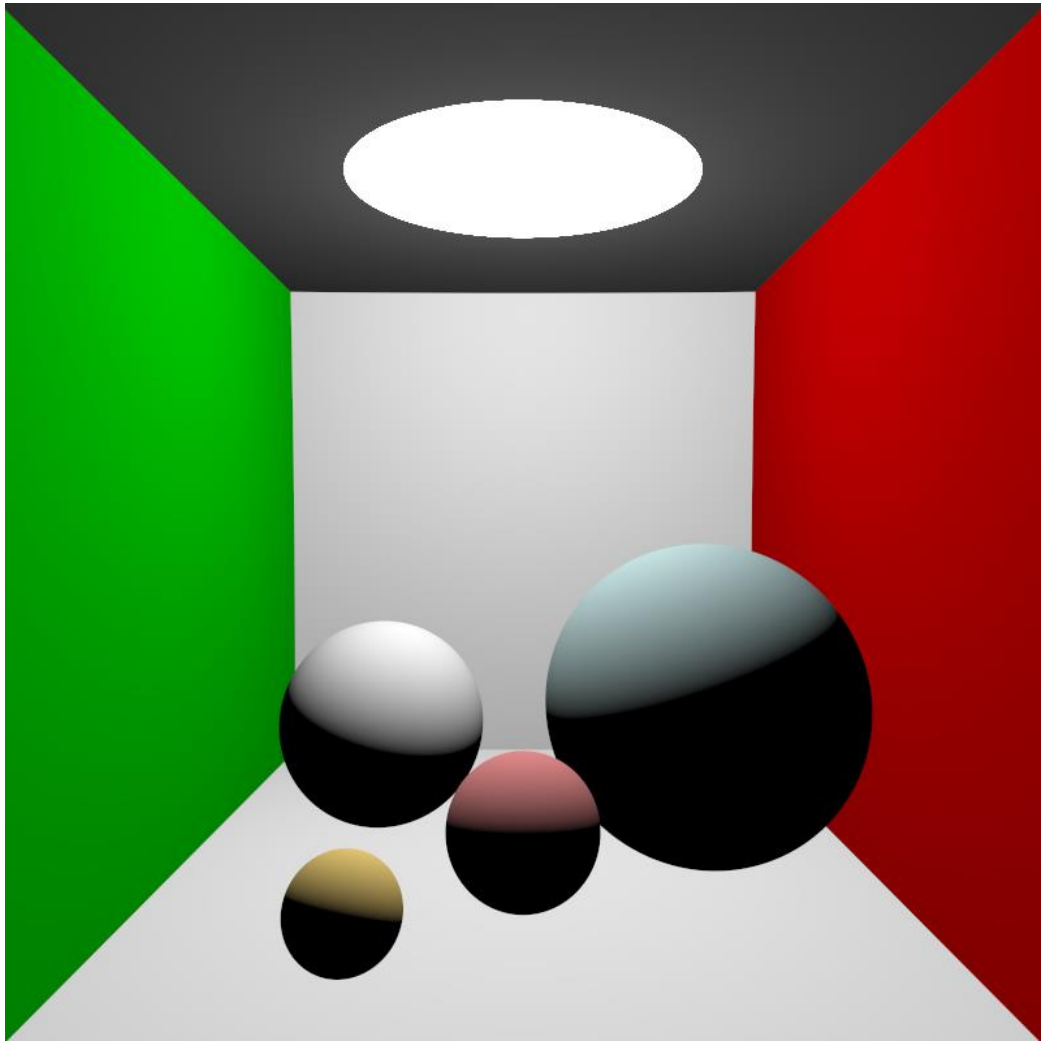
Smaller θ
Larger $\cos(\theta)$
Higher energy/area

Larger θ
Smaller $\cos(\theta)$
Lower energy/area

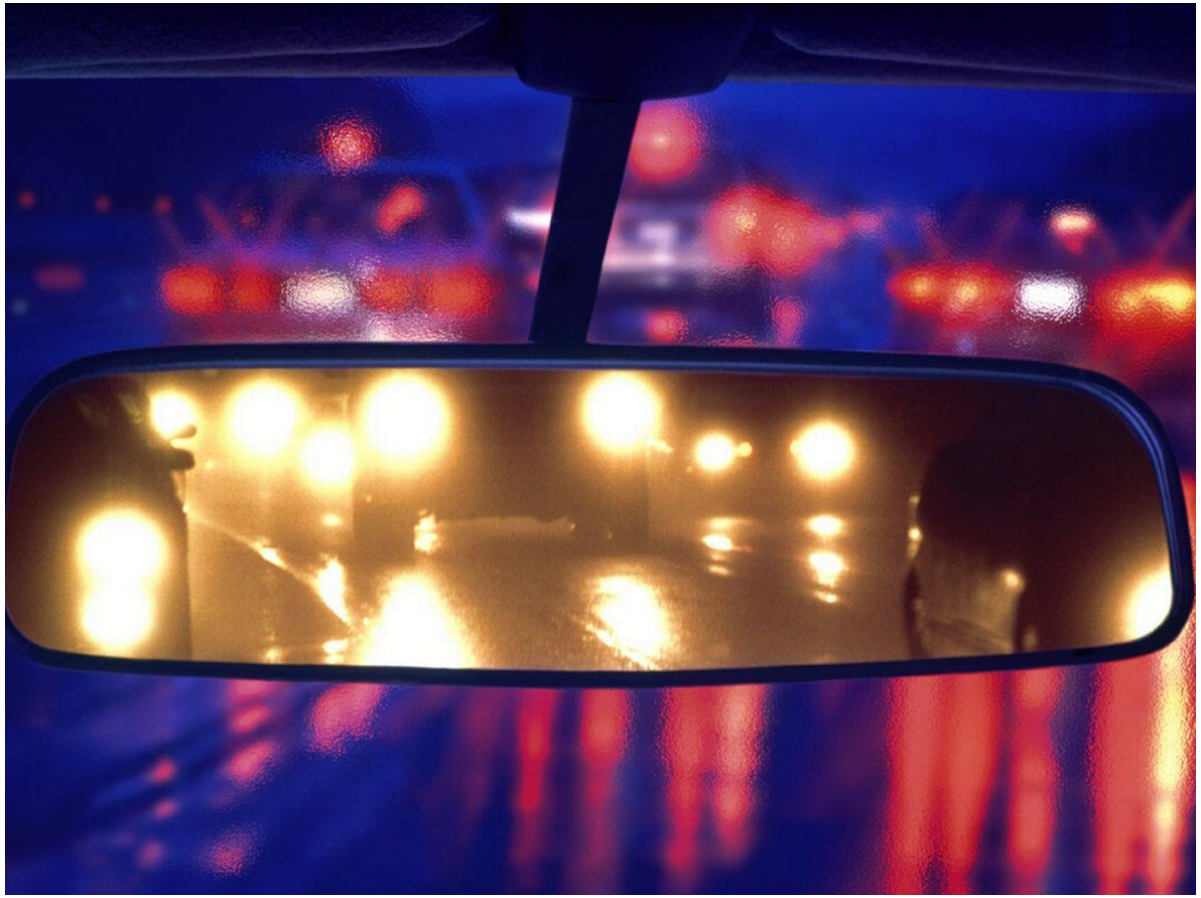


Looks like 3D!



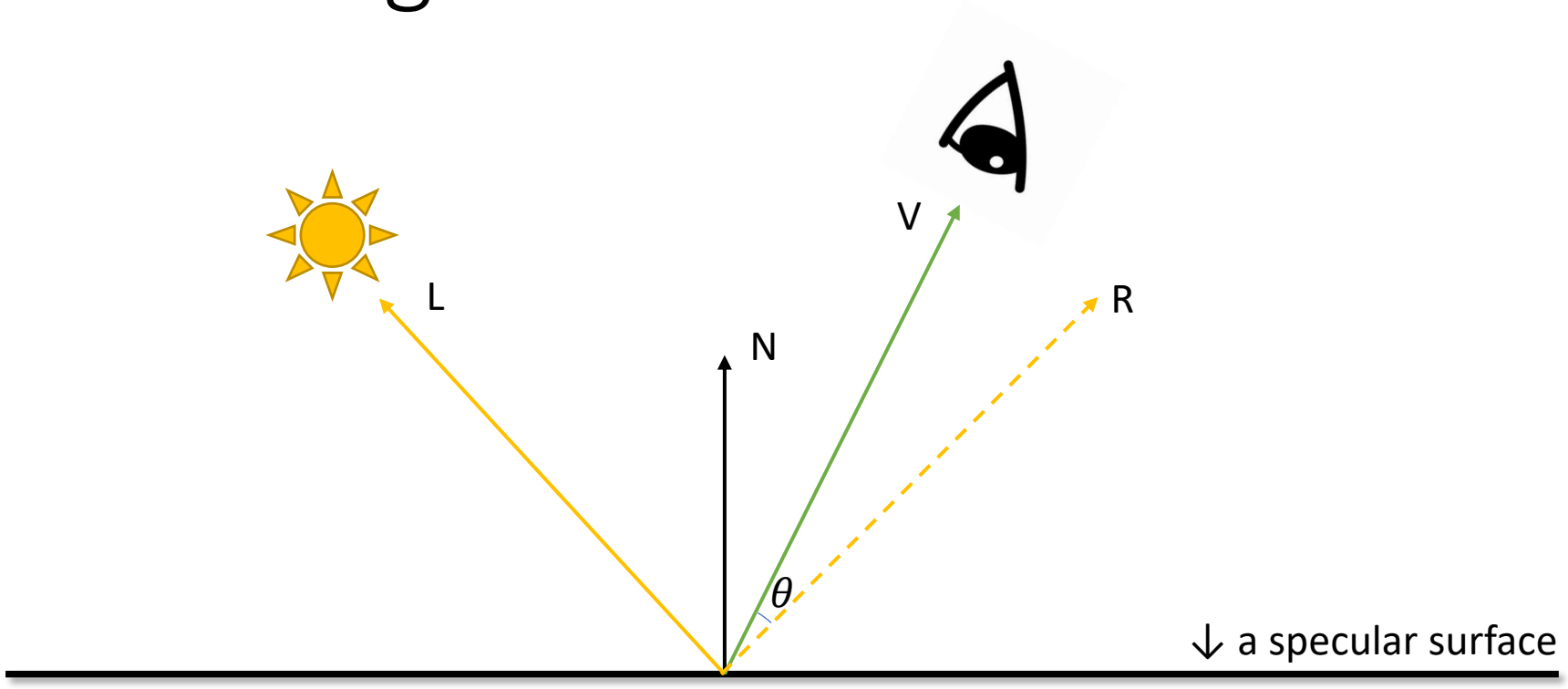


How about the specular surfaces?



[Image courtesy of mirror.co.uk]

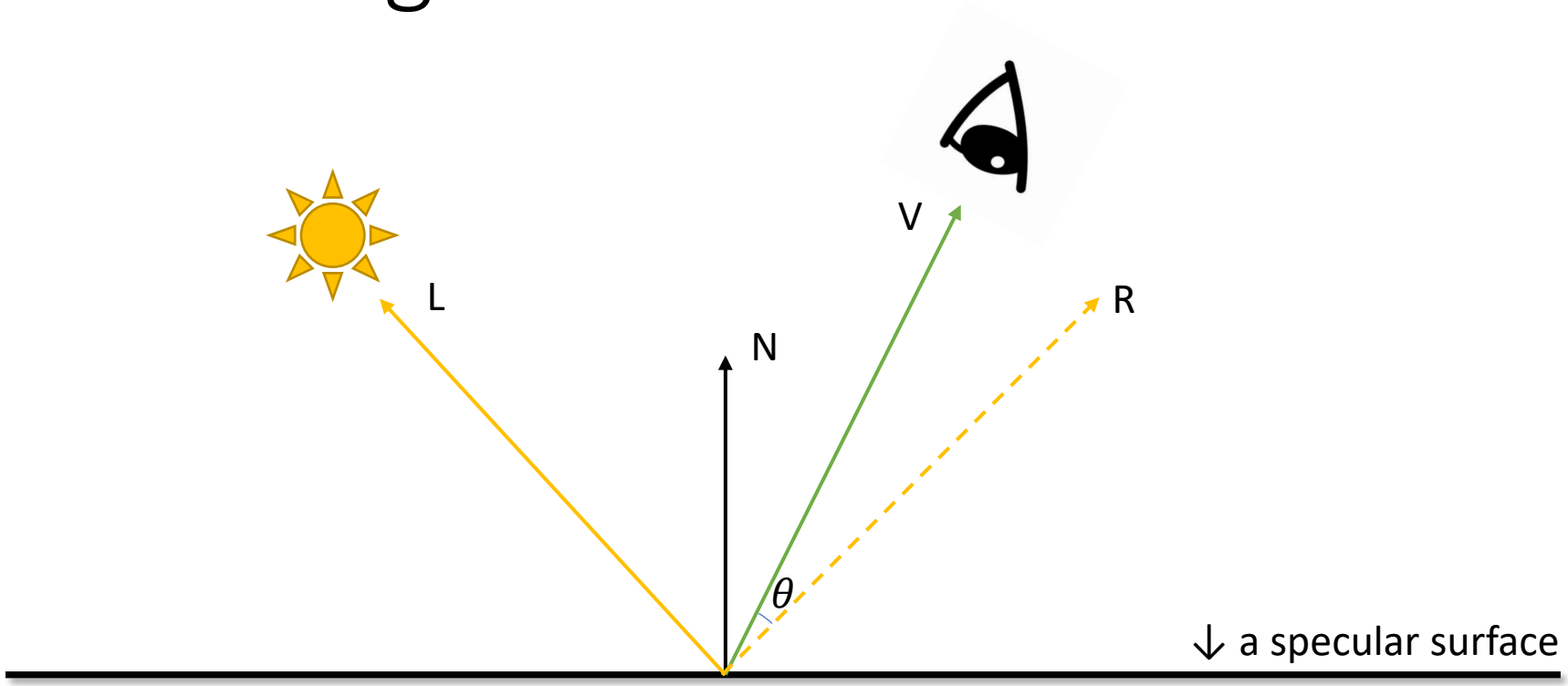
The Phong reflectance model

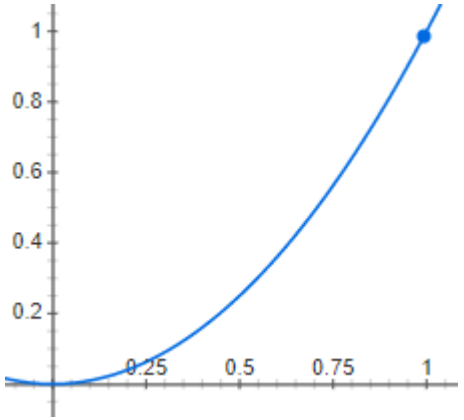


Brightness = $(V \cdot R)^{\alpha} = (\cos(\theta))^{\alpha}$,

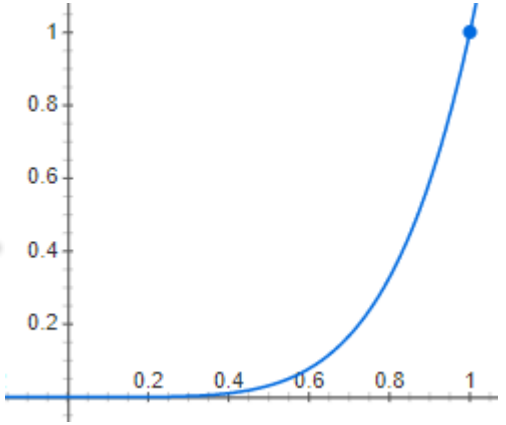
α is the hardness

The Phong reflectance model





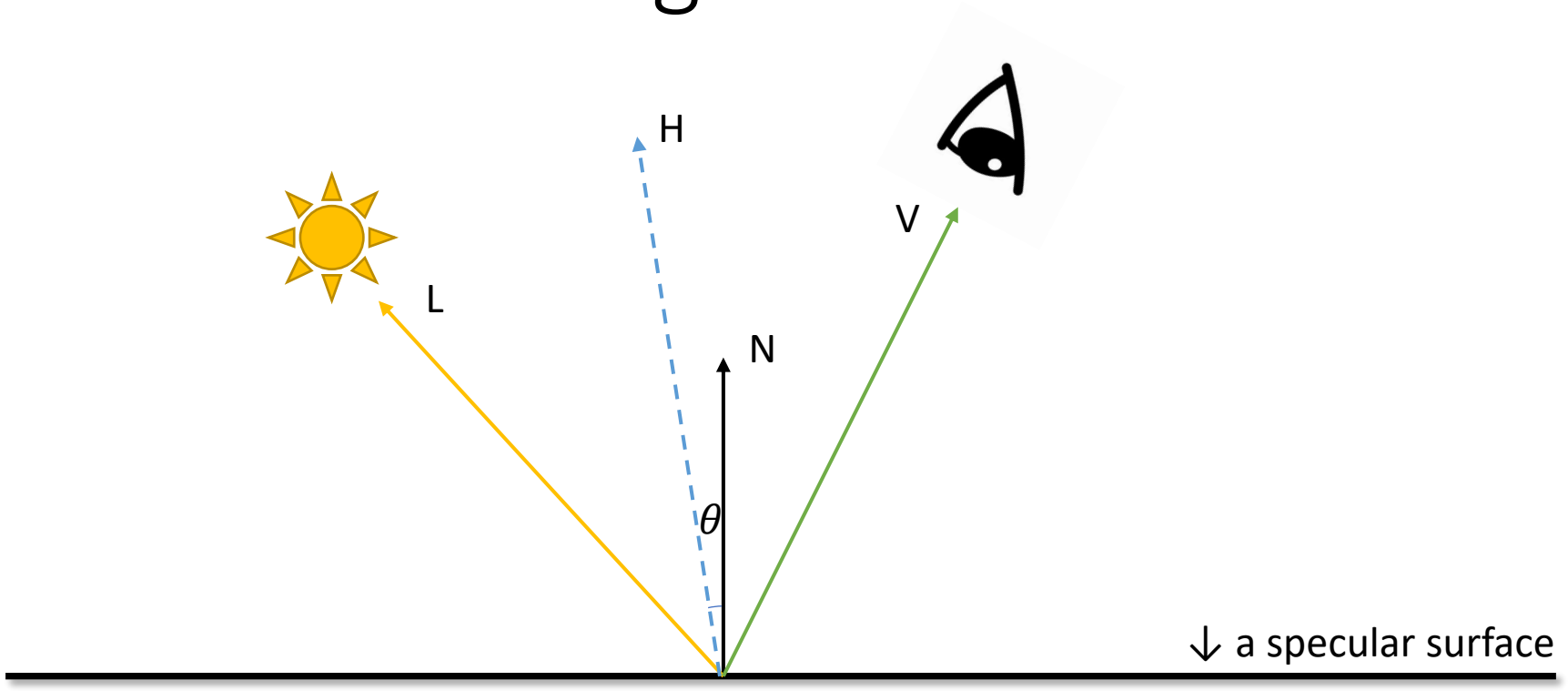
$\alpha = 2$



$\alpha = 5$

Brightness = $(V \cdot R)^{\alpha} = (\cos(\theta))^{\alpha}$,
α is the hardness

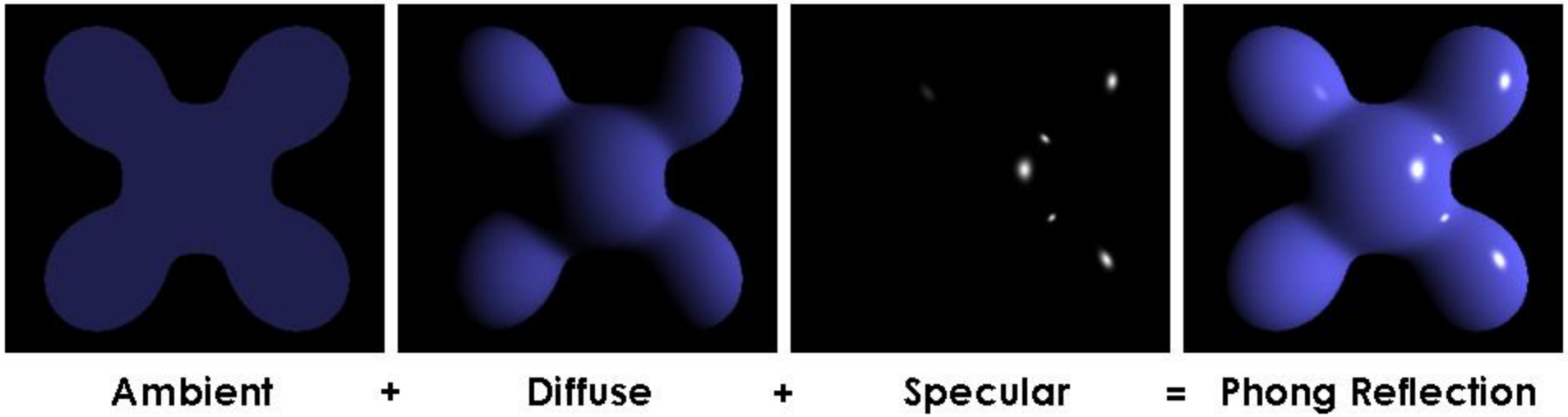
The Blinn–Phong reflectance model



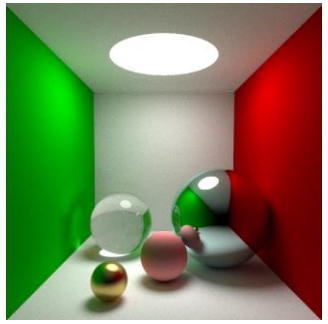
Brightness = $(N \cdot H)^{\alpha} = (\cos(\theta))^{\alpha'}, H = \frac{V+L}{\|V+L\|}$

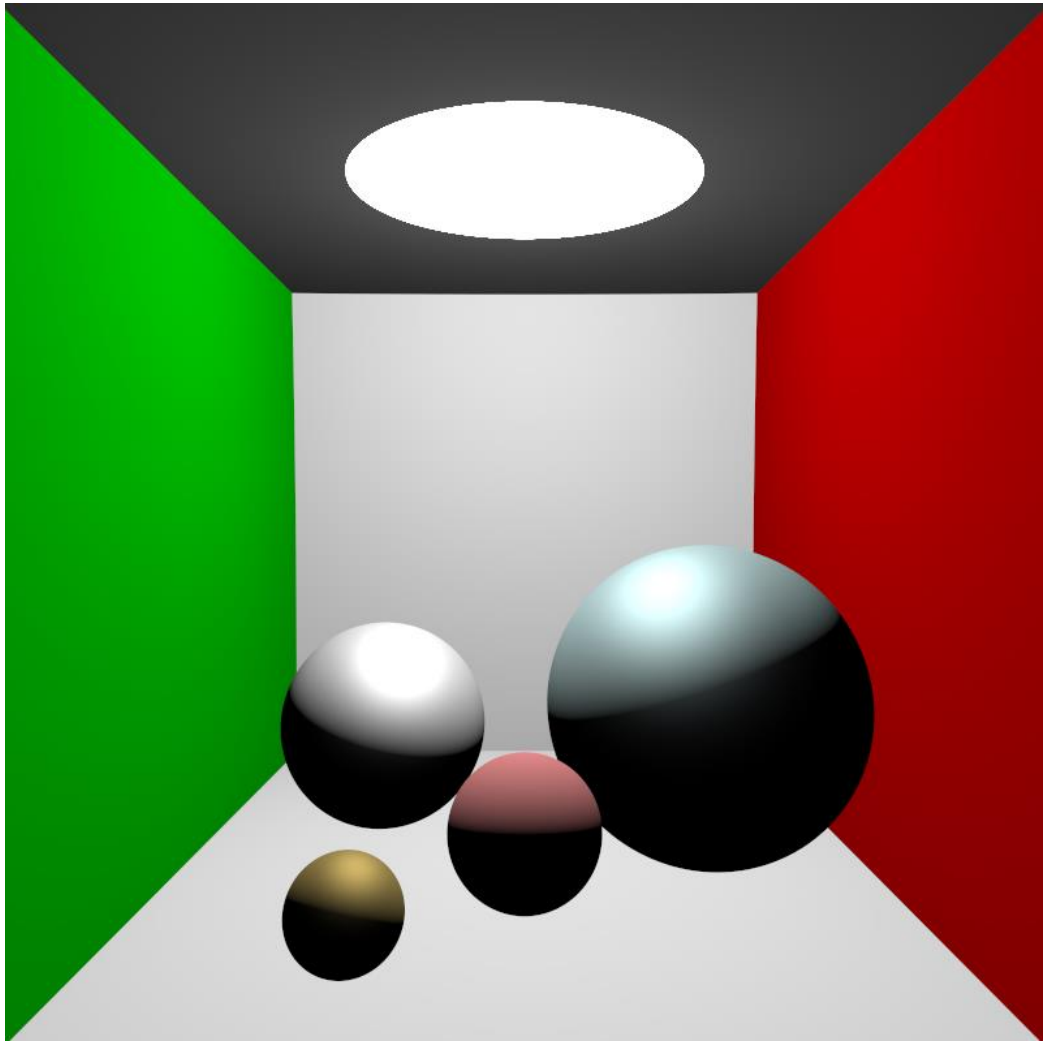
$\alpha' > \alpha$ is the hardness in Blinn-Phong model

Blend things together: the ***Blinn-Phong shading*** model

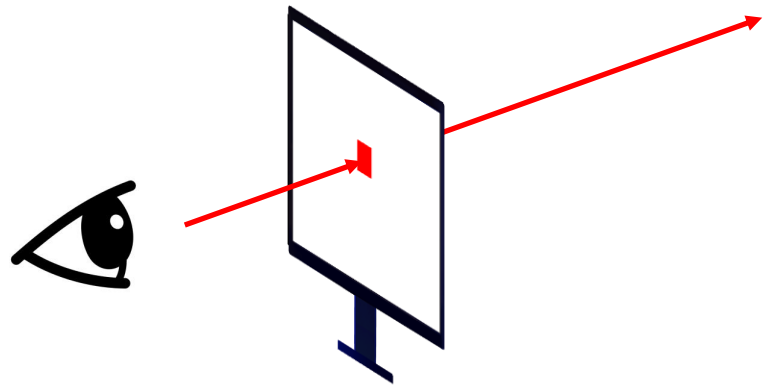


Shining! But floating!





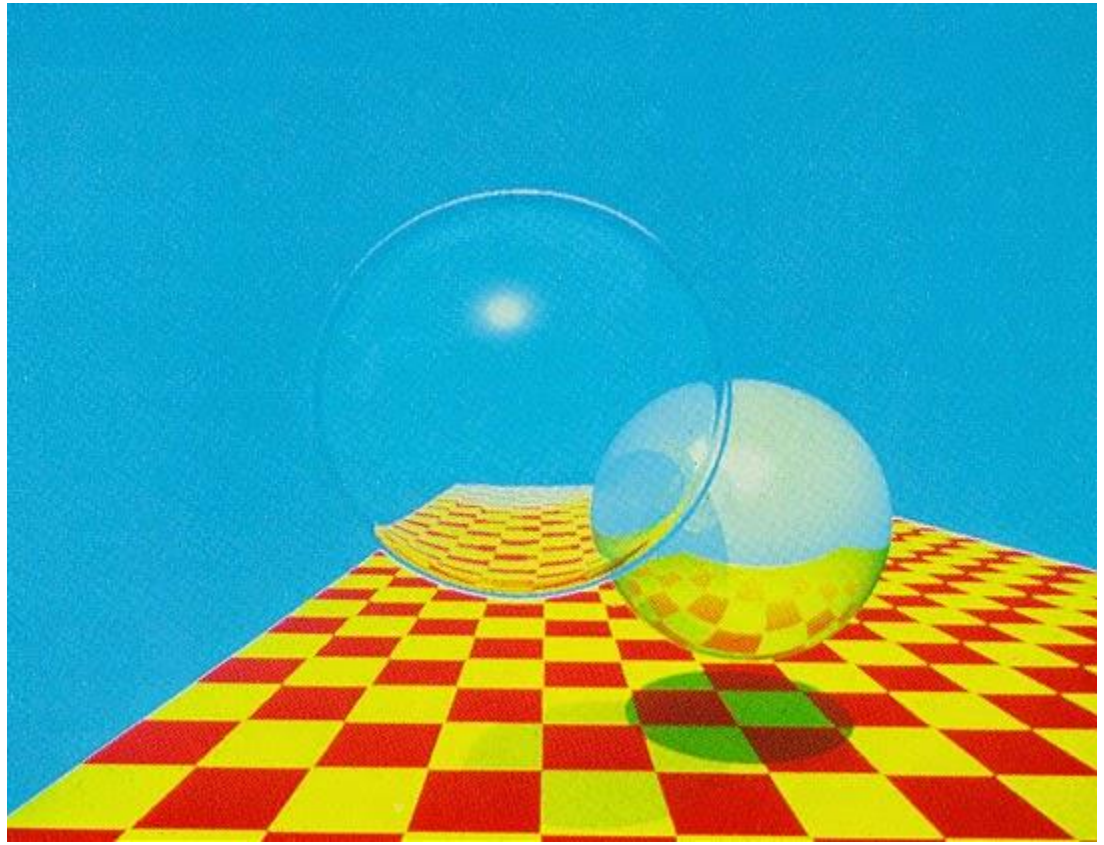
Option III: the Whitted-style ray tracer



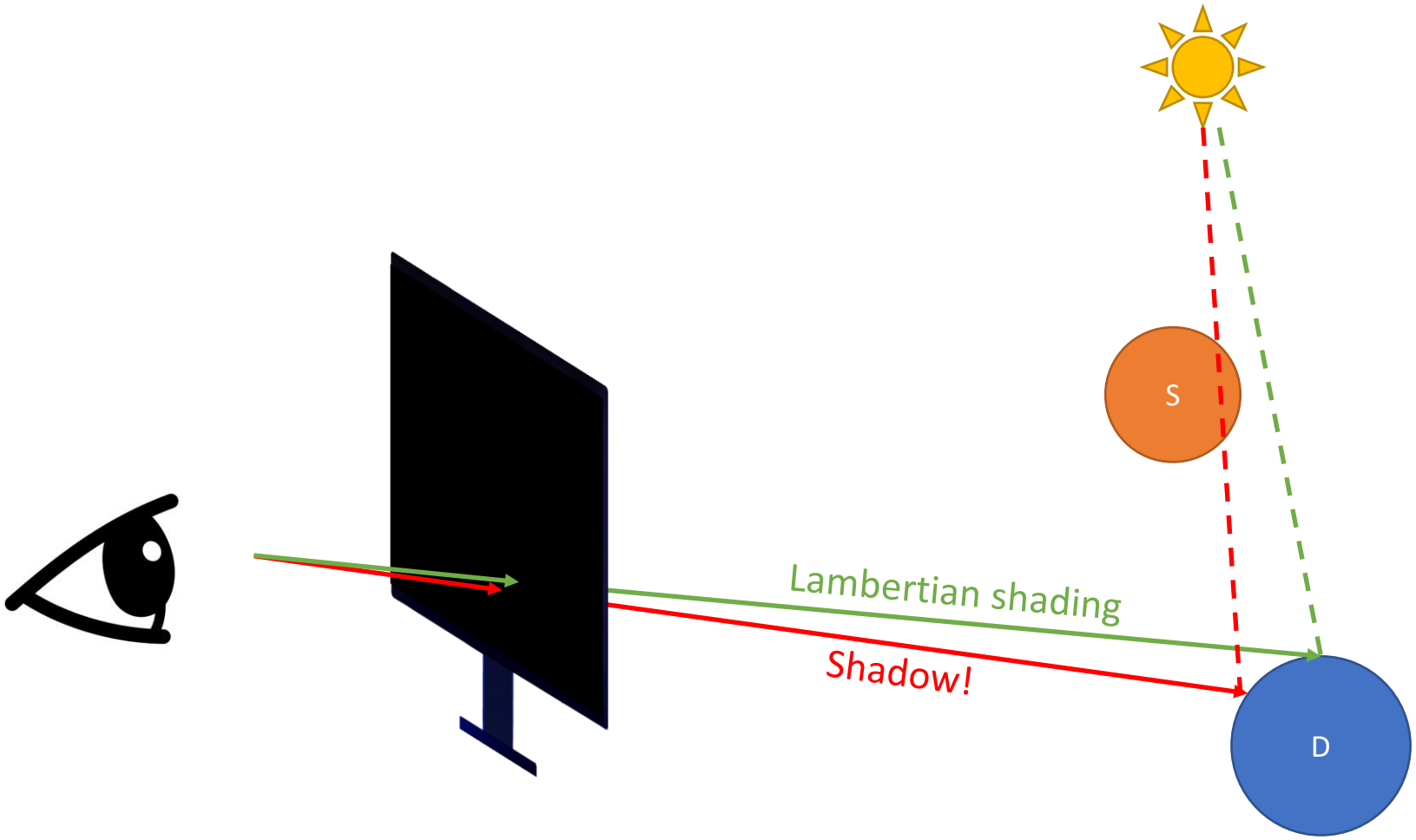
The Whitted-style ray tracer

- An improved illumination model for shaded display [Whitted, 1979][Link]

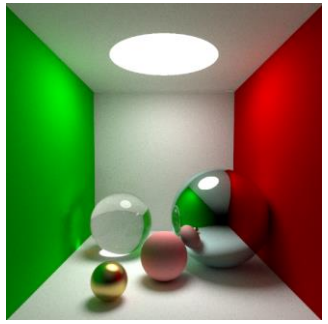
- Shadow? Mirror? Dielectric?
  - We've got what you want!

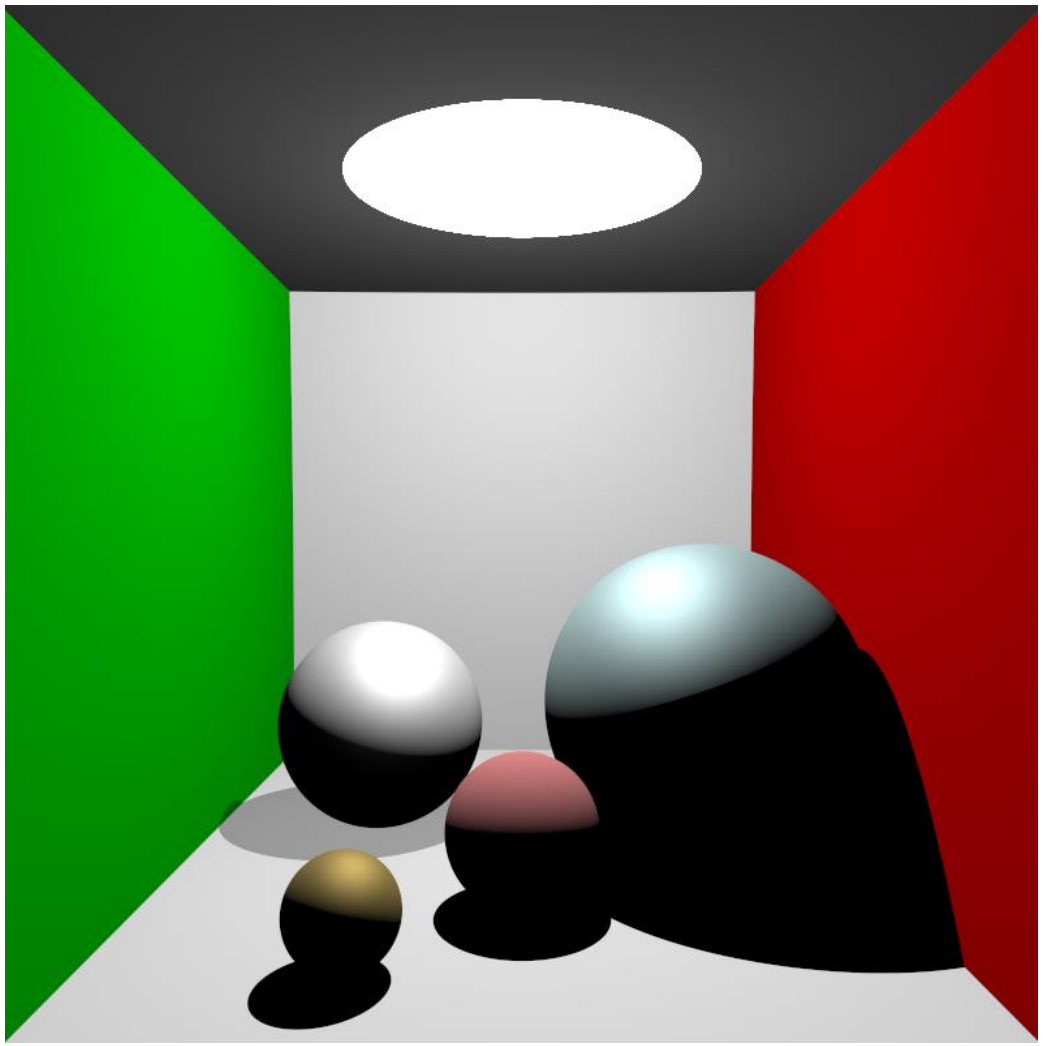


What color does the ray see?

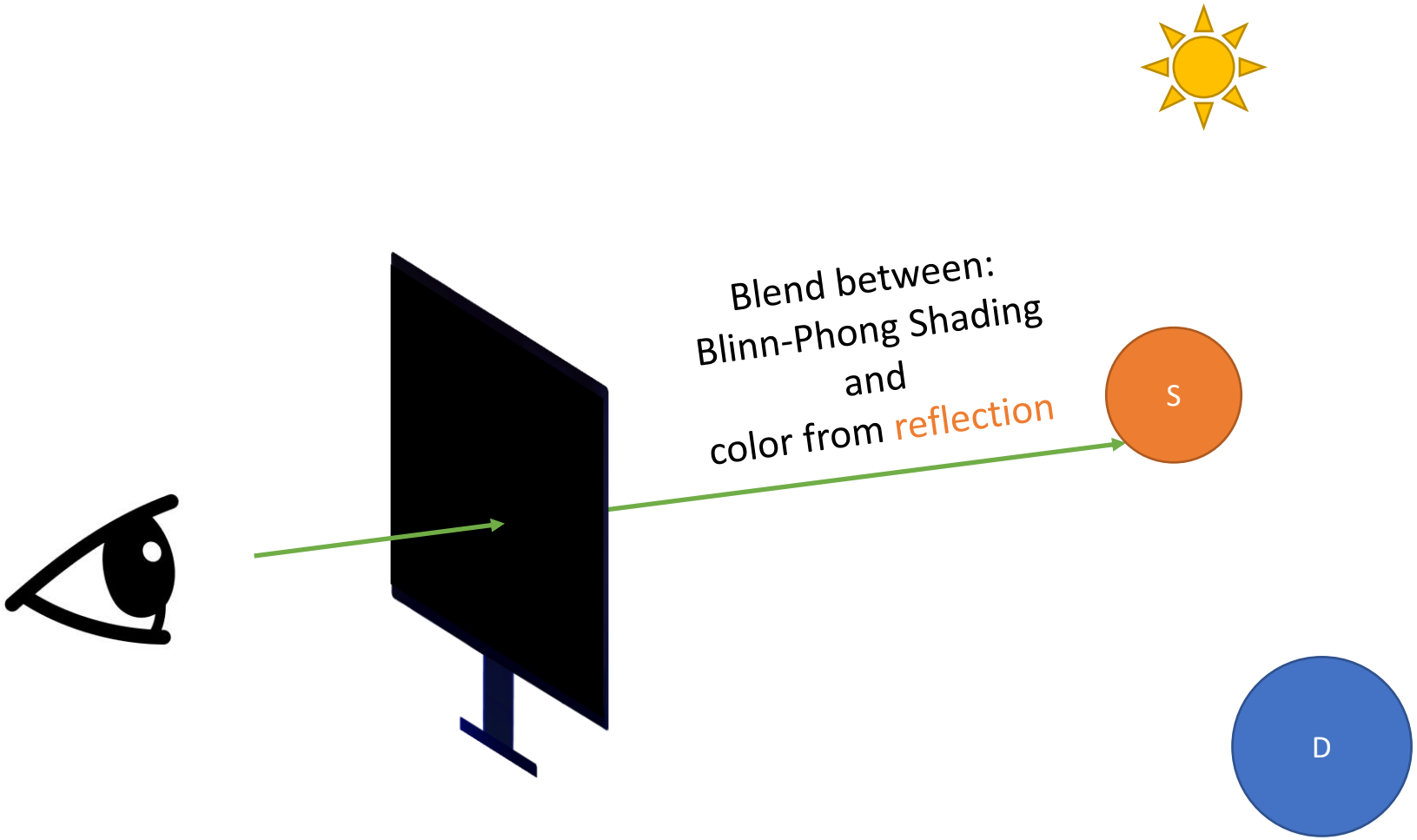


Shadows!

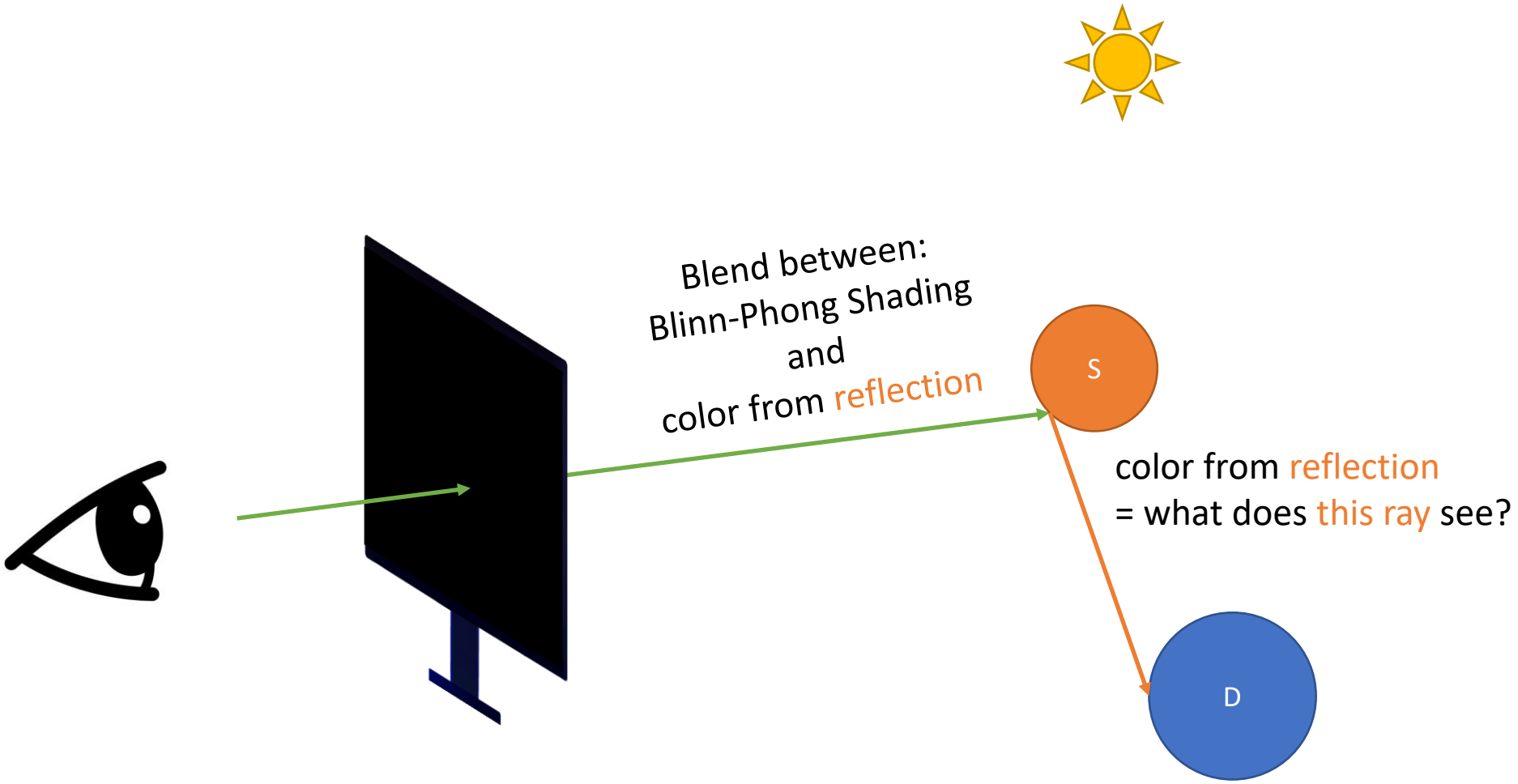




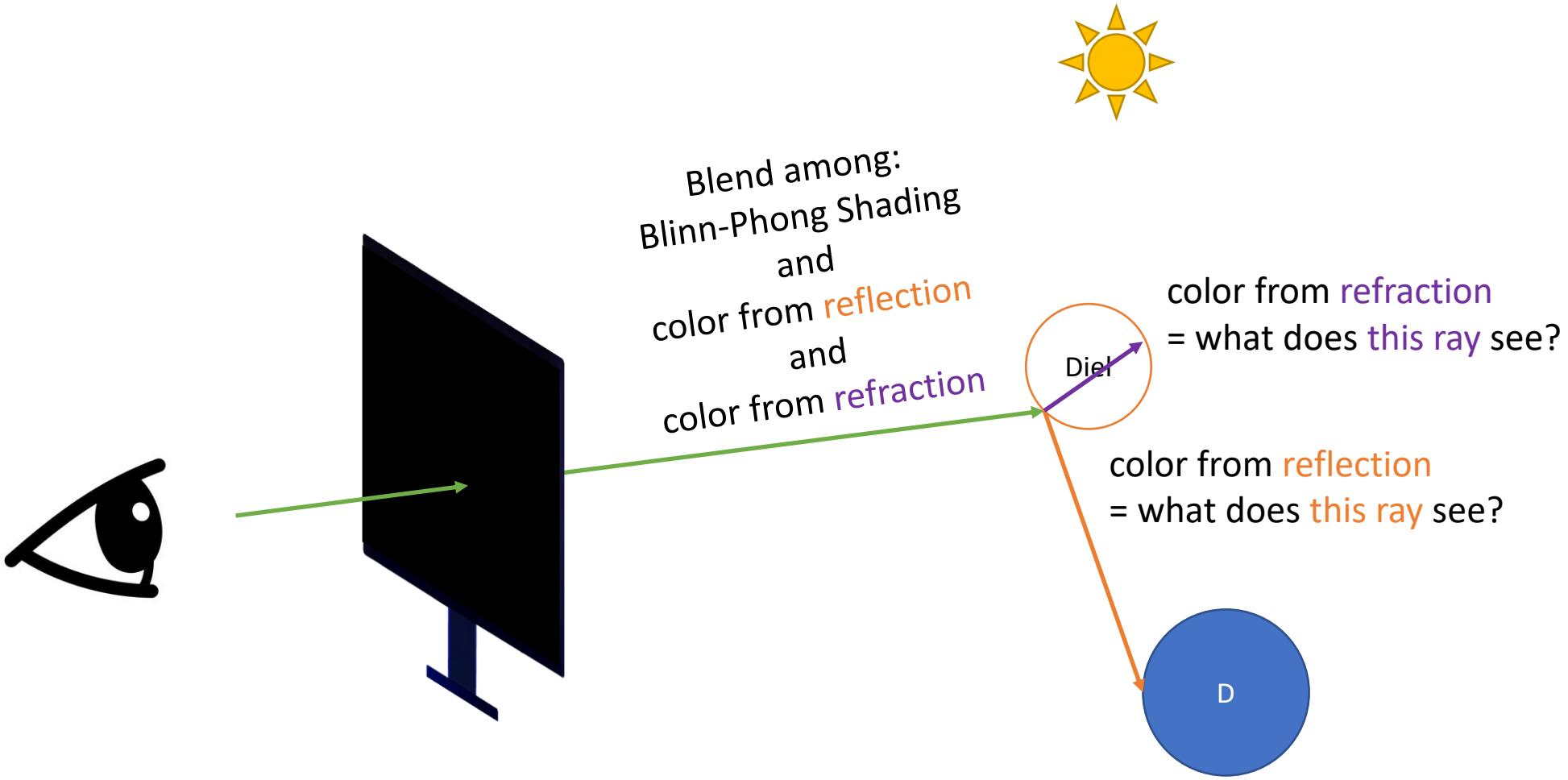
What color does the ray see?



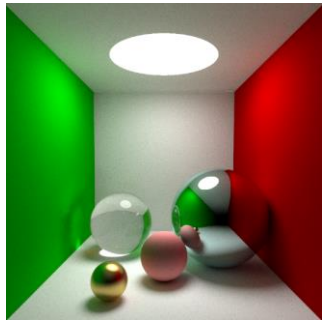
What color does the ray see?

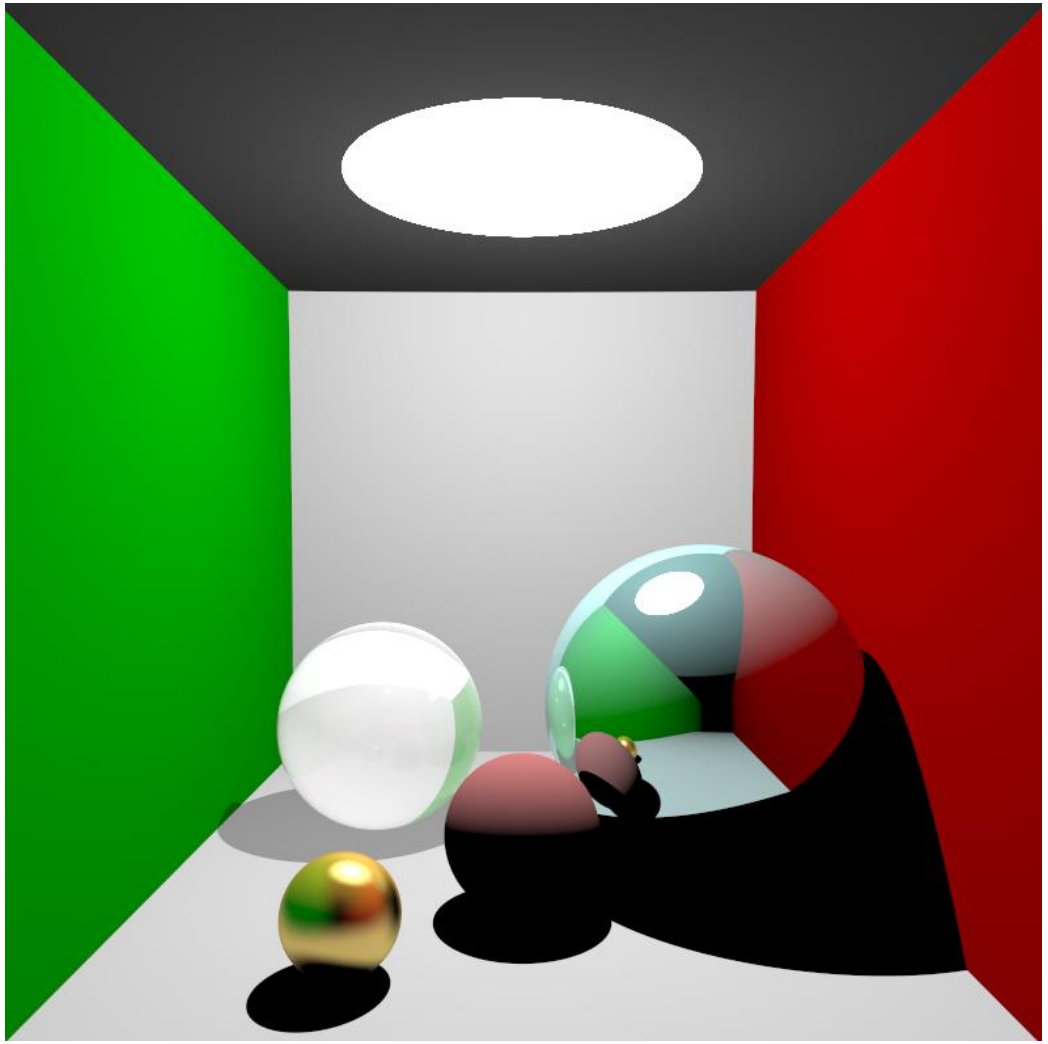


What color does the ray see?

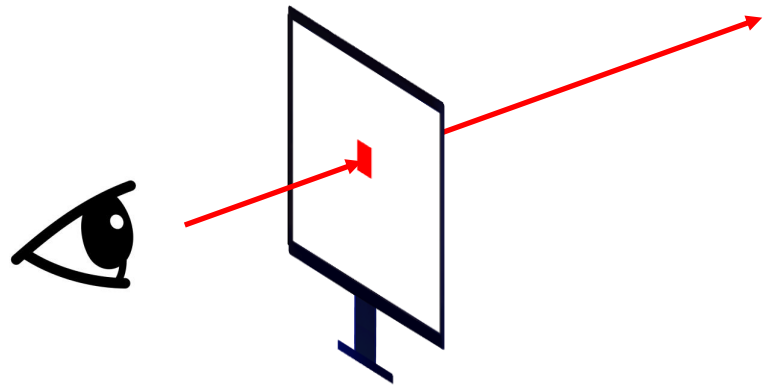


Shadow! Mirror! Dielectric!





Option IV: the path tracer



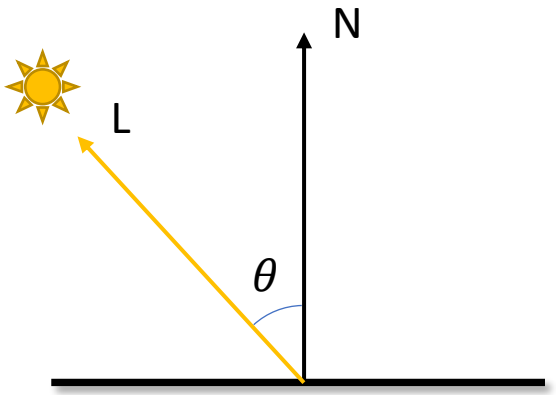
Take-aways from the previous approaches

- Shading models:
  - The brightness matters

- The Whitted-style ray tracing:
  - Getting color recursively: what color does this ray see?

Revisit the diffuse surfaces

- Lambertian shading model:
  - Brightness = $(N \cdot L)$
- The Whitted-style ray tracing:
  - Brightness = $(N \cdot L)$ * (is_occluded)
- Diffuse surfaces do not reflect the light rays in the previously mentioned methods.

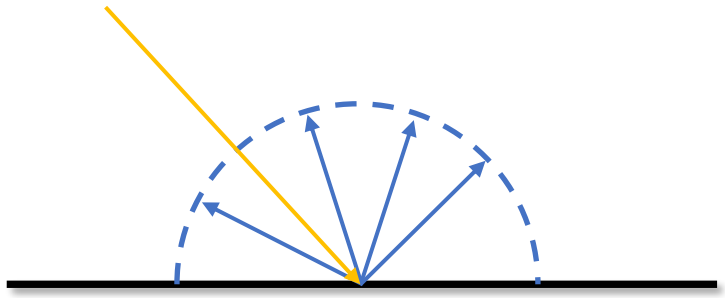


“Global illumination” (GI)

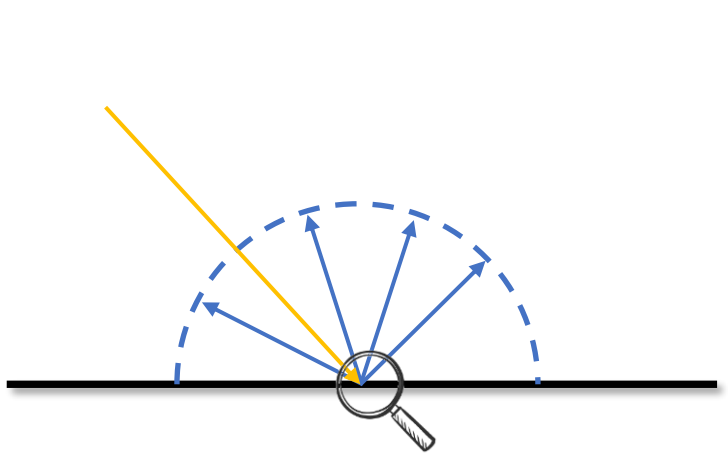
GI off

GI on

Diffuse surfaces can scatter rays as well

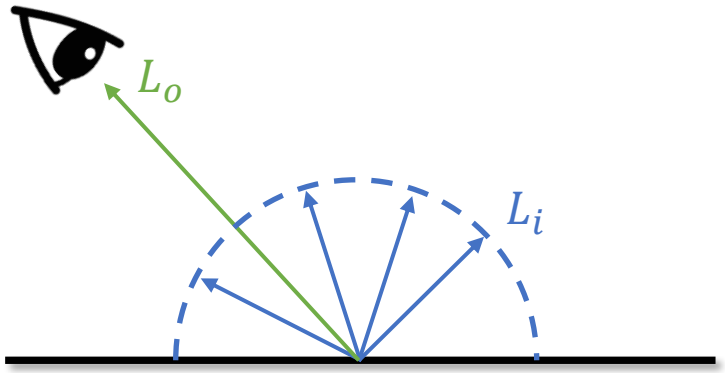


Diffuse surfaces can scatter rays as well

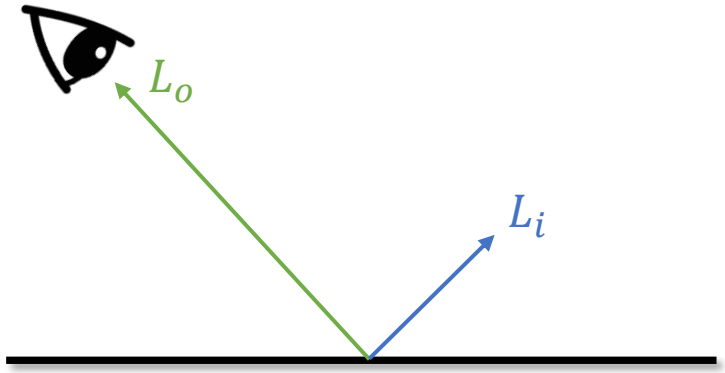


An “unified” model for different surfaces:

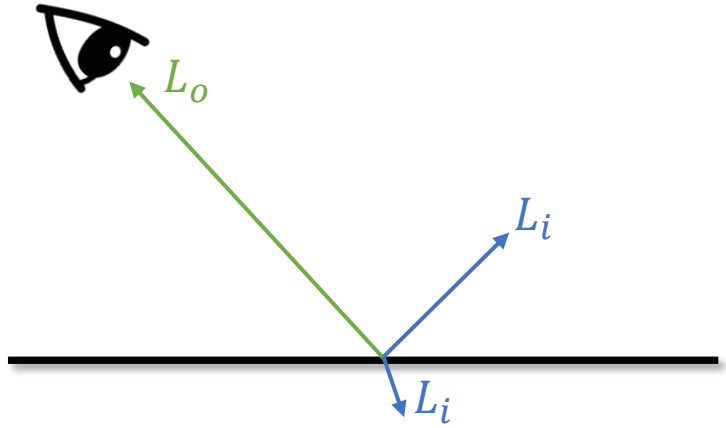
Diffuse



Specular

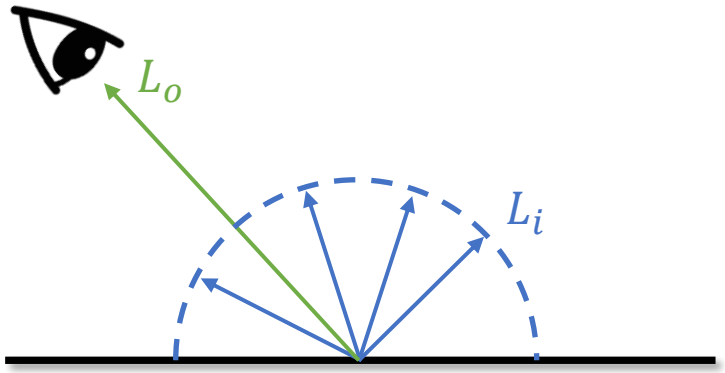


Dielectric



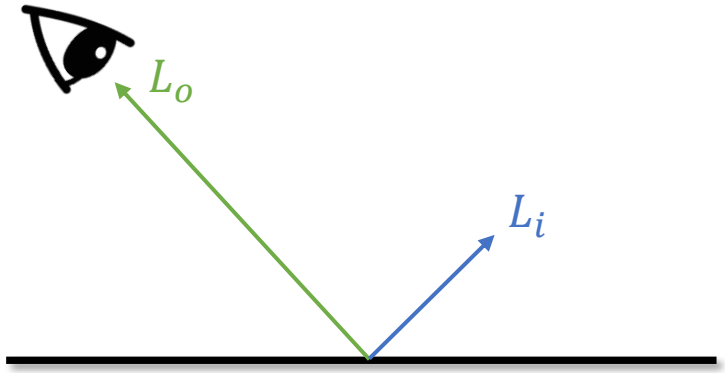
An “unified” model for different surfaces:

Diffuse



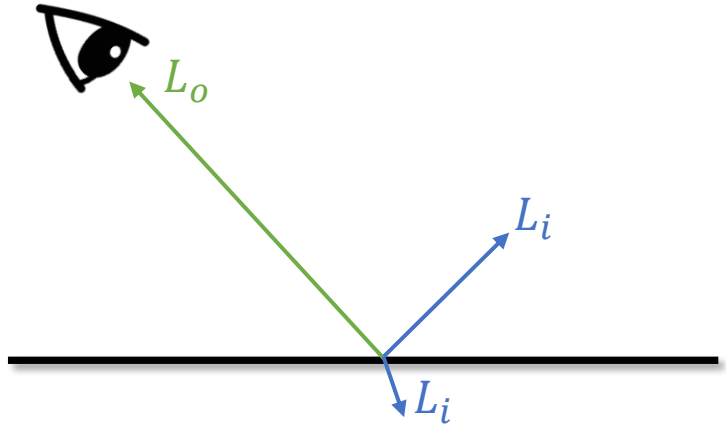
$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k} \cos(\theta_k)$$

Specular



$$L_o = L_i$$

Dielectric

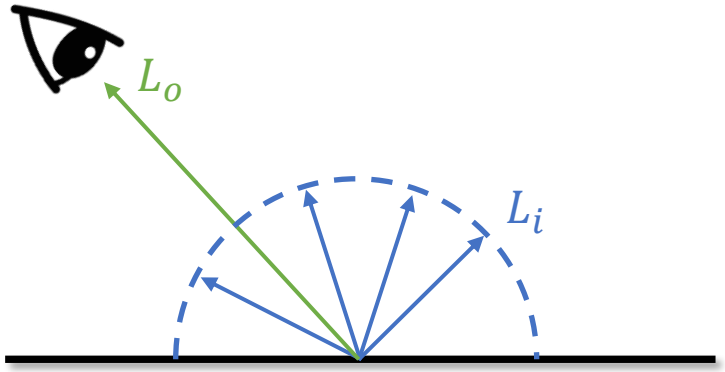


$$L_o = \frac{1}{\mathrm{N}} \sum_{k=1}^{N} L_{i,k}$$

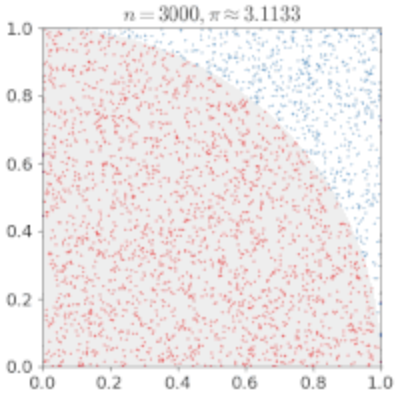
The Monte Carlo method

- …rely on **repeated random sampling** to obtain numerical results

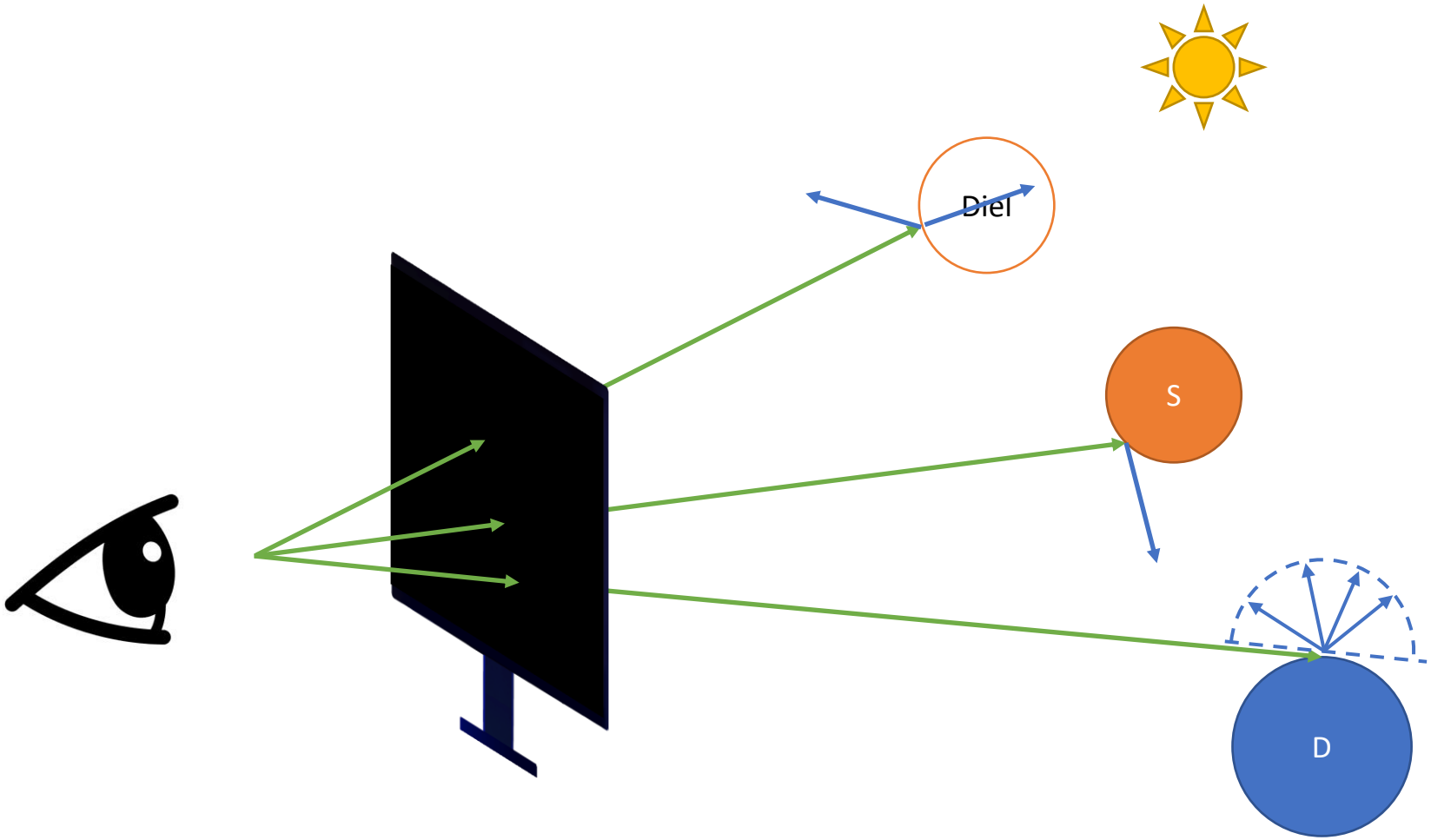
Diffuse



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



An “unified” ray tracer

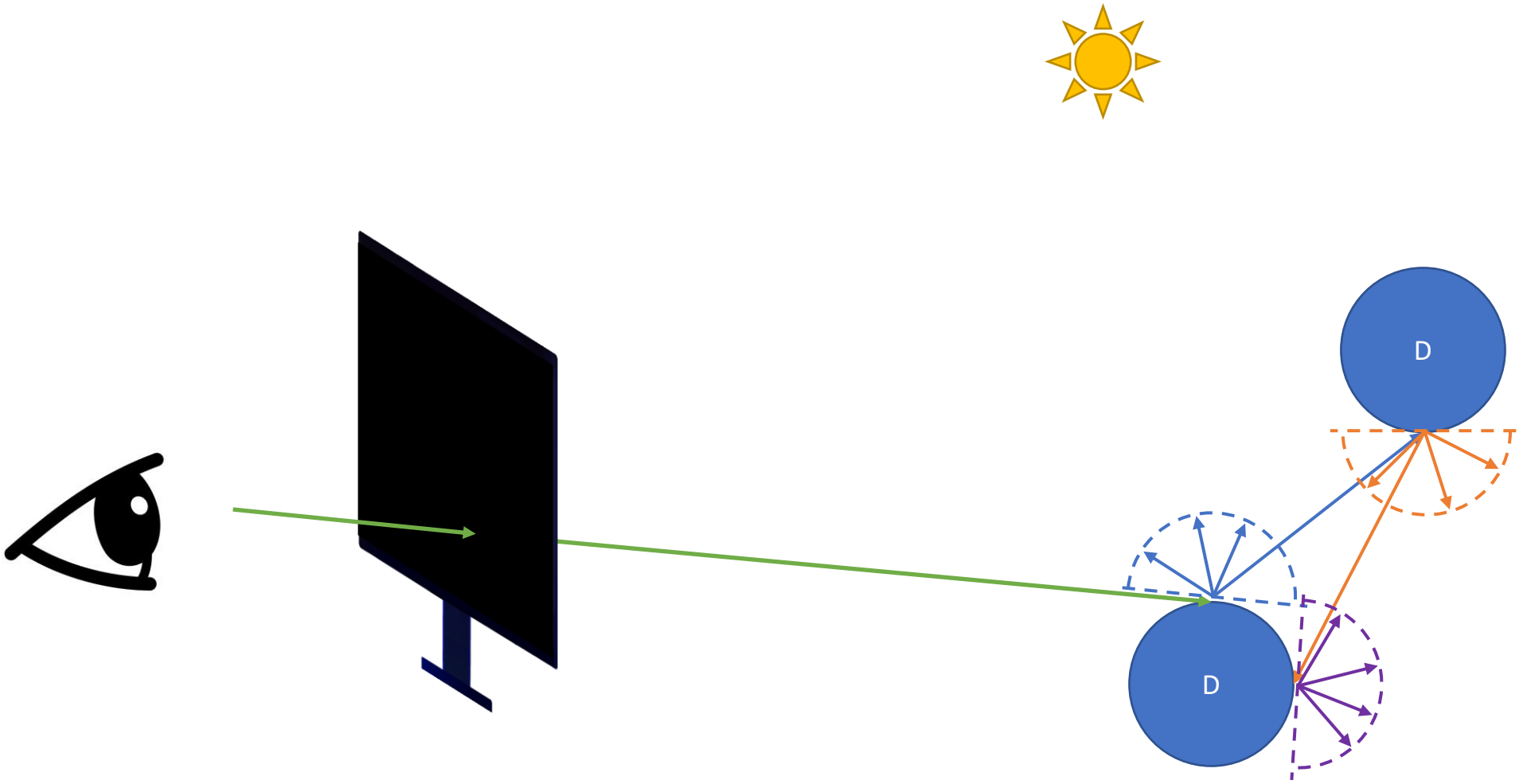


Problems?

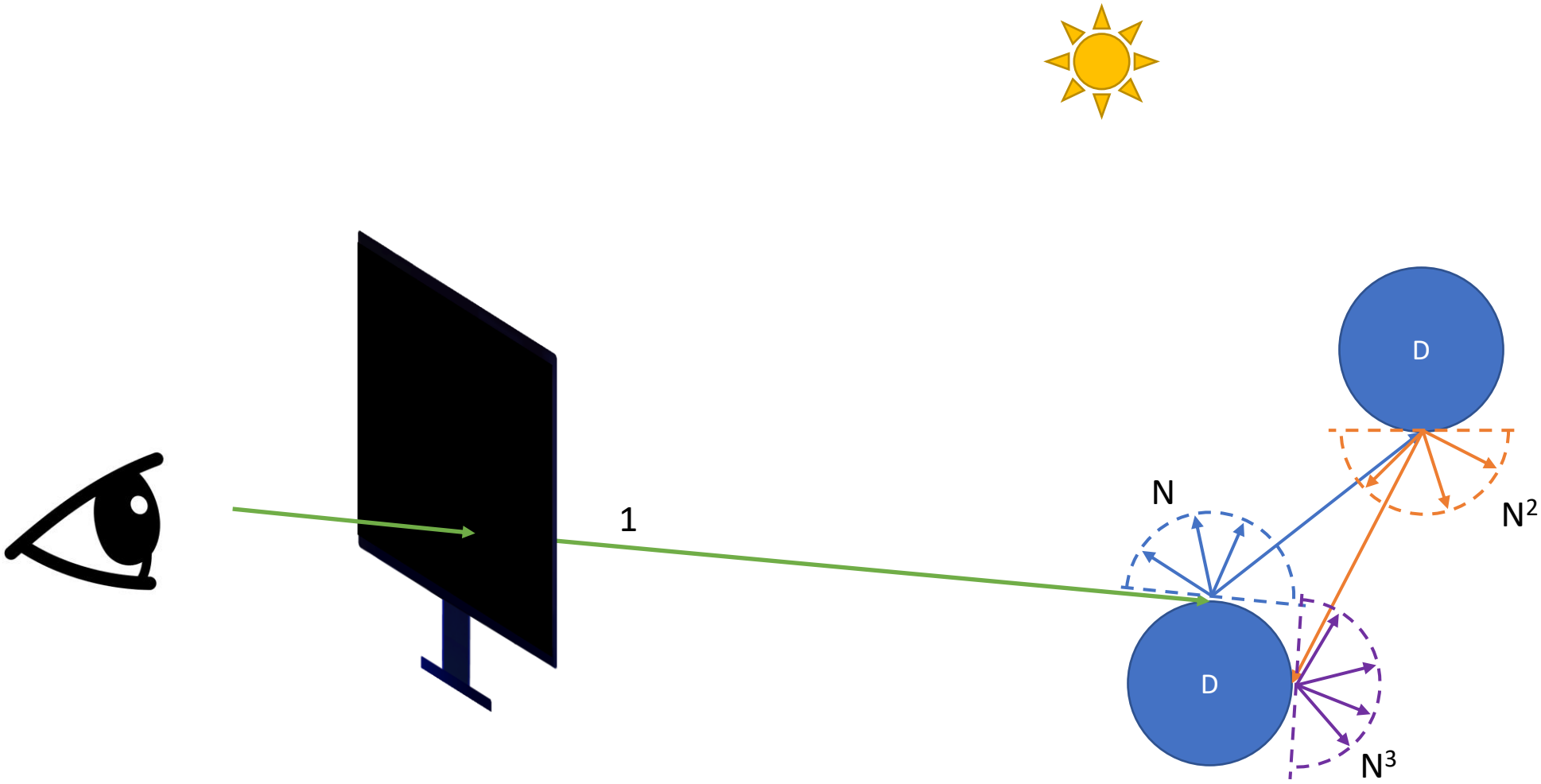
- Is unacceptably computationally heavy

- Lacks of stop criterion

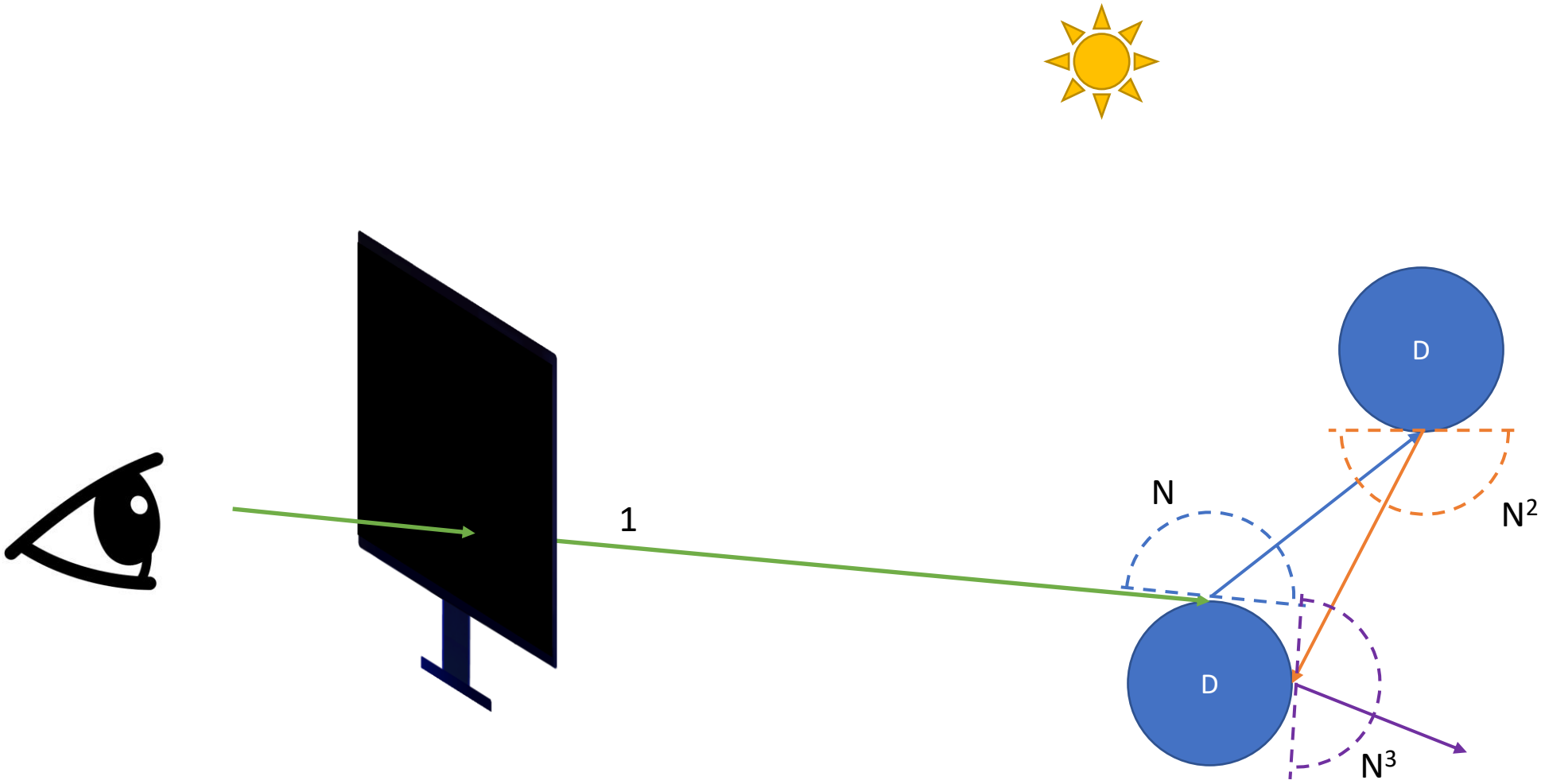
Problem: expensive



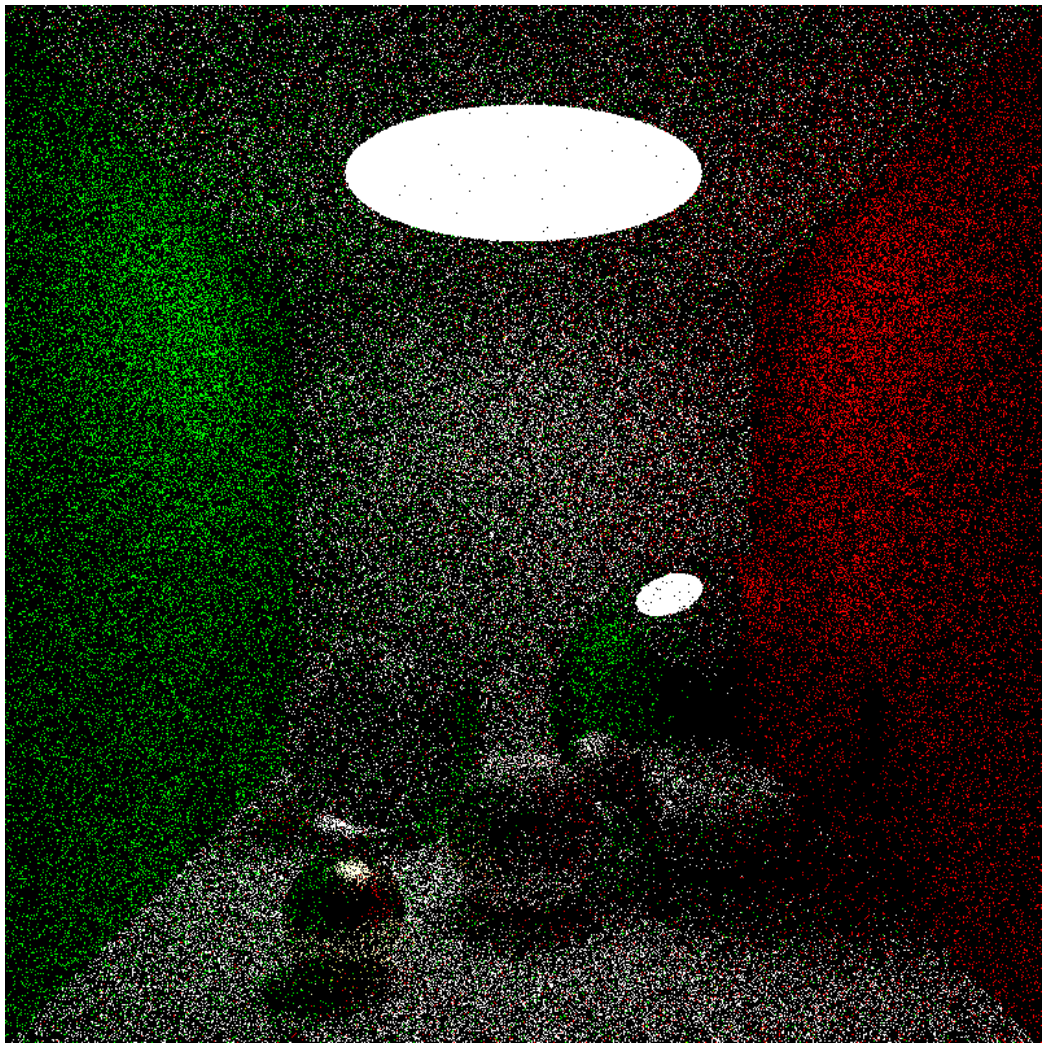
Problem: expensive

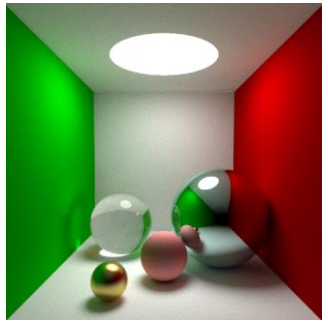


Solution: N = 1

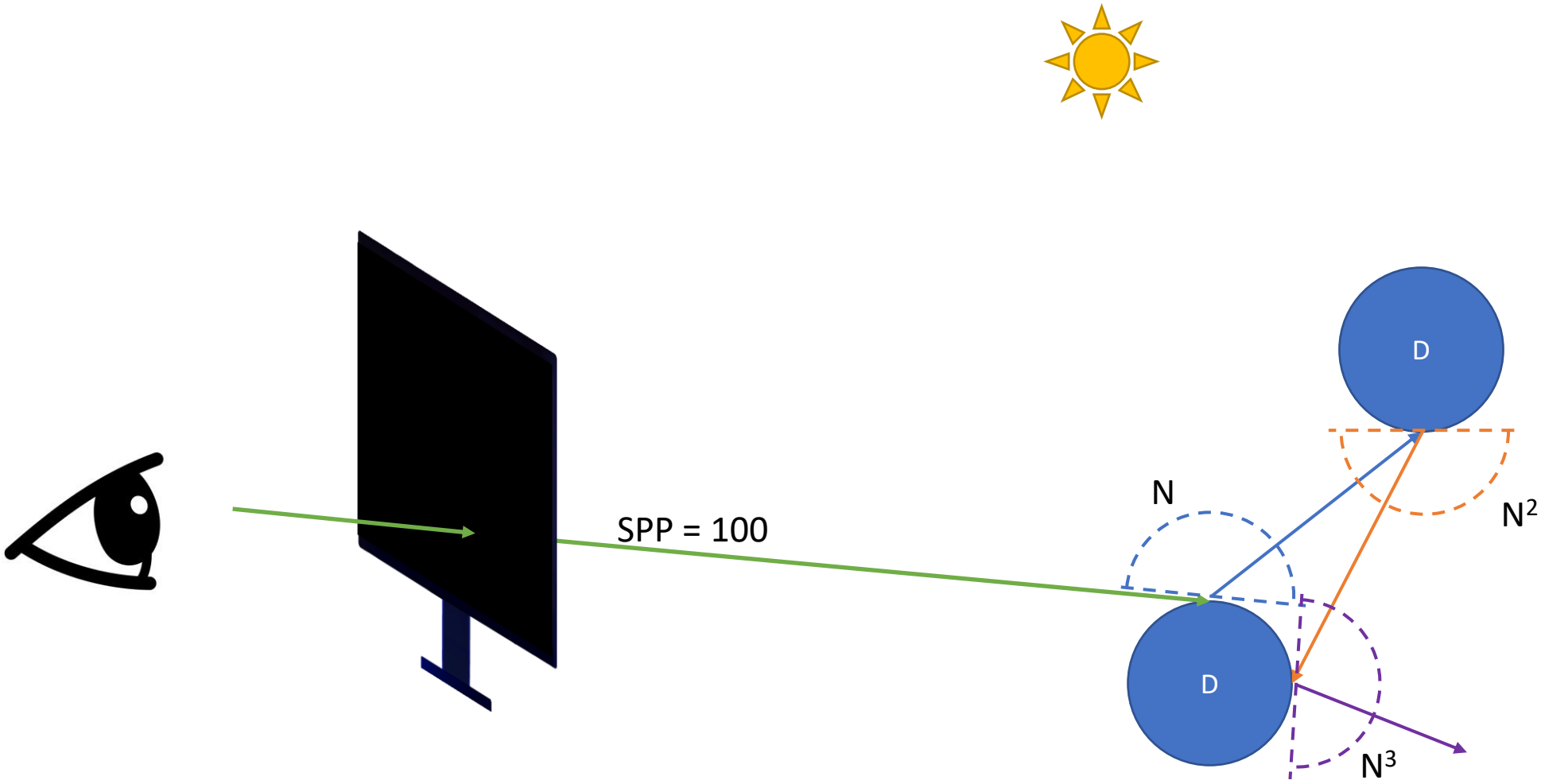


Noisy

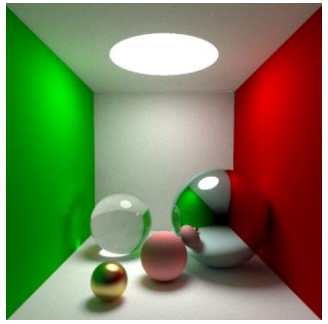


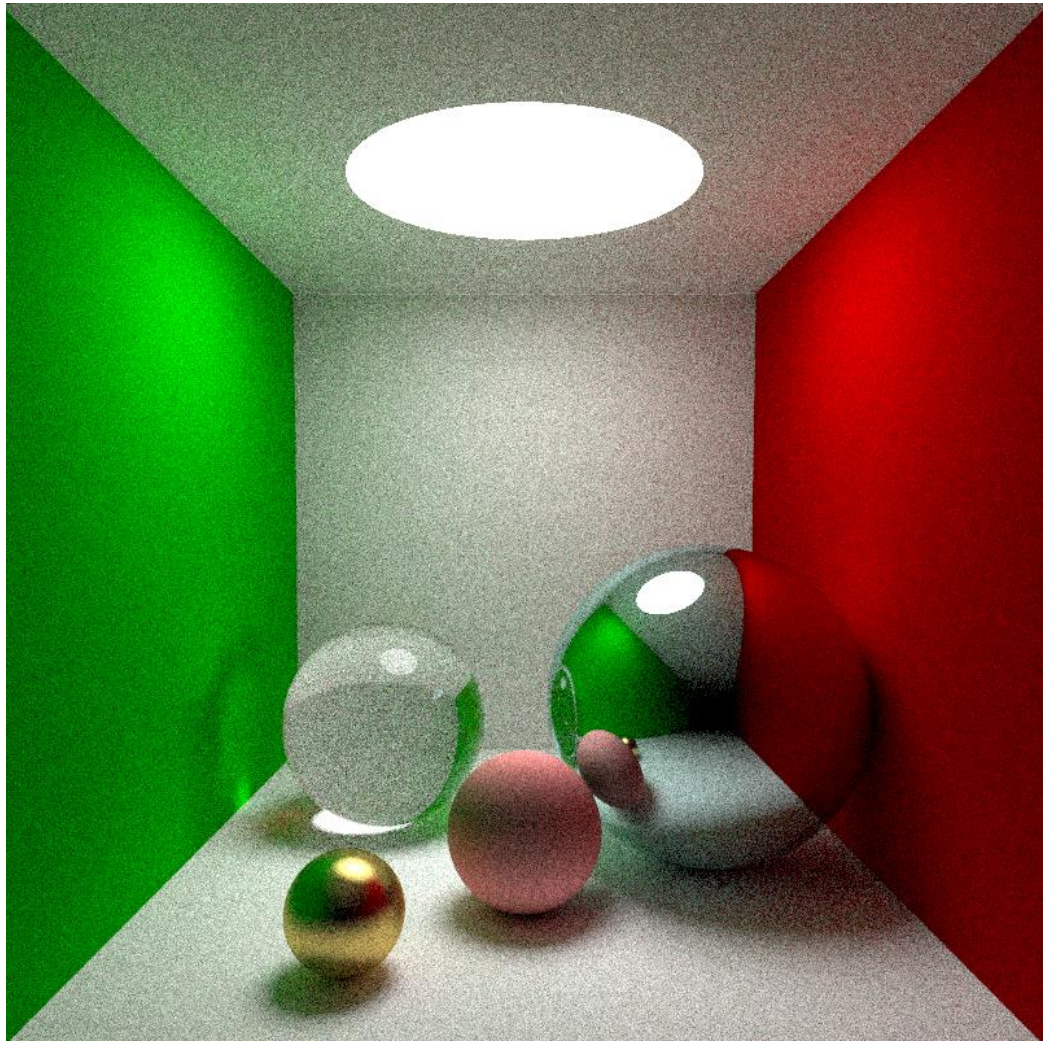


Solution:
N = 1, adding SPP (samples per pixel)



Better ☺





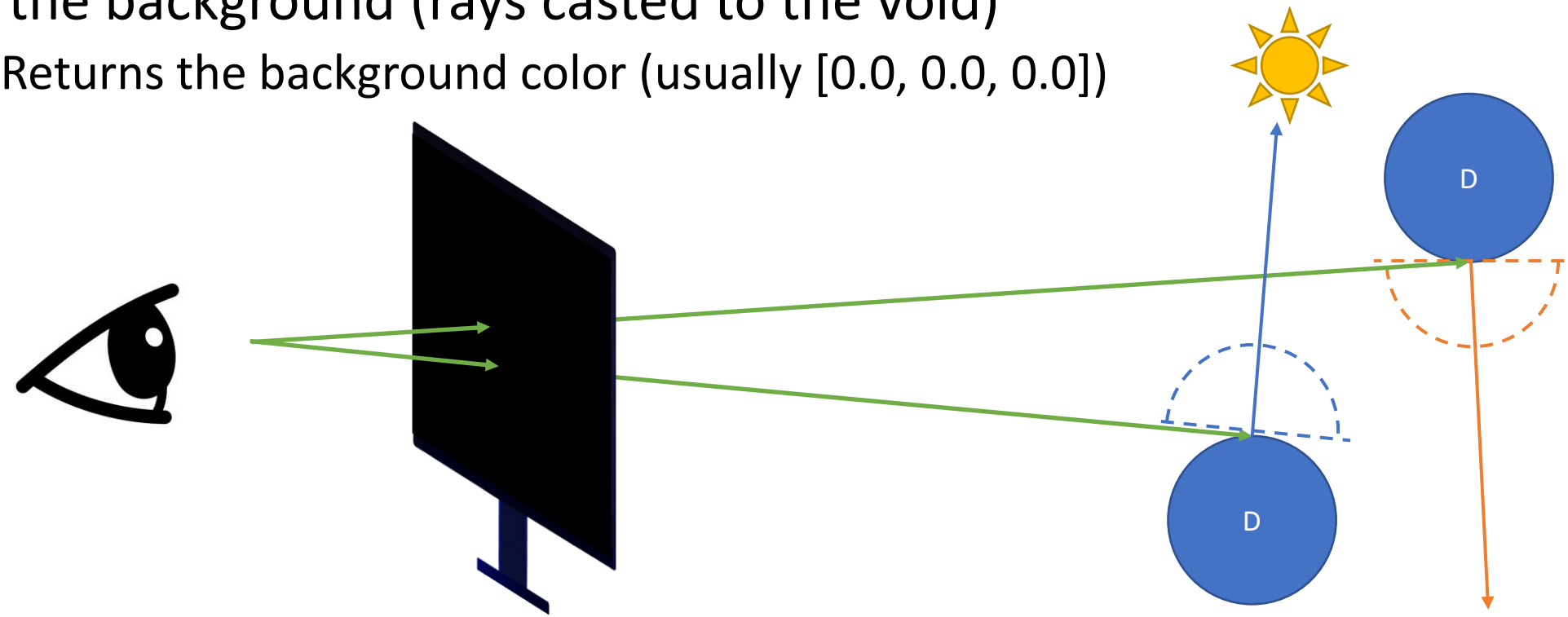
Problems?

- Is unacceptably computationally heavy

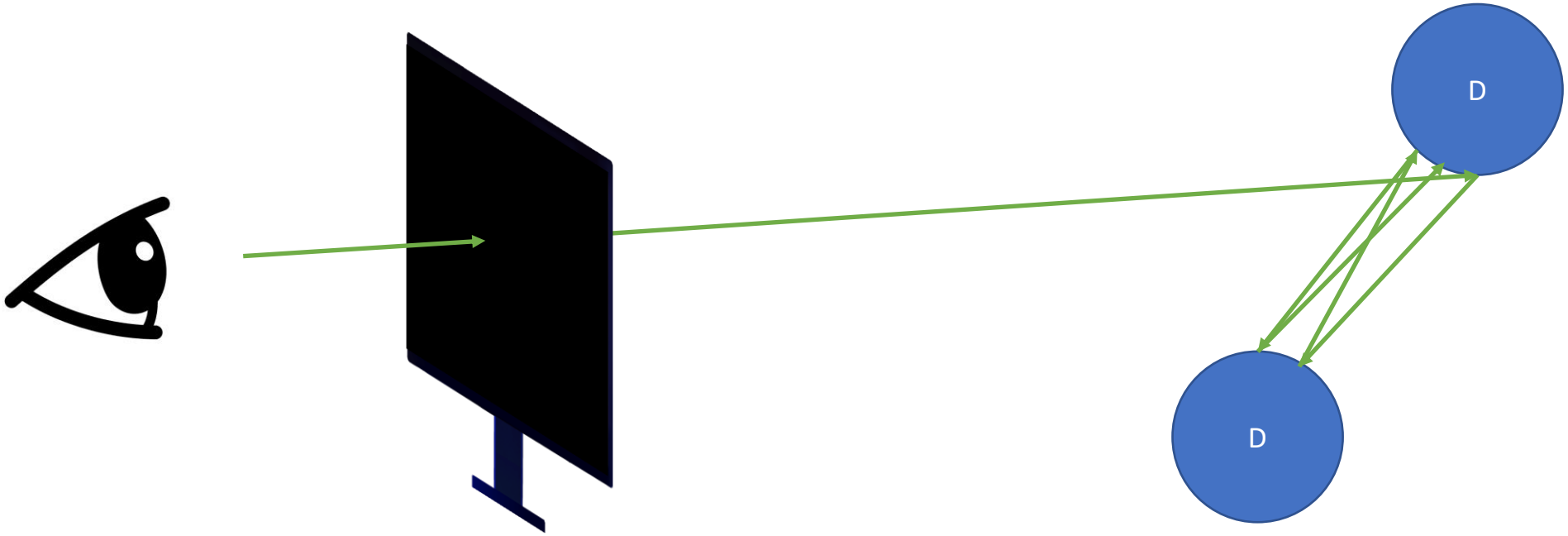
- Lacks of stop criterion

The current stop criterion

- Hit a light source
  - Returns the color of the light source (usually [1.0, 1.0, 1.0])
- Hit the background (rays casted to the void)
  - Returns the background color (usually [0.0, 0.0, 0.0])

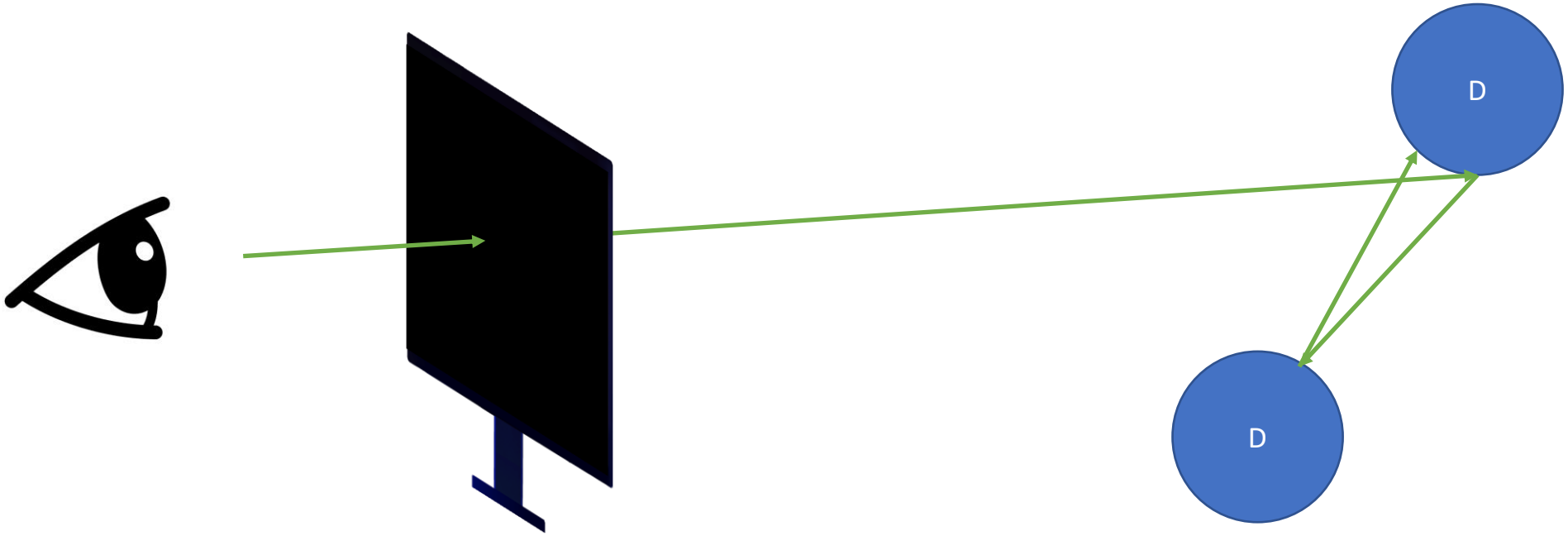


Problem: infinite loop

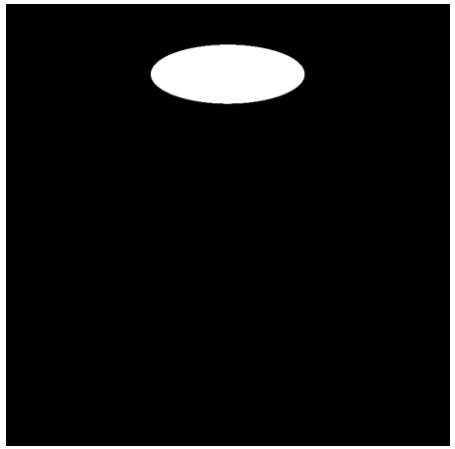


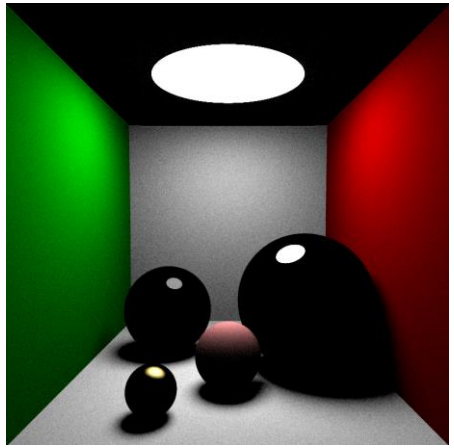
Solution: set depth of recursion

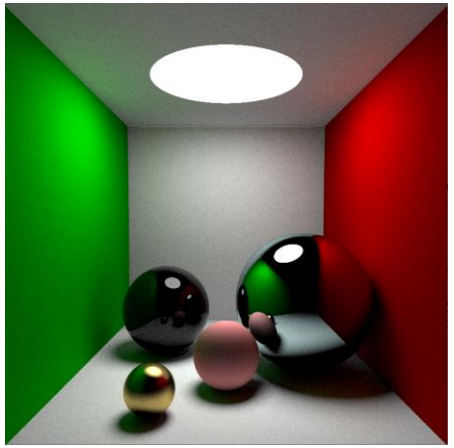
- "Stop recursion at the 2nd bounce"
  - But that loses energy!

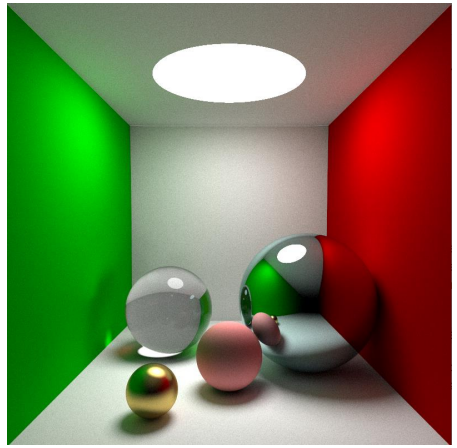


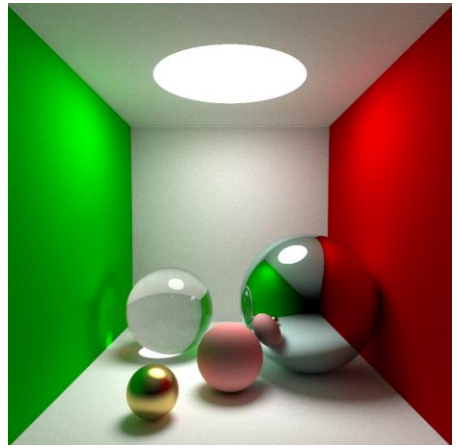
Cap the recursion depth


Depth=1

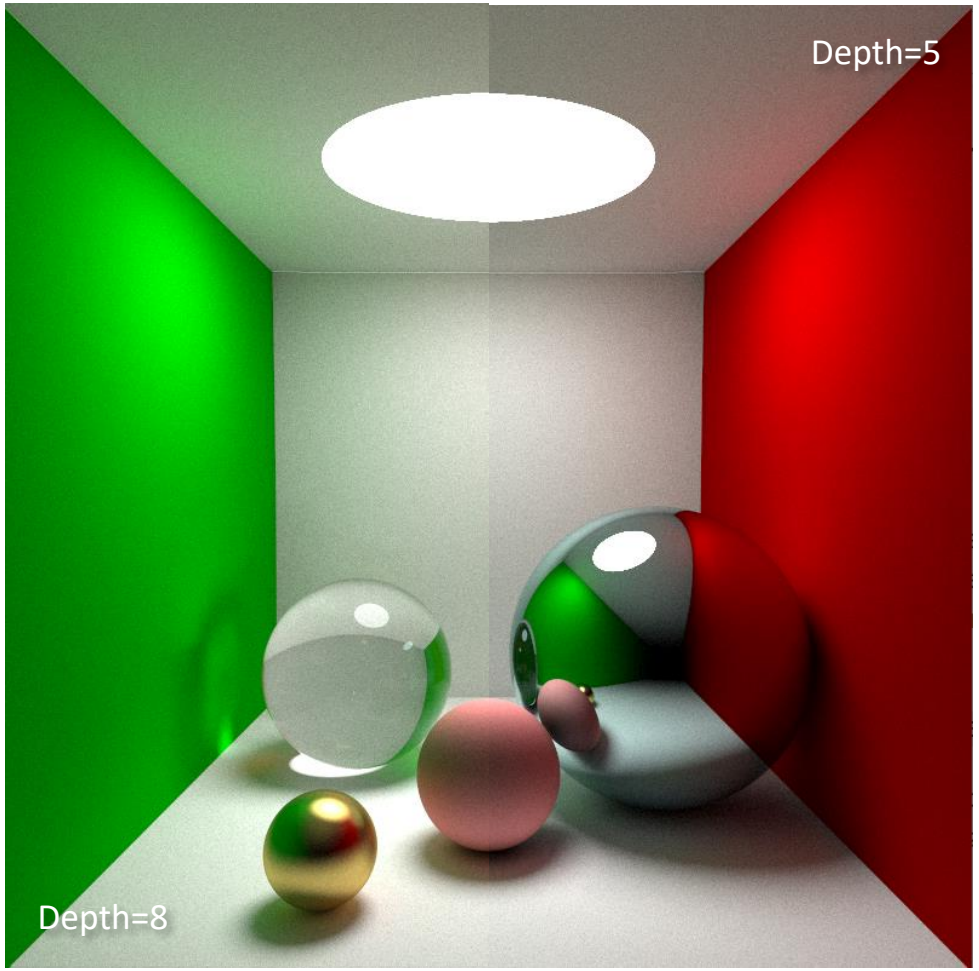

Depth=2


Depth=3


Depth=5

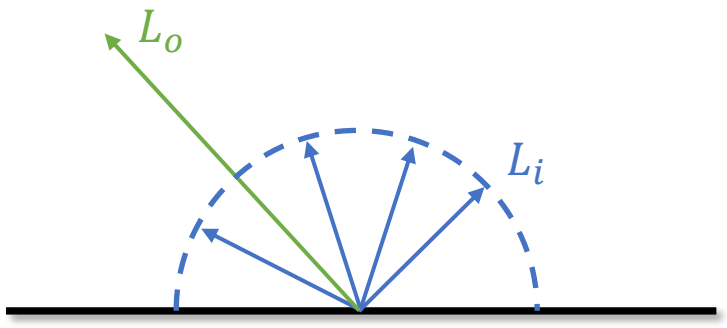

Depth=8

Cap the recursion depth (zoomed in)



Solution 2: Russian Roulette

- When asked "what color does this ray see?"
  - Set a probability p_RR (for instance 90%)
  - Roll a dice (between 0 and 1)
  - if roll() > p_RR:
    - Stop recursion
    - Return 0
  - else:
    - Go on recursion: what is L_i?
    - Return L_o/p_RR

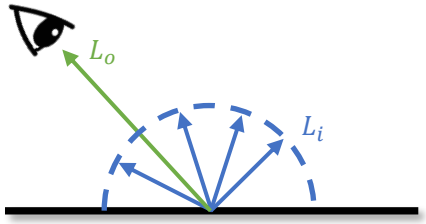


Put everything together

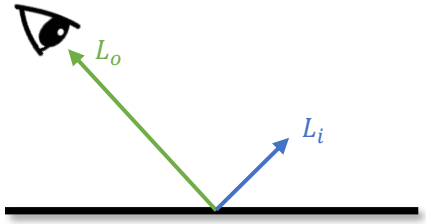
```
def what_color_does_this_ray_see(ray_o, ray_dir):
    if (random() > p_RR):
        return 0
    else:
        flag, P, N, material = first_hit(ray_o, ray_dir, scene)
        if flag == False:
            return 0
        if material.type == LIGHT_SOURCE:
            return 1
        else:
            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
```

The path tracer

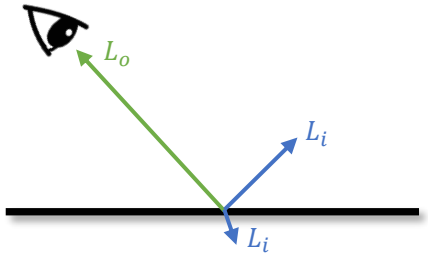
- Diffuse surfaces scatter light rays as well:
  - Monte Carlo!
- Every hit results in ONE scattered ray:
  - But we sample every pixel Multiple times
- Add the stop criterion:
  - Russian Roulette!



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



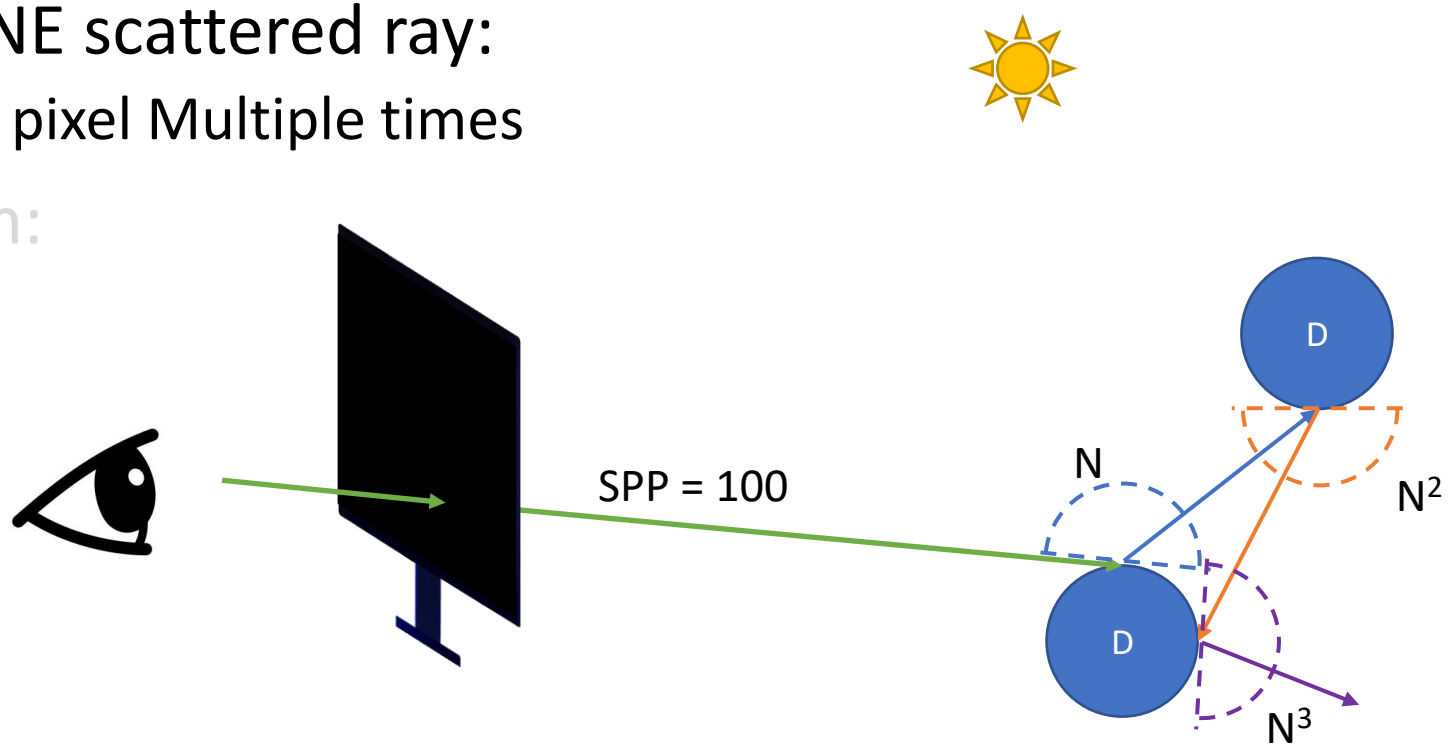
$$L_o = L_i$$



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

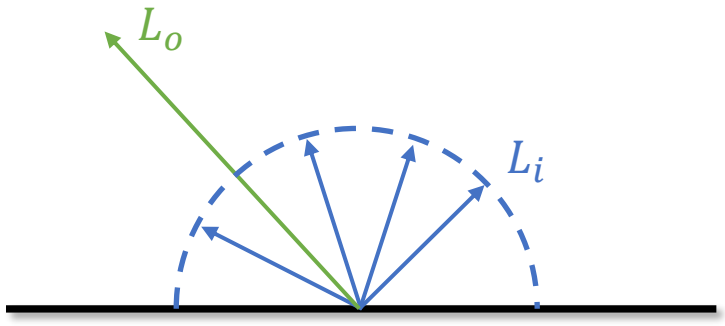
The path tracer

- Diffuse surfaces scatter light rays as well:
  - Monte Carlo!
- Every hit results in ONE scattered ray:
  - But we sample every pixel Multiple times
- Add the stop criterion:
  - Russian Roulette!



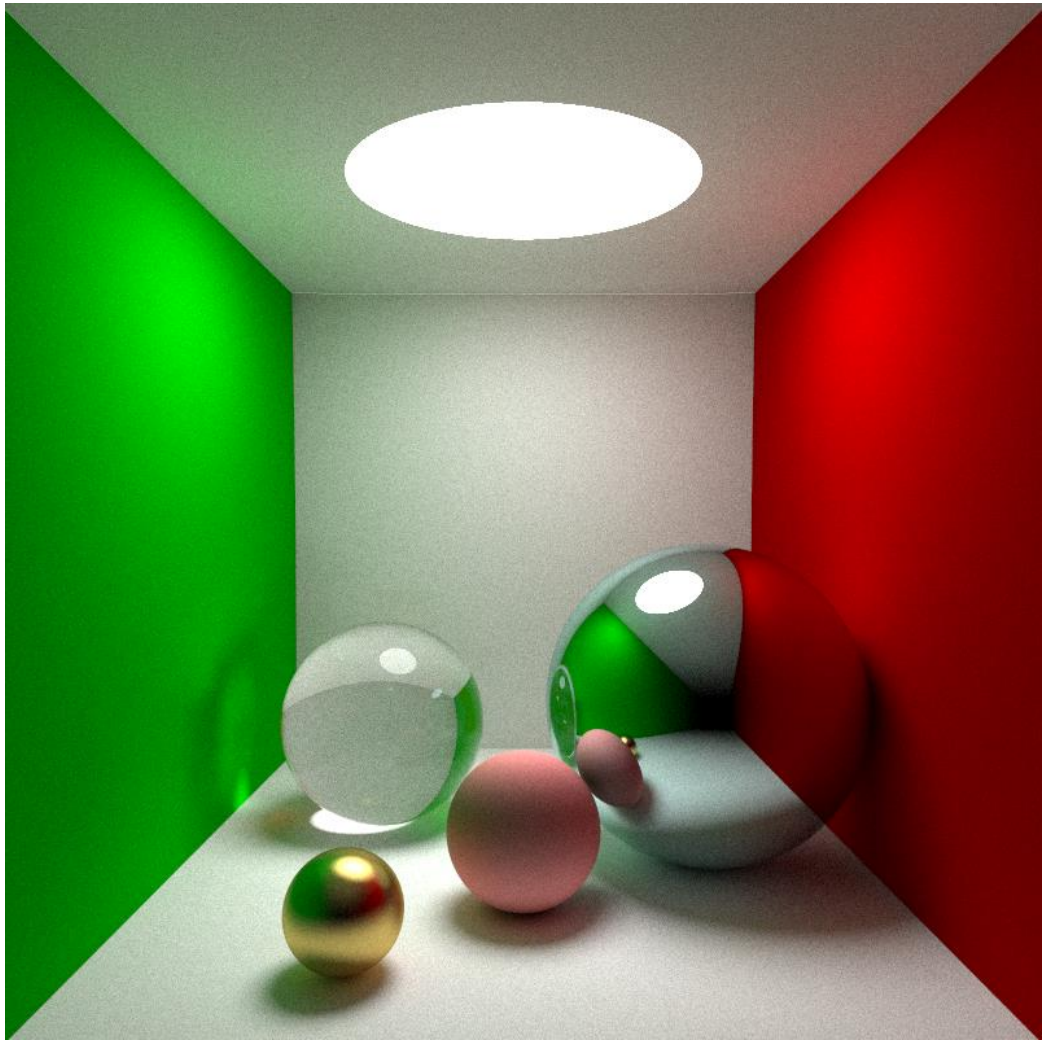
The path tracer

- Diffuse surfaces scatter light rays as well:
  - Monte Carlo!
- Every hit results in ONE scattered ray:
  - But we sample every pixel Multiple times



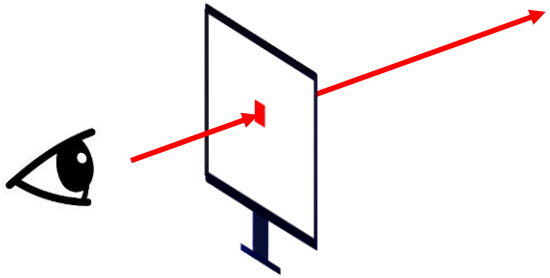
- Add the stop criterion:
  - Russian Roulette!
  - Depth caps are usually enabled too…

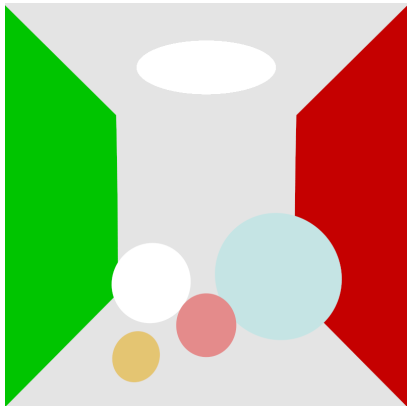
Finally: a path tracer



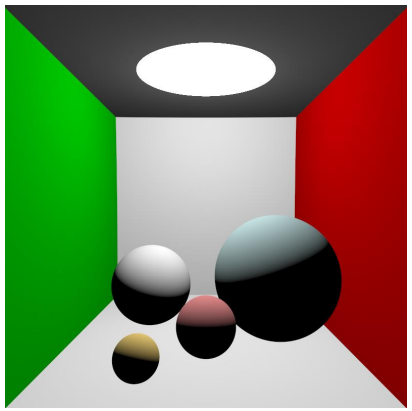
Remark

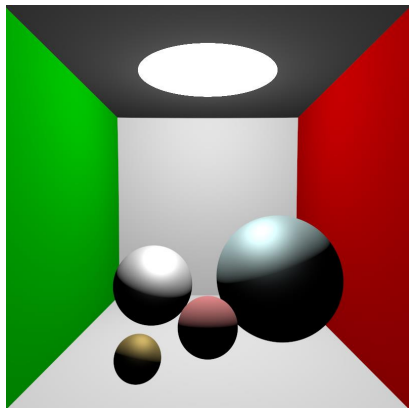
What color does the ray see?



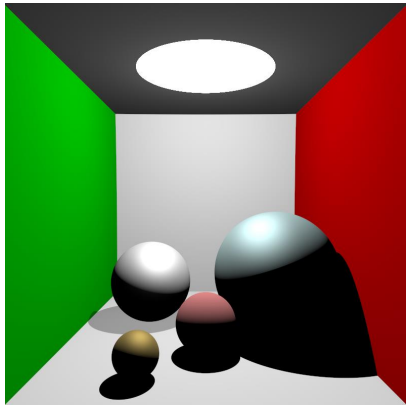


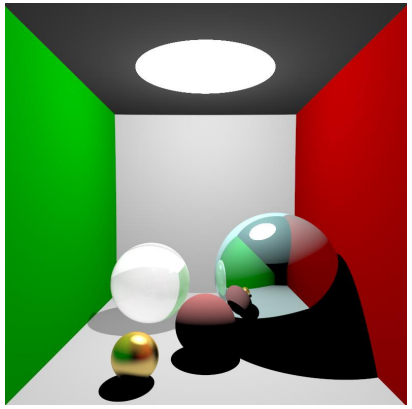
Color



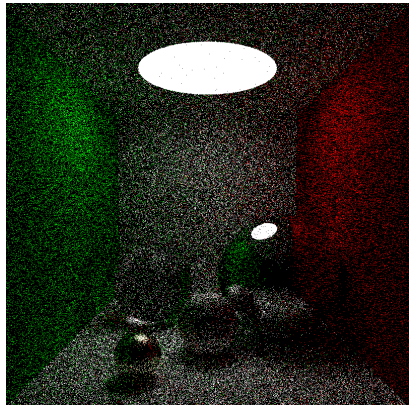


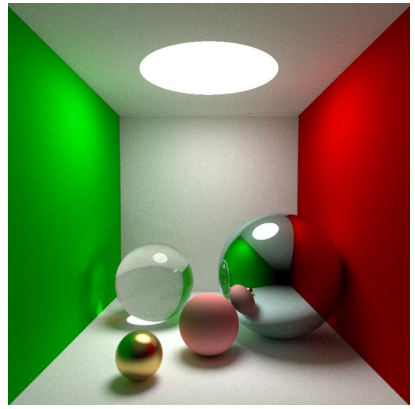
The Shading Models





The Whitted-style Ray Tracer





The Path Tracer

In the next week: implementations

- Ray-casting from the camera (eye) ?
- Anti-aliasing?
- Ray-object intersections?
- Reflection v.s. refraction?
- Uniform sampling? (Importance sampling?)
- Gamma correction?
- Recursions in Taichi?
- And more …

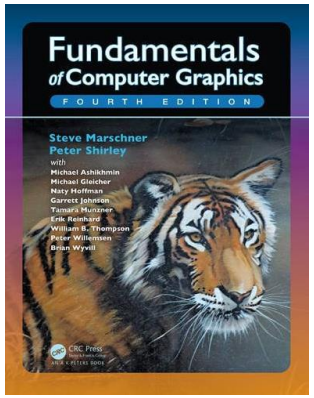
Further readings

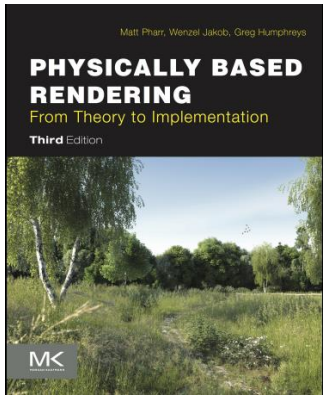
- Where do those cosines come from?
  - Radiometry!
  - The rendering equation [Kajiya 1986] [Link]

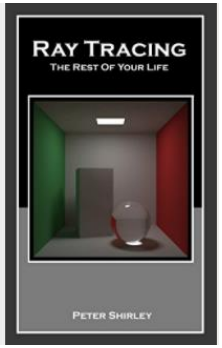
$$L_o = L_e + \int_\Omega L_i \cdot f_r \cdot \cos\theta \cdot d\omega$$

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]





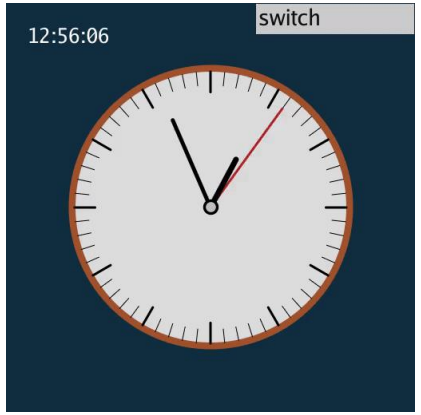


Homework

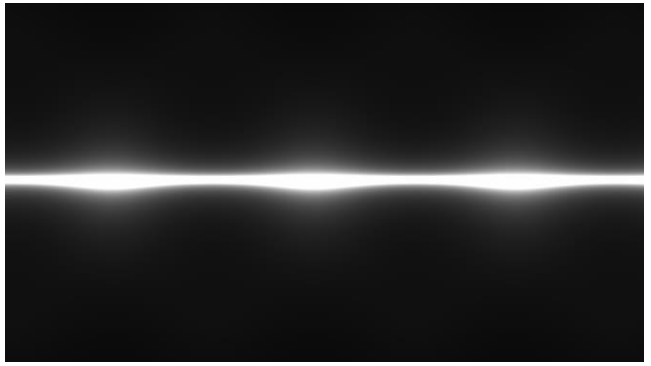
Homework Today

- Check our ray-tracer examples in Taichi
  - https://github.com/taichiCourse01/taichi_ray_tracing
- Try to connect the code with this lecture
  - You may want to watch this lecture multiple times
- Raise questions at the forum:
  - https://forum.taichi.graphics/

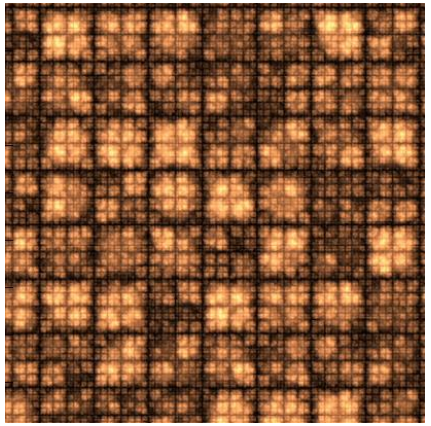
Excellent homework assignments



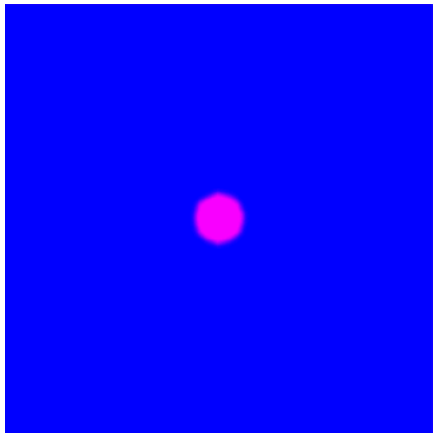
@0xzhang



@coxhin

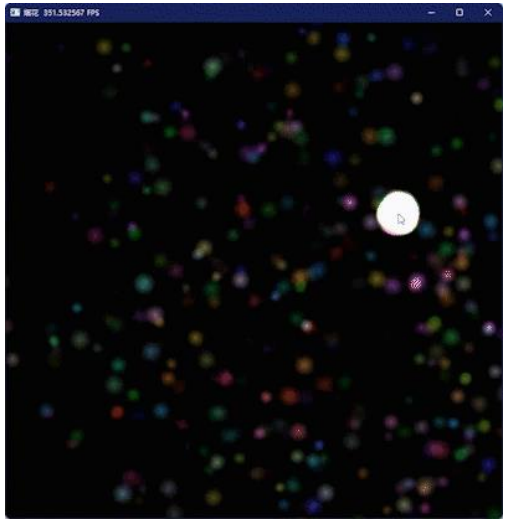


@Vineyo

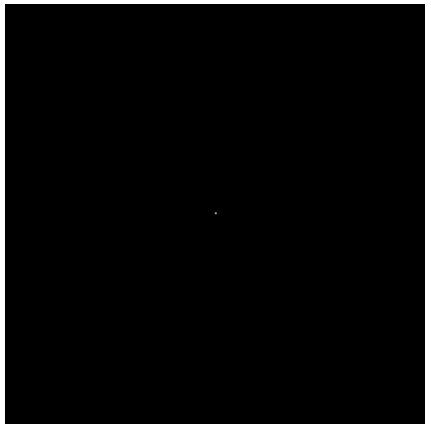


@tlcui

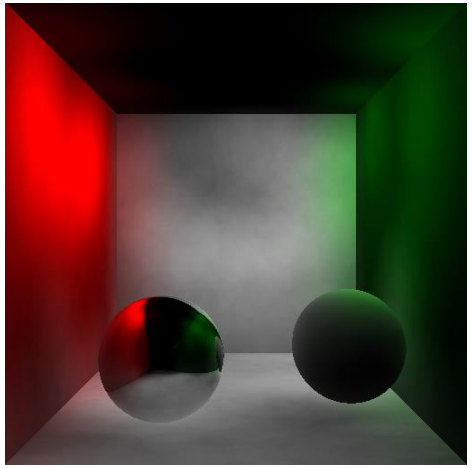
@Zydiii



@cflw

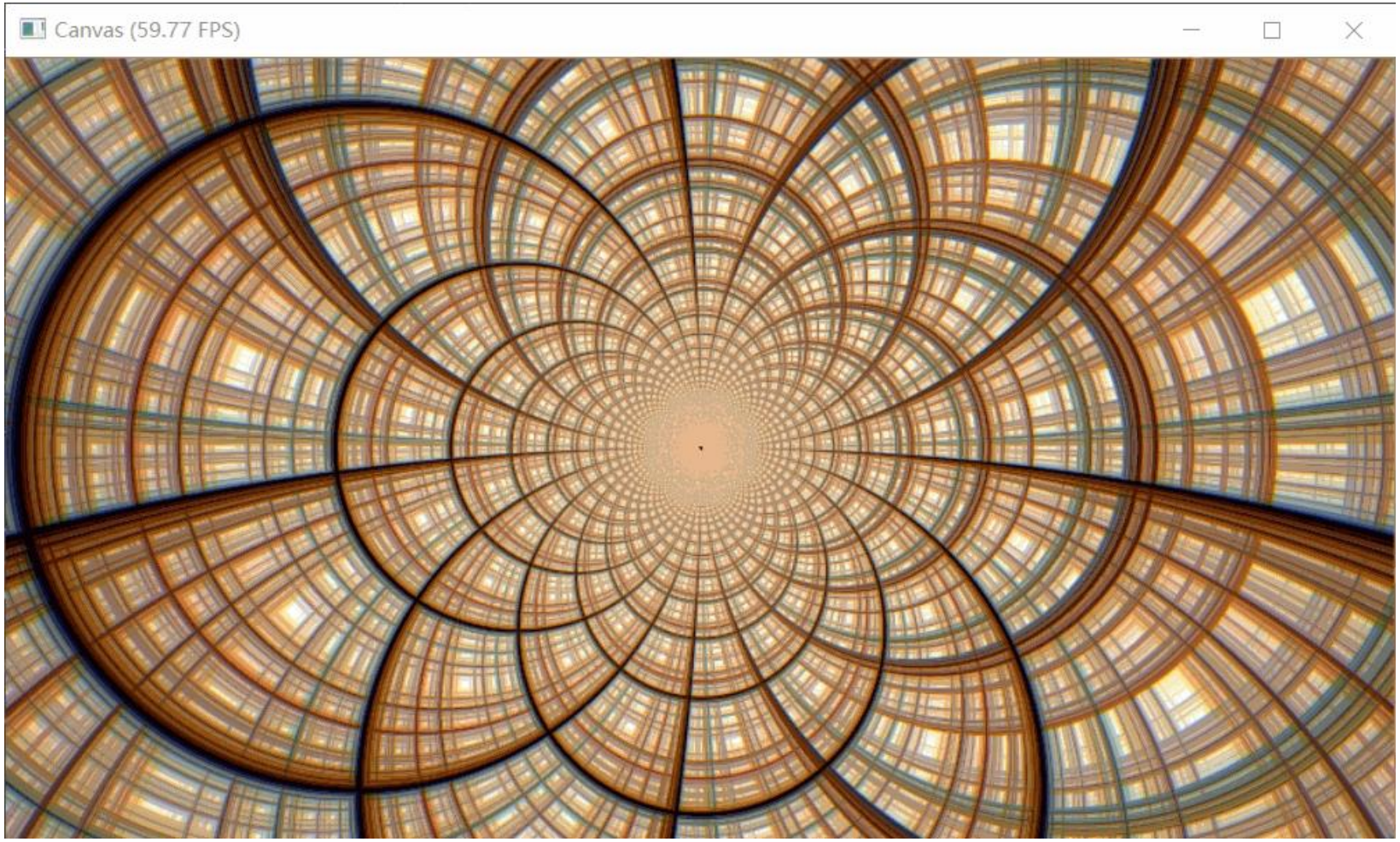


@tmzhang314159



@lightningbird

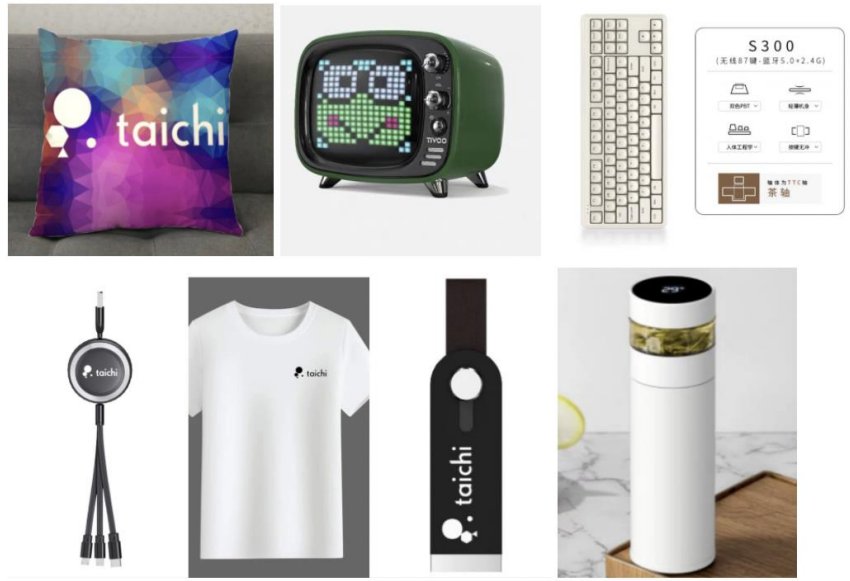
Excellent homework assignments

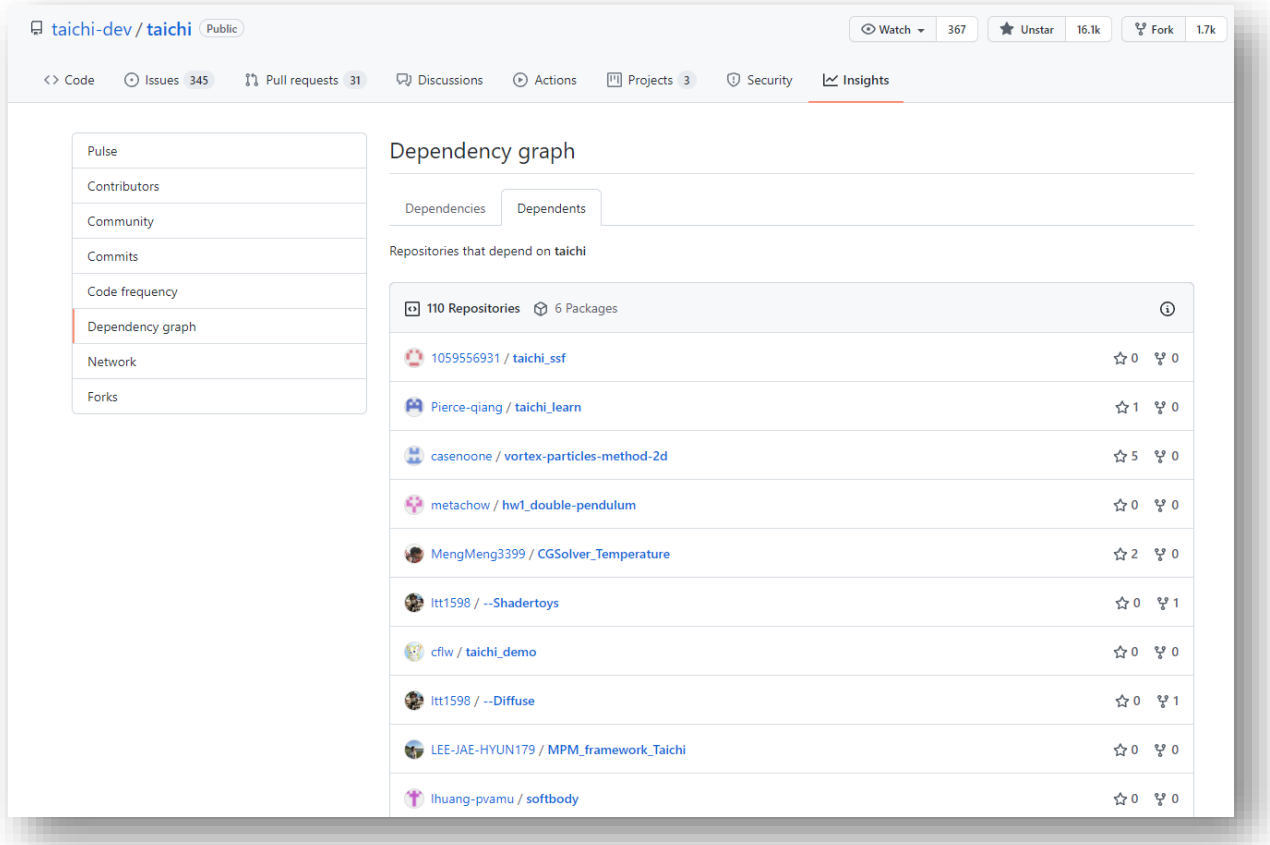


@Pierce-qiang

Gifts for the gifted

- Use Template for your homework
- Our next check date will be: Nov. 9th 2021





Questions?

本次答疑：11/04 ←作业分享也在这里

下次直播：11/09

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