Visibility Computations

EDAF80
Michael Doggett

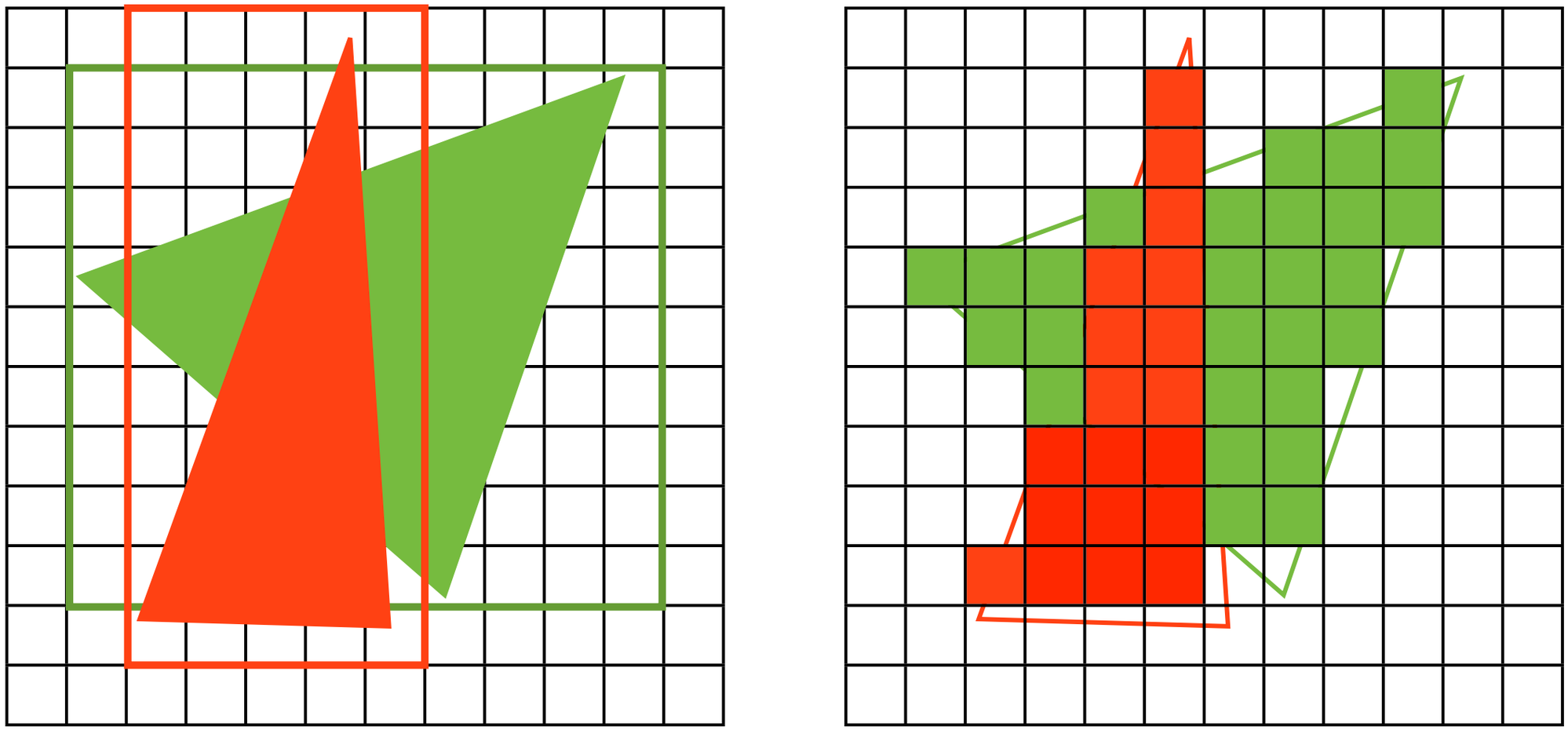
Some slides from Jacob Munkberg 2012-13

2024

Today

- Visibility computations
  - Rasterization & depth buffering
  - Ray tracing
- The rendering equation

Rasterization

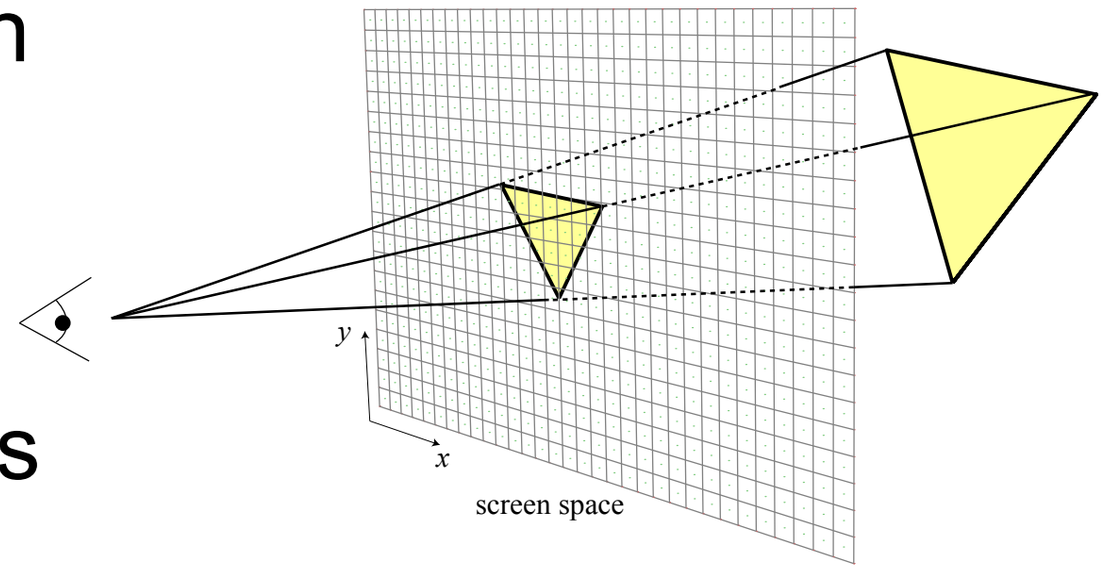


Convert triangles to pixels

Rasterization of Triangle

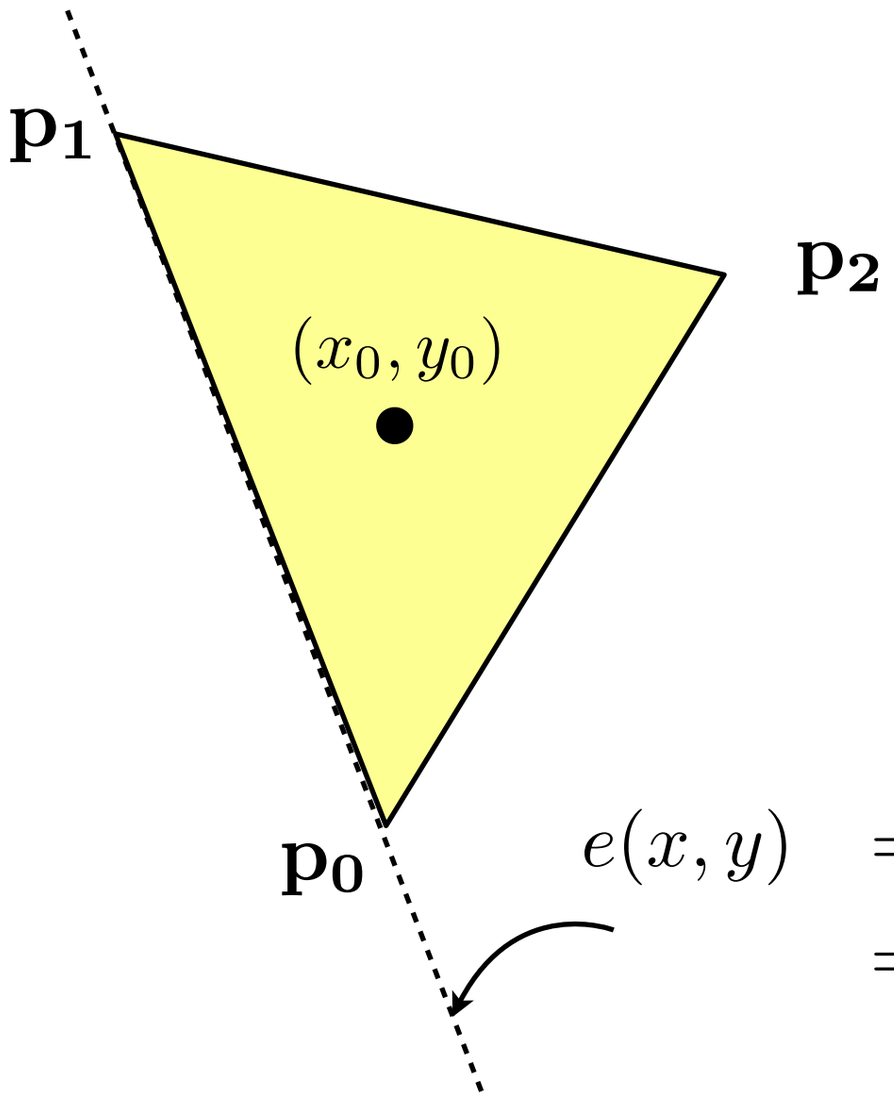
- Determine which pixels a triangle covers

1. Project triangle on screen
2. Setup *edge equations*
3. Test each pixel center against the triangle edges



Edge Equation

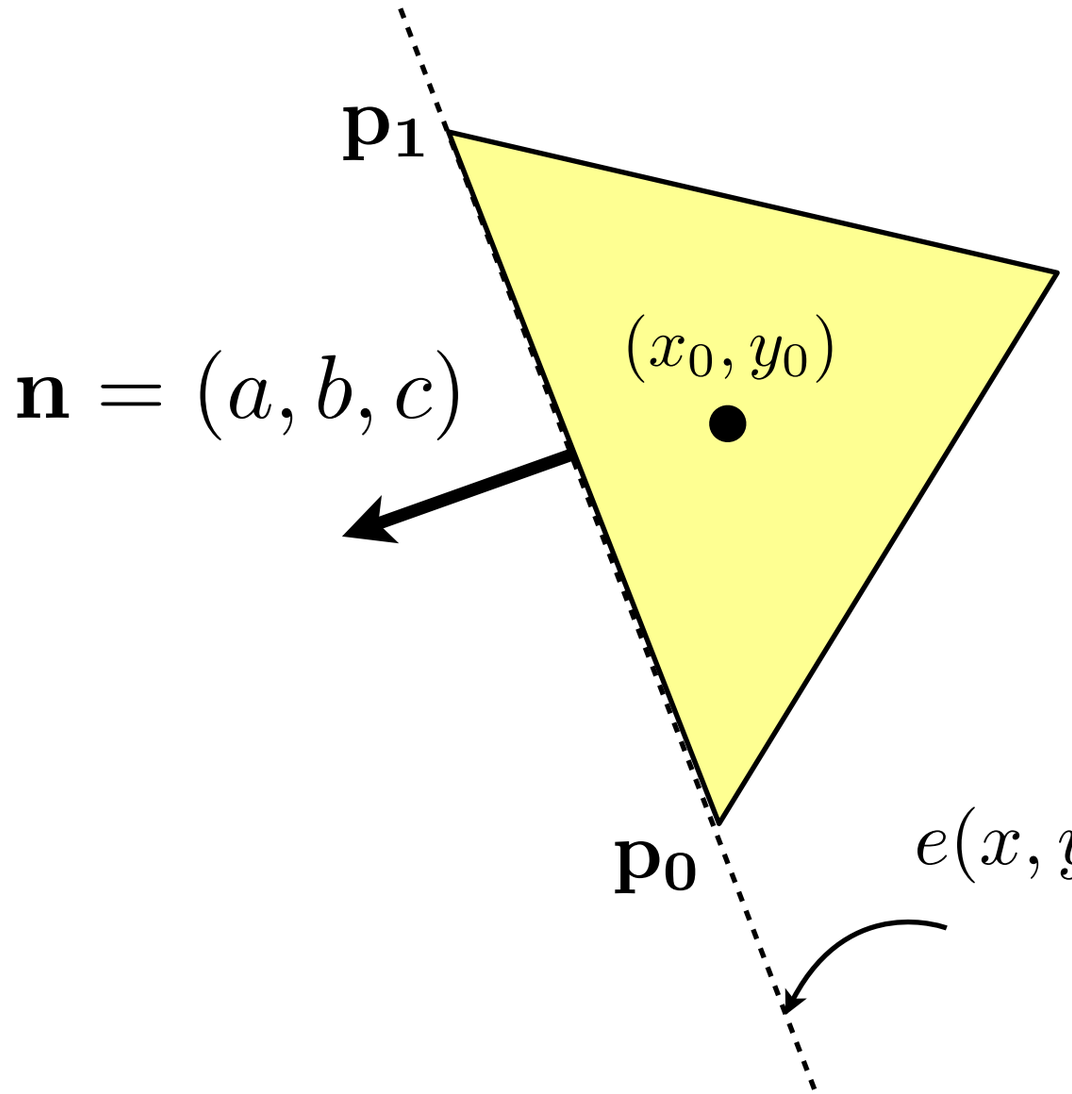
- Point inside triangle test



$$
\begin{aligned}
e(x,y) &= (\mathbf{p}_0 \times \mathbf{p}_1) \cdot (x, y, 1) \\
&= ax + by + c
\end{aligned}
$$

Edge Equation

- Point inside edge test



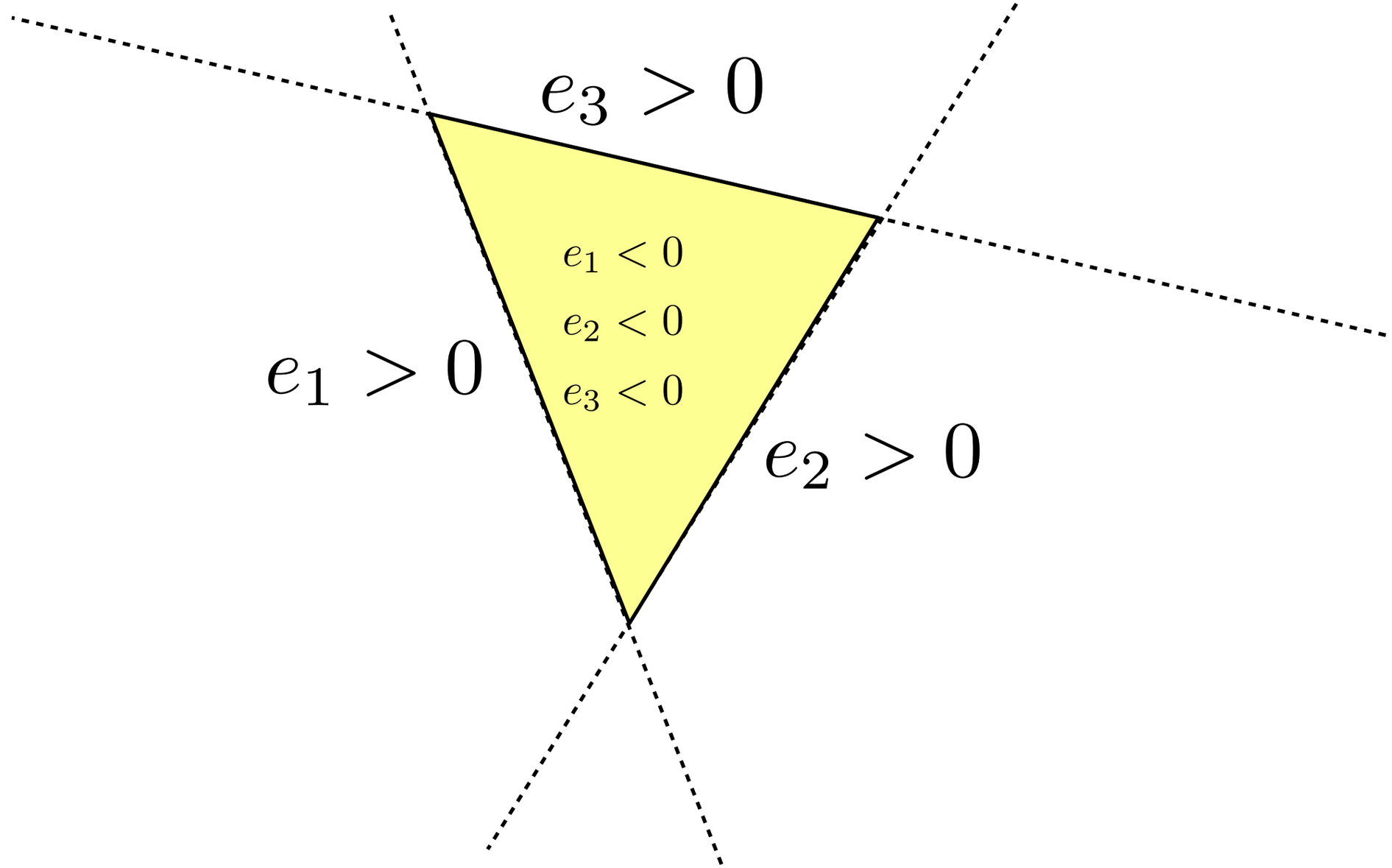
Point (x_0, y_0) inside edge between $\mathbf{p}_0$ and $\mathbf{p}_1$ if:

$$\begin{aligned} \mathbf{n} \cdot (x_0, y_0, 1) &< 0 \\ ax_0 + by_0 + c &< 0 \end{aligned}$$

$$\begin{aligned} e(x, y) &= (\mathbf{p}_0 \times \mathbf{p}_1) \cdot (x, y, 1) \\ &= ax + by + c \end{aligned}$$

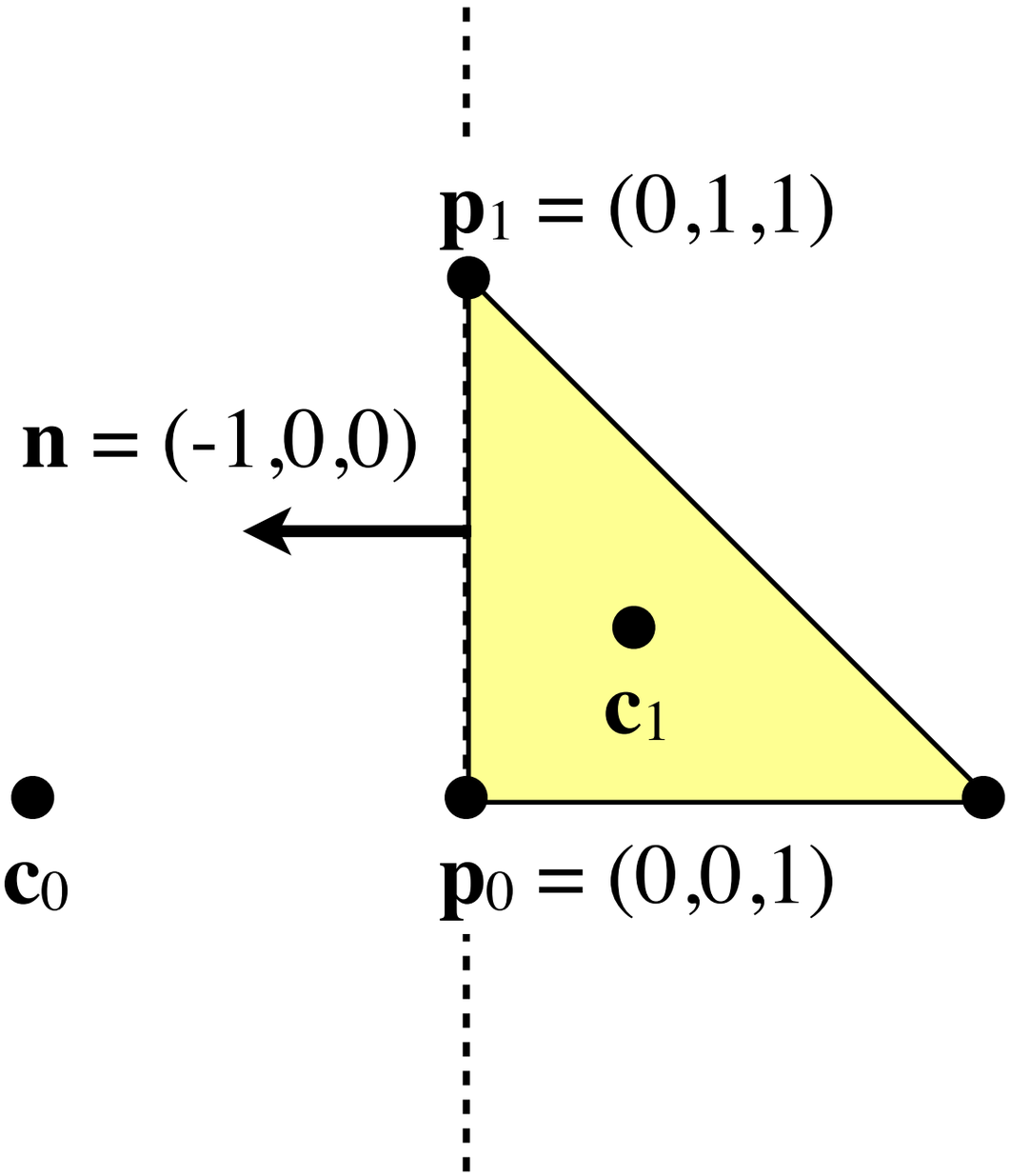
Point Inside Triangle Test

Inside all three edges: **Hit**



Edge Equation Example

Consider edge between $\mathbf{p}_0$ and $\mathbf{p}_1$



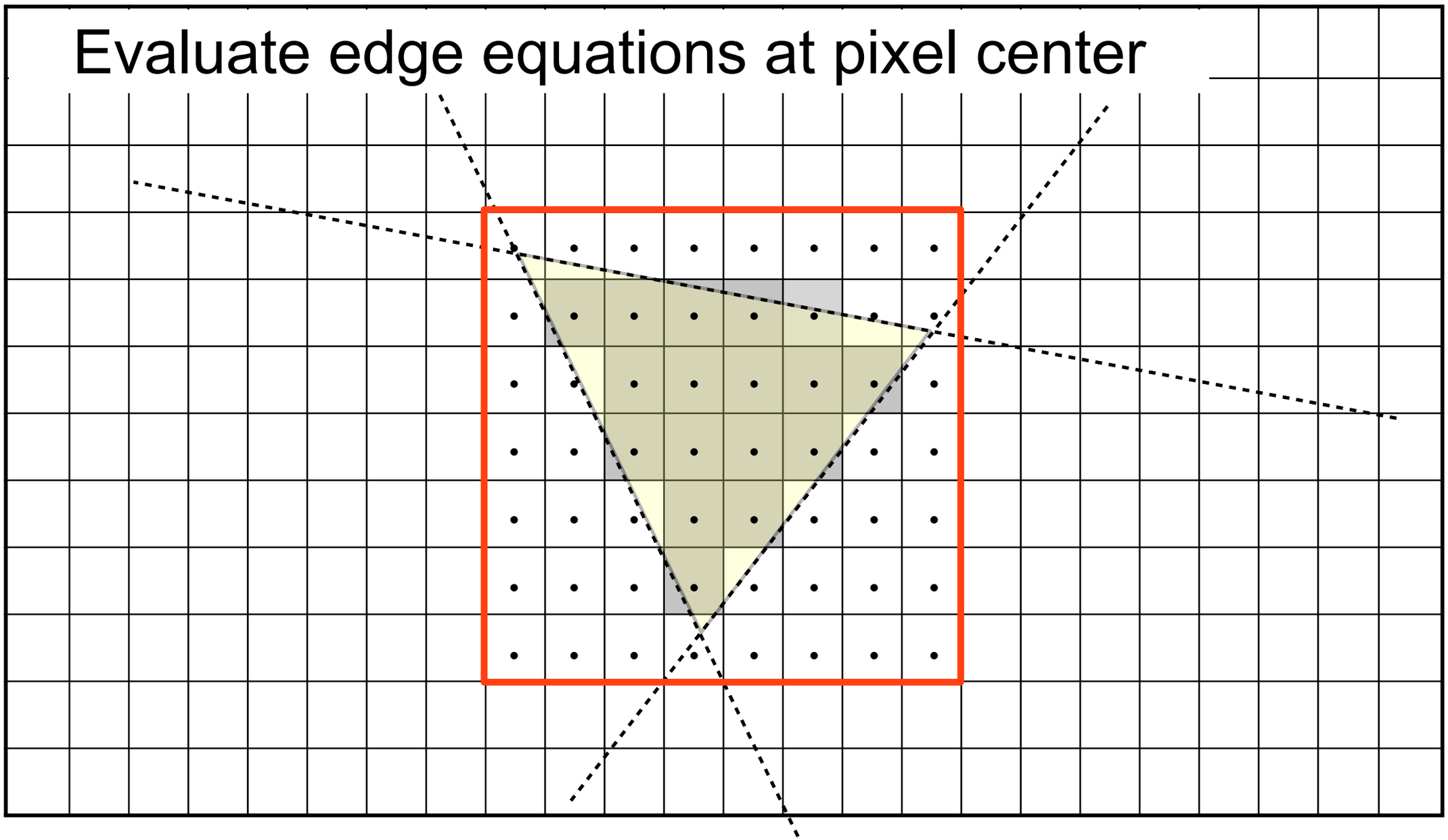
$$\mathbf{n} = \mathbf{p}_0 \times \mathbf{p}_1 = (-1,0,0)$$

$$e(x,y) = \mathbf{n} \cdot (x,y,1)$$

Point $\mathbf{c}_0$=(-1,0,1) outside edge, as $e(\mathbf{c}_0) = \mathbf{n} \cdot \mathbf{c}_0 > 0$

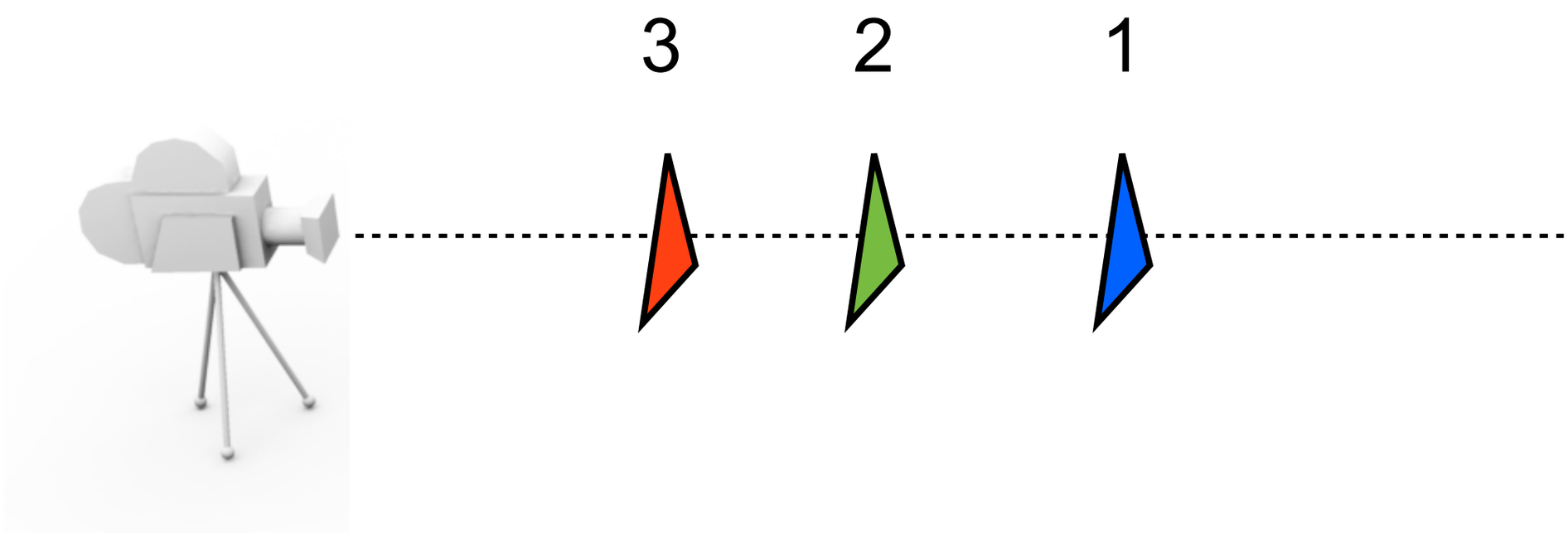
Point $\mathbf{c}_1$=(0.3,0.3,1) inside edge, as $e(\mathbf{c}_1) = \mathbf{n} \cdot \mathbf{c}_1 < 0$

Test Samples



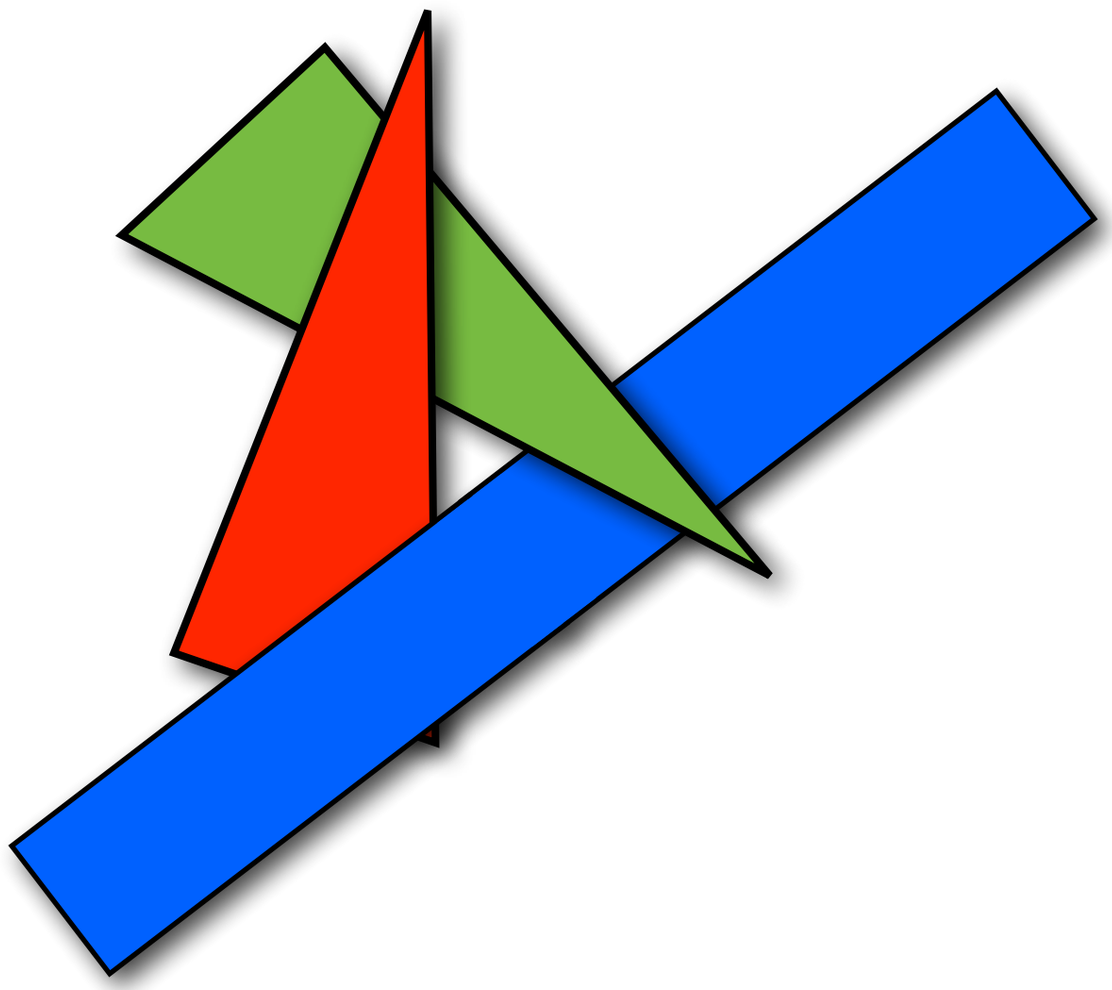
Overlapping Primitives

- Keep the closest triangle at each pixel
- Simple solution: Sort objects in depth and render back to front
- Painter's algorithm



Harder case

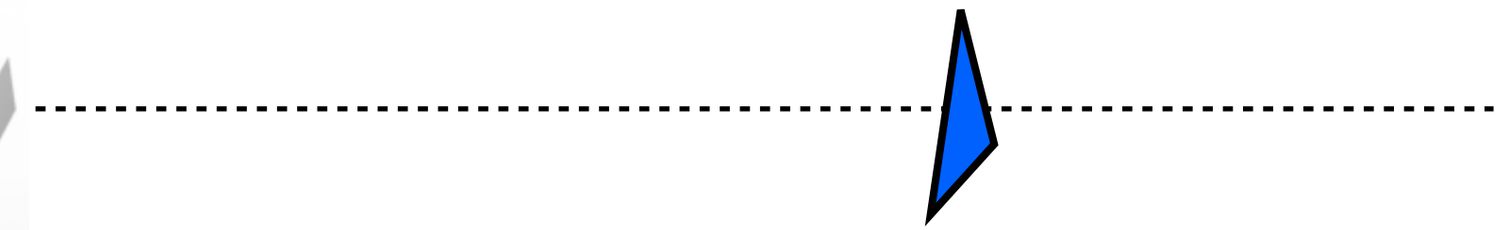
- Not always possible to sort objects in depth

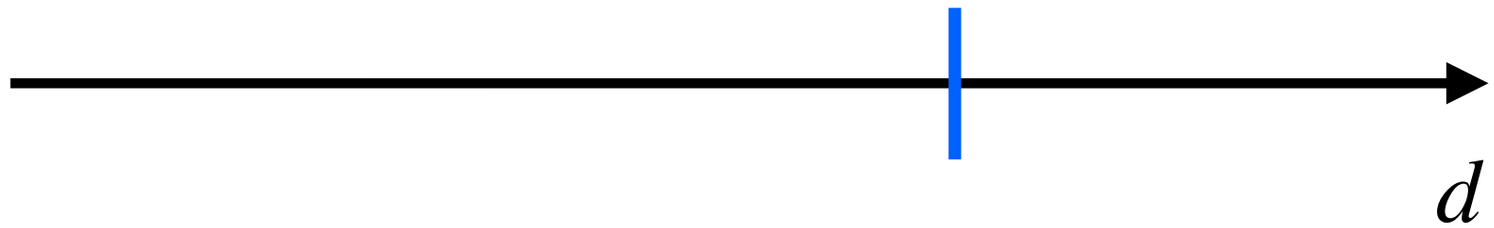


Depth Buffering

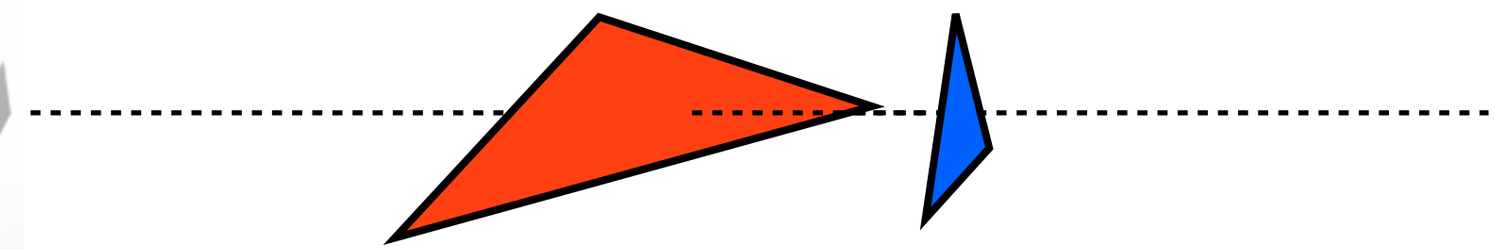
- For each pixel, store a depth value
  - Initialize to large value: $d_{\text{stored}} := \text{FLT\_MAX}$
- For each pixel, compute depth value d_{new}, of **current triangle** at hitpoint
  - If $d_{\text{new}} < d_{\text{stored}}$ we have a hit. Update the depth buffer: $d_{\text{stored}} := d_{\text{new}}$, and call the pixel shader
  - Otherwise, the triangle is covered by already drawn primitives. Move to next pixel.
  - In OpenGL : `glEnable(GL_DEPTH_TEST);`

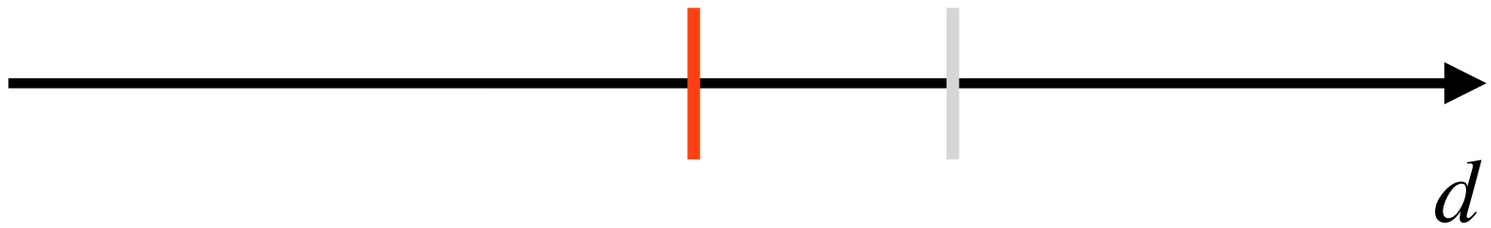
Overlapping Primitives



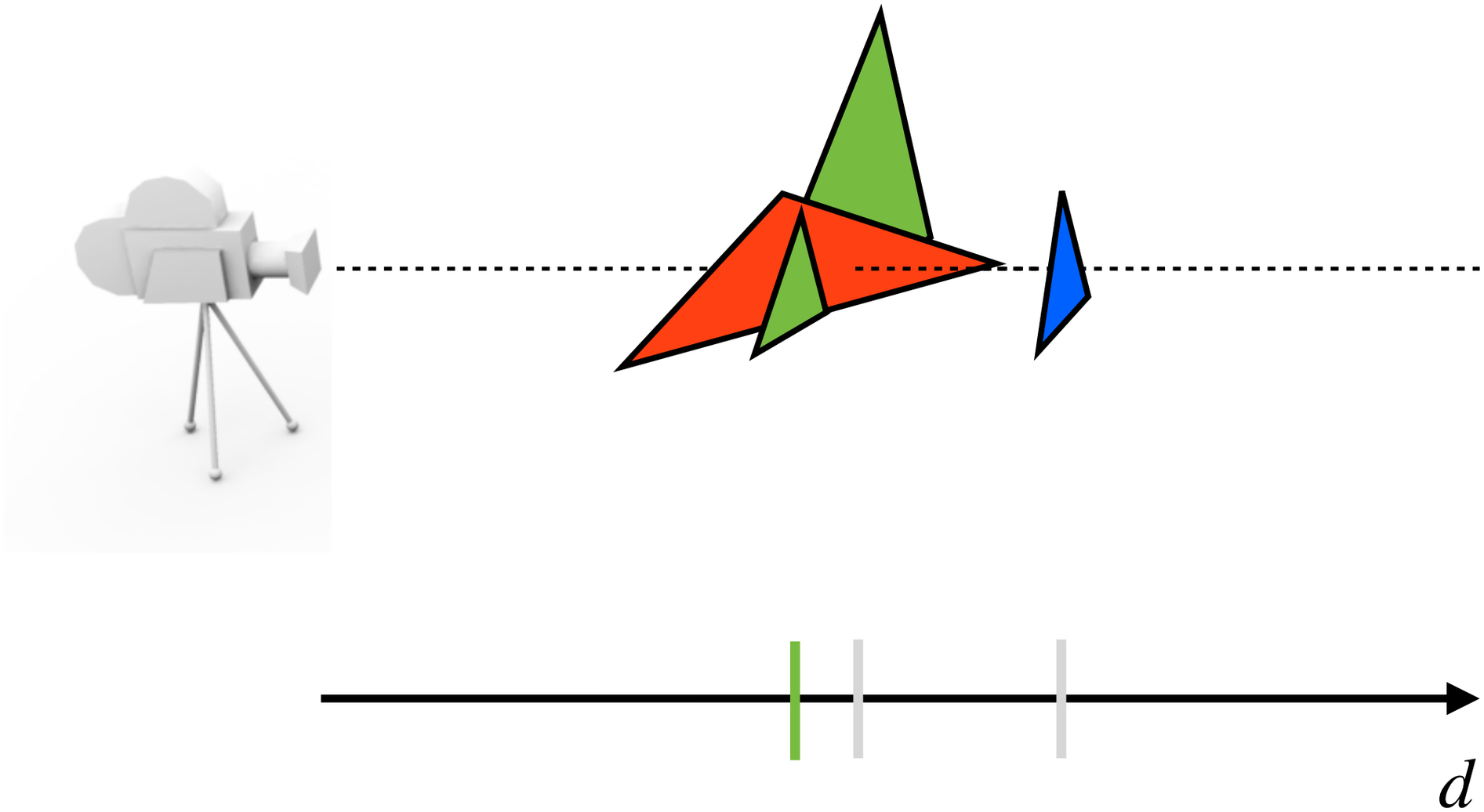


Overlapping Primitives

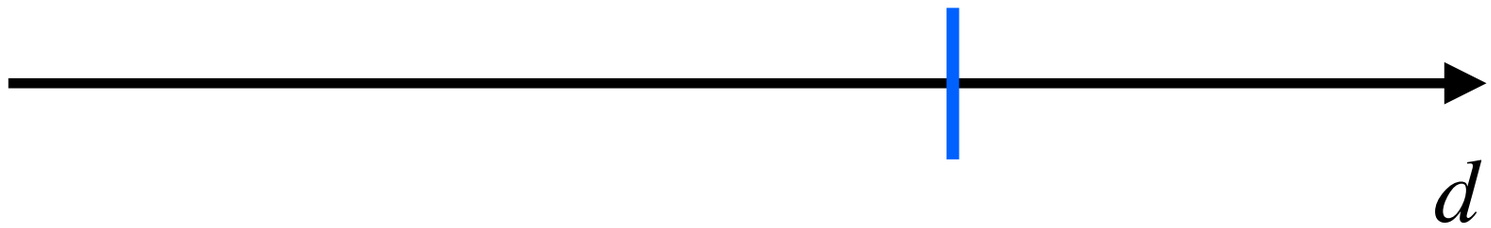




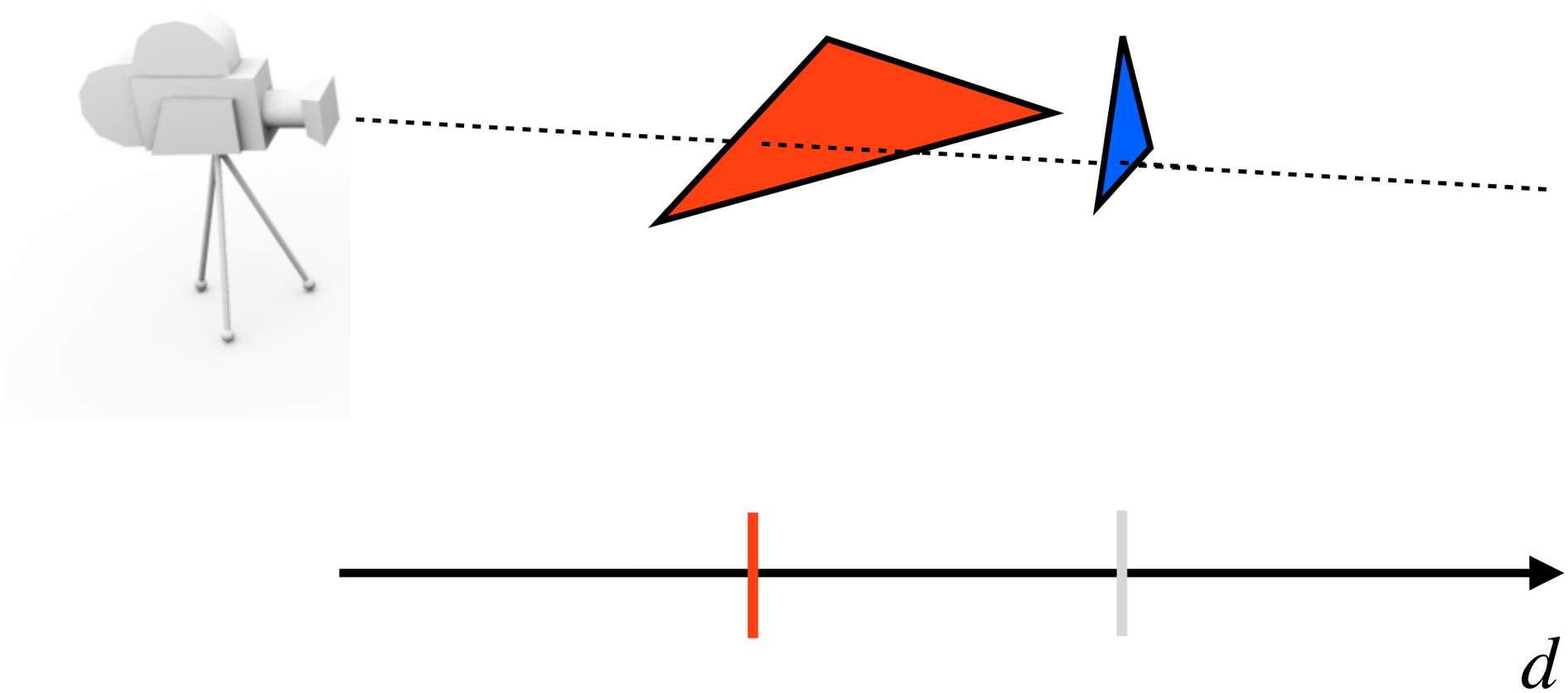
Overlapping Primitives



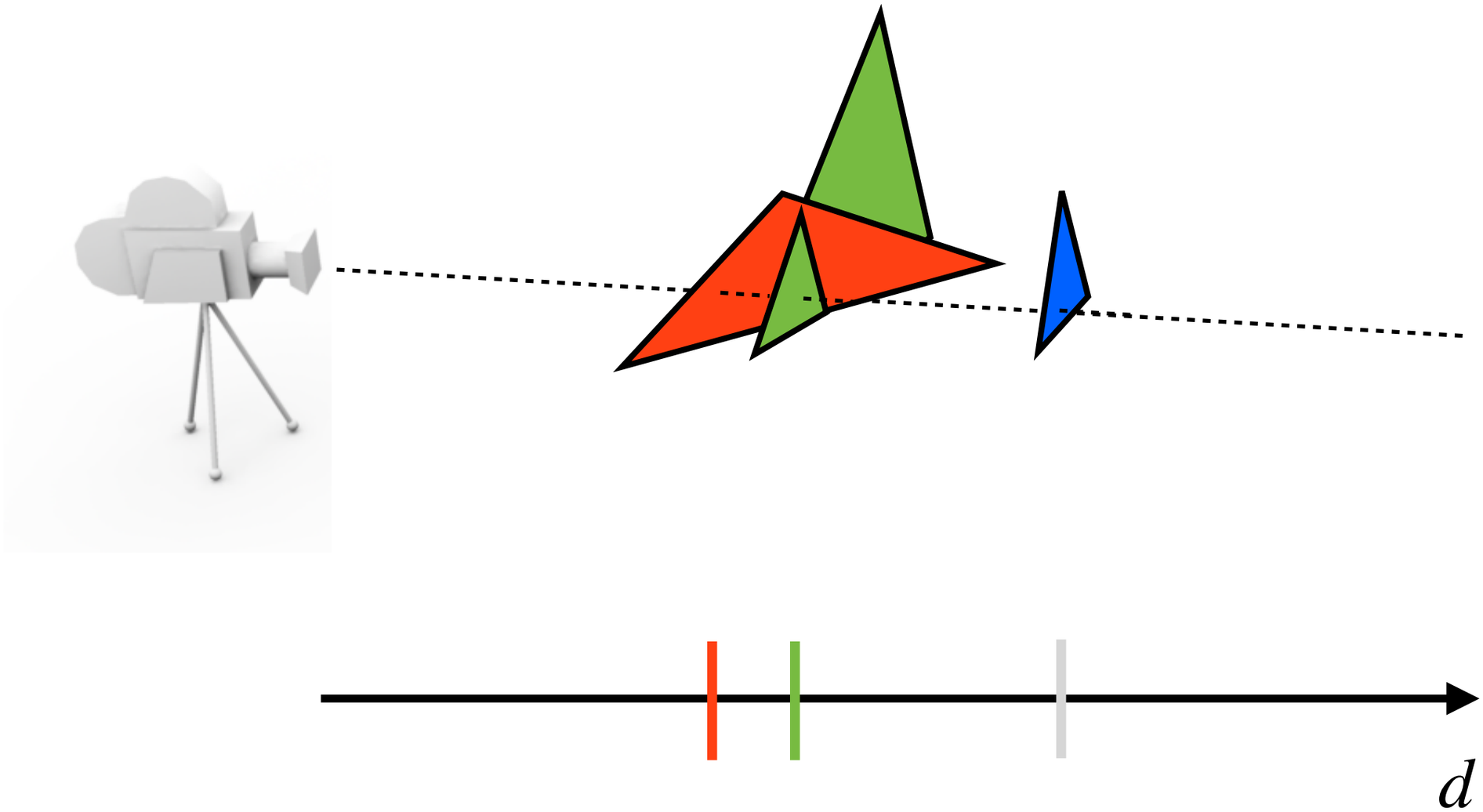
Overlapping Primitives



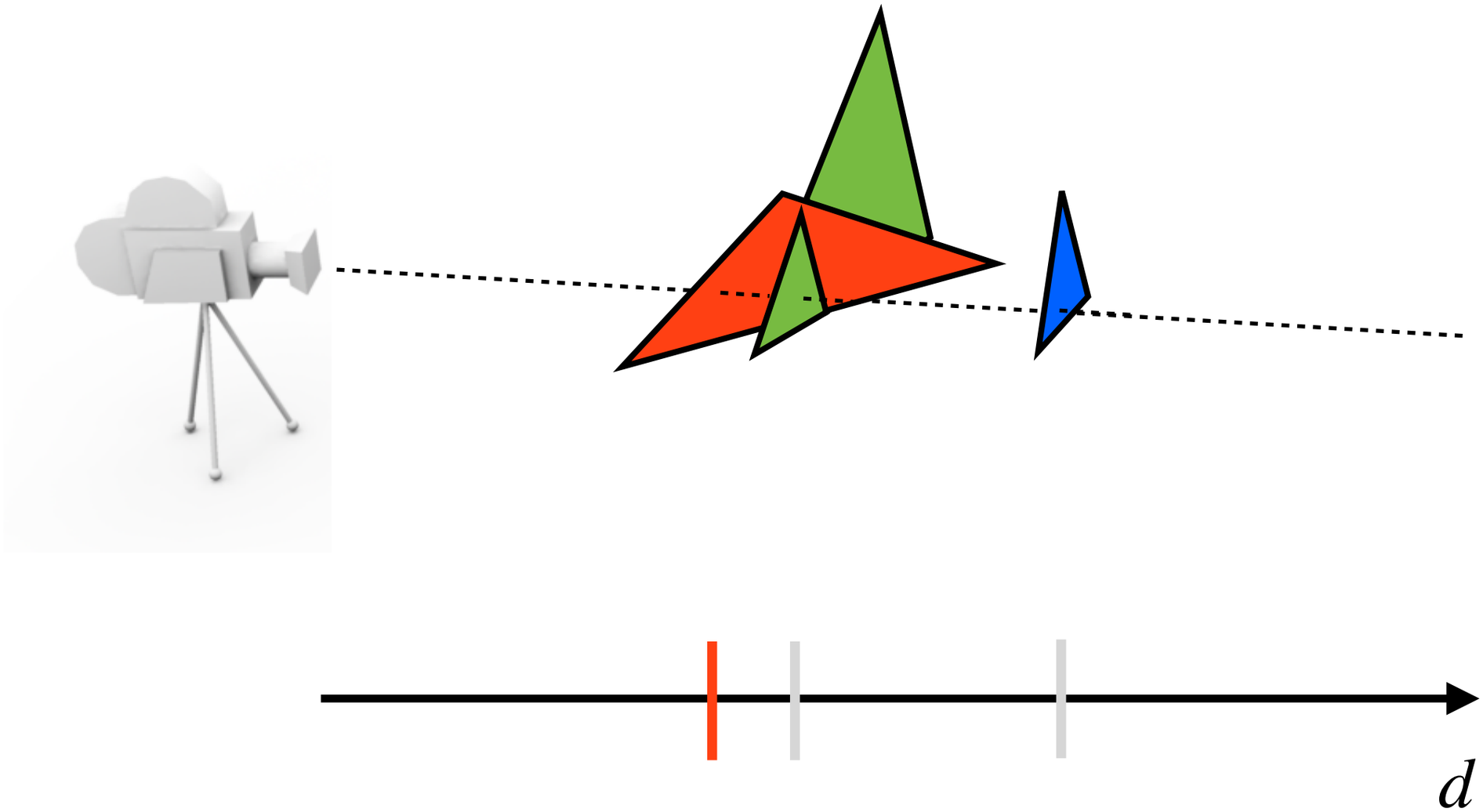
Overlapping Primitives



Overlapping Primitives



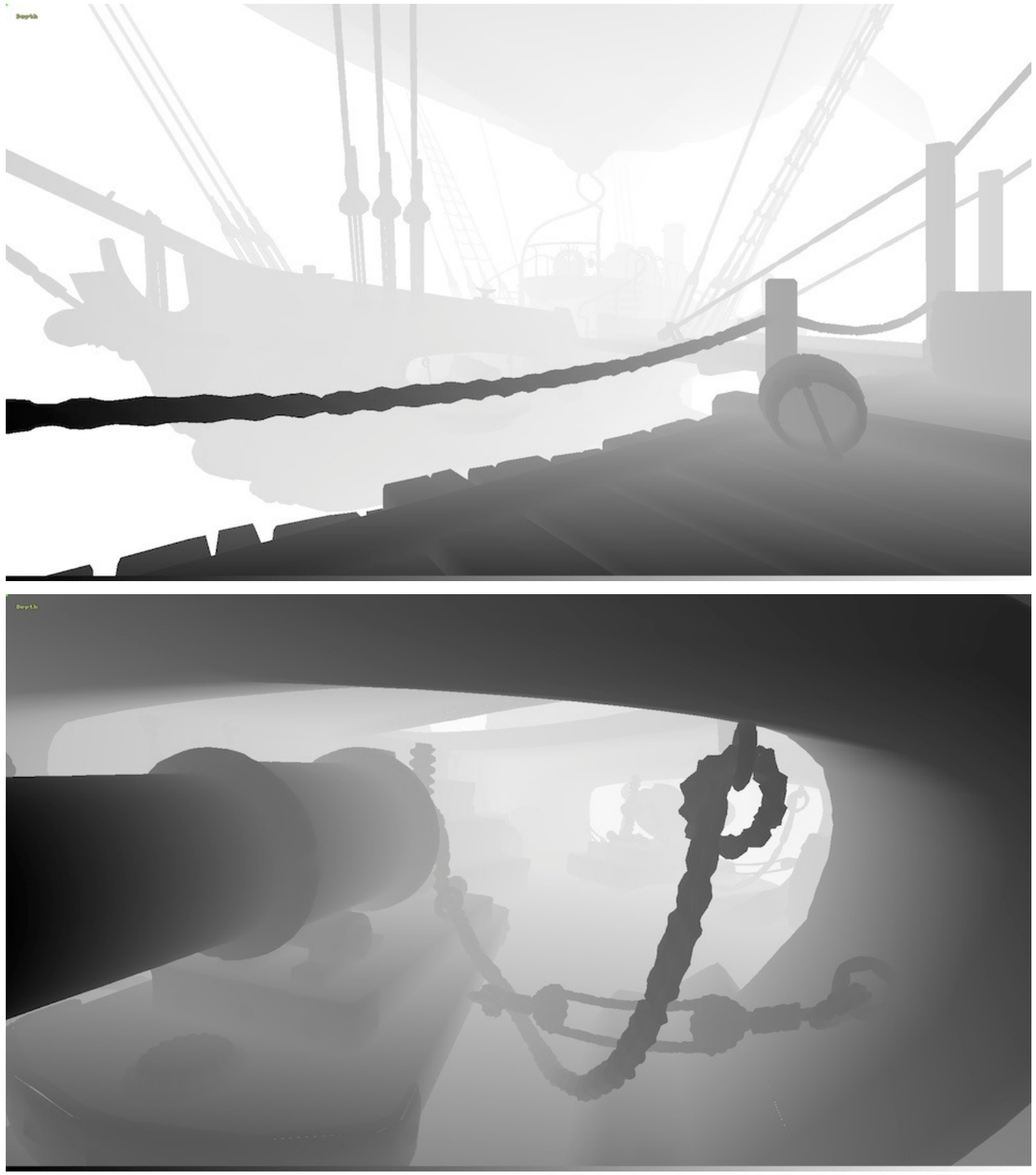
Overlapping Primitives



Depth Buffering

- Correct visibility regardless of in which order the triangles are rendered!
- Done under the hood by the graphics hardware
  - Depth values are commonly stored as $d = z/w$ (after projection matrix and perspective divide, i.e., in Normalized Device Coordinates)
  - z/w can be linearly interpolated in screen space (called **perspective-correct** interpolation).

Depth Buffer Examples



Heaven 2 DX11 Benchmark © Unigine

Graphics Pipeline Summary

- Application setup
  - Geometry, shaders and transform matrices
- For each triangle:
  - Apply transforms in vertex shader (**MVP**)
  - Project on screen (divide by w)
  - Rasterize: Evaluate edge equations at pixel center
  - For each covered pixel:
    - Depth buffer test to check if closest object
    - If visible: run pixel shader, store in color buffer

Ray Tracing

Rasterization vs Ray Tracing

Rasterization

```
For each triangle
   For each pixel
      Is pixel in triangle?
         Store closest pixel
```

Ray Tracing

```
For each pixel
   For each triangle
      Does ray hit triangle?
         Store closest pixel
```

Order of loops is switched!

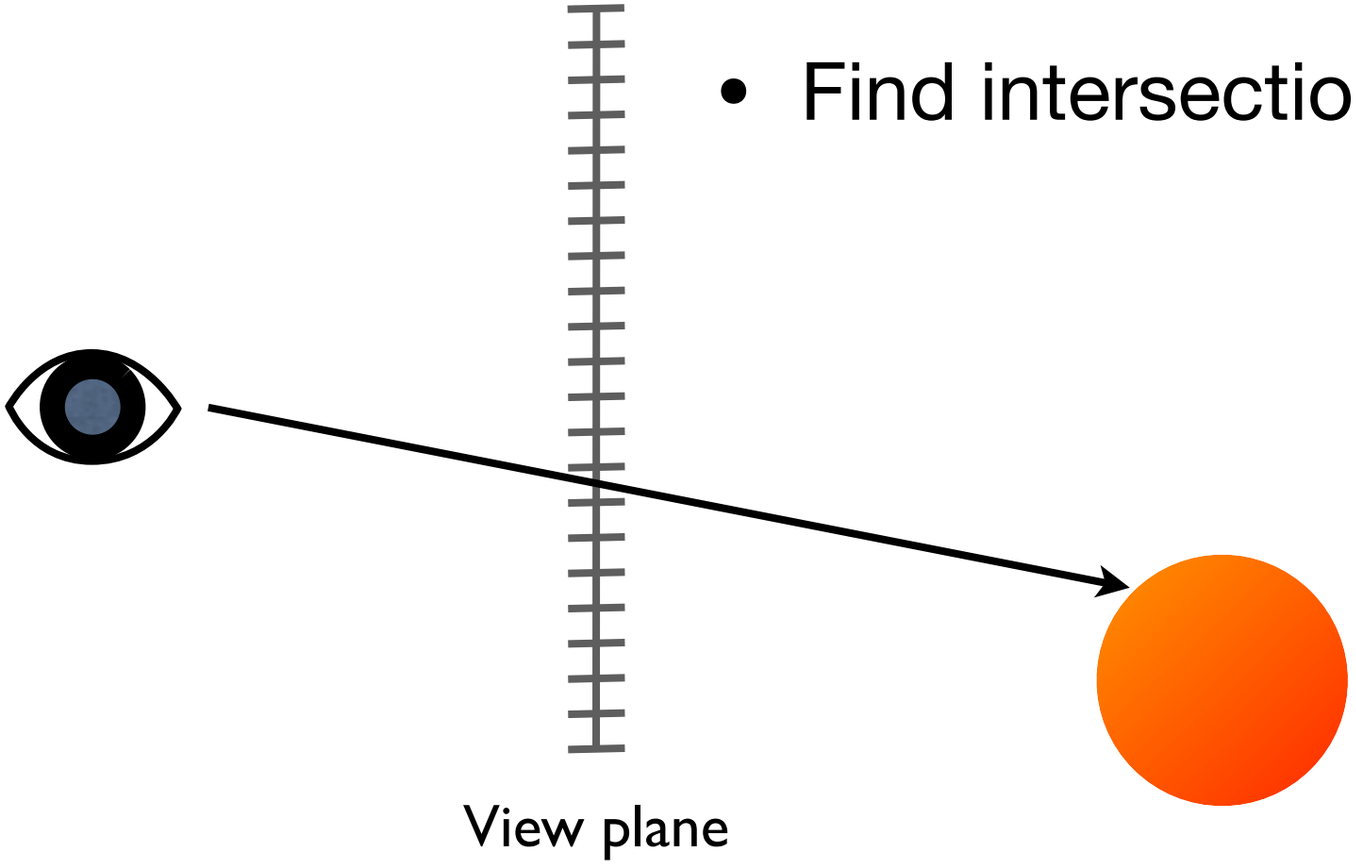
Rasterization vs Ray Tracing

- Rasterization - does one triangle at a time
  - Needs color and depth for screen pixels
  - + *Streams* over the triangles - good for parallelism and caching
  - - Shadows, reflections and transparency are not simple

- Ray Tracing - does one pixel at a time
  - + Easy recursion, shadows, reflection and transparency
  - - Could hit any triangle in scene
    - Whole scene needs to be accessible
    - Ray Tracing requires more memory

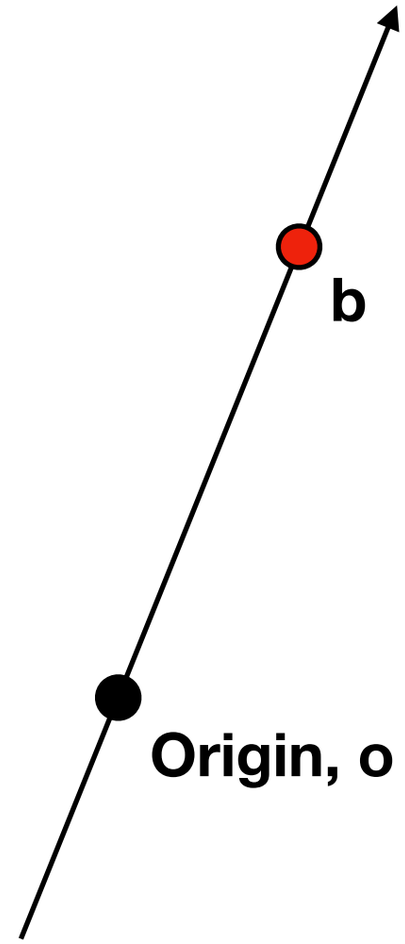
Ray Tracing

- Construct a line (ray) from the eye
  - through the view plane and into the scene
- Find intersection with objects

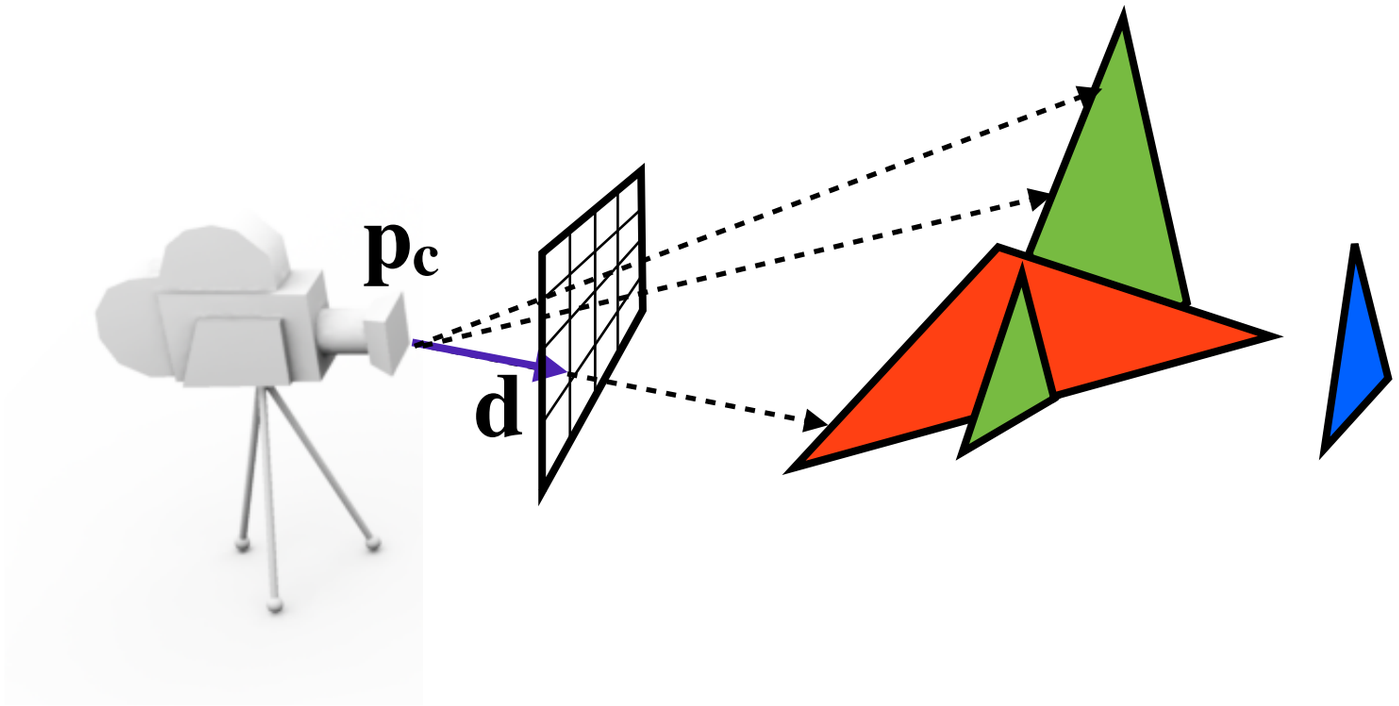


Ray definition

- Origin (**o**) and direction (**d**)
- Position on ray is represented using a Parameter, *t*
- **c** = (1-*t*)**o** + *t***b**
  - **c** = **o** + *t*(**b** - **o**) = **o** + *t***d**
- **c** is a point along the ray



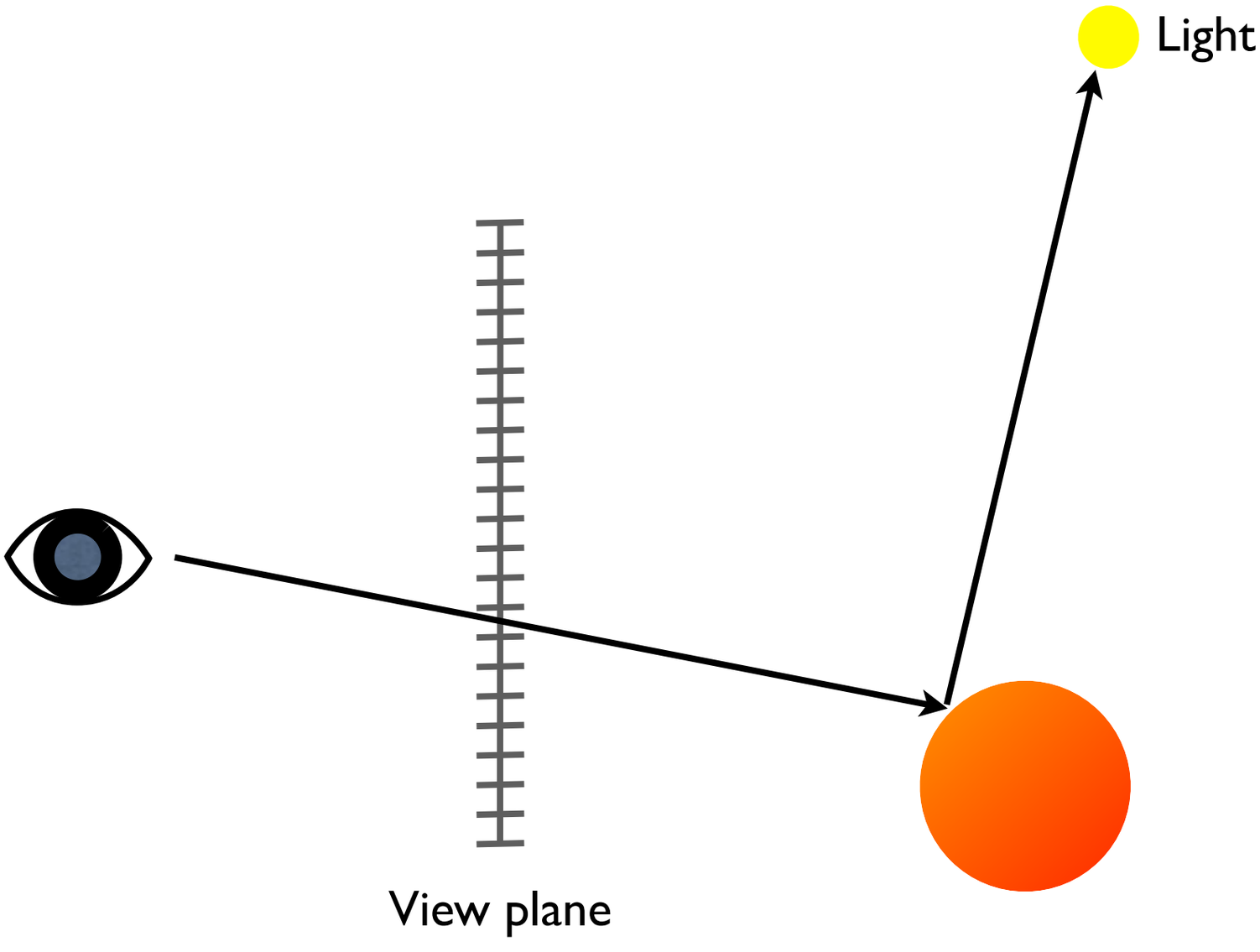
Trace Rays from Camera



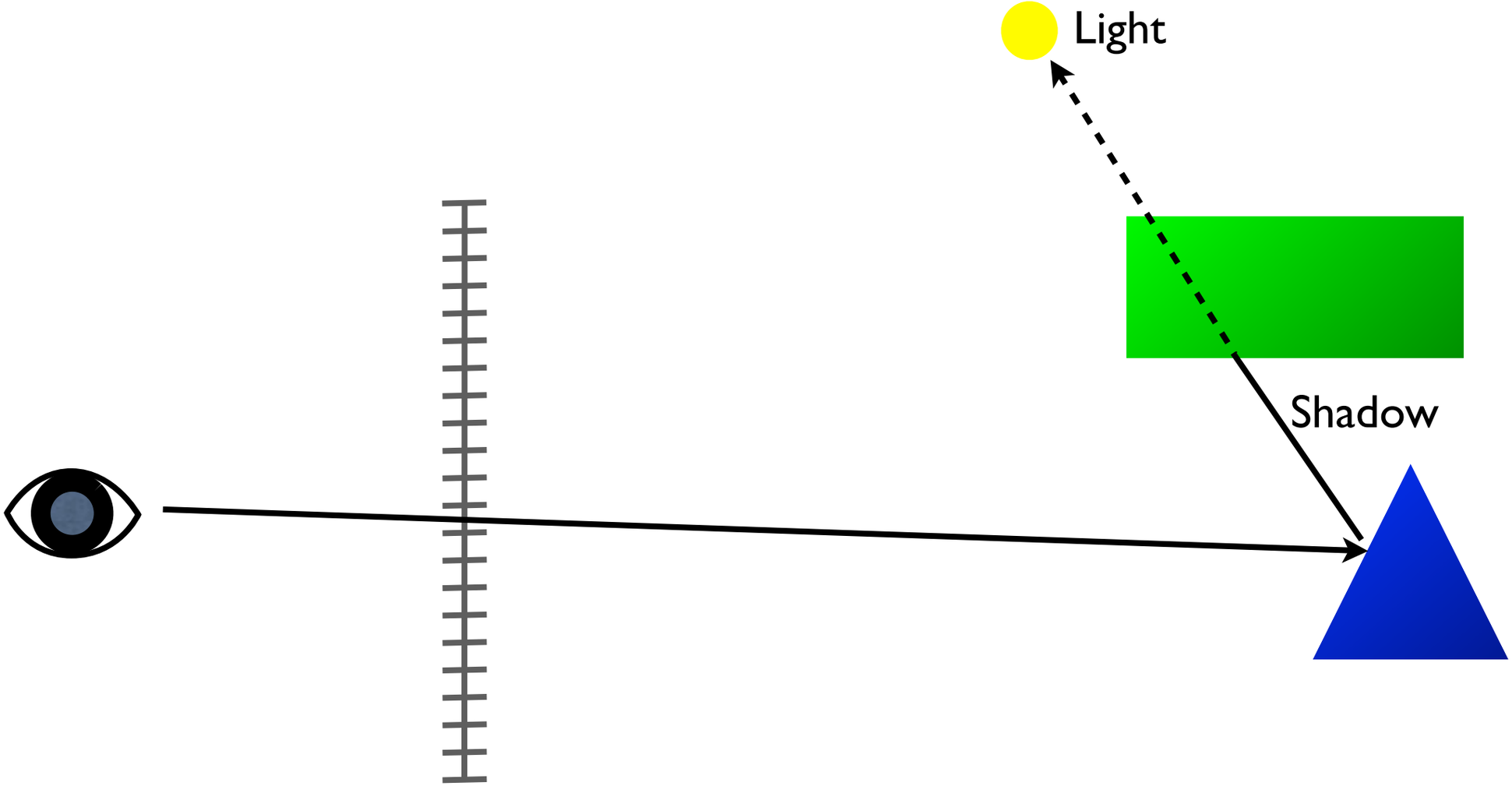
- Compute a ray starting from $\mathbf{p}_c$, going through a pixel on the image plane
  - Find intersection with ray : $\mathbf{r}(t) = \mathbf{p}_c + \mathbf{d}t$ and triangle, save closest hit (smallest t)

Shadows

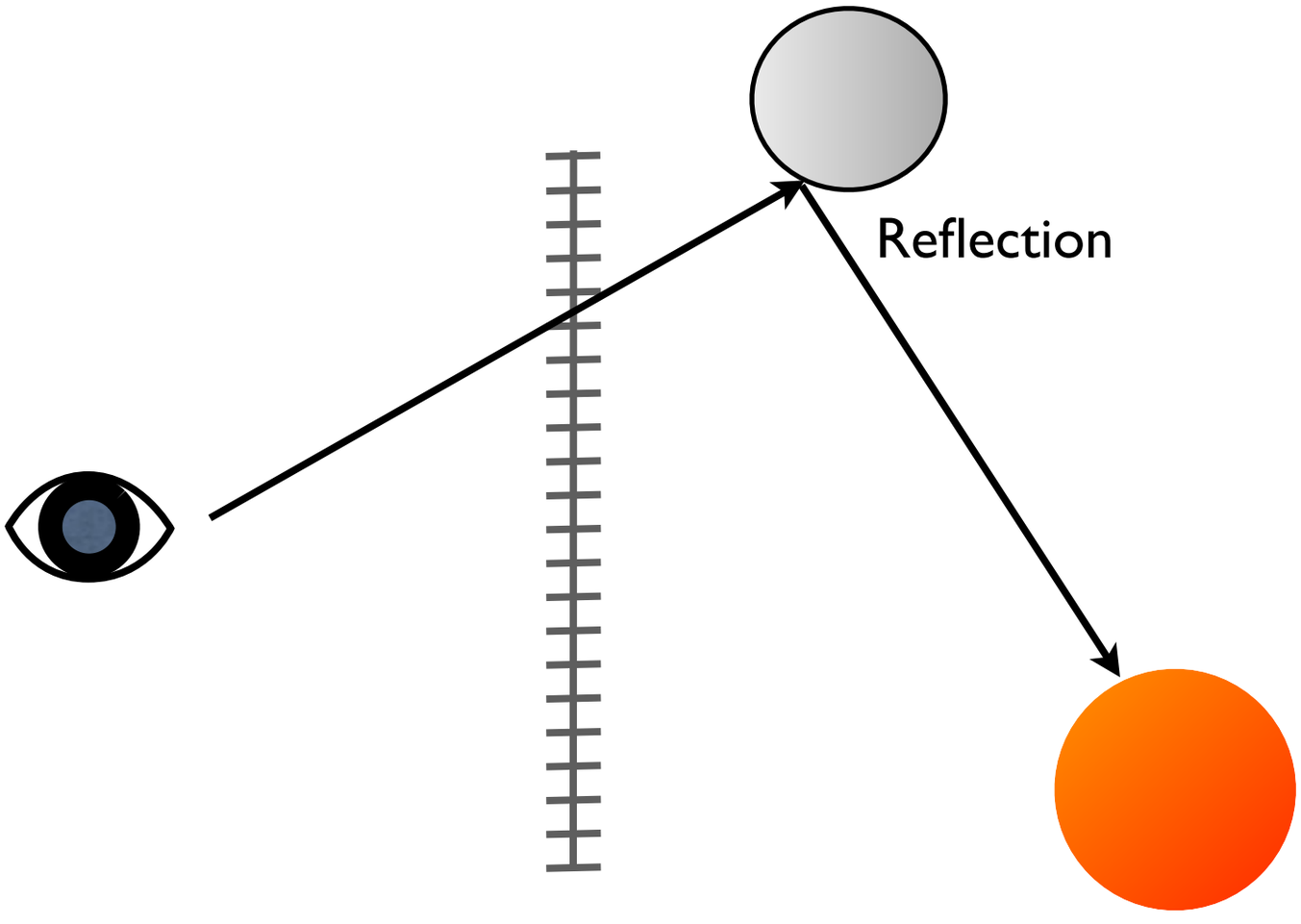
- Trace a ray from intersection to every light to check for shadows and shade surface



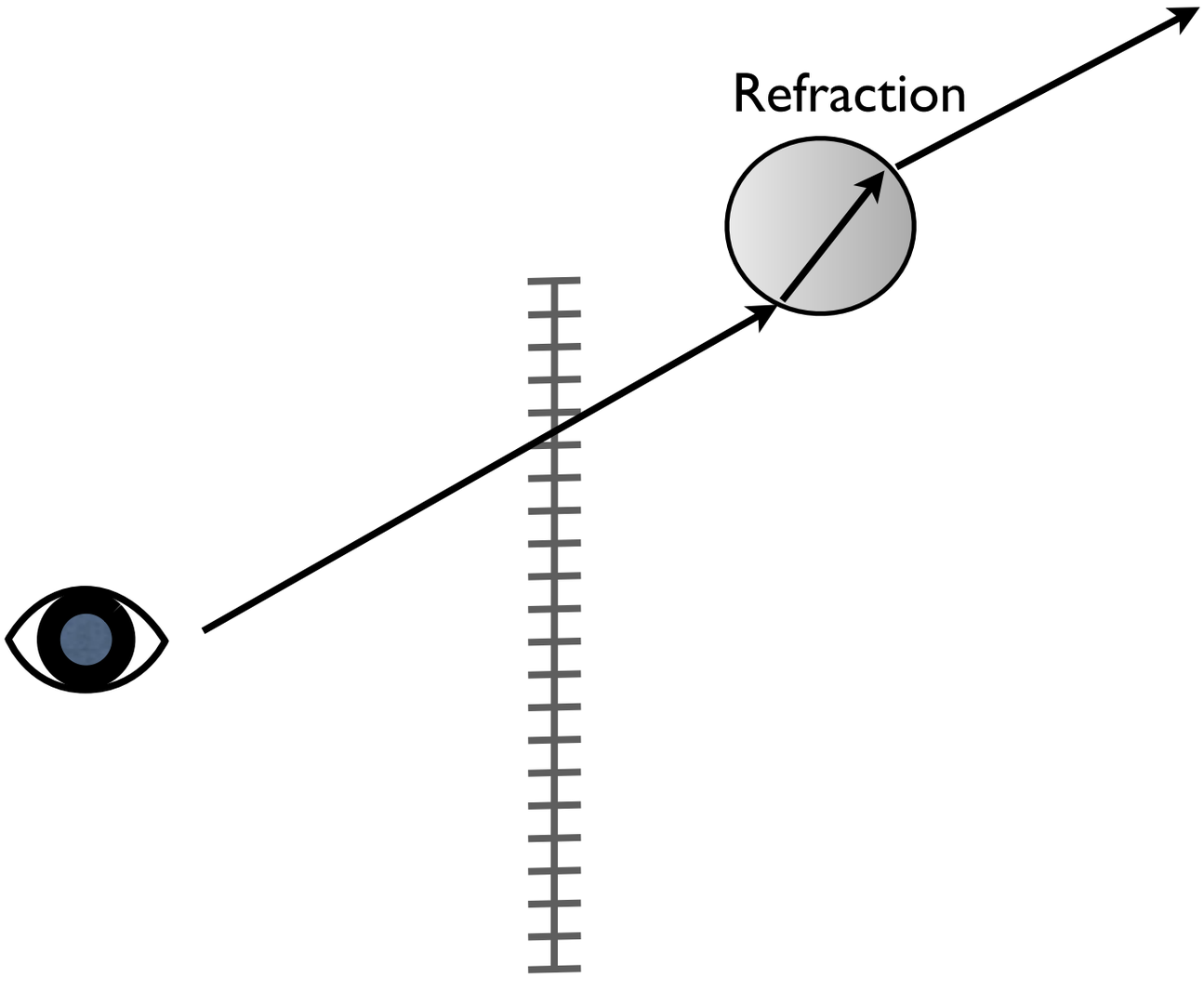
Shadows



Reflection

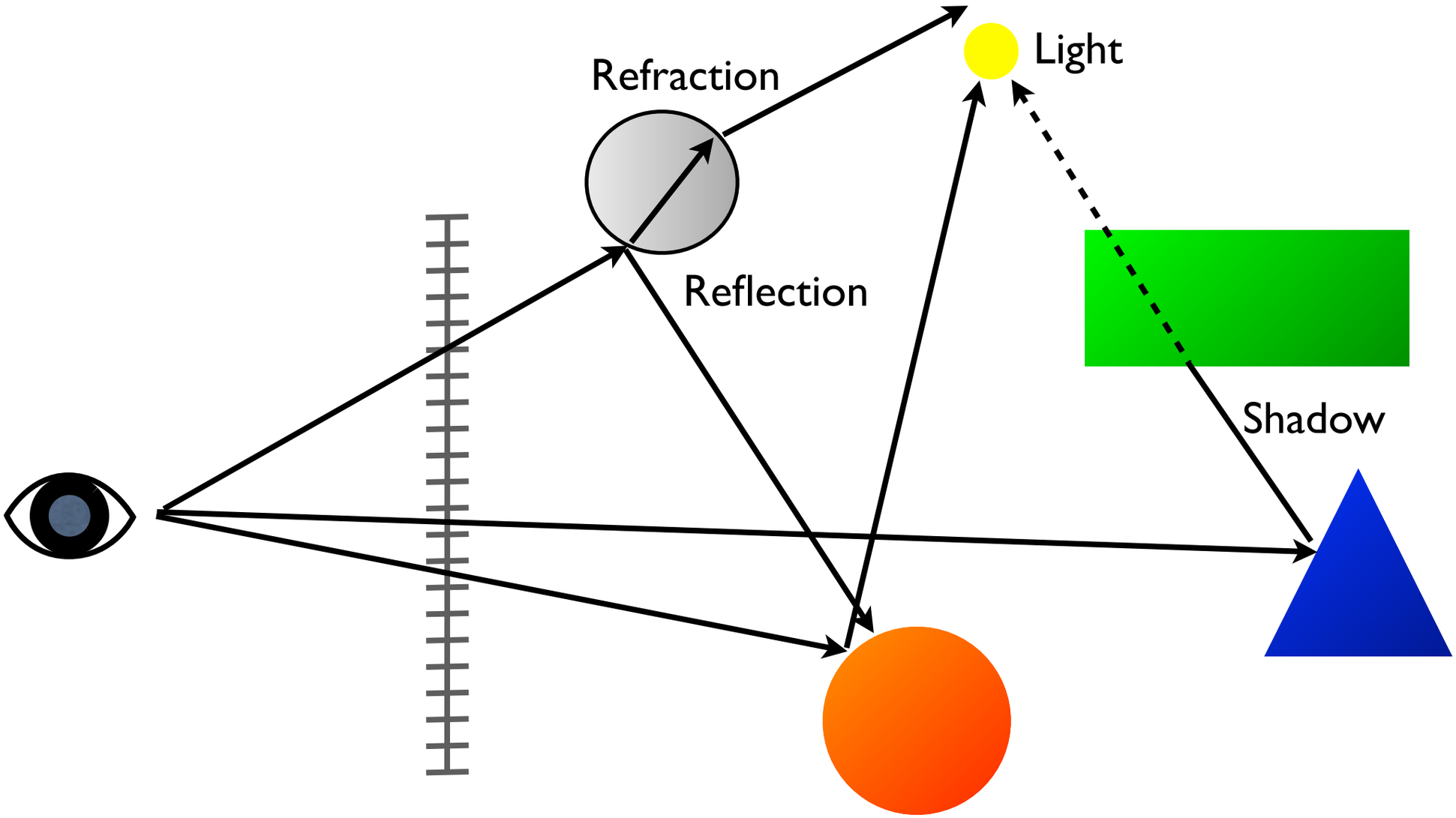


Refraction

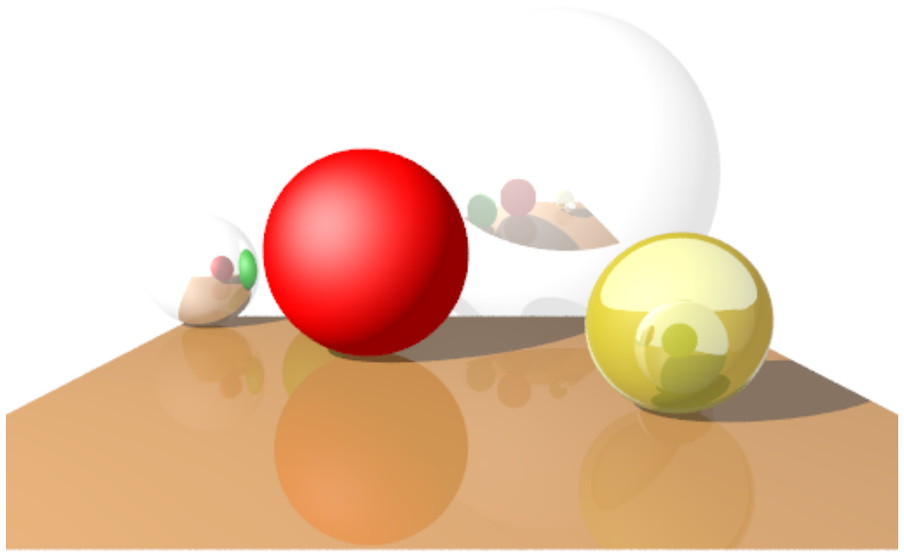


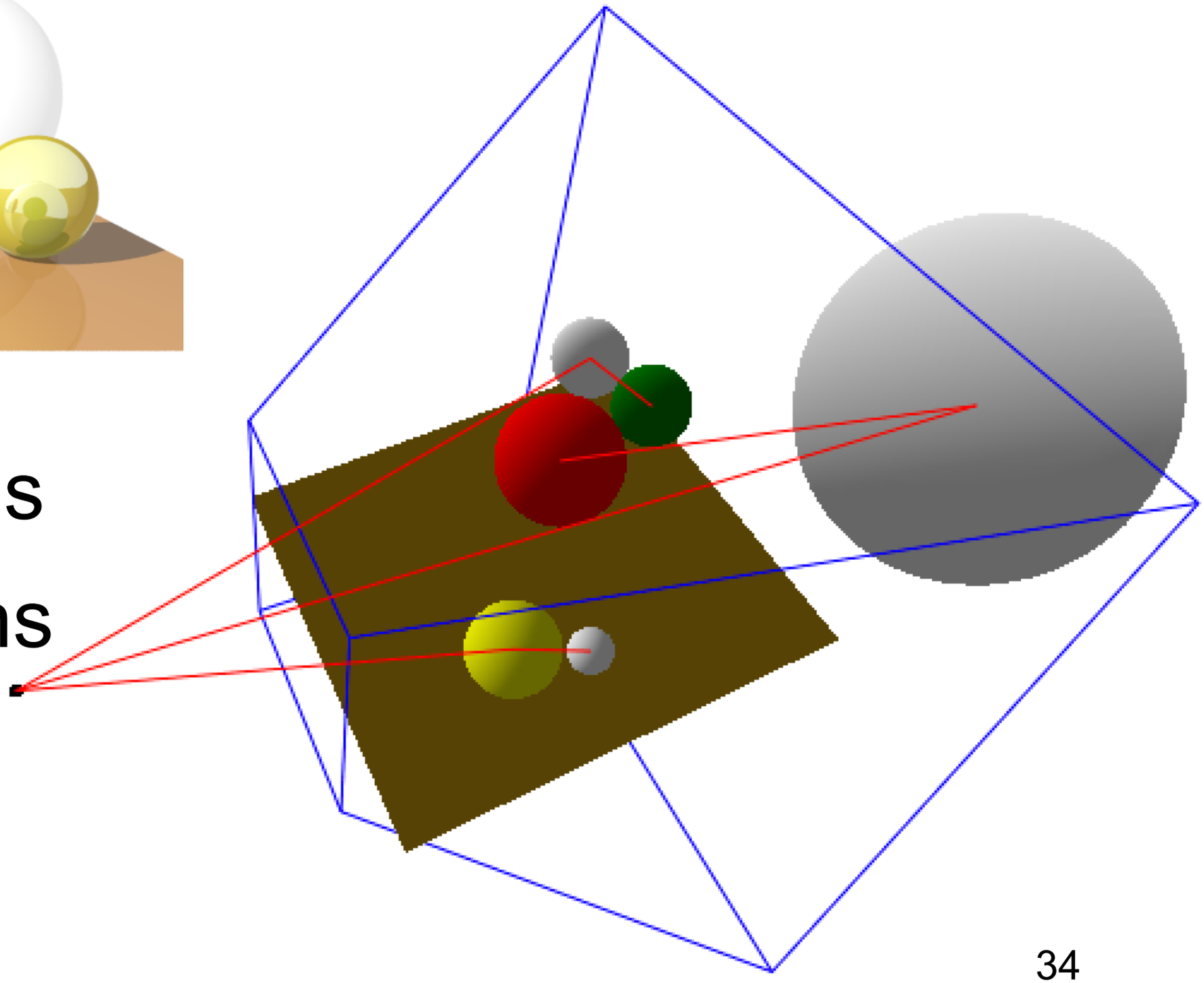
Recursive Ray Tracing

- At each intersection, trace shadow, reflection, and refraction rays



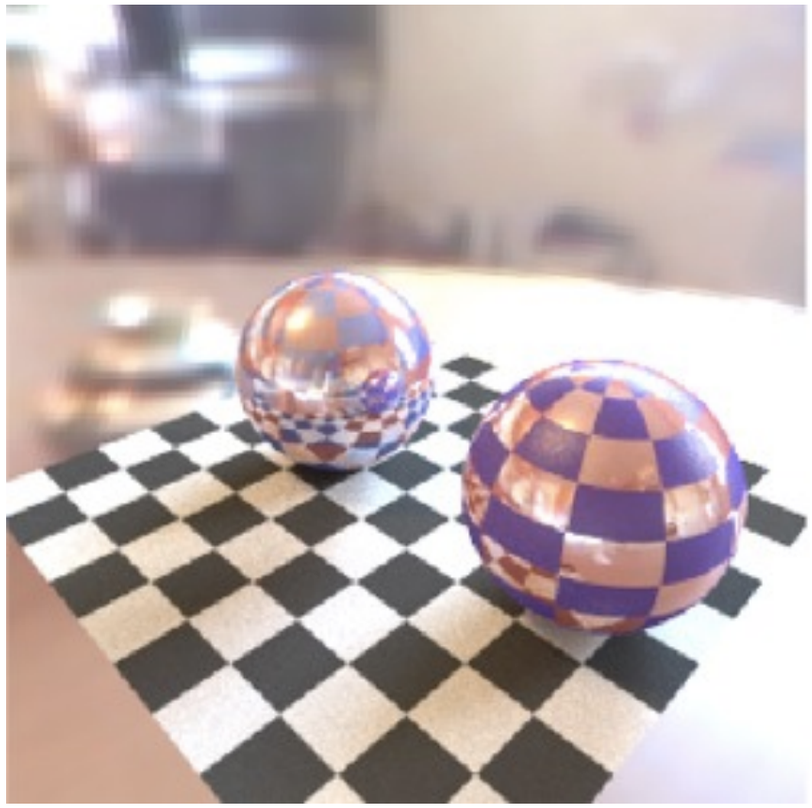
Recursive Ray Tracing





- Reflections
- Refractions
- Shadows

Reflections and Refractions

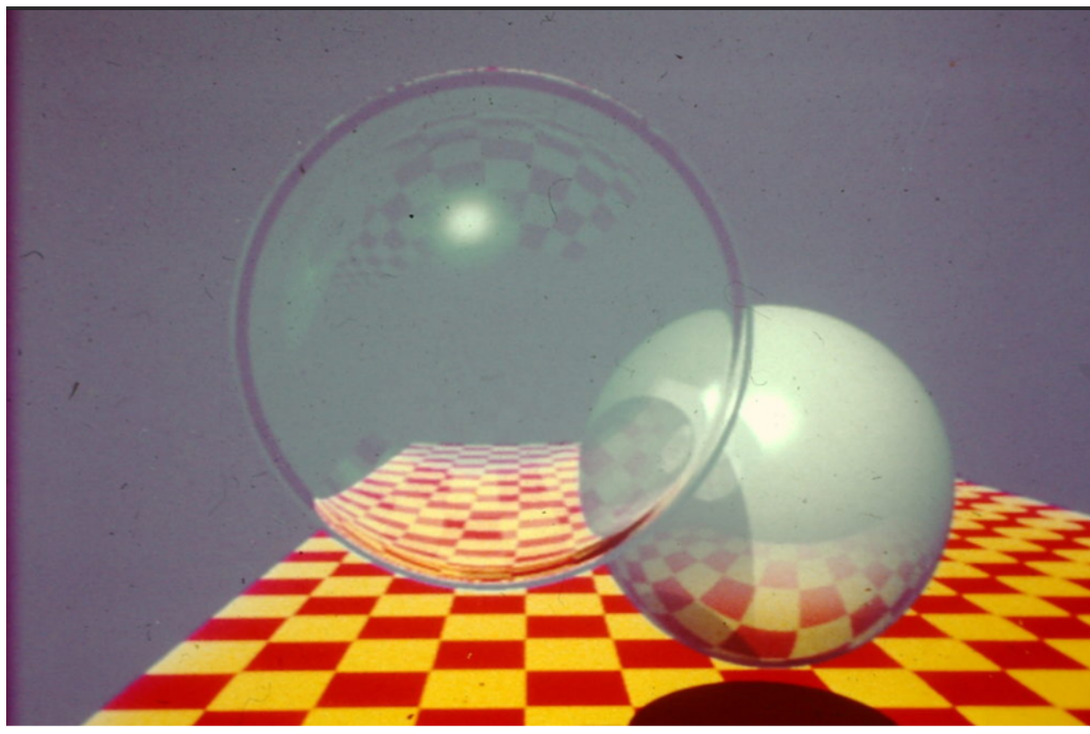


Whitted Ray Tracing

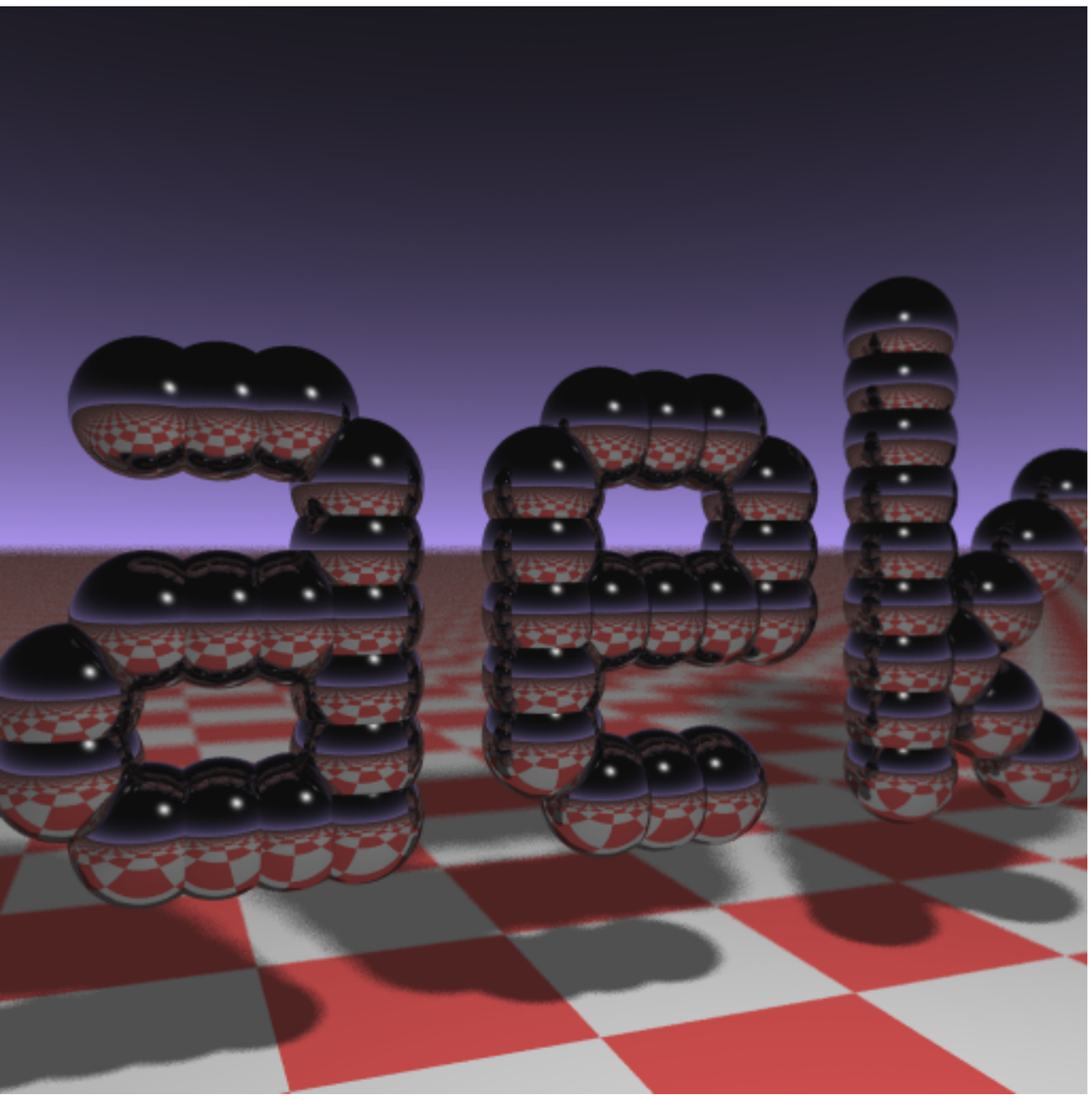
- 1980, “An Improved Illumination Model for Shaded Display”, Turner Whitted, CACM

- Simple surface model, perfect reflection

- Shadow rays trace to point light sources

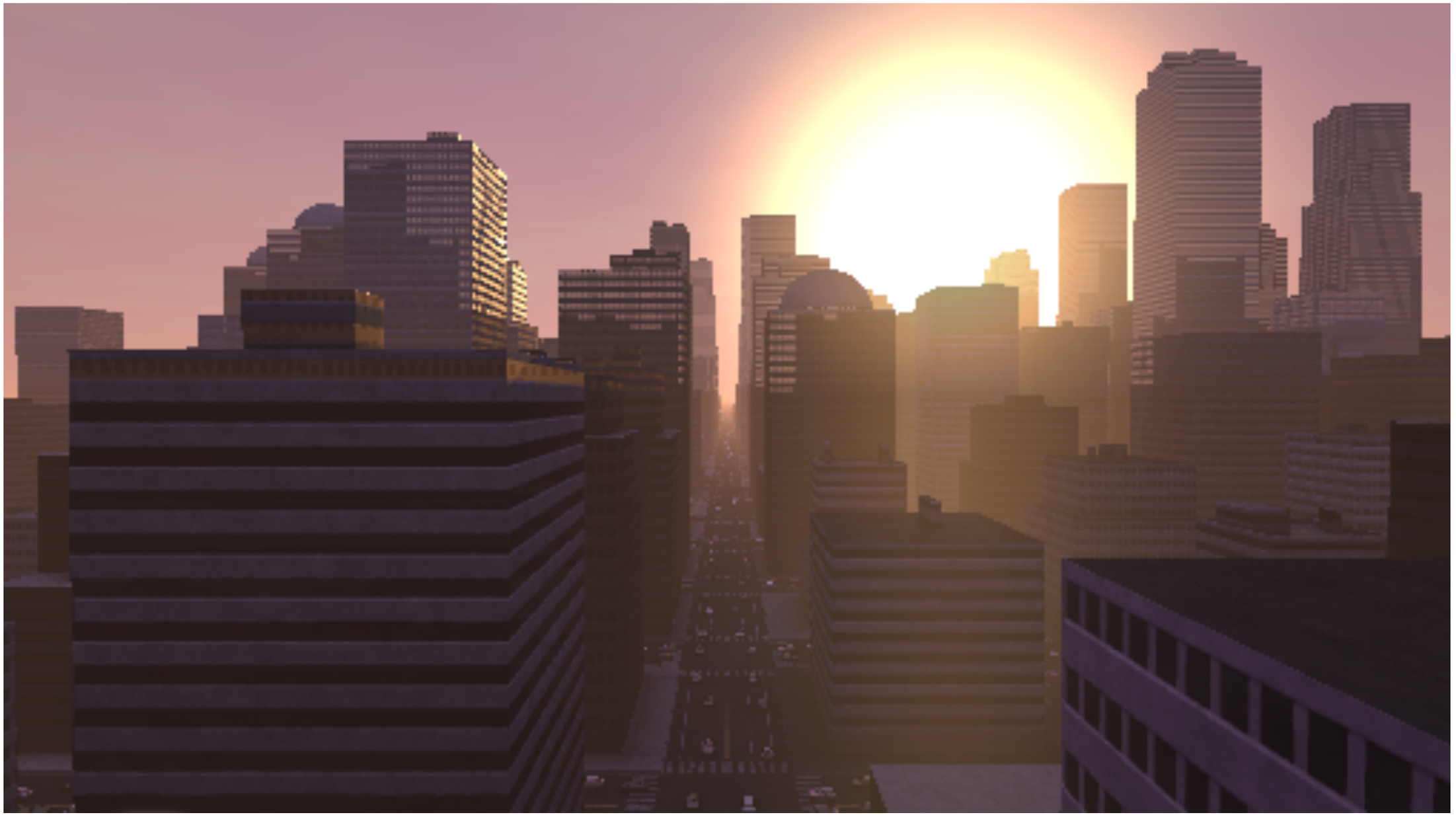


Card Ray Tracer



```
#include <stdlib.h>   // card > aek.ppm
#include <stdio.h>
#include <math.h>
typedef int i;typedef float f;struct v{f x,y,z;v
operator+(v r){return v(x+r.x,y+r.y,z+r.z);}v
operator*(f r){return v(x*r,y*r,z*r);}f operator%
(v r){return x*r.x+y*r.y+z*r.z;}v(){}v operator^(v
r){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.y-y*r.x);}
v(f a,f b,f c){x=a;y=b;z=c;}v operator!()
{return*this*(1/sqrt(*this%*this));}};i
G[]={247570,280596,280600,249748,18578,18577,23118
4,16,16};f R(){return(f)rand()/RAND_MAX;}i T(v o,v
d,f&t,v&n){t=1e9;i m=0;f p=-o.z/
d.z;if(.01<p)t=p,n=v(0,0,1),m=1;for(i
k=19;k--;)for(i j=9;j--;)if(G[j]&1<<k){v p=o+v(-
k,0,-j-4);f b=p%d,c=p%p-1,q=b*b-c;if(q>0){f s=-b-
sqrt(q);if(s<t&&s>.01)t=s,n=!(p+d*t),m=2;}}return
m;}v S(v o,v d){f t;v n;i m=T(o,d,t,n);if(!
m)return v(.7,.6,1)*pow(1-d.z,4);v h=o+d*t,l=!
(v(9+R(),9+R(),16)+h*-1),r=d+n*(n%d*-2);f
b=l%n;if(b<0||T(h,l,t,n))b=0;f
p=pow(l%r*(b>0),99);if(m&1){h=h*.2;return((i)
(ceil(h.x)+ceil(h.y))&1?
v(3,1,1):v(3,3,3))*(b*.2+.1);}return v(p,p,p)
+S(h,r)*.5;}i main(){printf("P6 512 512 255 ");v
g=!v(-6,-16,0),a=!(v(0,0,1)^g)*.002,b=!
(g^a)*.002,c=(a+b)*-256+g;for(i y=512;y--;)for(i
x=512;x--;){v p(13,13,13);for(i r=64;r--;){v
t=a*(R()-.5)*99+b*(R()-.5)*99;p=S(v(17,16,8)+t,!
(t*-1+(a*(R()+x)+b*(y+R())+c)*16))*3.5+p;}
printf("%c%c%c",(i)p.x,(i)p.y,(i)p.z);}}
```

courtesy Andrew Kensler from http://eastfarthing.com/blog/2016-01-12-card/



Skyline

procedural, raymarching

https://www.shadertoy.com/view/XtsSWs

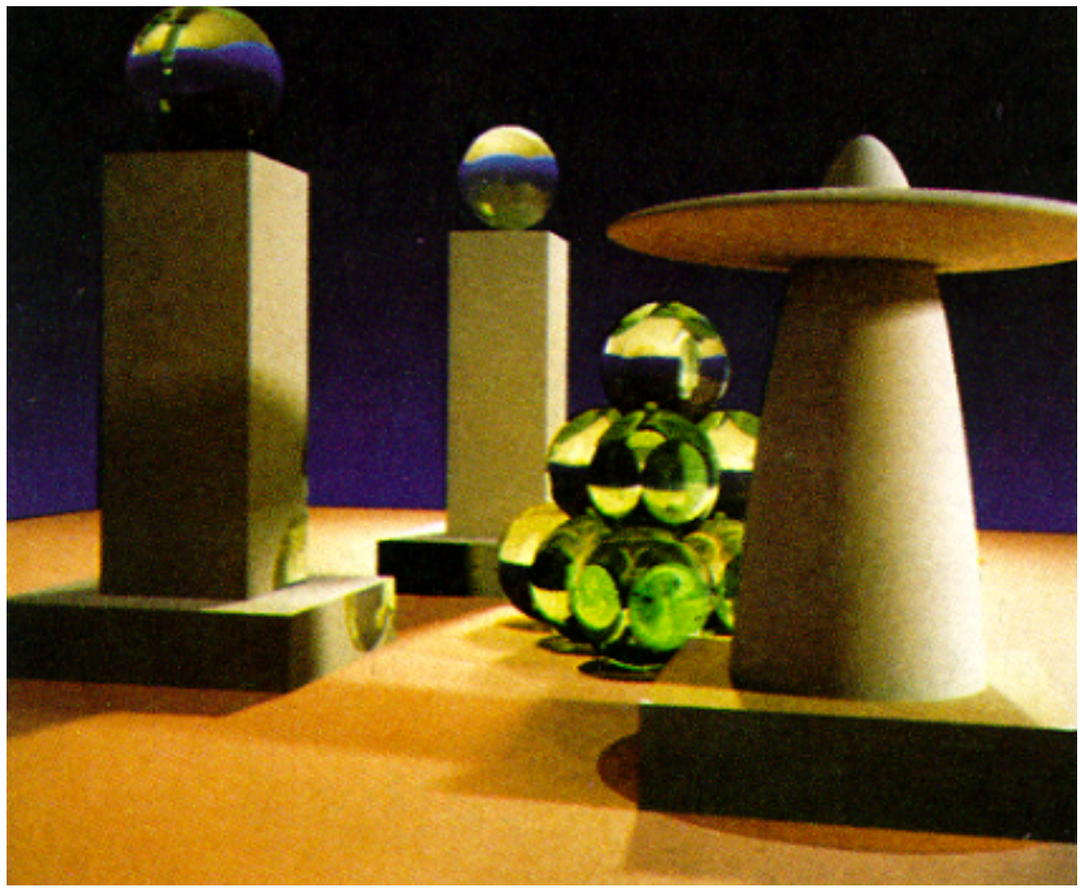
The Rendering Equation

Beyond the Phong shading model

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

Path Tracing
Kajiya 1986

- "The Rendering Equation"
- Diffuse inter-reflection
- Fully random on diffuse surfaces



PBRT - Path Tracing book and code that inspired movie Path Tracing

directions for making pictures using numbers
(explained using only the ten hundred words people use most often)

position on the stuff

direction towards the eye

light made by the stuff (sometimes because it is very hot)

the answer to how much light from an interesting direction that will keep going in the direction towards the eye, after hitting stuff at the position (this is easy for mirrors, not so easy for everything else)

the light that comes from an interesting direction towards the position on the stuff

how much the light becomes less bright because the stuff leans away from the interesting direction

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i \rightarrow \omega_o)\, L_i(x, \omega_i)\, (\omega_i \cdot n)\, d\omega_i$$

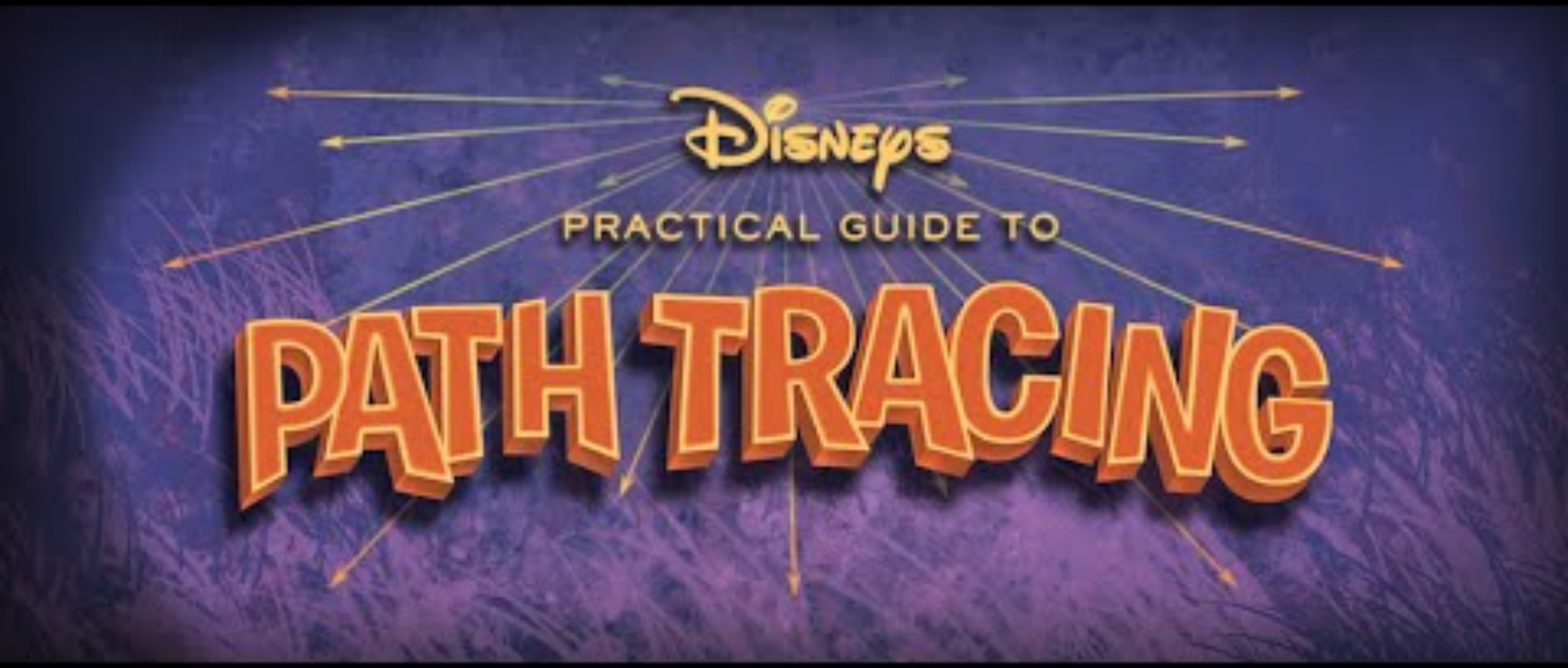
light that leaves the position on the stuff and reaches the eye

light can be added said a man who sat under a tree many years ago

for lots of interesting directions inside half a ball facing up from the stuff, add up all the answers in between

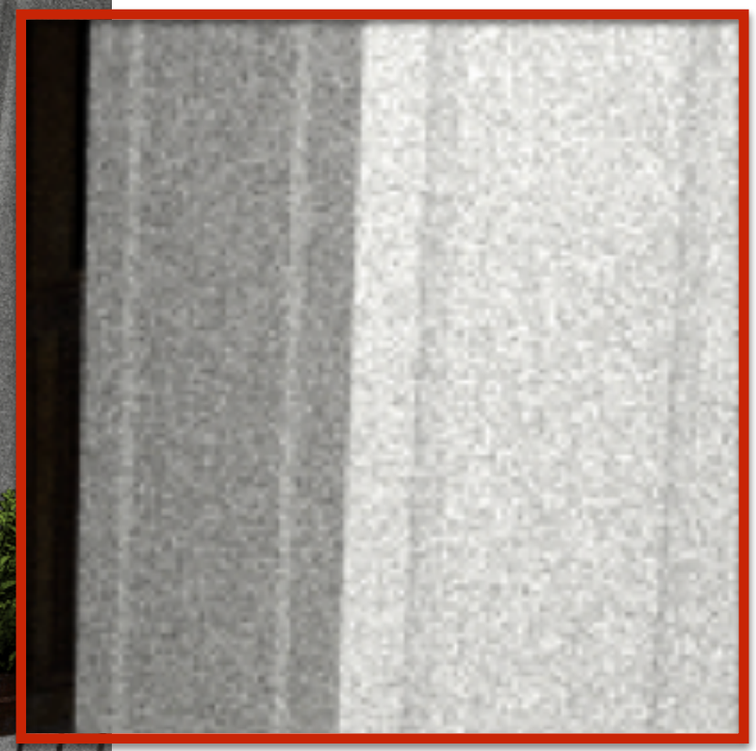
this idea came from http://xkcd.com/1133/

@levork



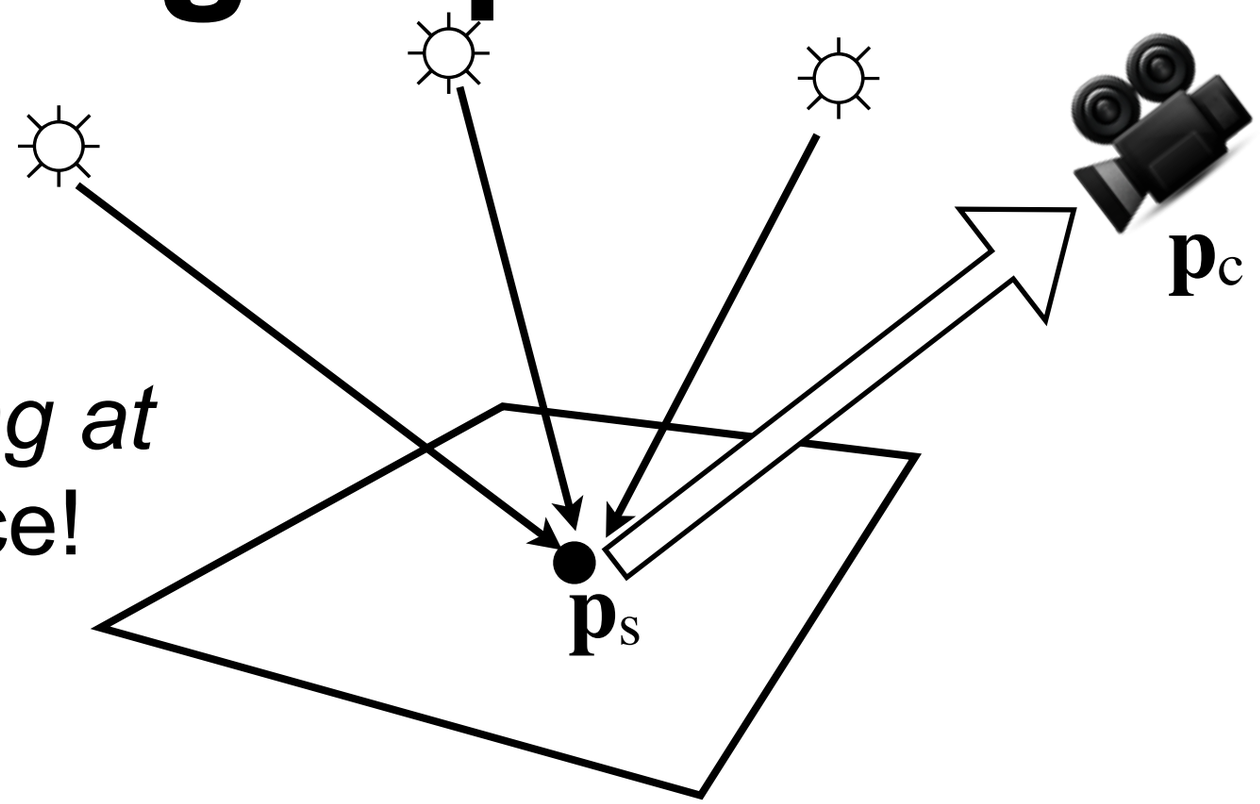
- https://youtu.be/frLwRLS_ZR0
- Disney's Hyperion Renderer https://www.disneyanimation.com/technology/innovations/hyperion

Noise



MD15

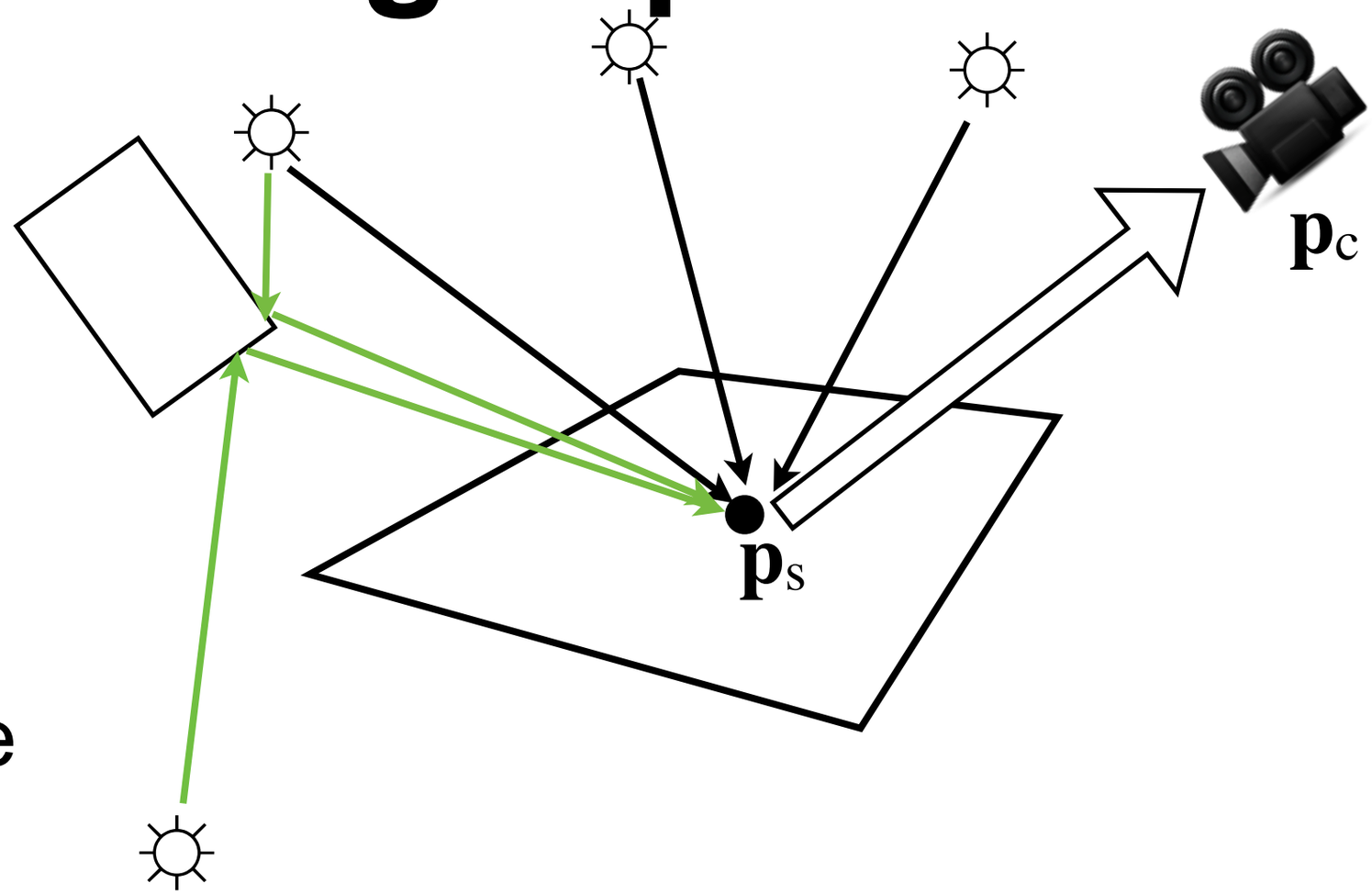
The Rendering Equation



Energy conservation:
Total amount of light *arriving at* and *leaving* $\mathbf{p}_S$ must balance!

The total intensity of light leaving point $\mathbf{p}_S$ is equal to the incoming light energy from all possible points plus the emission of light from $\mathbf{p}_S$ itself (if $\mathbf{p}_S$ is a light source)

The Rendering Equation



Also, light may have bounced several times before arriving at $\mathbf{p}_S$

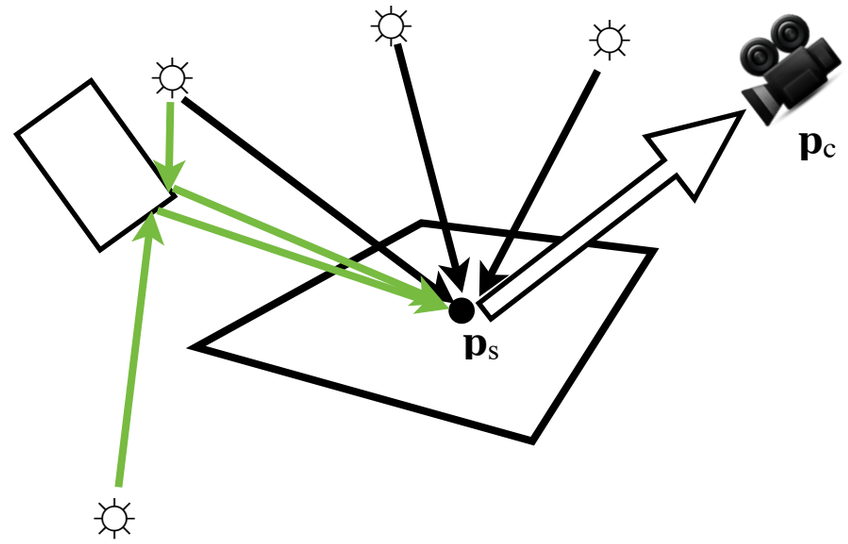
The Rendering Equation

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p}\right]$$

$I(\mathbf{p}_c, \mathbf{p}_s)$ Intensity of light from $\mathbf{p}_s$ towards $\mathbf{p}_c$

$v(\mathbf{p}_c, \mathbf{p}_s)$ Visibility between $\mathbf{p}_s$ and $\mathbf{p}_c$ (0 or 1)

$\epsilon(\mathbf{p}_c, \mathbf{p}_s)$ Self-emitted light from $\mathbf{p}_s$ toward $\mathbf{p}_c$



$\rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p})$ BRDF: Fraction of light from $\mathbf{p}$ towards $\mathbf{p}_s$ scattered in the direction of $\mathbf{p}_c$.
The BRDF describes the material properties at $\mathbf{p}_s$

$\int_S \ldots d\mathbf{p}$ Integral over all possible points in the scene

Recursive Equation

Unfold recursion in the rendering equation one step

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s) \left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p} \right]$$

Recursive Equation

Unfold recursion in the rendering equation one step

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p}\right]$$

$$\begin{aligned} I(\mathbf{p}_c, \mathbf{p}_s) \quad = \quad & v(\mathbf{p}_c, \mathbf{p}_s)[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p})[\\ & v(\mathbf{p}_s, \mathbf{p})\left[\epsilon(\mathbf{p}_s, \mathbf{p}) + \int_S \rho(\mathbf{p}_s, \mathbf{p}, \mathbf{p}') I(\mathbf{p}, \mathbf{p}') d\mathbf{p}'\right]] d\mathbf{p}] \end{aligned}$$

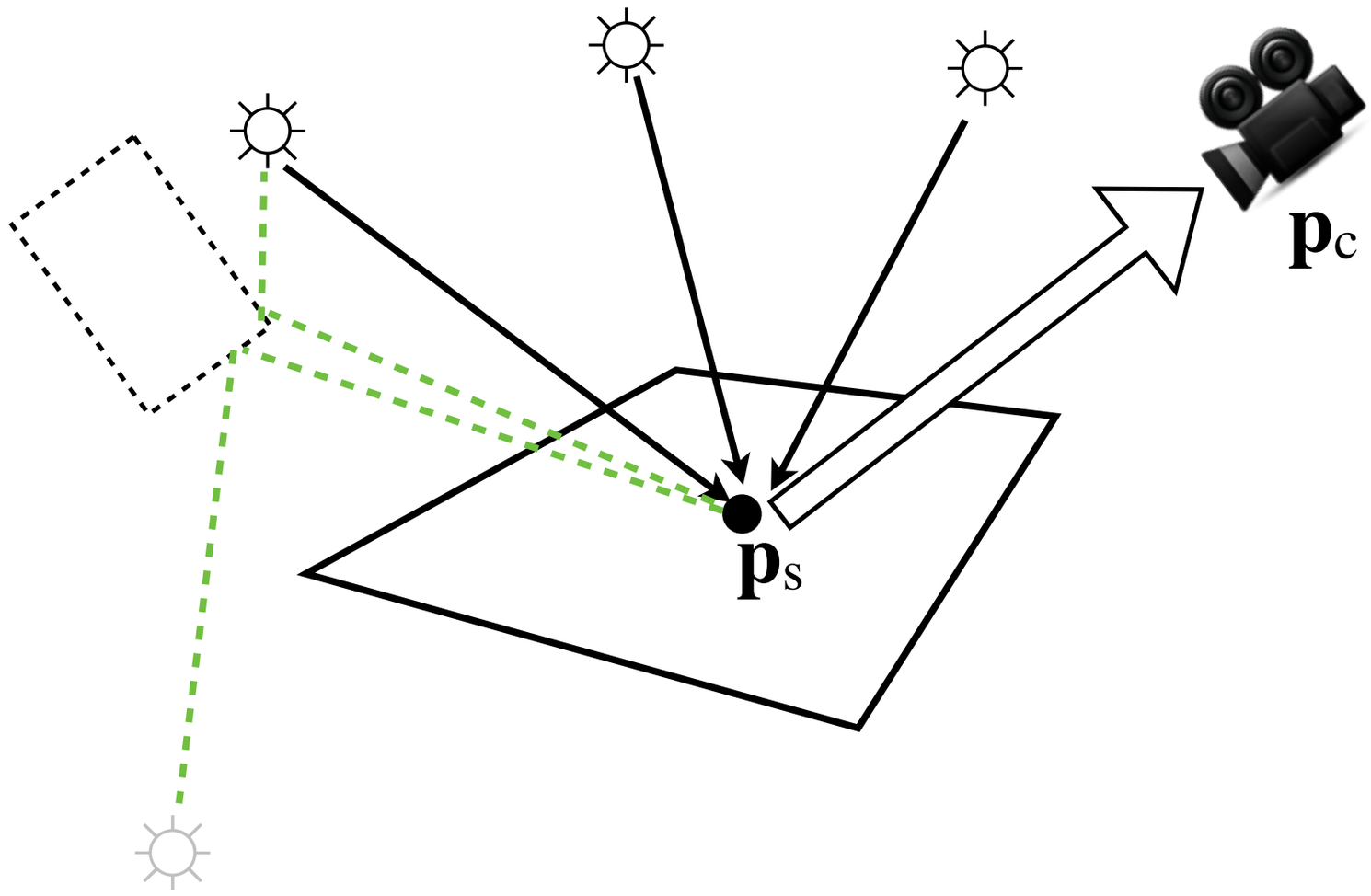
Recursive Equation

Unfold recursion in the rendering equation

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p}\right]$$

$$\begin{aligned} I(\mathbf{p}_c, \mathbf{p}_s) \quad = \quad & v(\mathbf{p}_c, \mathbf{p}_s)[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p})[\\ & v(\mathbf{p}_s, \mathbf{p})\left[\epsilon(\mathbf{p}_s, \mathbf{p}) + \int_S \rho(\mathbf{p}_s, \mathbf{p}, \mathbf{p}') I(\mathbf{p}, \mathbf{p}') d\mathbf{p}'\right]] d\mathbf{p}] \end{aligned}$$

Local Reflection Model



Disregard the secondary bounces of light.

Only accumulate light **directly** from light sources.

Local Reflection Model

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p}\right]$$

Unfold recursion in the rendering equation one step

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p})[$$

$$v(\mathbf{p}_s, \mathbf{p})\left[\epsilon(\mathbf{p}_s, \mathbf{p}) + \int_S \rho(\mathbf{p}_s, \mathbf{p}, \mathbf{p}') I(\mathbf{p}, \mathbf{p}') d\mathbf{p}'\right]]d\mathbf{p}]$$

Drop remaining terms, i.e., only include emitted light in second step. $\mathbf{p}_\mathrm{L}$: position of light source. L : all lights.

$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_L \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) v(\mathbf{p}_s, \mathbf{p}_L) \epsilon(\mathbf{p}_s, \mathbf{p}_L) d\mathbf{p}_L\right]$$

Local Reflection Model

$$I(\mathbf{p}_c, \mathbf{p}_s) \quad = \quad v(\mathbf{p}_c, \mathbf{p}_s) \left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_L \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) v(\mathbf{p}_s, \mathbf{p}_L) \epsilon(\mathbf{p}_s, \mathbf{p}_L) d\mathbf{p}_L \right]$$

Assume: **1**. Non-emitting surfaces $\epsilon(\mathbf{p}_c, \mathbf{p}_s) = 0$

2. Perfect visibility $v(\mathbf{p}_c, \mathbf{p}_s) = 1$

3. Finite number of lights $\int_L \rightarrow \sum_L$

Local Reflection Model

$$I(\mathbf{p}_c, \mathbf{p}_s) \quad = \quad v(\mathbf{p}_c, \mathbf{p}_s) \left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_L \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) v(\mathbf{p}_s, \mathbf{p}_L) \epsilon(\mathbf{p}_s, \mathbf{p}_L) d\mathbf{p}_L \right]$$

Assume: **1**. Non-emitting surfaces $\epsilon(\mathbf{p}_c, \mathbf{p}_s) = 0$

2. Perfect visibility $v(\mathbf{p}_c, \mathbf{p}_s) = 1$

3. Finite number of lights $\int_L \rightarrow \sum_L$

Light intensity of visible light: $v(\mathbf{p}_s, \mathbf{p}_L)\epsilon(\mathbf{p}_s, \mathbf{p}_L) = L(\mathbf{p}_L)$

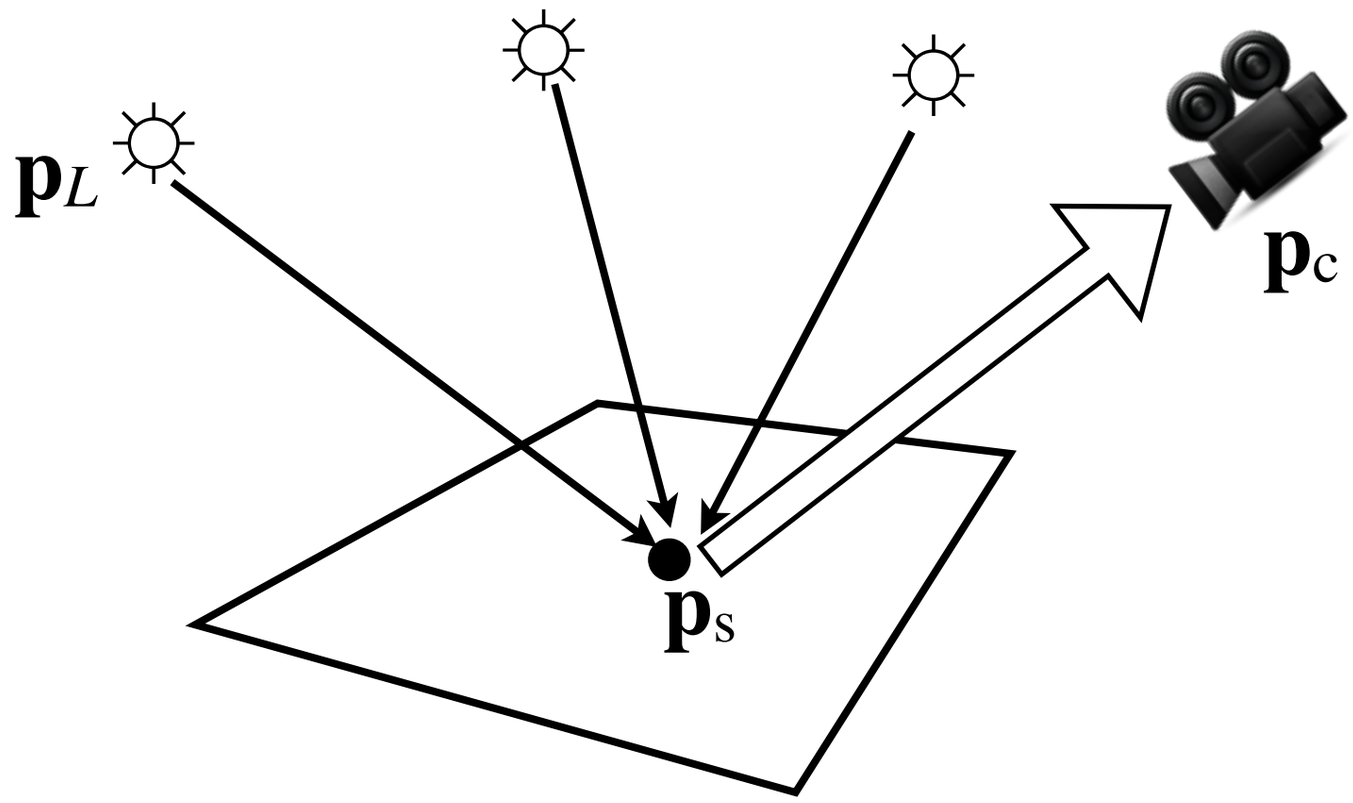
$$I(\mathbf{p}_c, \mathbf{p}_s) \quad = \quad \sum_L \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) L(\mathbf{p}_L)$$

Local Reflection Model

Material properties

Light intensity

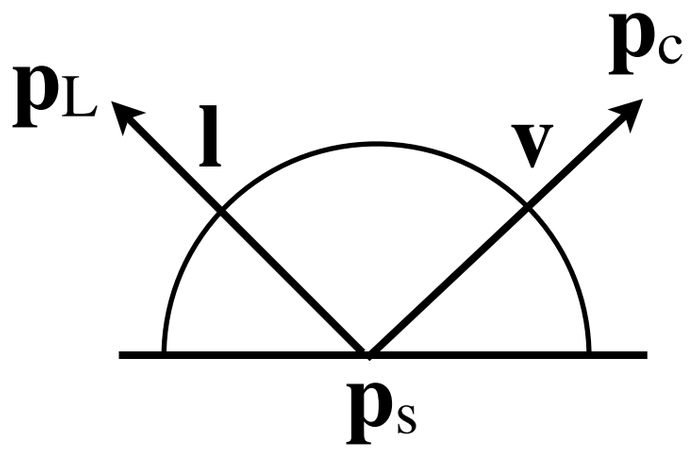
$$I(\mathbf{p}_c, \mathbf{p}_s) = \sum_L \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) L(\mathbf{p}_L)$$



Material Properties: BRDF

- Bidirectional Reflection Distribution Function

$$\begin{aligned}\rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L) &= \rho(\mathbf{v}, \mathbf{l}) \\ \mathbf{v} &= \text{normalize}(\mathbf{p}_c - \mathbf{p}_s) \\ \mathbf{l} &= \text{normalize}(\mathbf{p}_L - \mathbf{p}_s)\end{aligned}$$



The BRDF defines the fraction of light coming from direction **l** which is reflected in direction **v.**

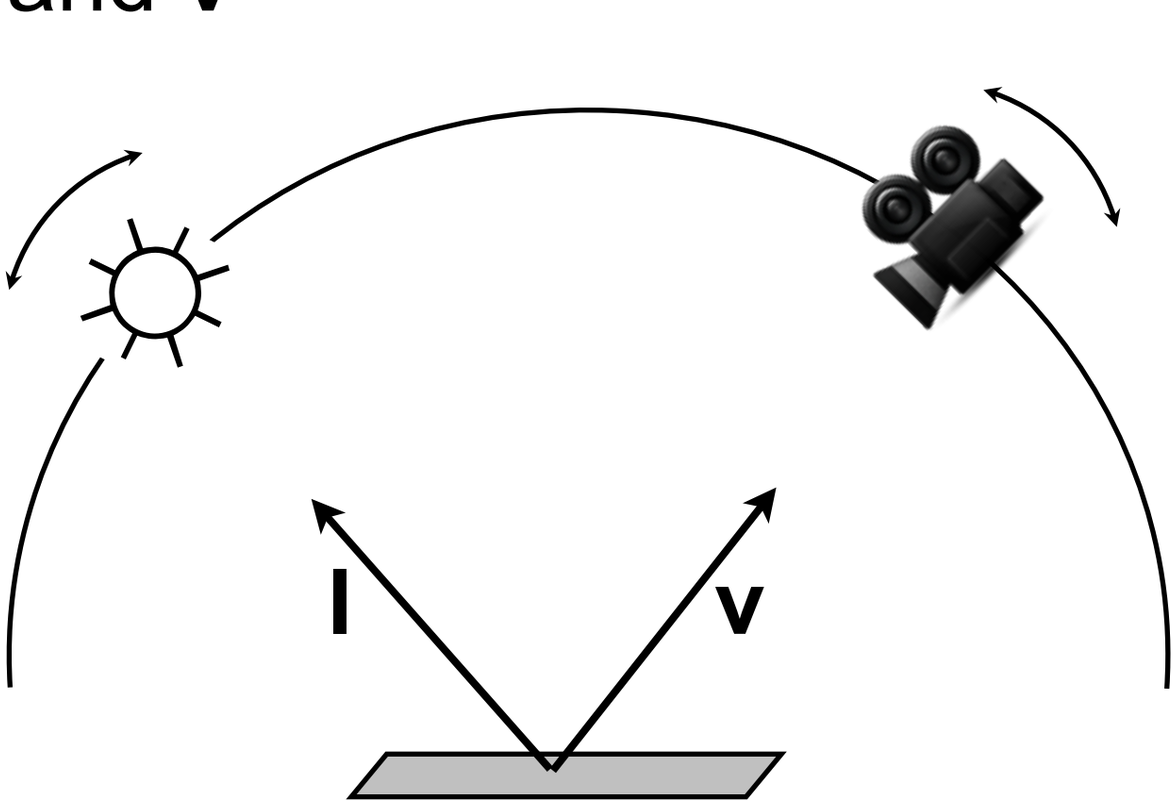
Example: Phong shading

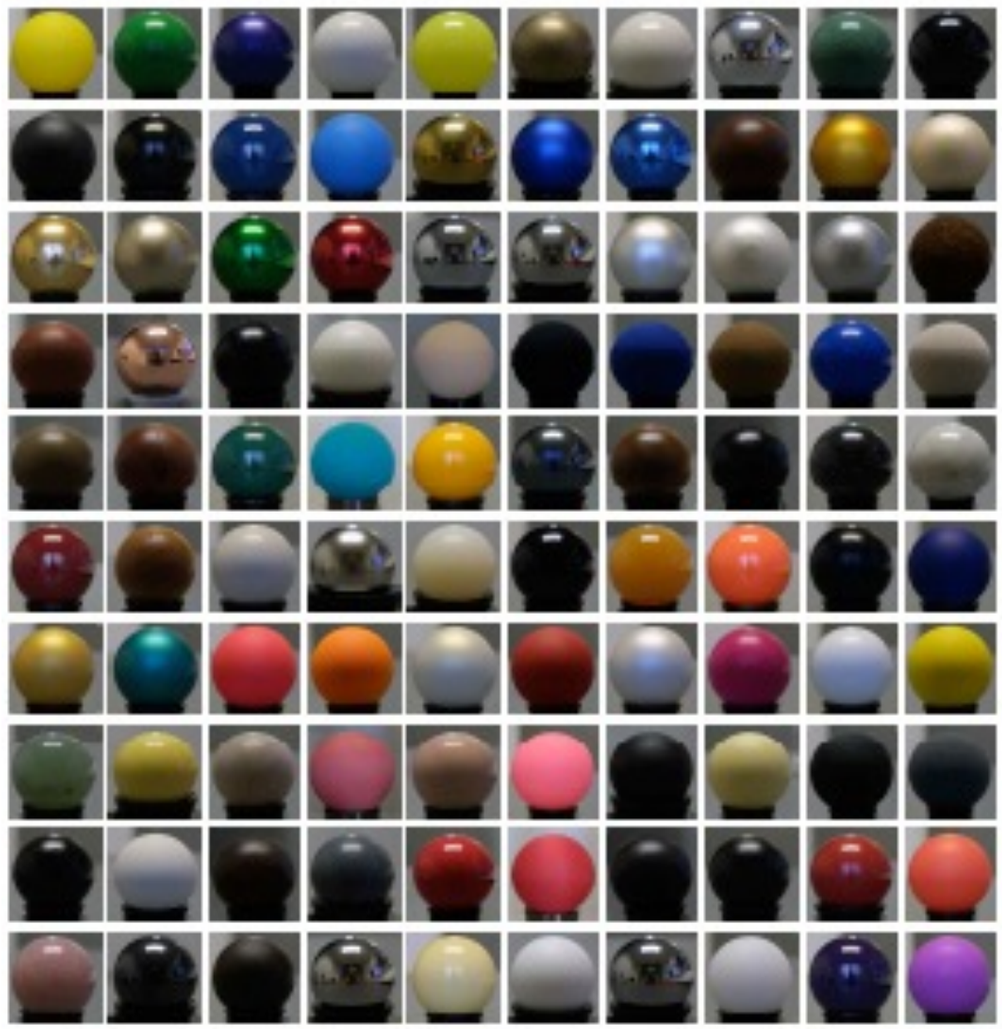
$$I(\mathbf{p}_s, \mathbf{p}_c) \approx \sum_i \rho(\mathbf{v}, \mathbf{l}_i) L_i = \sum_i (k_a + k_d(\mathbf{l}_i \cdot \mathbf{n}) + k_s(\mathbf{r}_i \cdot \mathbf{v})^\alpha) L_i$$

Example: Measure BRDFs

Move camera around in hemisphere
Move light around in hemisphere
BRDF has a unique value $\rho(\mathbf{l},\mathbf{v})$

for each combination of
l and **v**





http://www.merl.com/brdf/

Rendering Equation Summary

- The general equation is very complex
  - Active research on how to solve it approximately
- Simplifications leads to plausible approximations, such as Phong

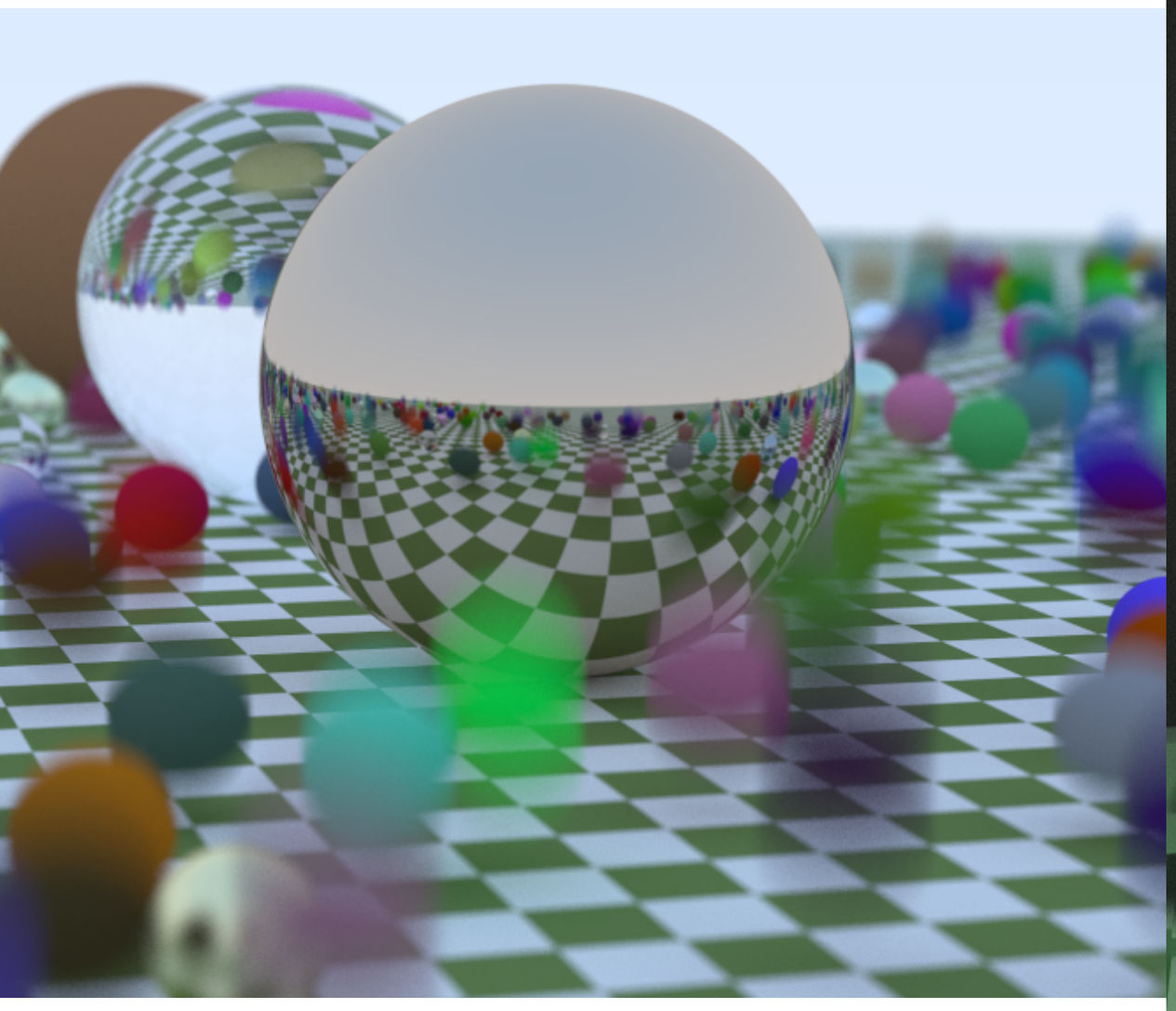
General form

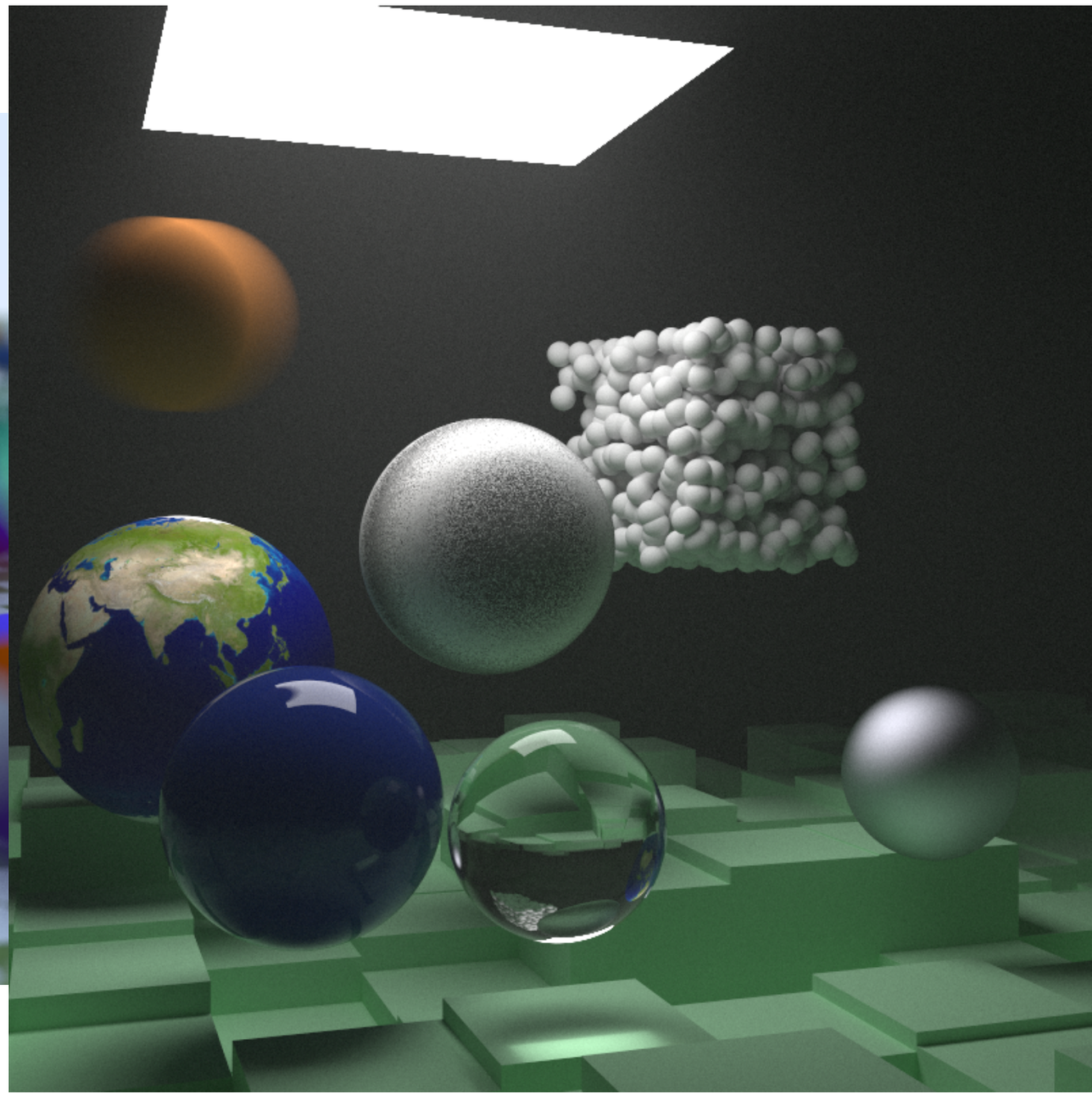
$$I(\mathbf{p}_c, \mathbf{p}_s) = v(\mathbf{p}_c, \mathbf{p}_s)\left[\epsilon(\mathbf{p}_c, \mathbf{p}_s) + \int_S \rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}) I(\mathbf{p}_s, \mathbf{p}) d\mathbf{p}\right]$$

Simplified (Phong)

$$I(\mathbf{p}_s, \mathbf{p}_c) \approx \sum_i \rho(\mathbf{v}, \mathbf{l}_i) L_i = \sum_i (k_a + k_d(\mathbf{l}_i \cdot \mathbf{n}) + k_s(\mathbf{r}_i \cdot \mathbf{v})^\alpha) L_i$$

Ray Tracing in One Weekend





https://raytracing.github.io/