Visibility Computations

EDAF80

Jacob Munkberg

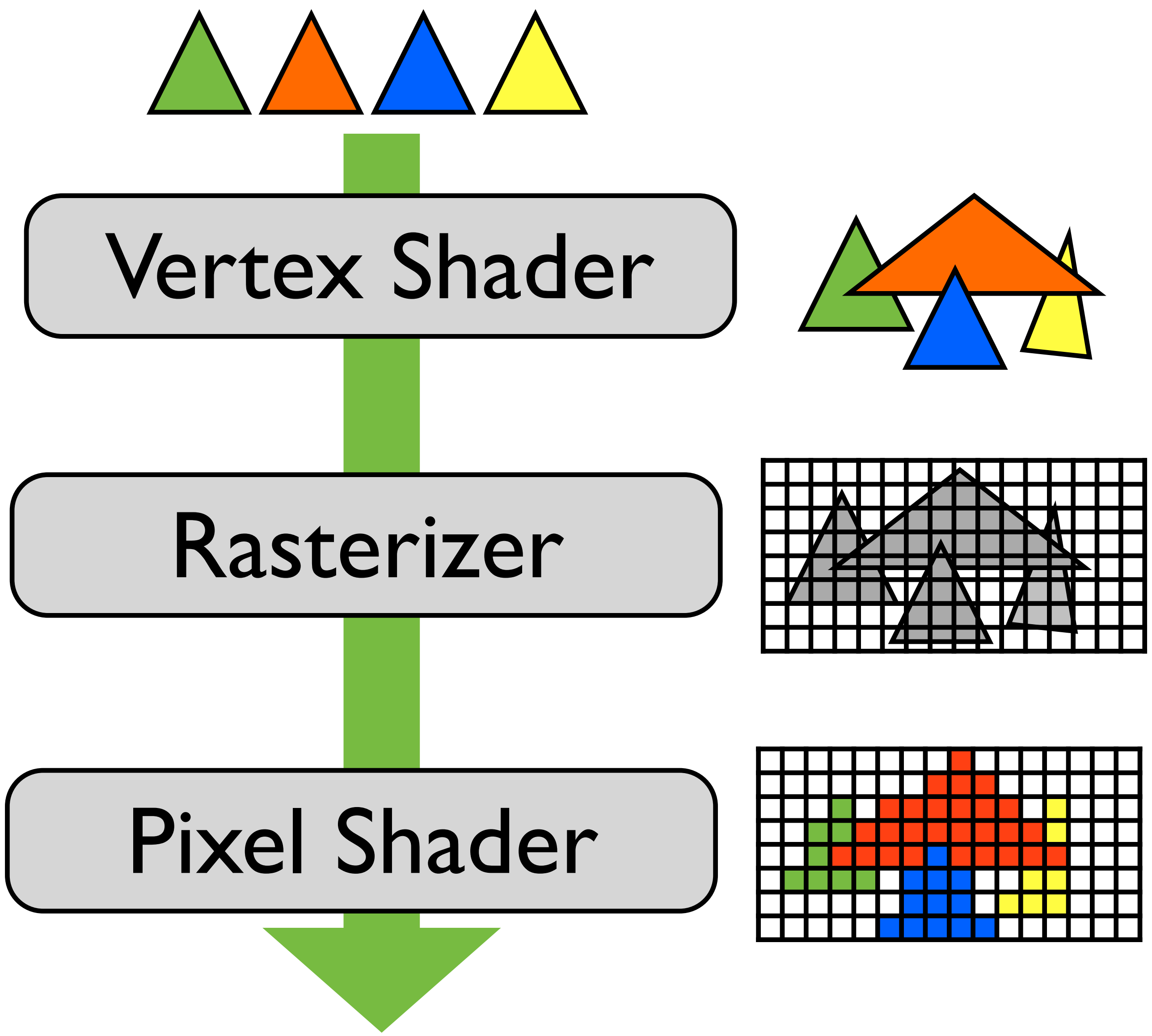
2018

Today

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

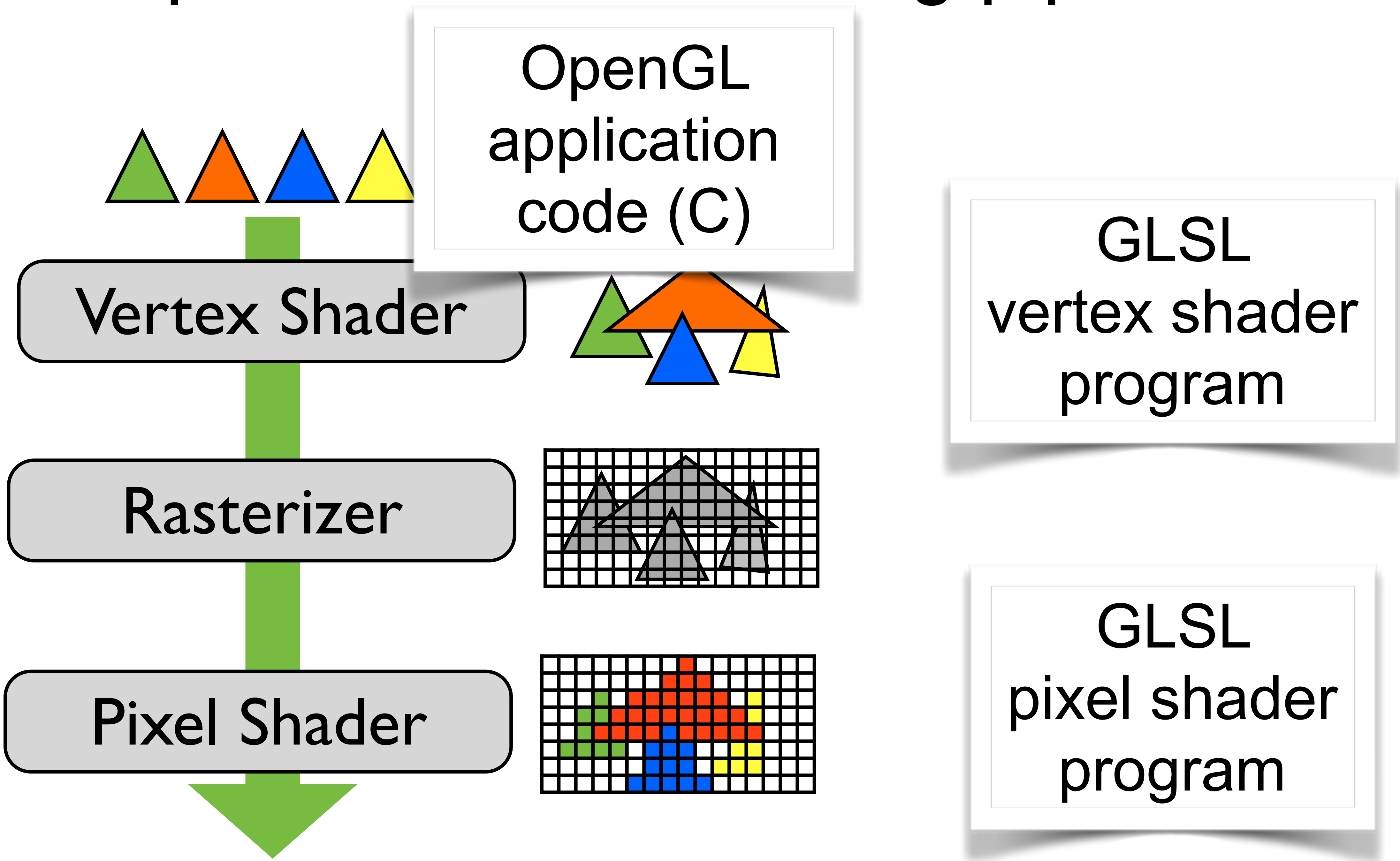
Graphics Hardware

- Expresses the rendering pipeline



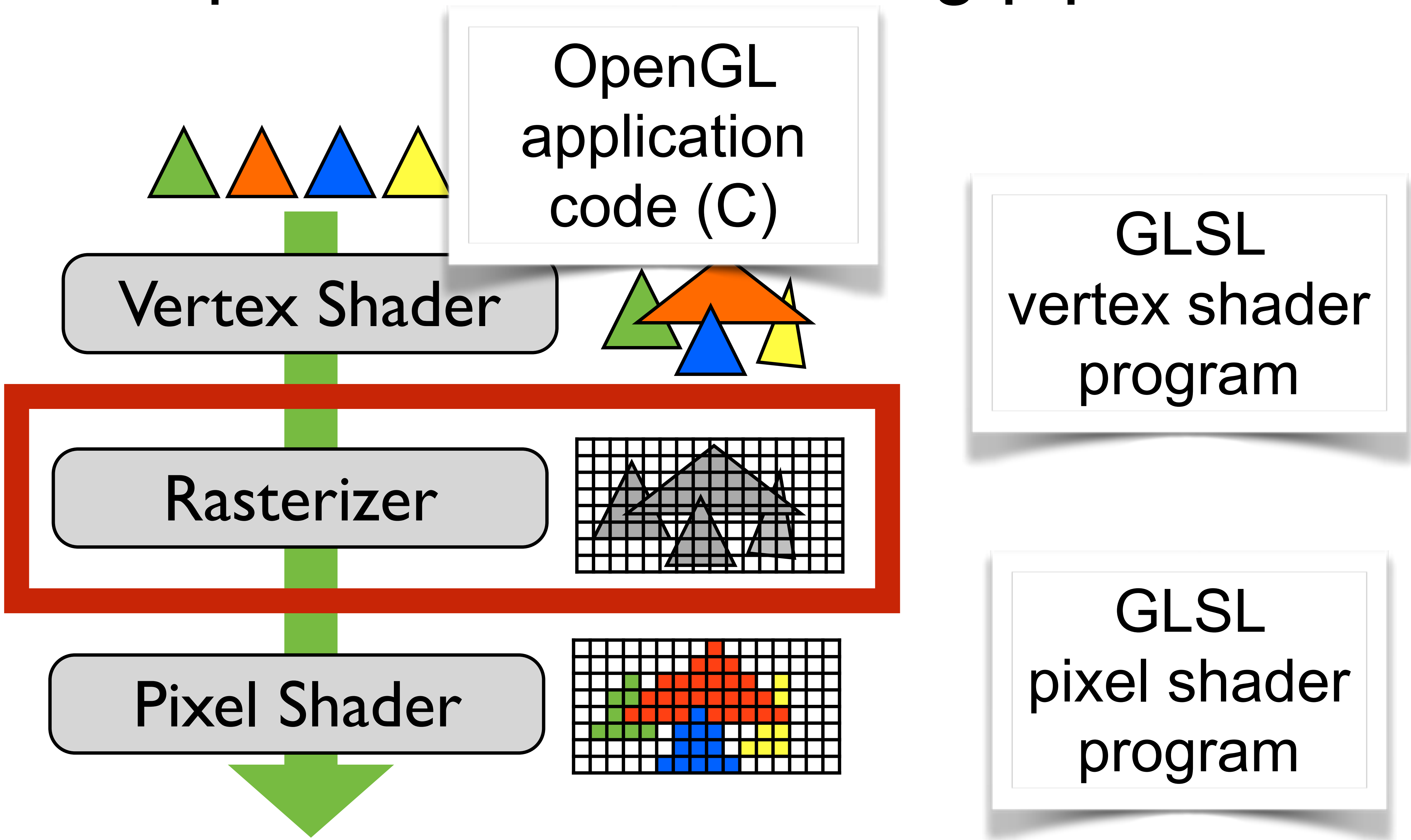
Graphics Hardware

- Expresses the rendering pipeline

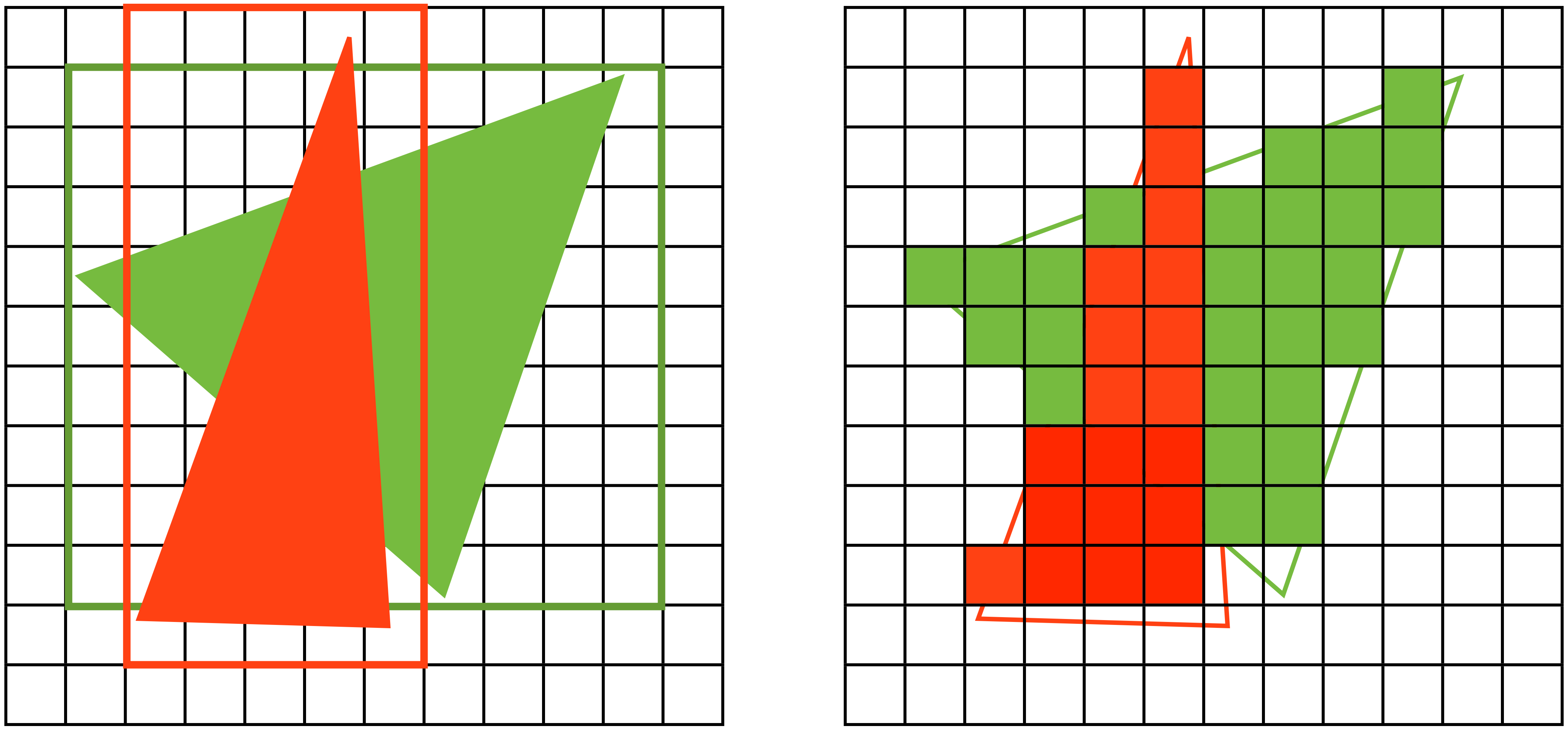


Graphics Hardware

- Expresses the rendering pipeline



Rasterization

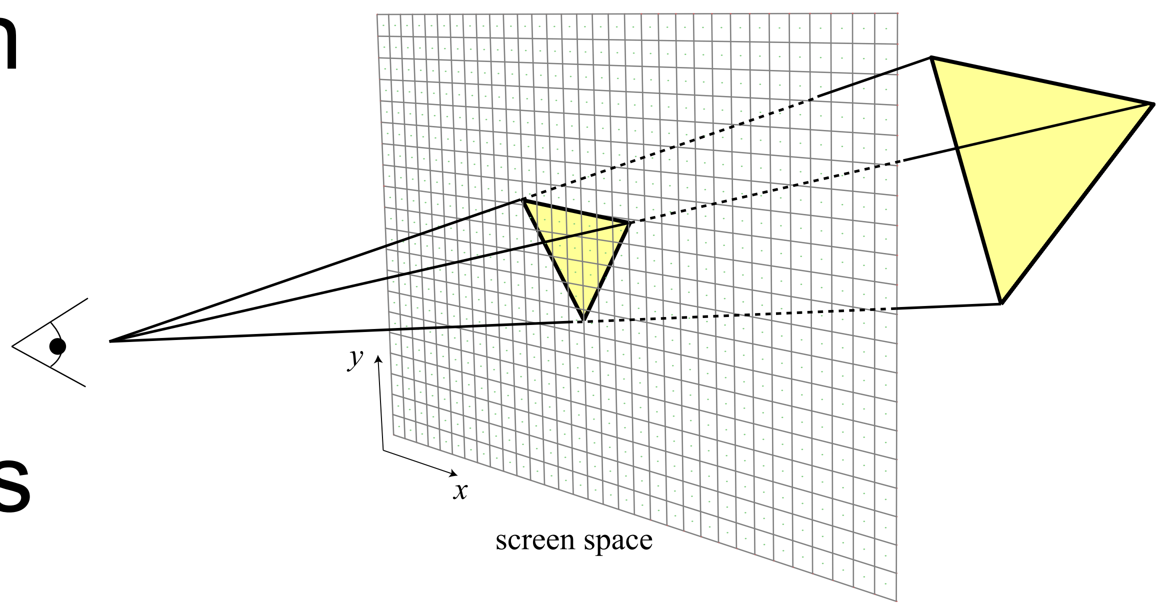


Convert triangles to pixels

Rasterization of Triangle

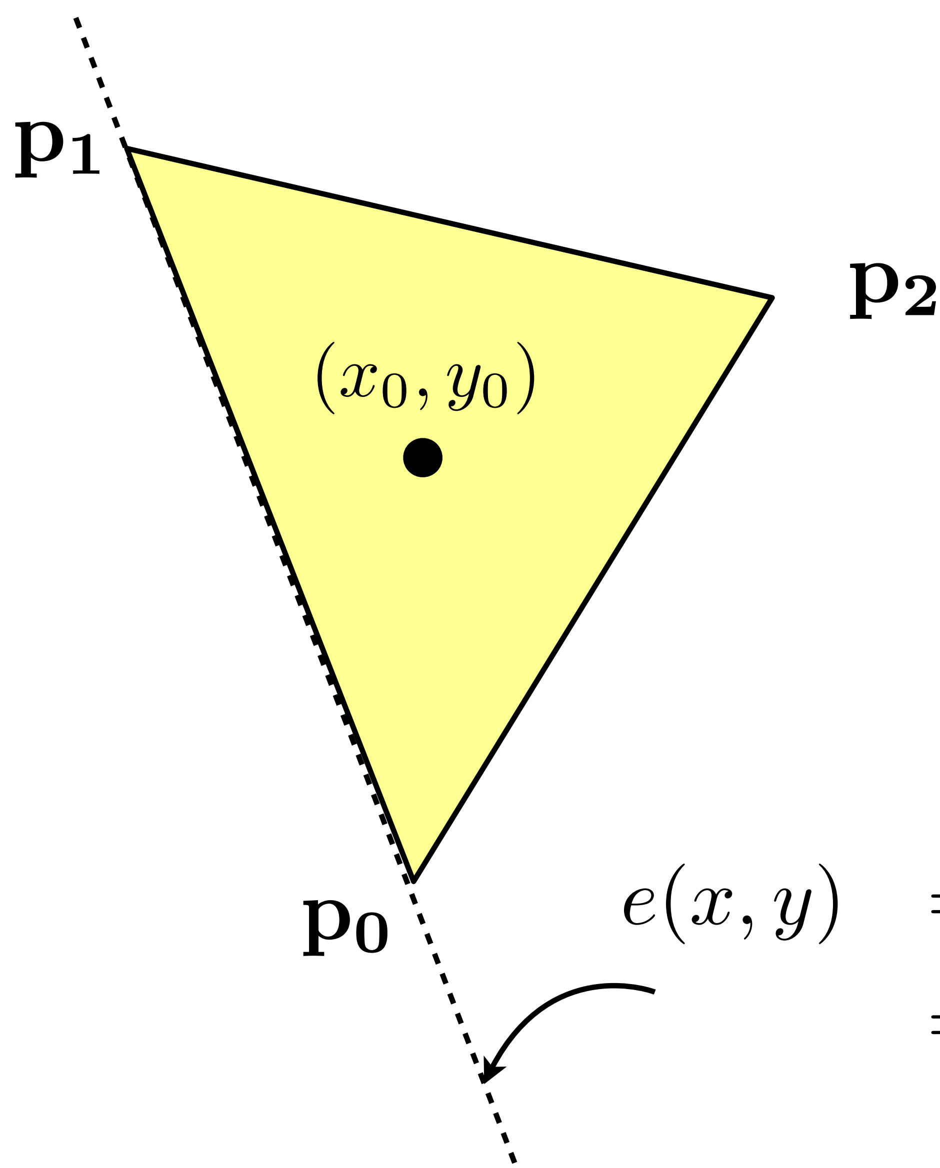
- Determine which pixels a triangle covers

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



Edge Equation

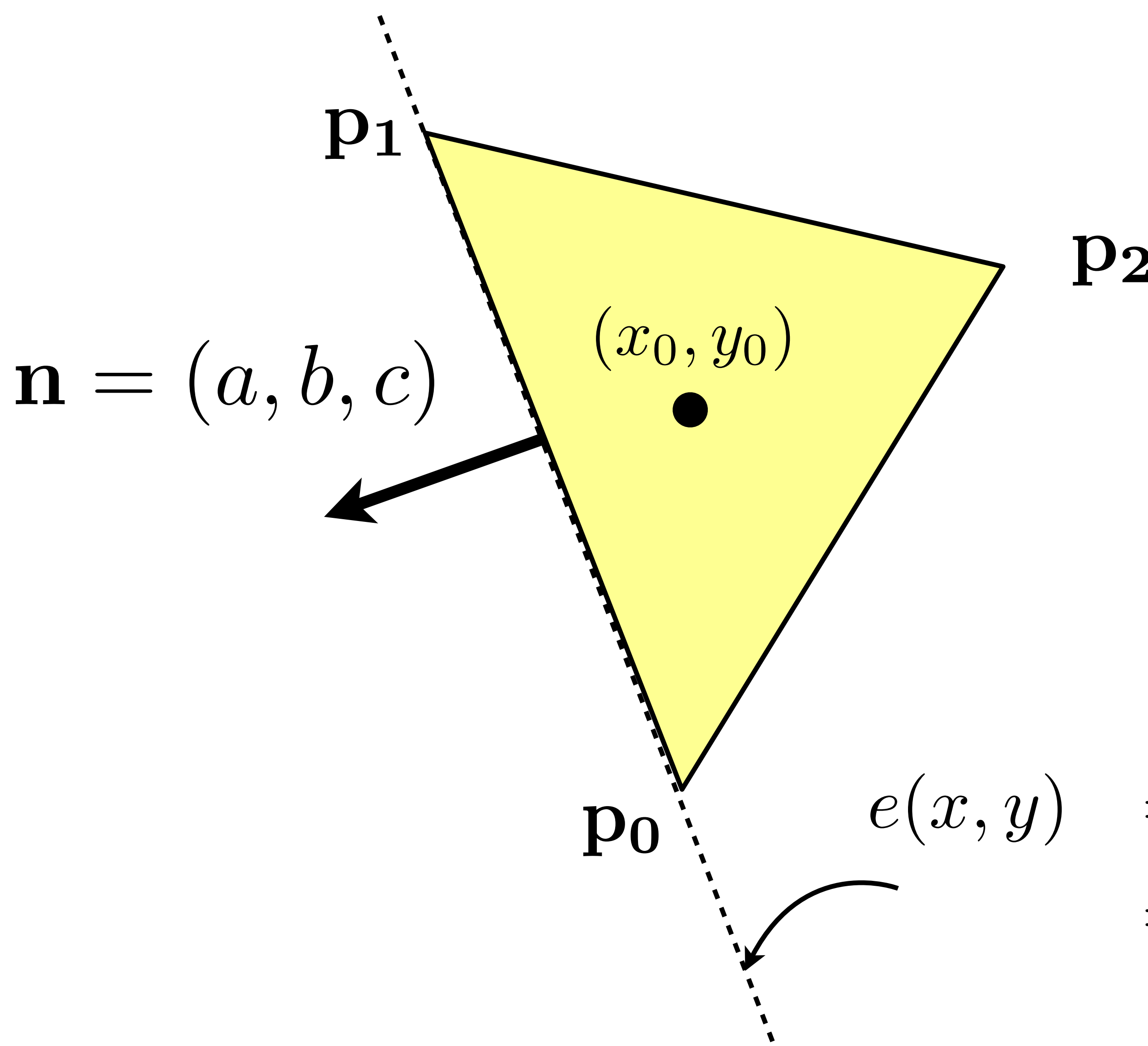
- Point inside triangle test



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

Edge Equation

- Point inside edge test



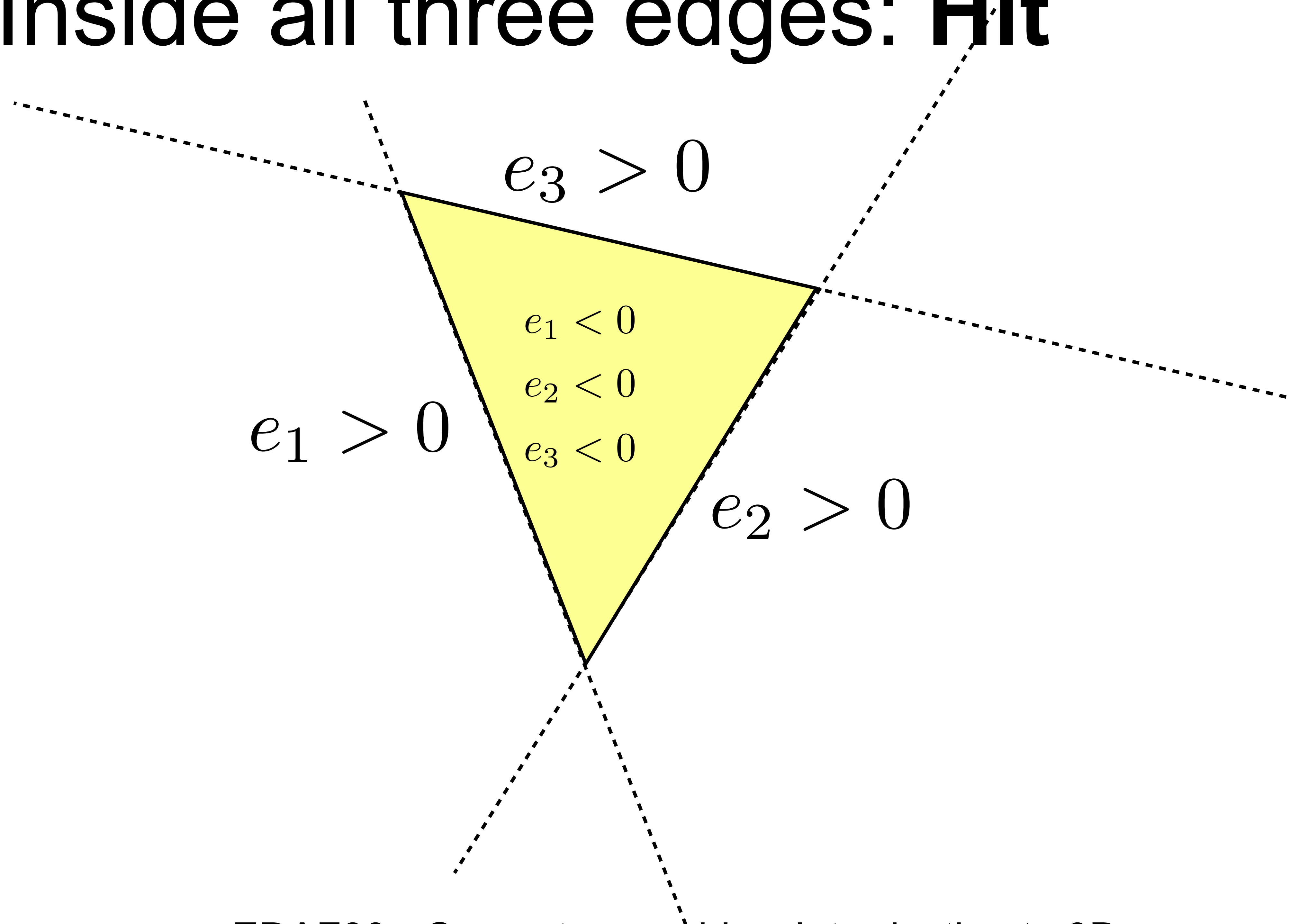
Point (x_0, y_0) inside edge between $\mathbf{p}_0$ and $\mathbf{p}_1$ if:

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

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

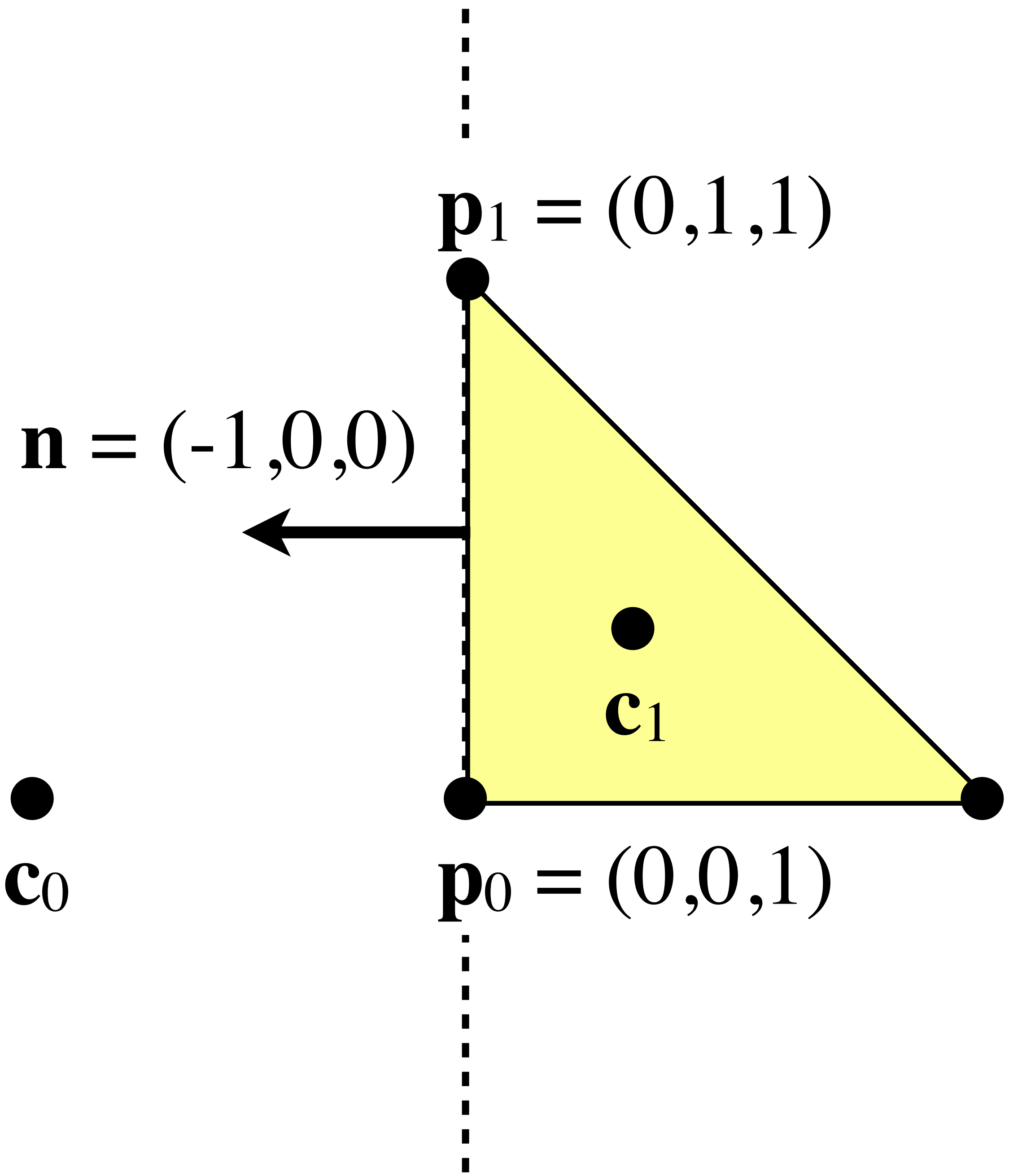
Point Inside Triangle Test

Inside all three edges: **Hit**



Edge Equation Example

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



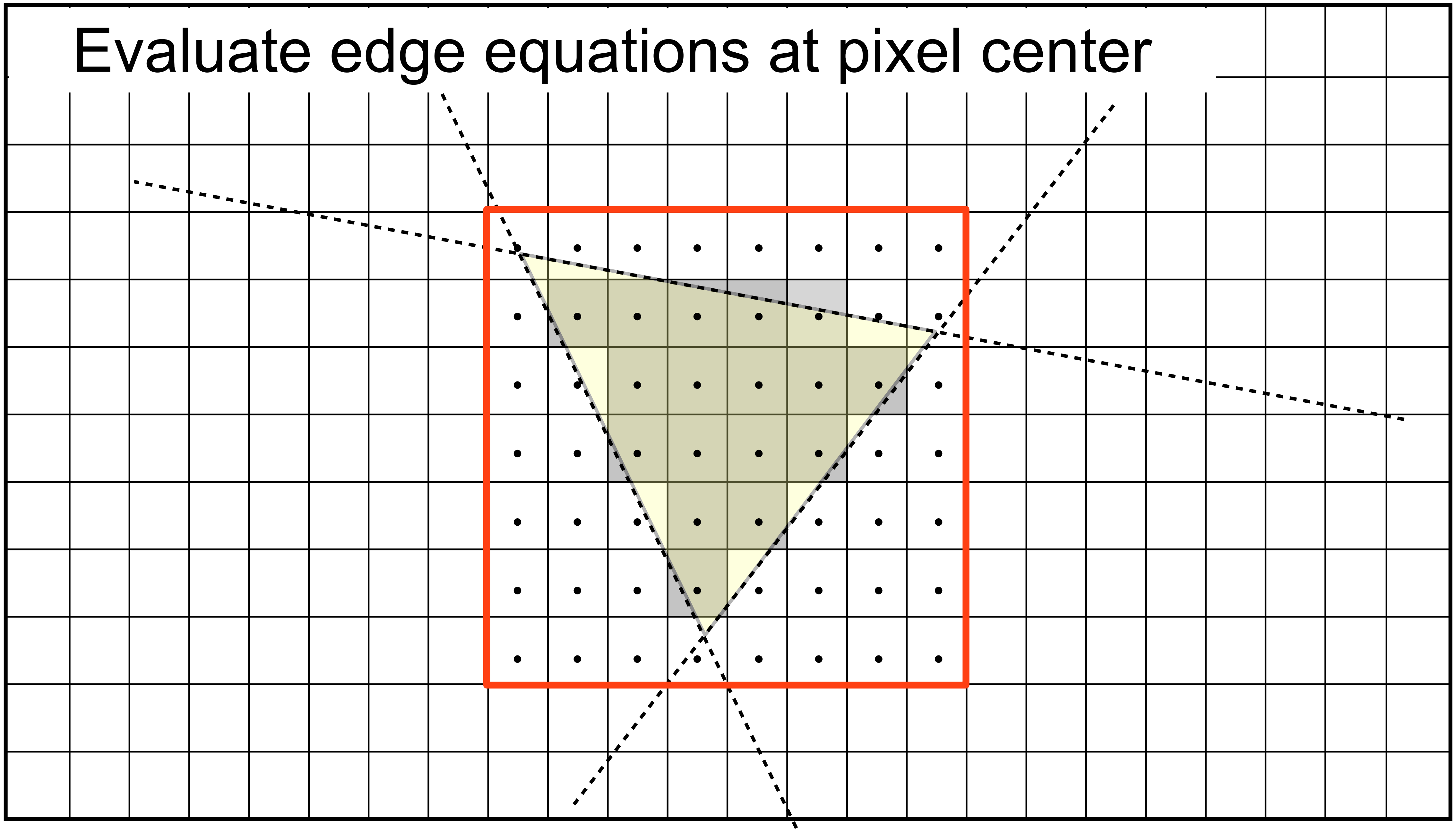
$$\mathbf{n} = \mathbf{p}_0 \times \mathbf{p}_1 = (-1,0,0)$$

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

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

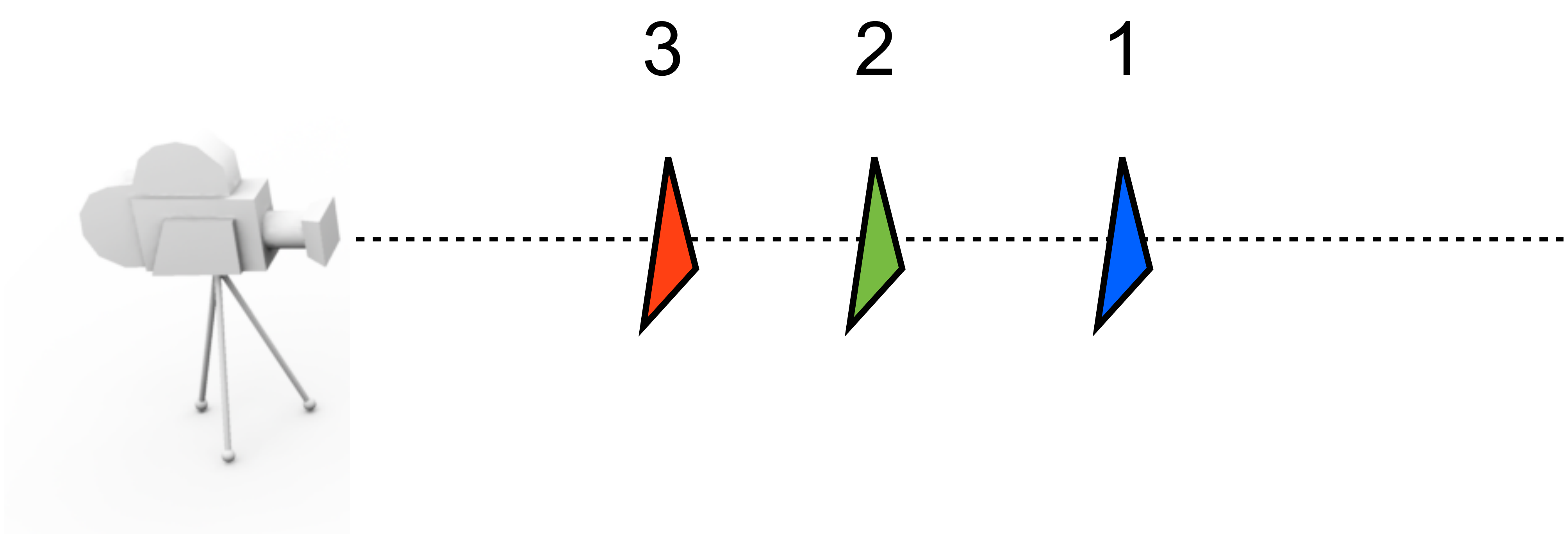
Point $\mathbf{c}_1$=(0.3,0.3,1) inside edge, as e($\mathbf{c}_1$) = $\mathbf{n} \cdot \mathbf{c}_1 < 0$

Test Samples



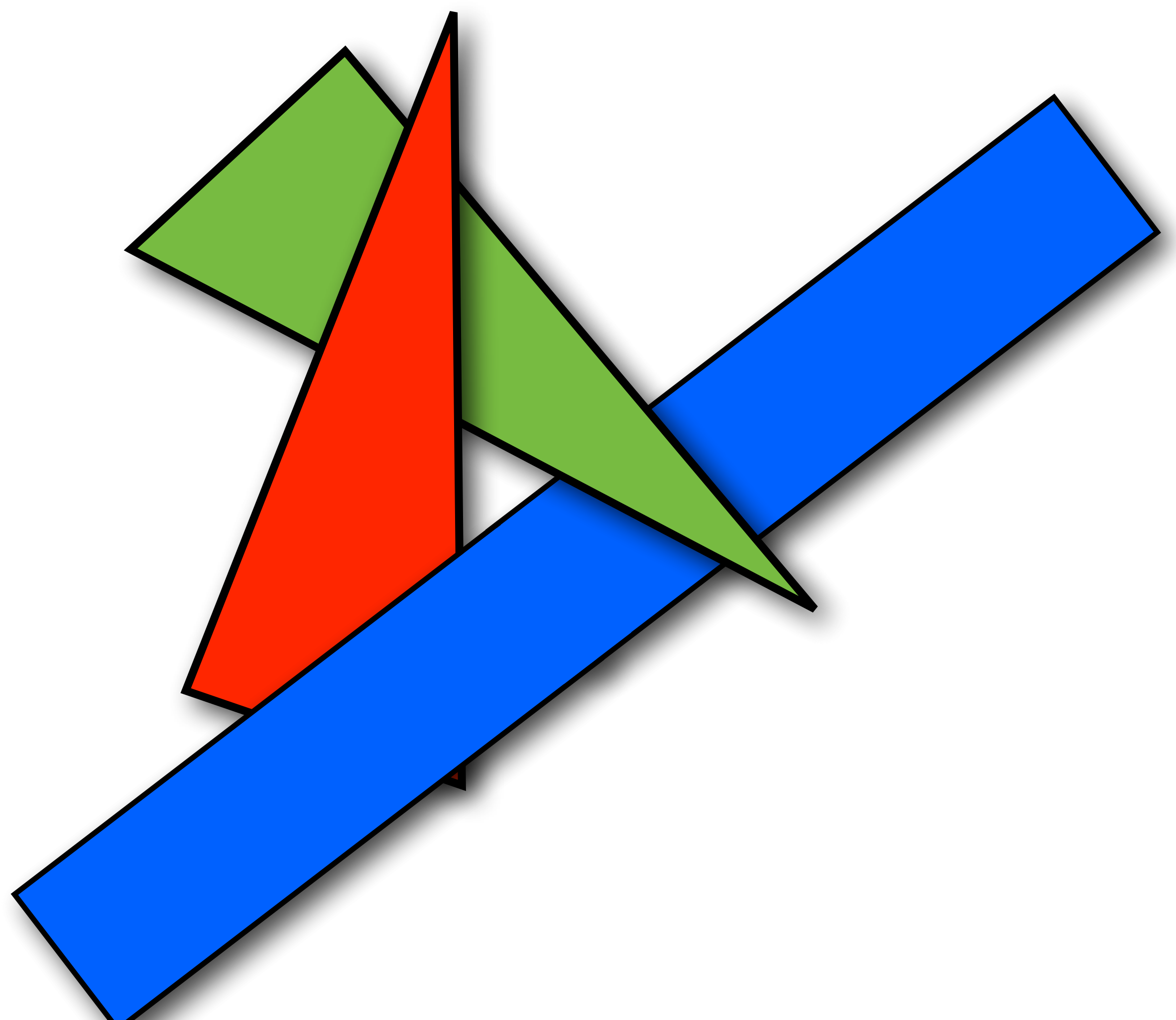
Overlapping Primitives

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



Harder case

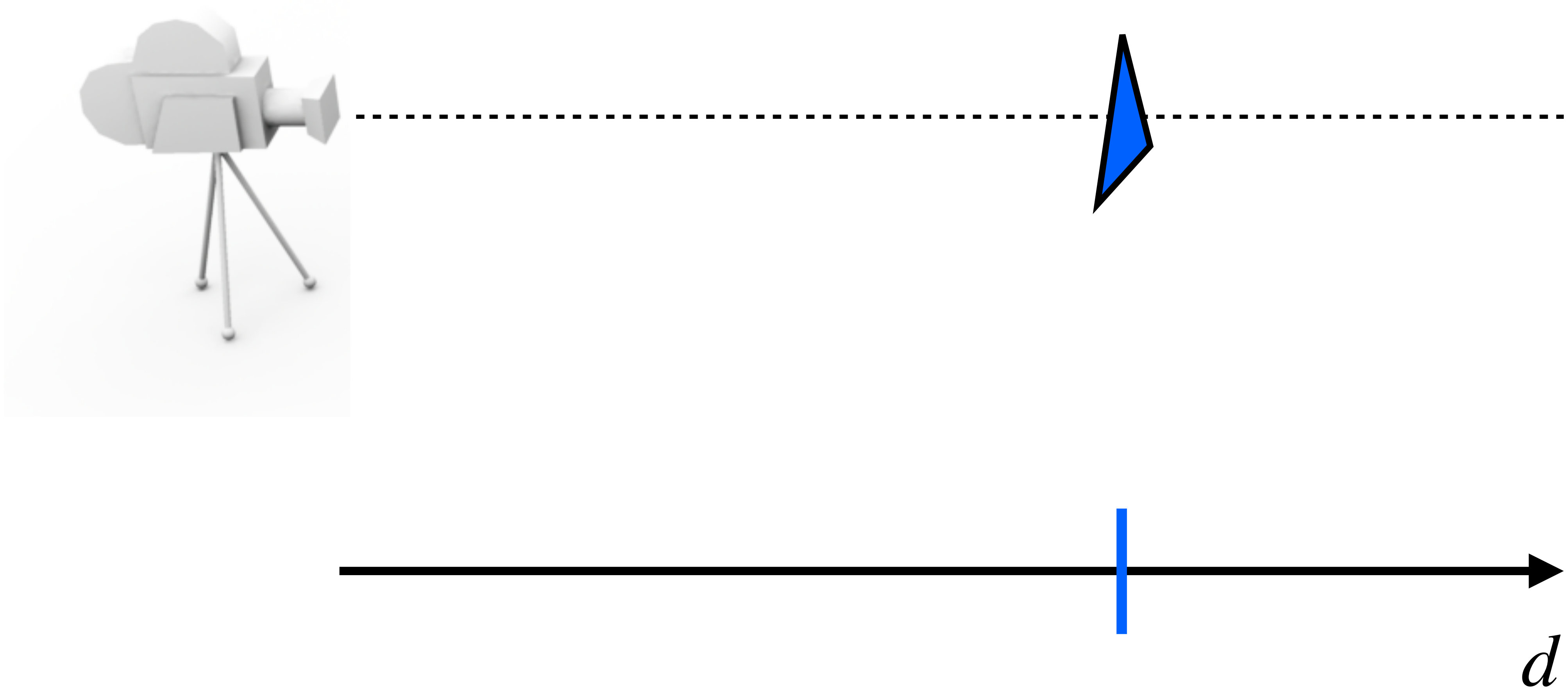
- Not always possible to sort objects in depth



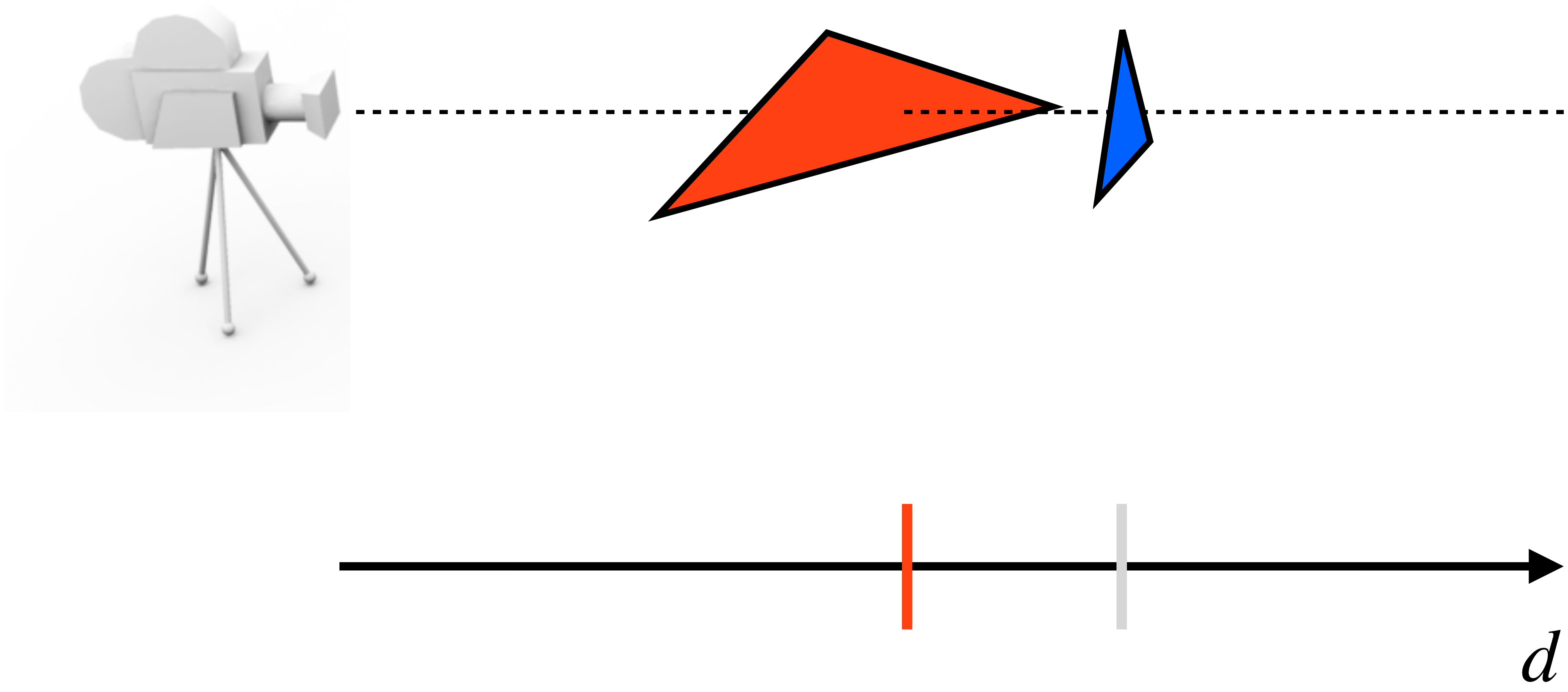
Depth Buffering

- 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);`

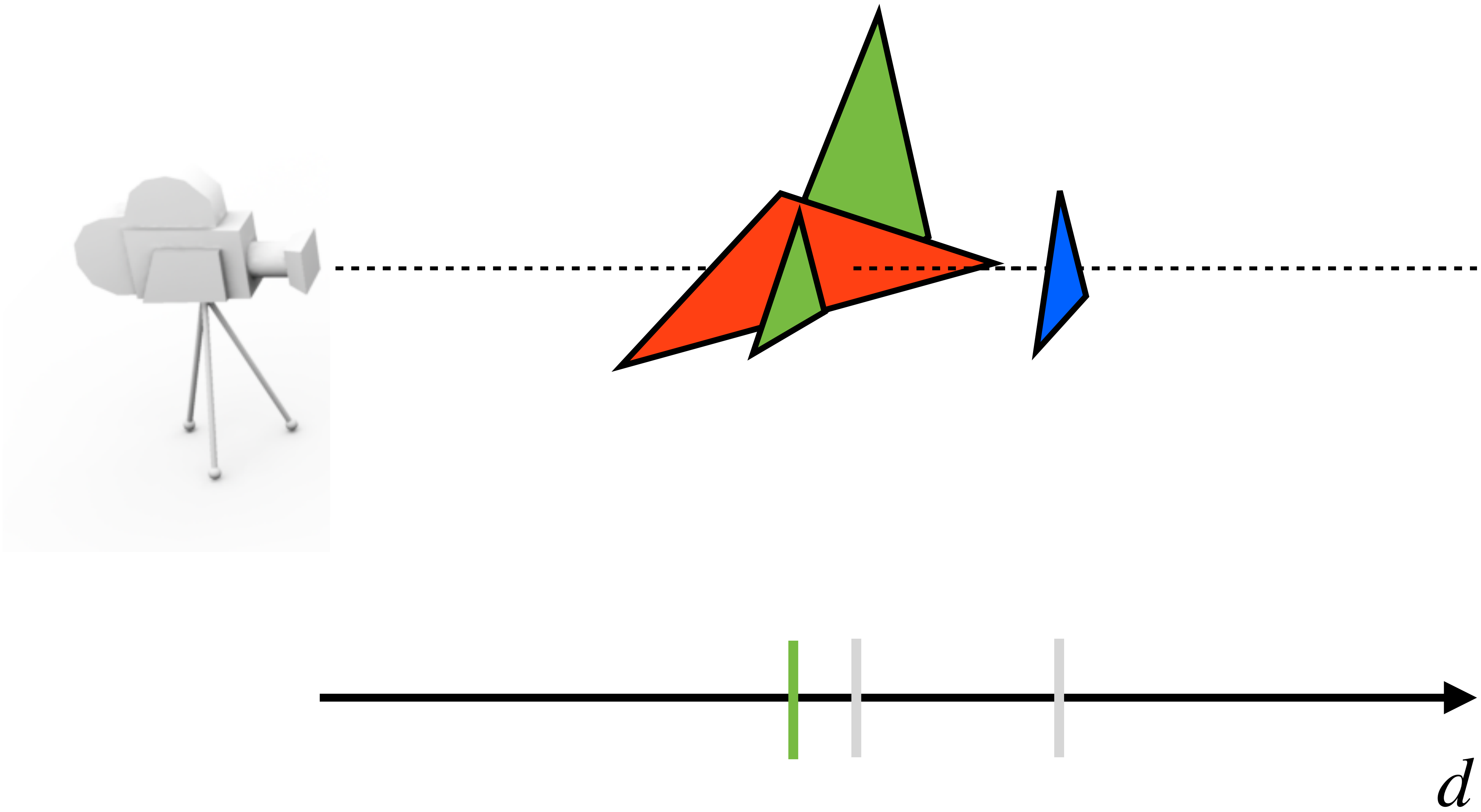
Overlapping Primitives



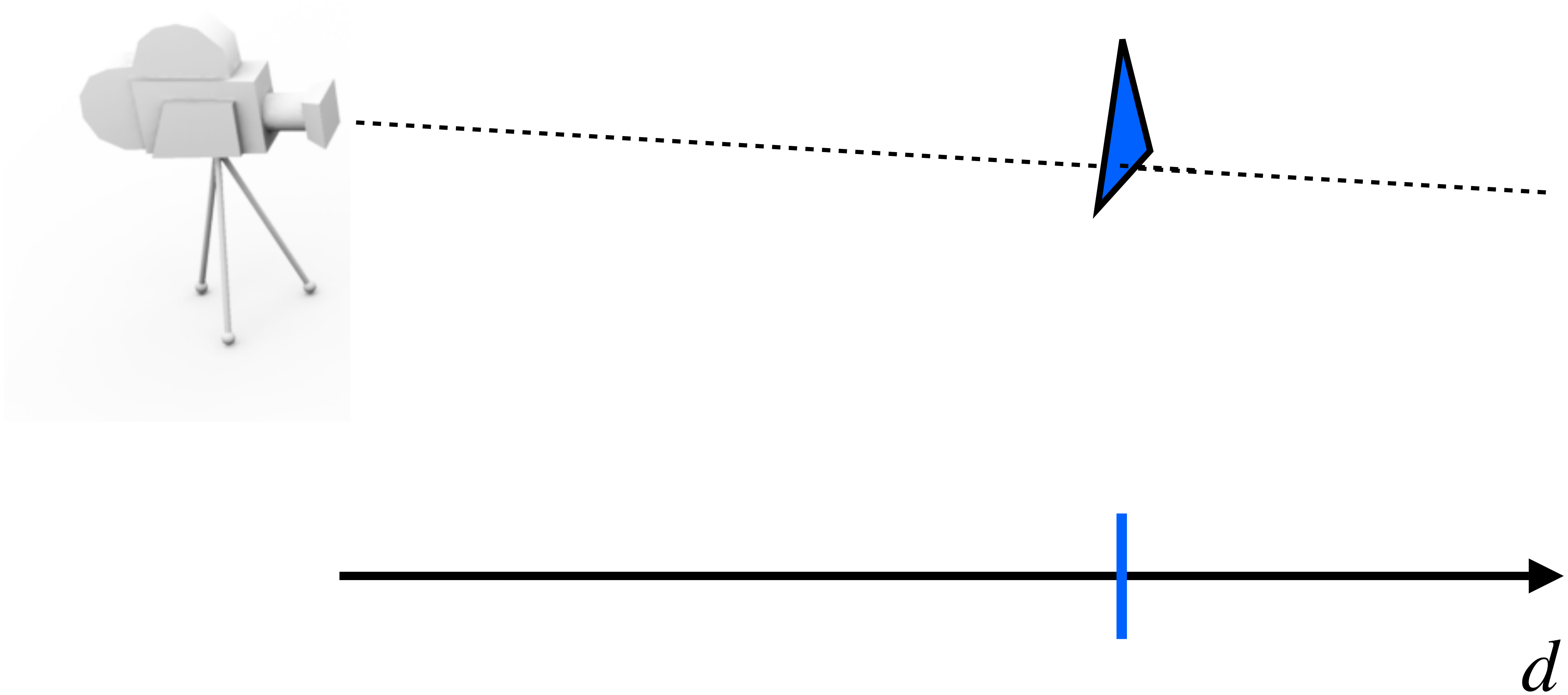
Overlapping Primitives



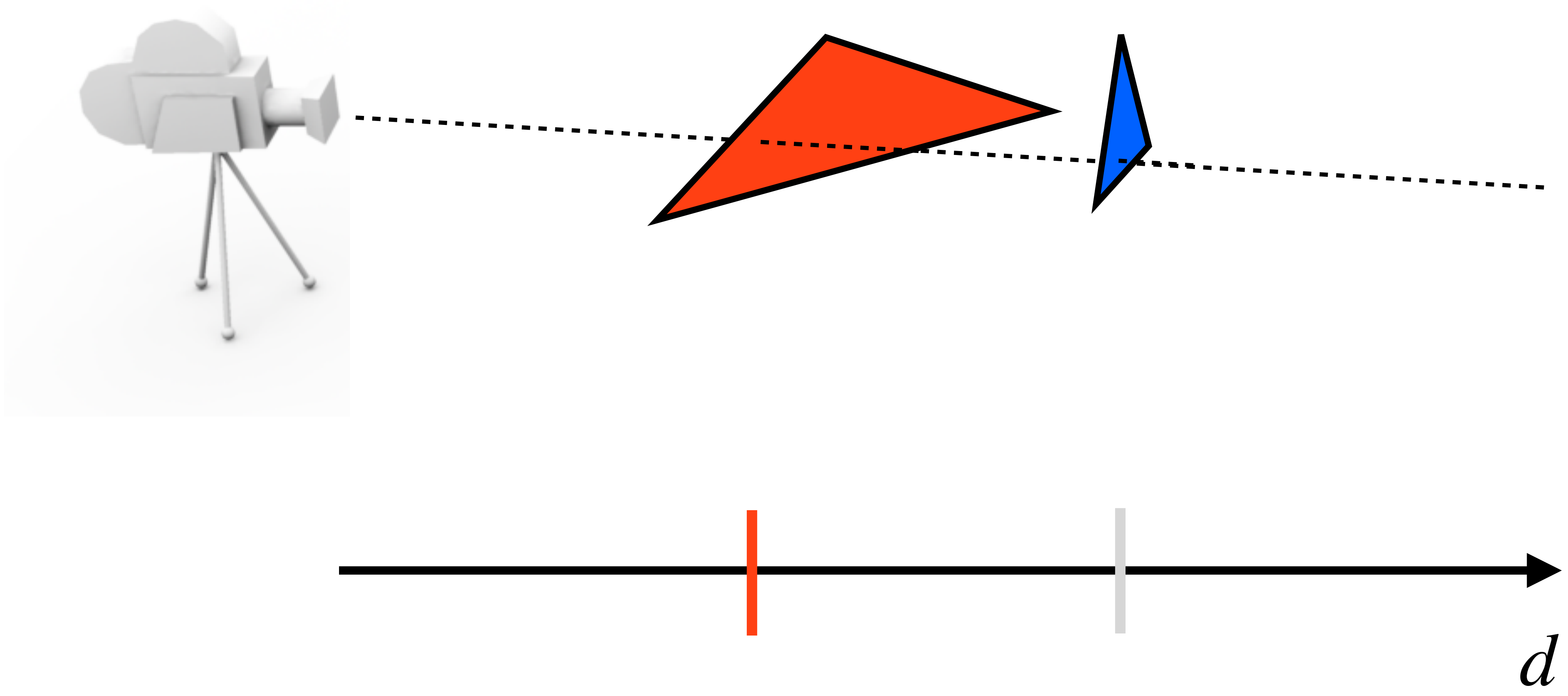
Overlapping Primitives



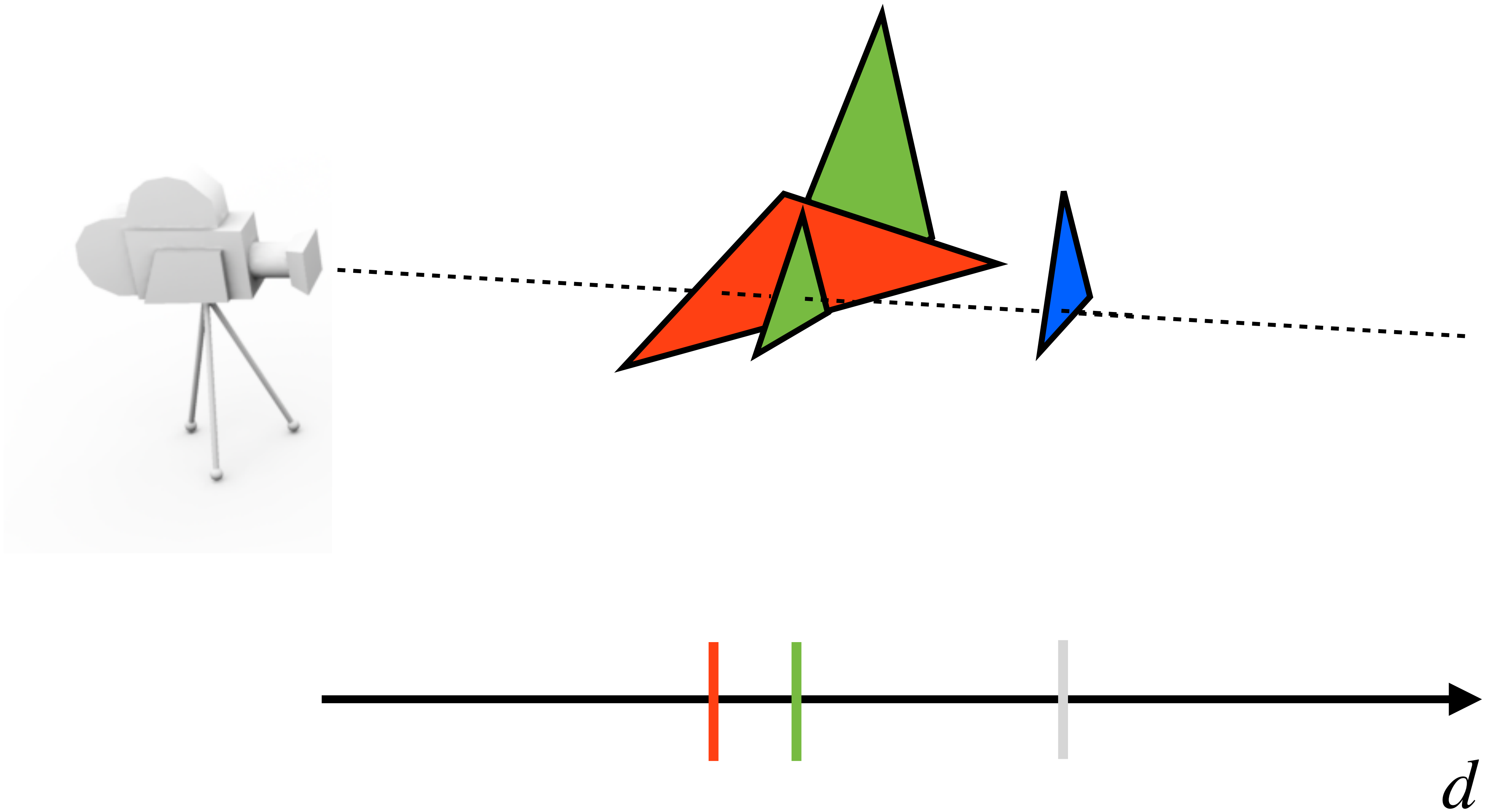
Overlapping Primitives



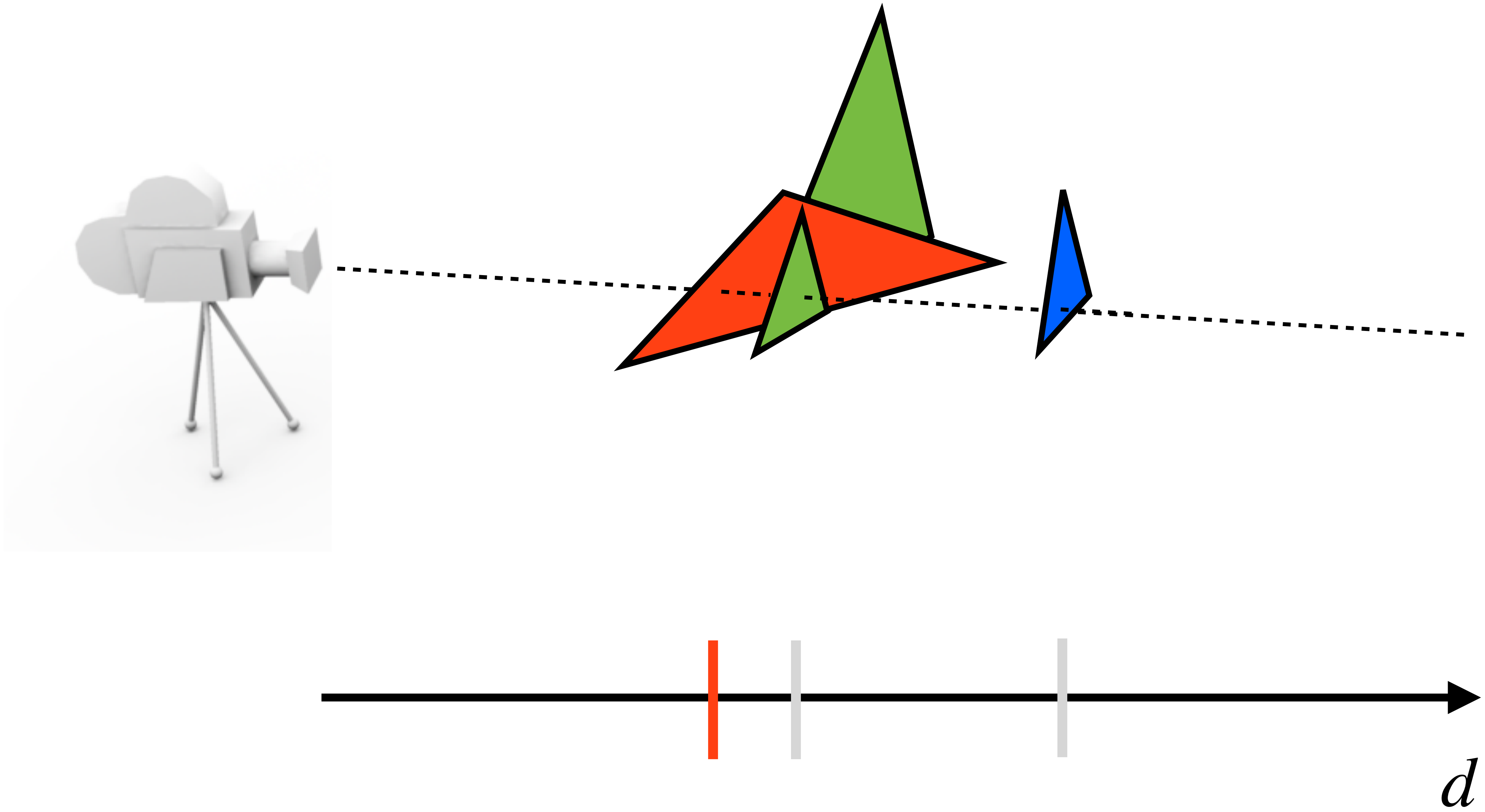
Overlapping Primitives



Overlapping Primitives



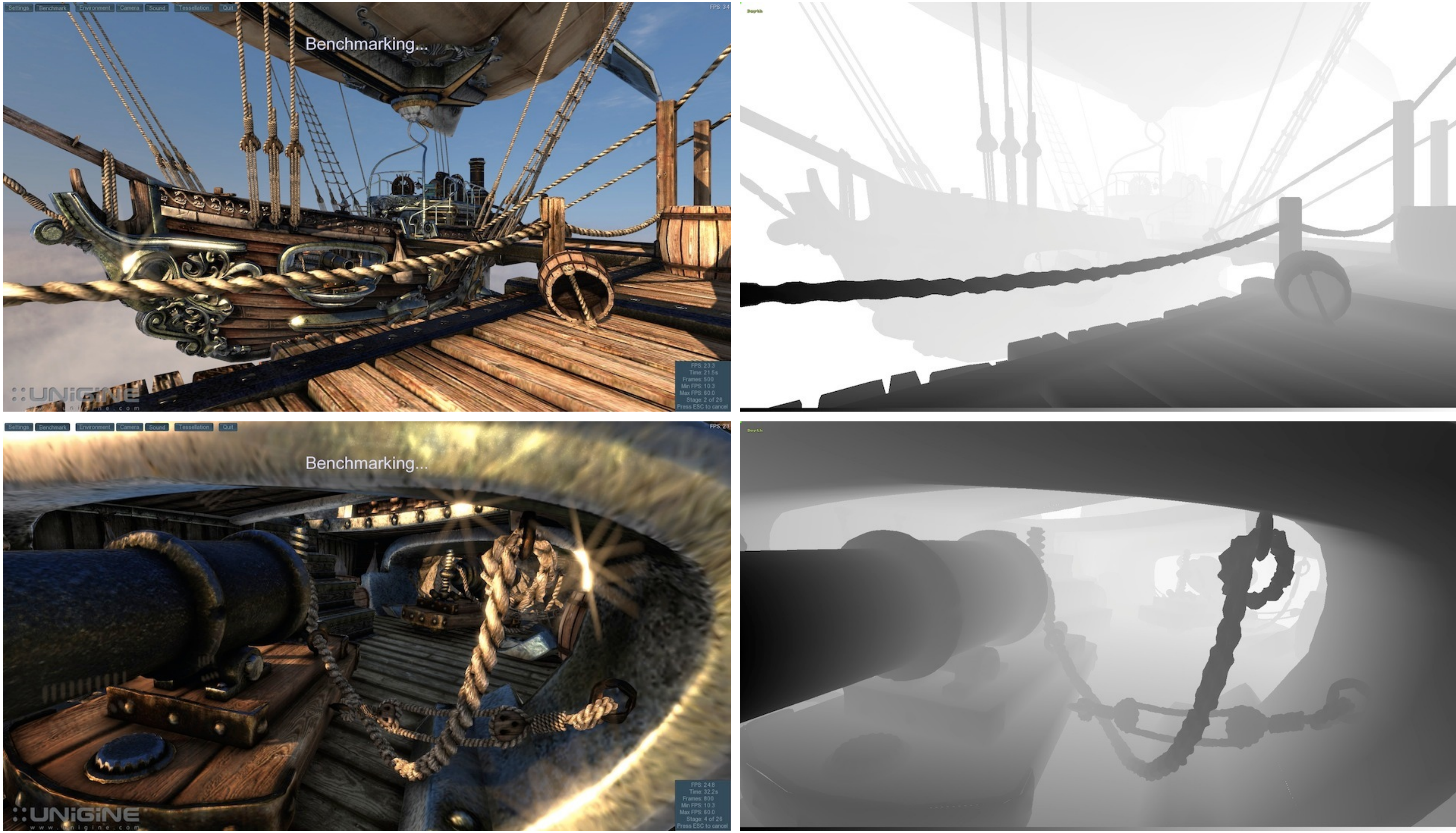
Overlapping Primitives



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

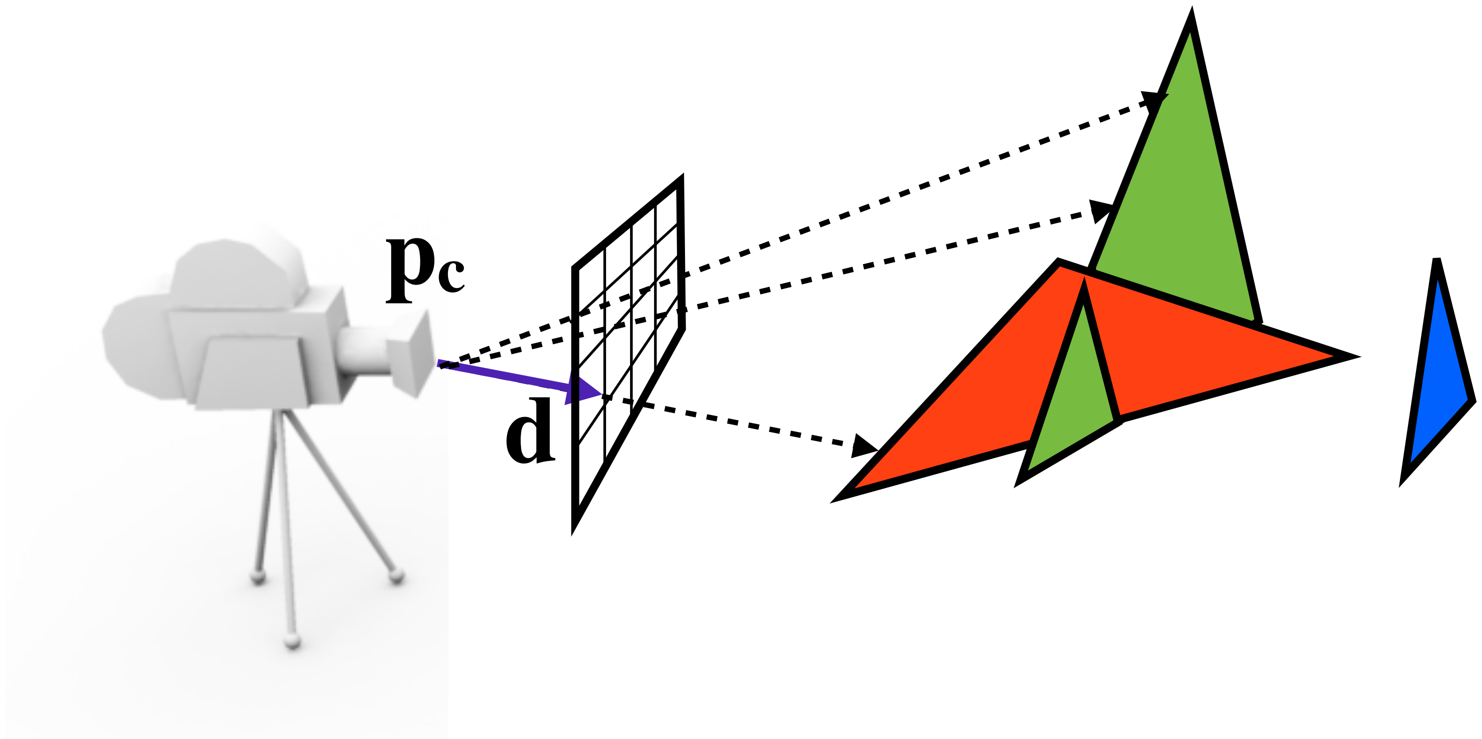
Ray Tracing

A brief introduction

Visibility Computations

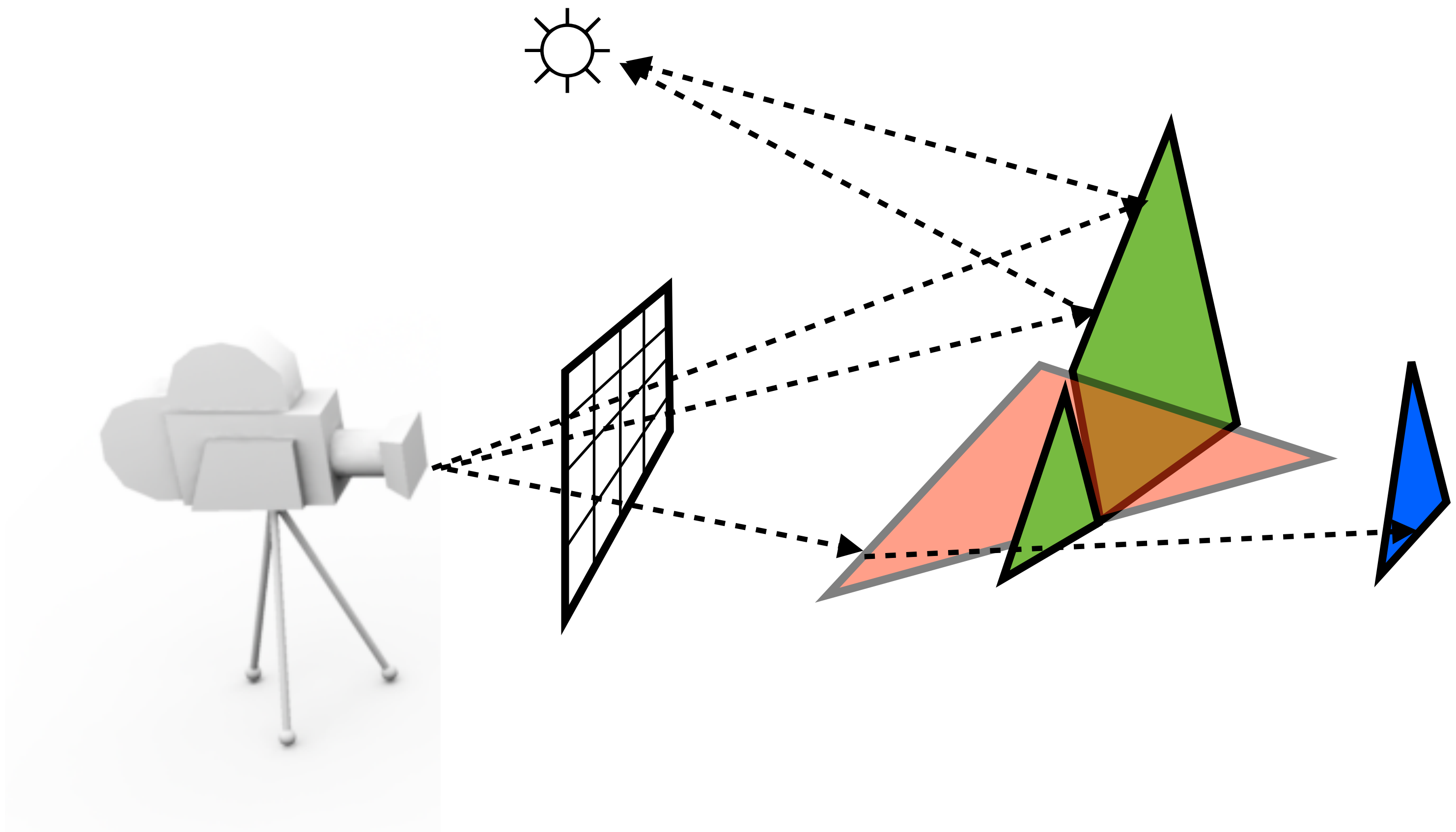
- Rasterization pipeline
  - For **each triangle**, find **covered pixels**
  - Check with depth buffer if closest hit
- Ray tracing
  - For **each pixel**, find **overlapping triangles**
  - Shoot a ray from the camera through the pixel
    - Find the triangle that first intersects the ray
- Loops are reversed!

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)

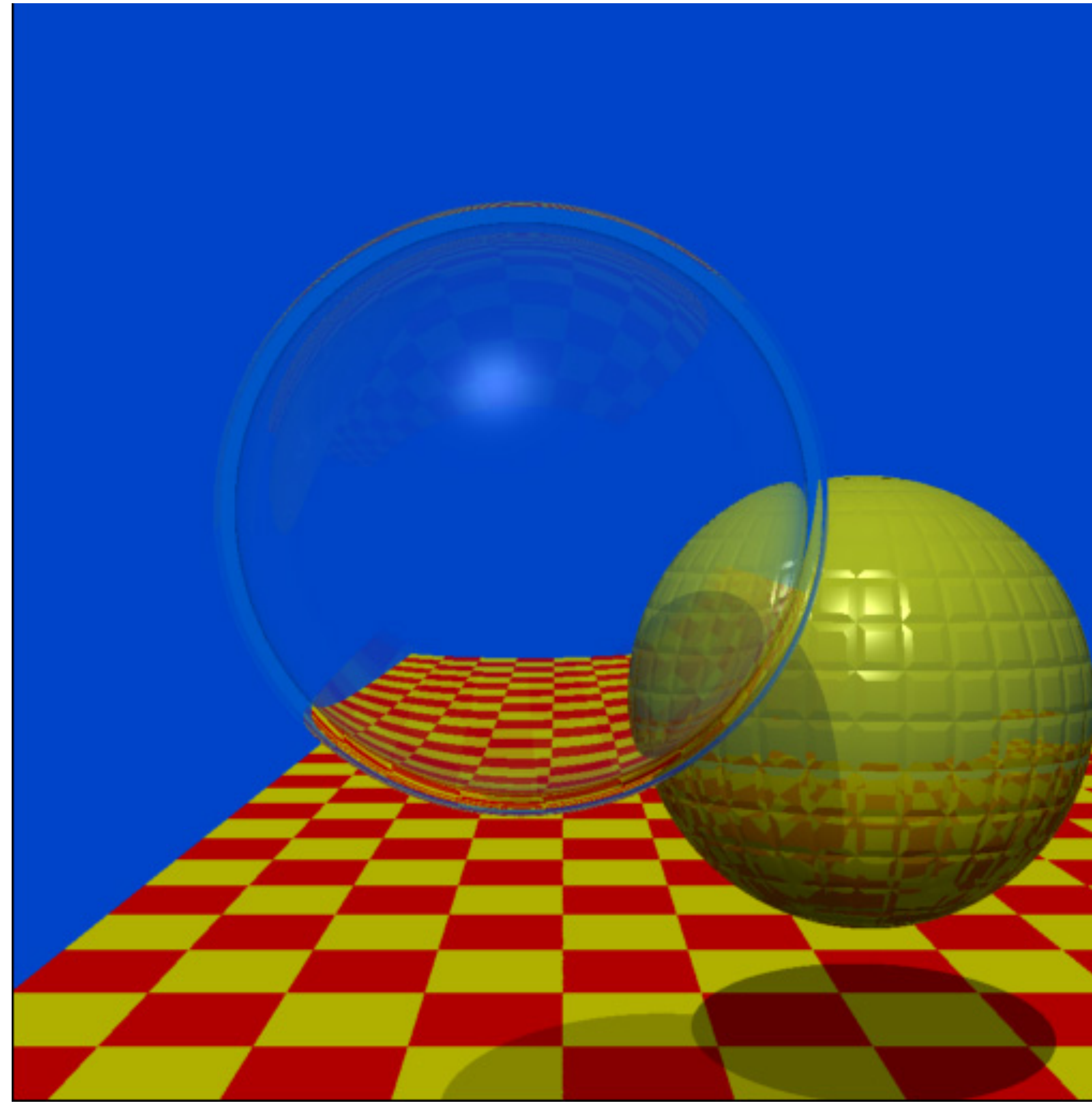
Recursive Ray Tracing



- At intersection points, trace new rays
  - Towards light sources - shadows
  - In reflection direction - reflections
  - In refraction direction - refractions

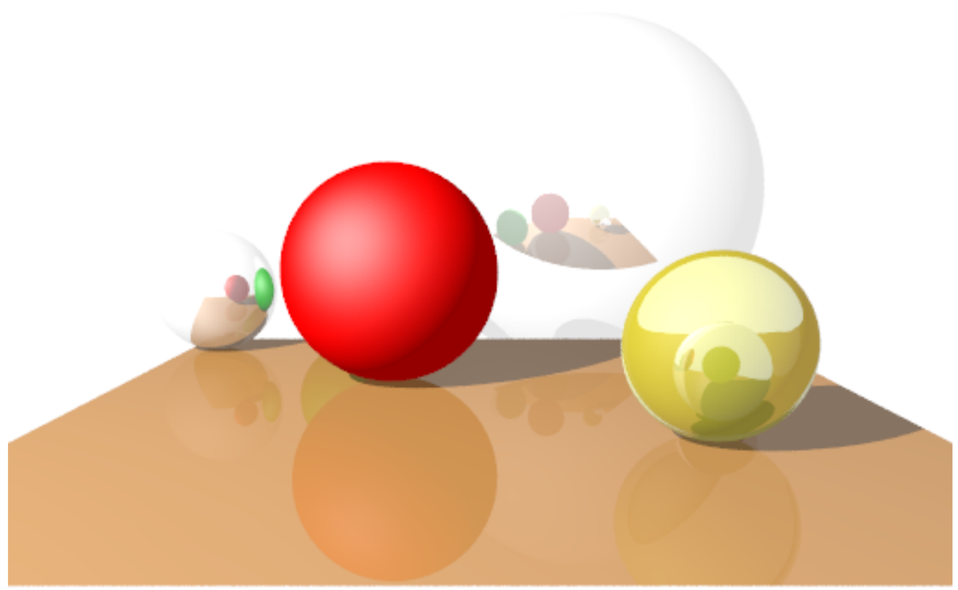
Recursive Ray Tracing

Turner Whitted 1979

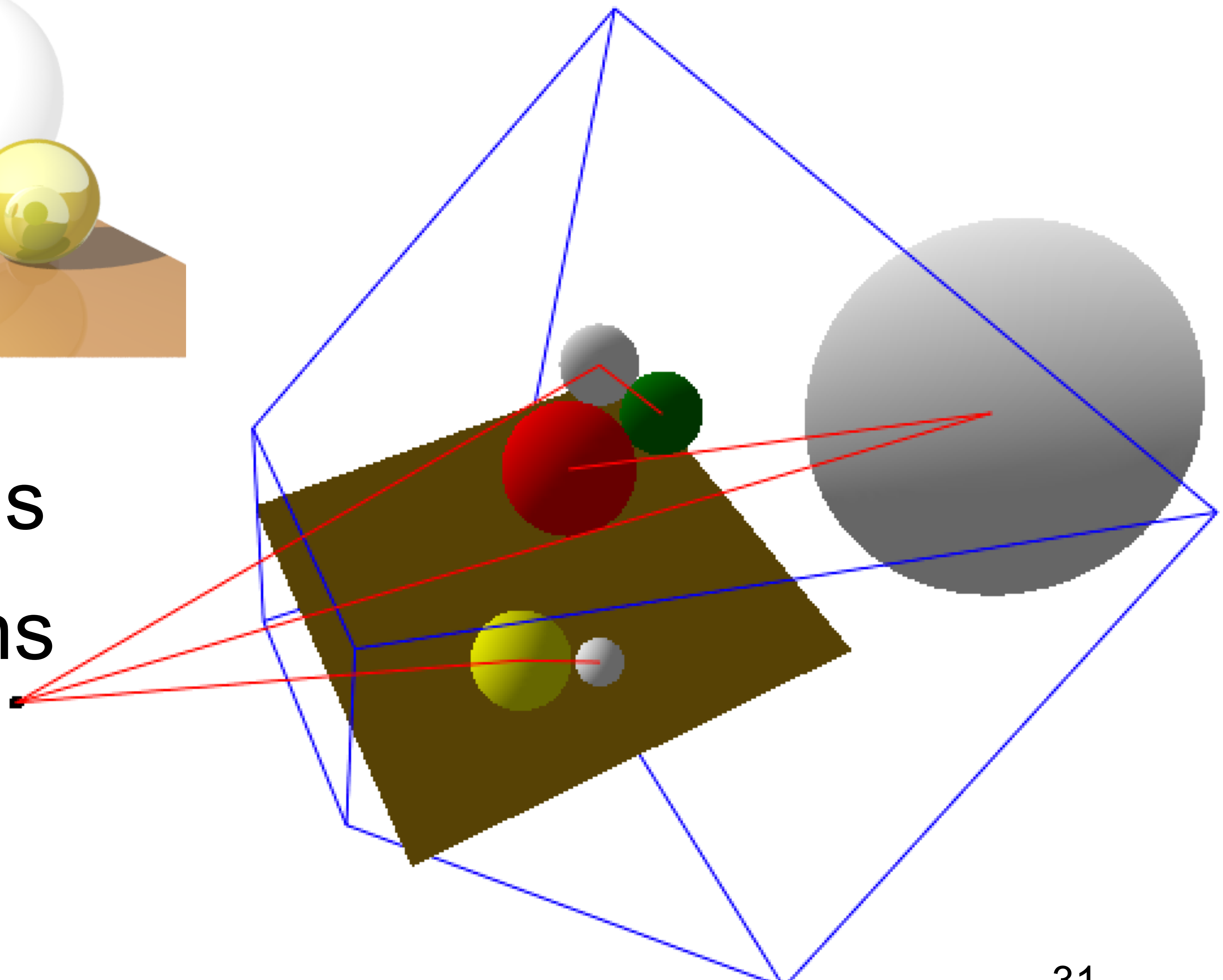


- Reflections
- Refractions
- Shadows

Recursive Ray Tracing



- Reflections
- Refractions
- Shadows



SOL Demo

Real-time ray tracing

https://www.youtube.com/watch?v=KJRZTkttgLw

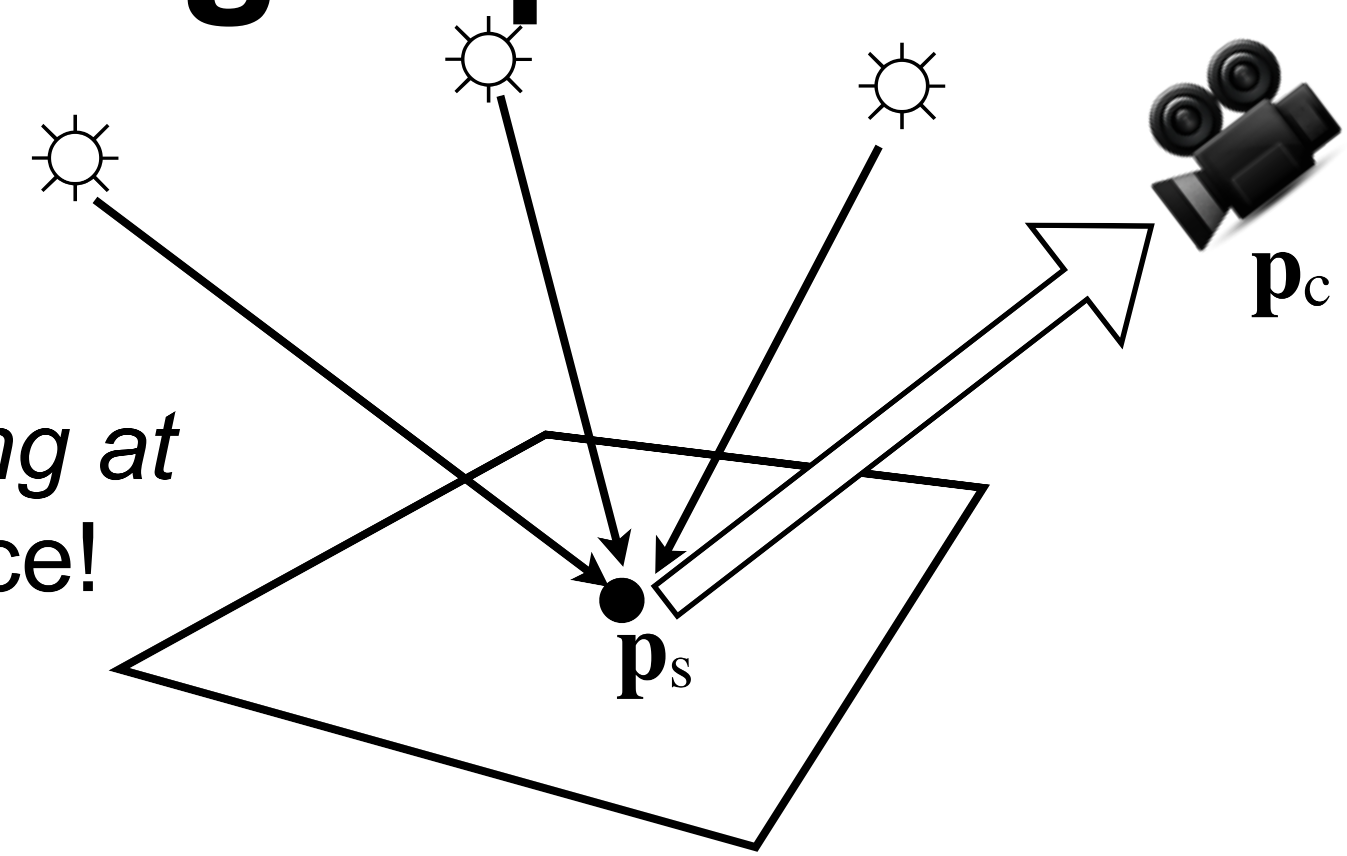
https://www.youtube.com/watch?v=J3ue35ago3Y

The Rendering Equation

Beyond the Phong shading model

MD15

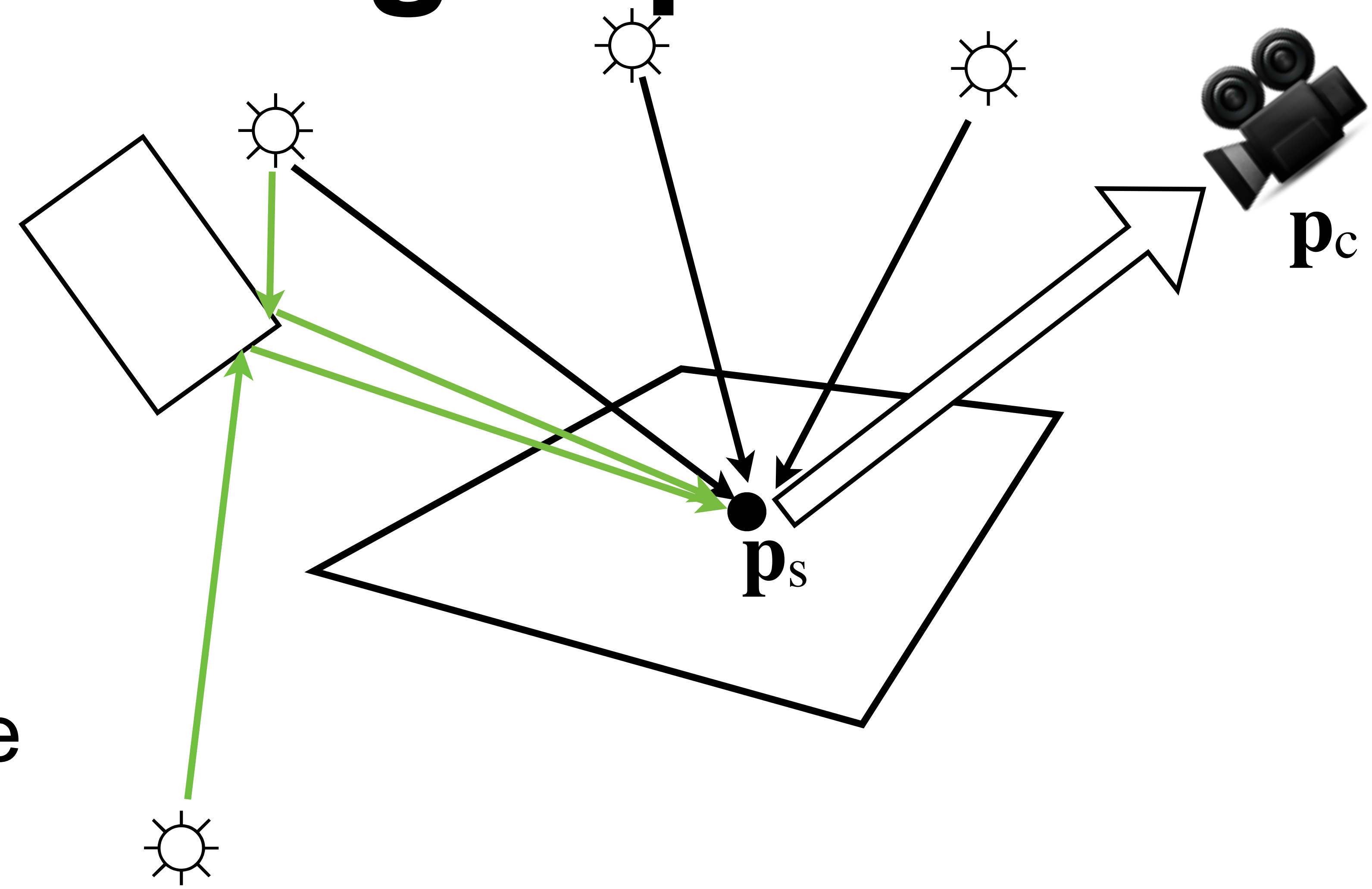
The Rendering Equation



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)

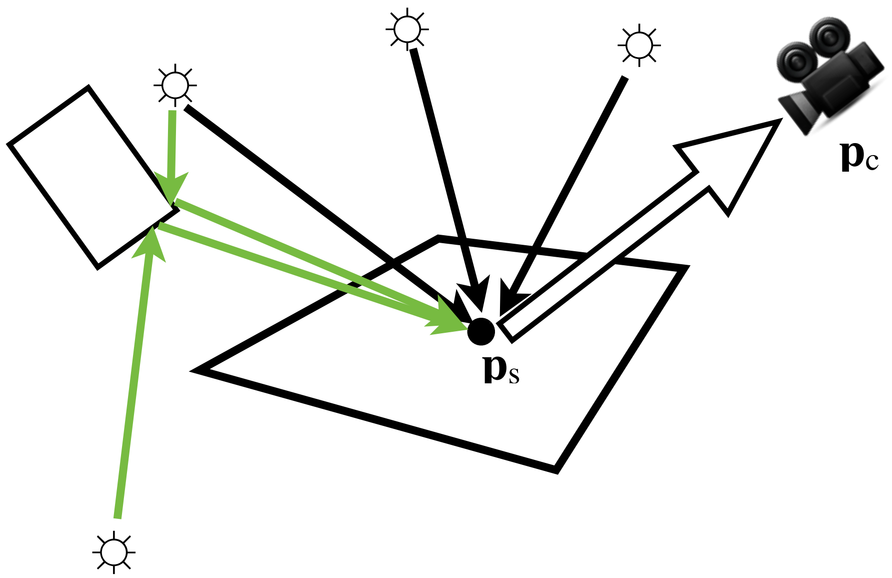
The Rendering Equation



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

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

Recursive Equation

Unfold recursion in the rendering equation one step

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

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

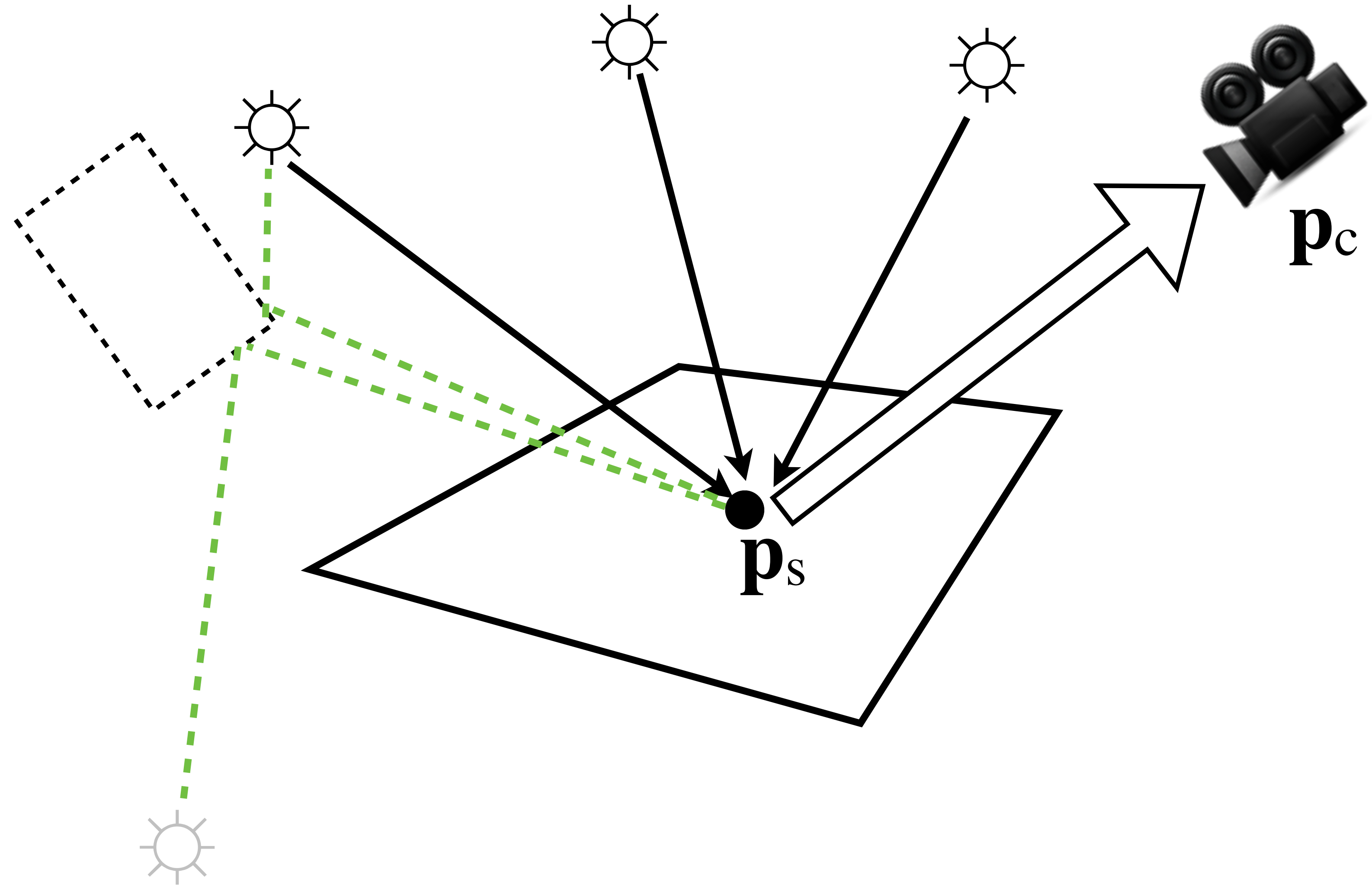
Recursive Equation

Unfold recursion in the rendering equation

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

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

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

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

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

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

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 $\int_L \rightarrow \sum_L$

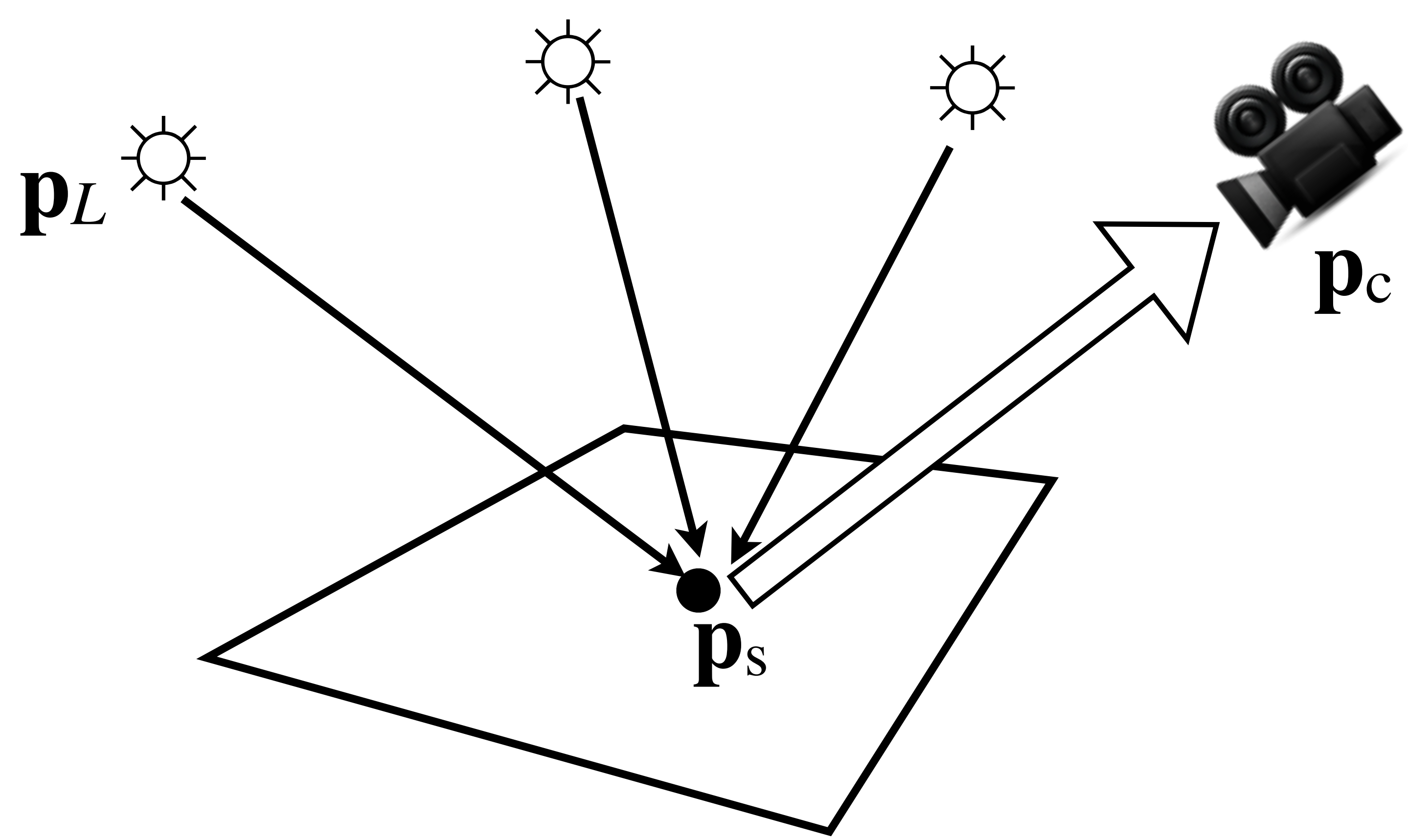
Light intensity of visible light: $v(\mathbf{p}_s, \mathbf{p}_L)\epsilon(\mathbf{p}_s, \mathbf{p}_L) = L(\mathbf{p}_L)$

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

Local Reflection Model

Material properties Light intensity

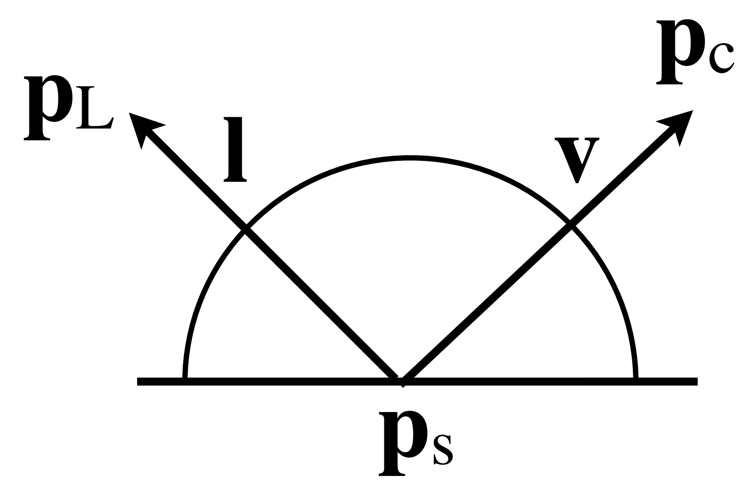
$$I(\mathbf{p}_c, \mathbf{p}_s) = \sum_L \boxed{\rho(\mathbf{p}_c, \mathbf{p}_s, \mathbf{p}_L)} L(\mathbf{p}_L)$$



Material Properties: BRDF

- Bidirectional Reflection Distribution Function

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



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

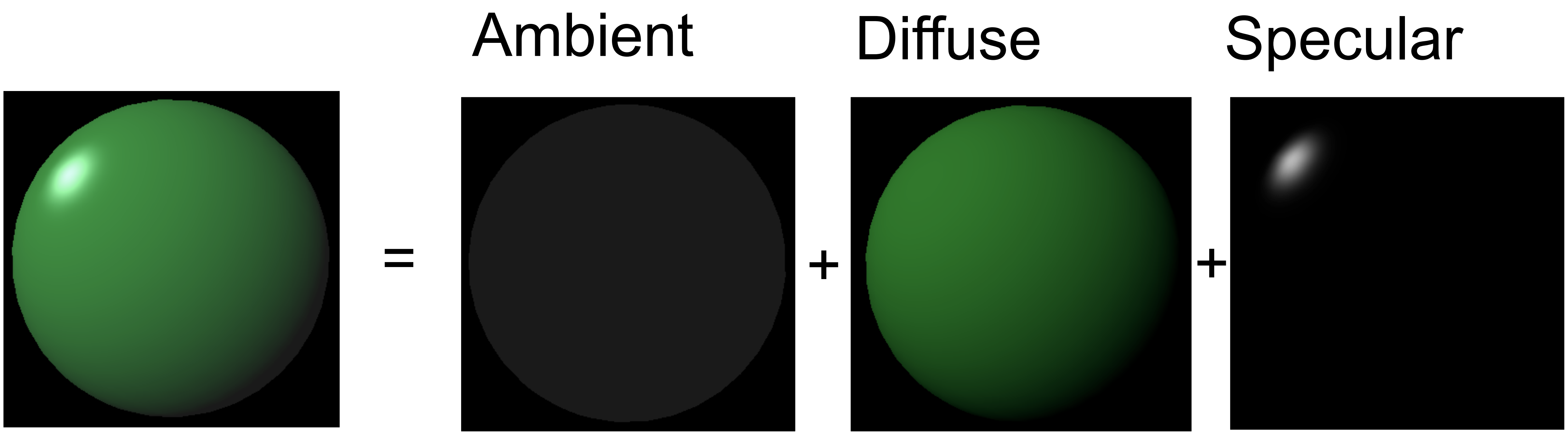
Example: Phong shading

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

Phong Model

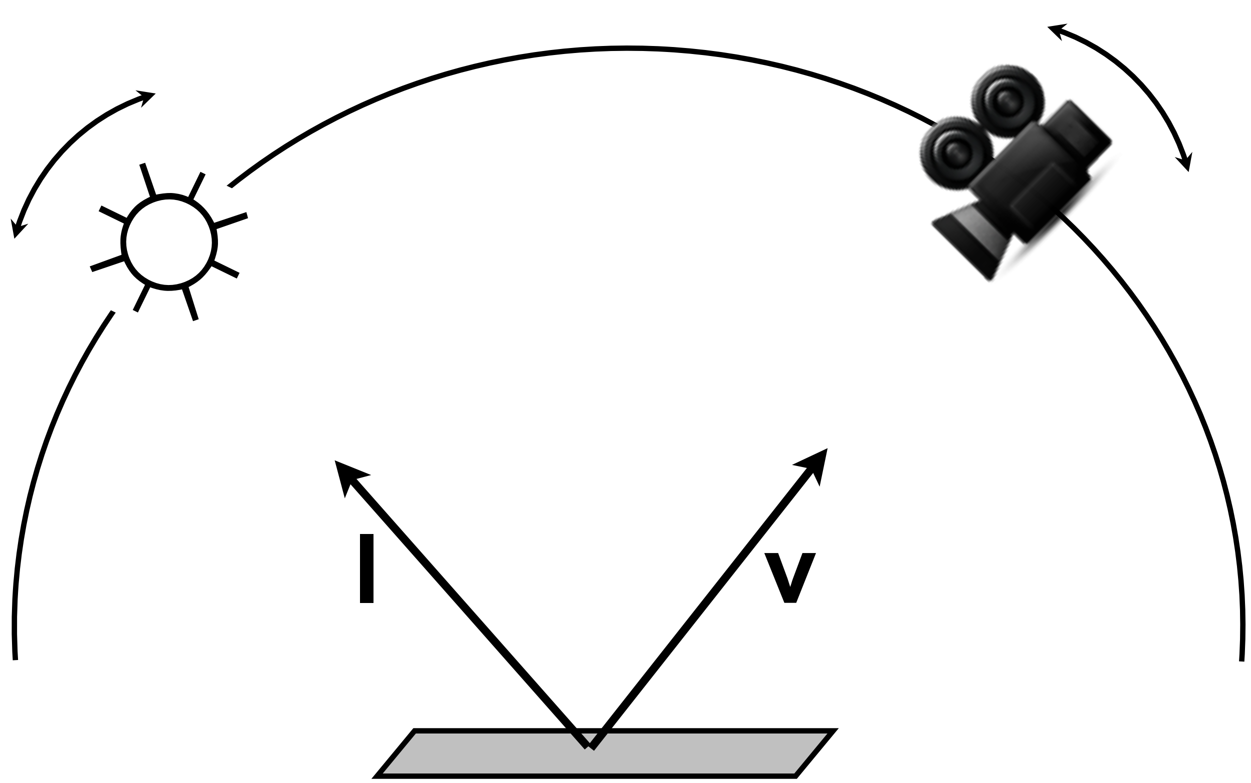
- Combine the three terms

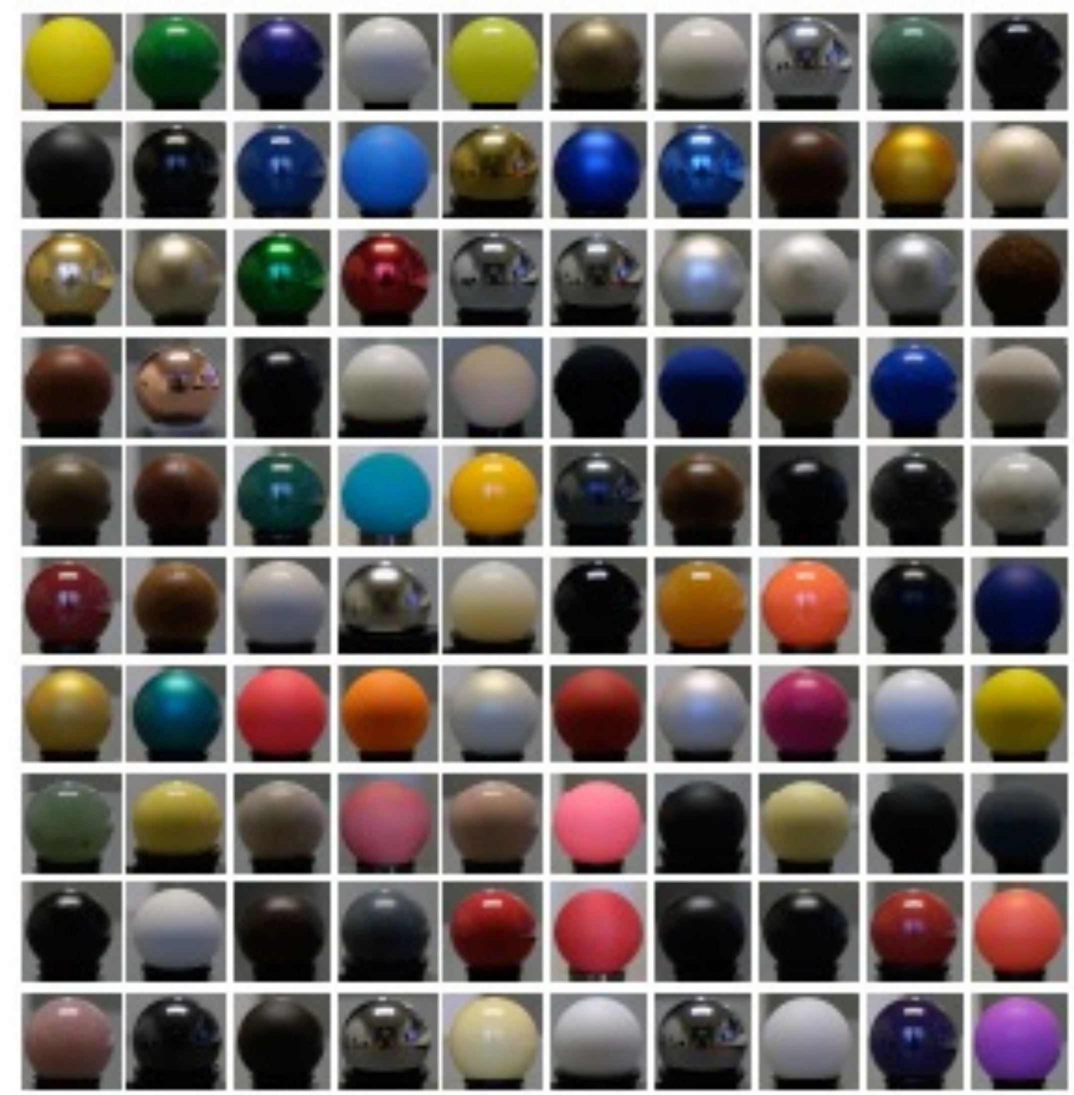
$$I = k_a L_a + k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max((\mathbf{r} \cdot \mathbf{v})^{\alpha}, 0)$$



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
$\mathbf{l}$ and $\mathbf{v}$





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

Rendering Equation Summary

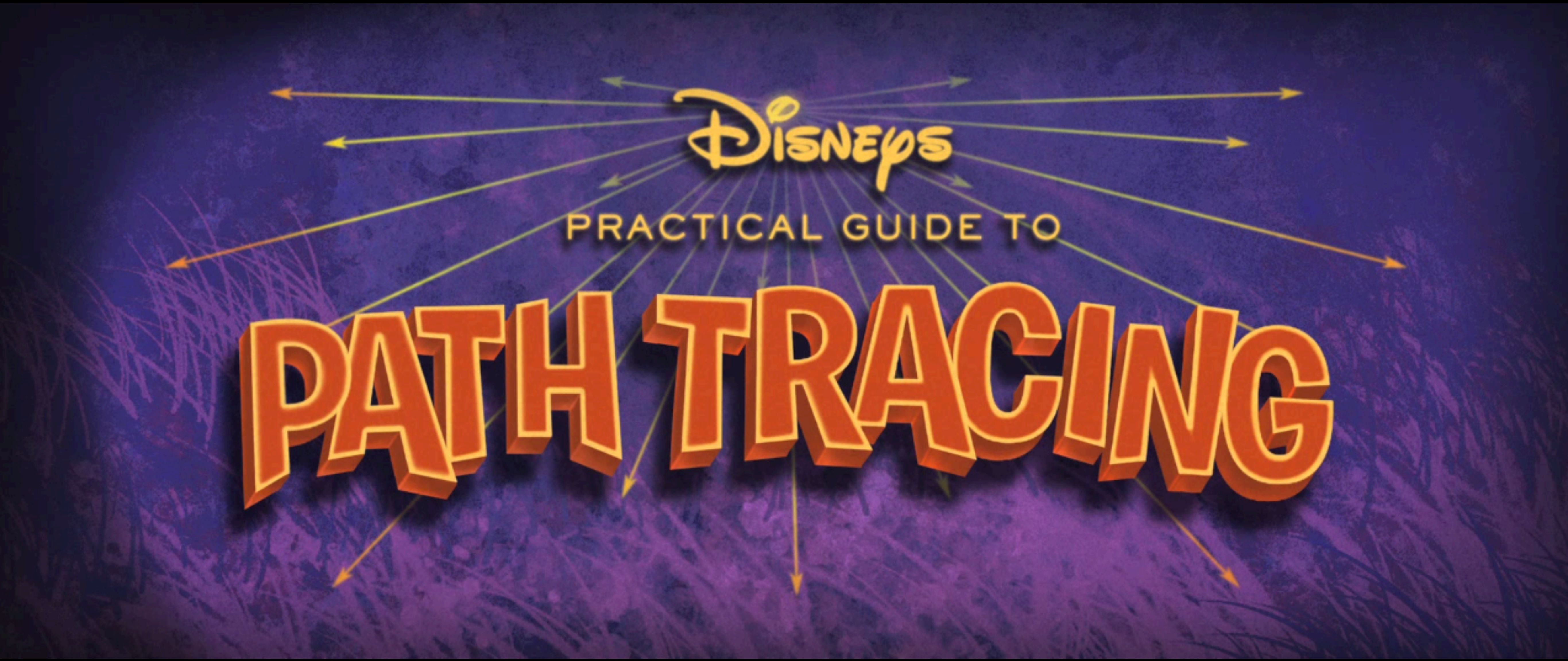
- The general equation is very complex
  - Active research on how to solve it approximately
- Simplifications leads to plausible approximations, such as Phong

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)

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



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

Fake or Photo

https://area.autodesk.com/fakeorfoto/

Procedural Techniques

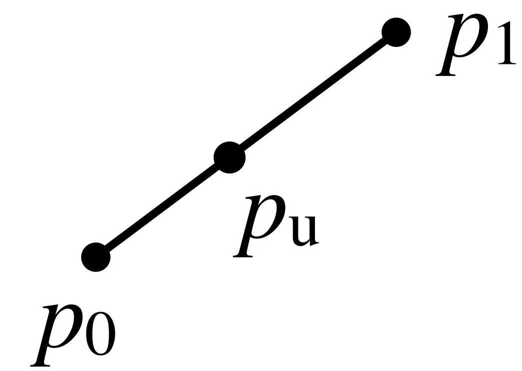
Procedural Techniques

- Instead of describing detail with textures, use mathematical functions
  - Clouds, smoke, fire
- Human visual system very good at detecting patterns/repetitions
- Add randomness for more realism!

Bi-linear interpolation

- Remember linear interpolation

$$p_u = (1-u)p_0 + up_1$$

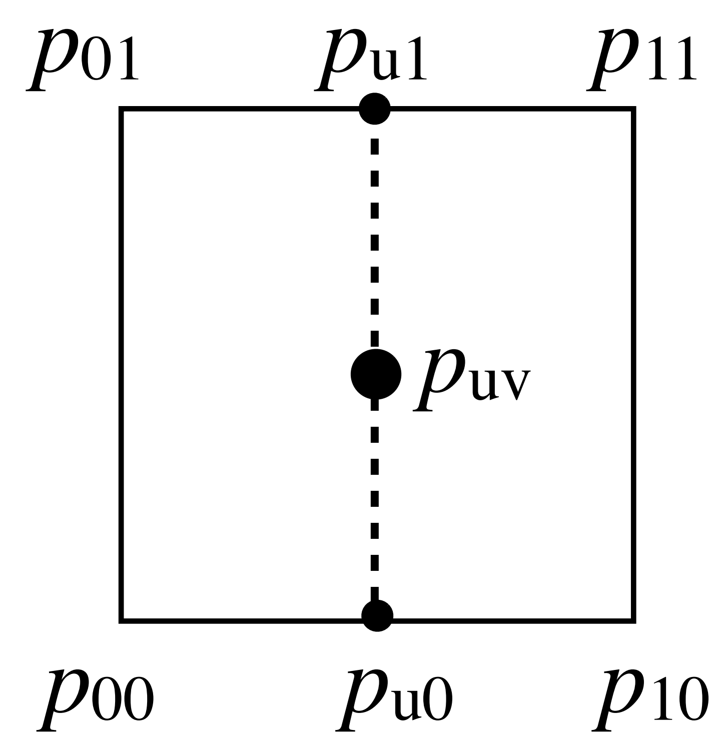


- Bilinear interpolation:

$$p_{u0} = (1-u)p_{00} + up_{10}$$

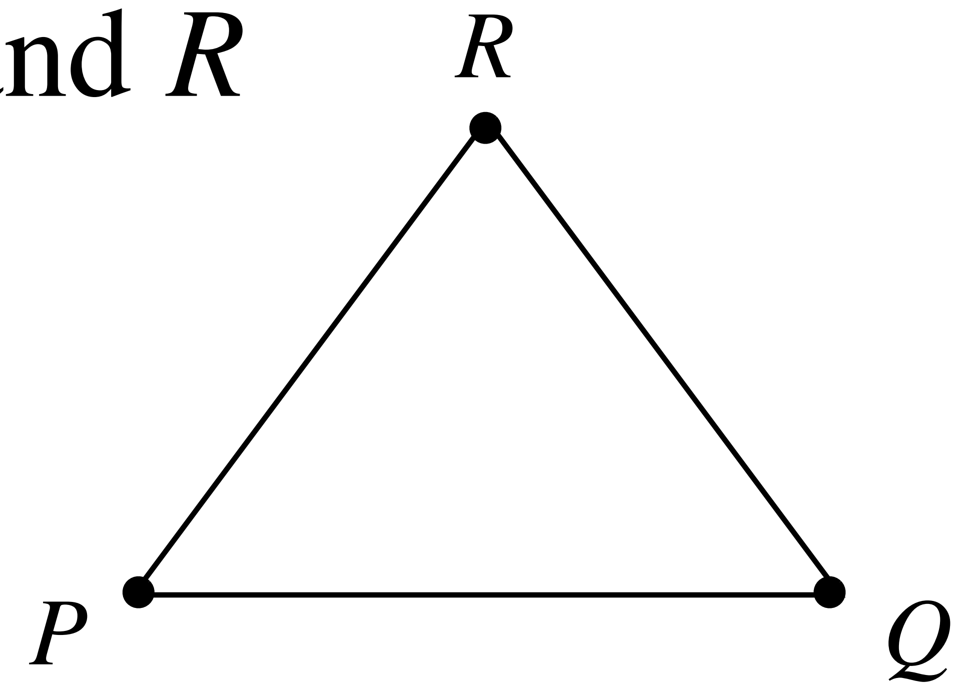
$$p_{u1} = (1-u)p_{01} + up_{11}$$

$$p_{uv} = (1-v)p_{u0} + vp_{u1}$$



Triangle

- Defined by three points
  P, *Q* and *R*



- Points inside triangle:

$$wP + uQ + vR$$

$$u + v + w = 1$$

$$u, v, w \geq 0$$

- *u*,*v*,*w* : barycentric coordinates

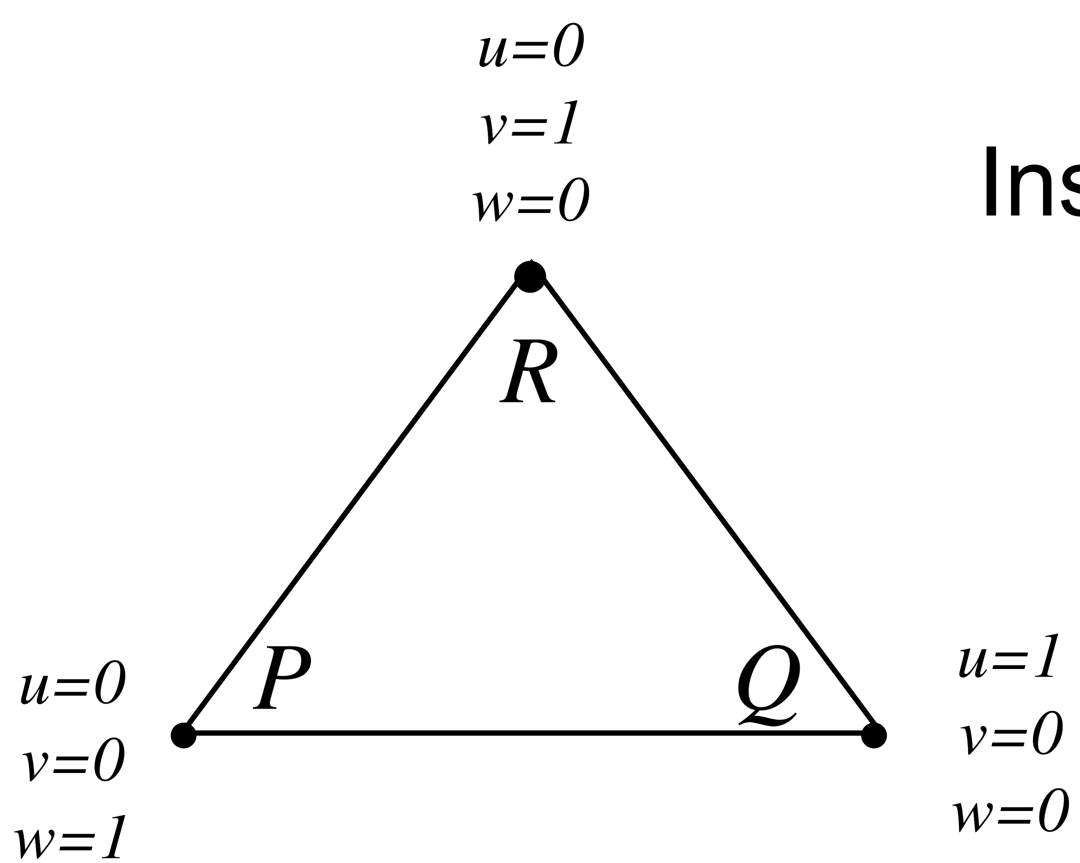
Barycentric Interpolation

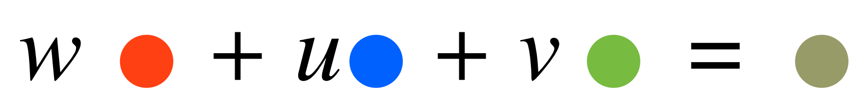
- *u,v,w* : barycentric coordinates

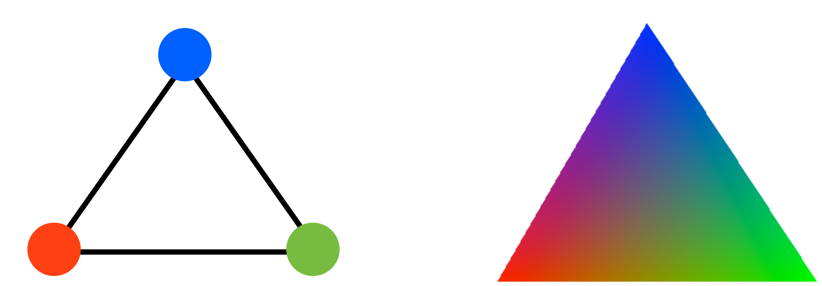
$$wP + uQ + vR$$

$$u + v + w = 1$$

Inside triangle if: $u, v, w \geq 0$

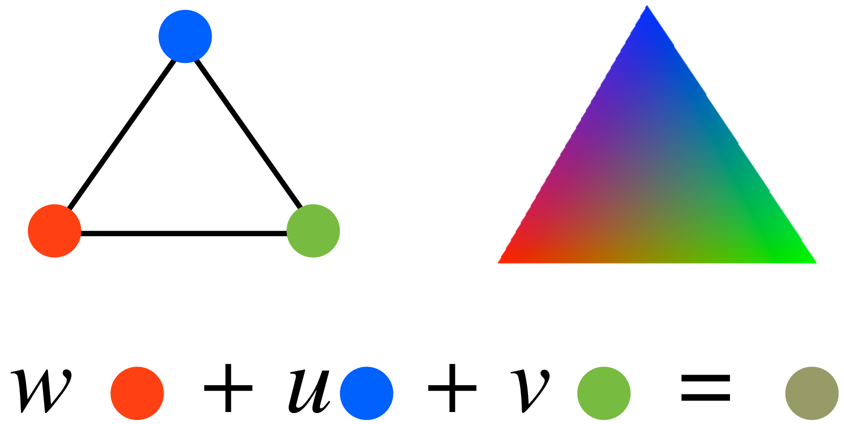


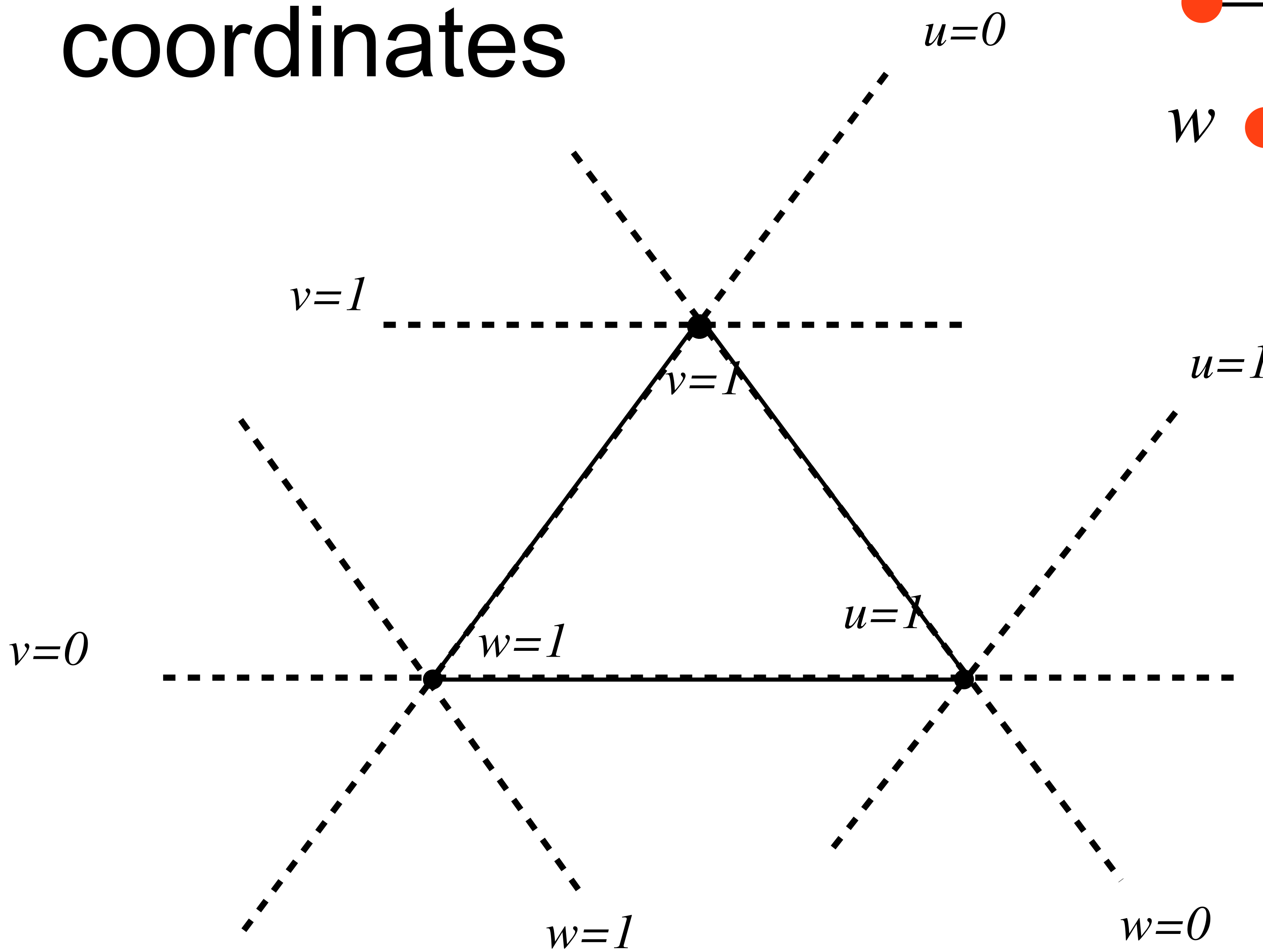




Barycentric Interpolation

- *u*,*v*,*w* : barycentric coordinates

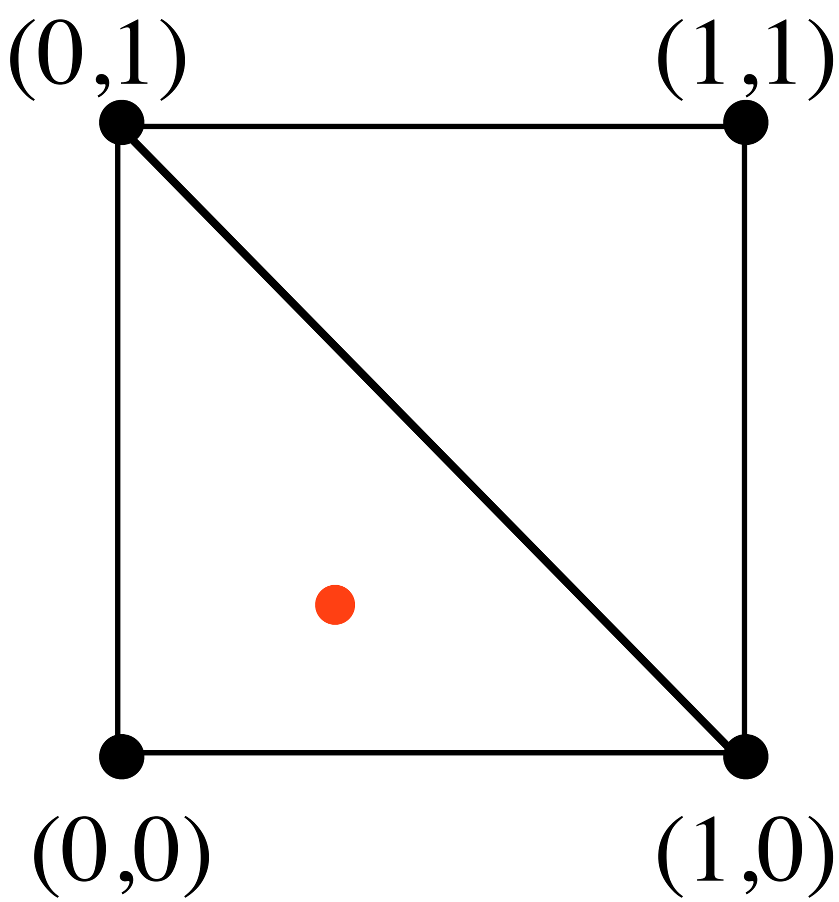


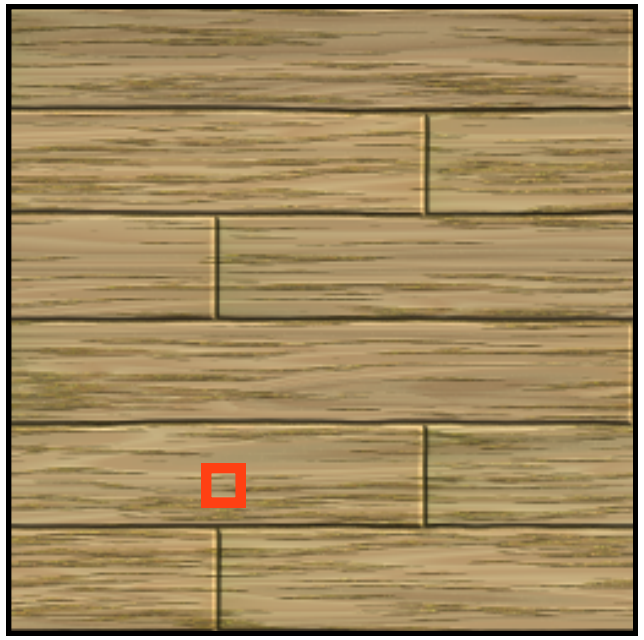


$$u + v + w = 1$$

Texture Coordinates

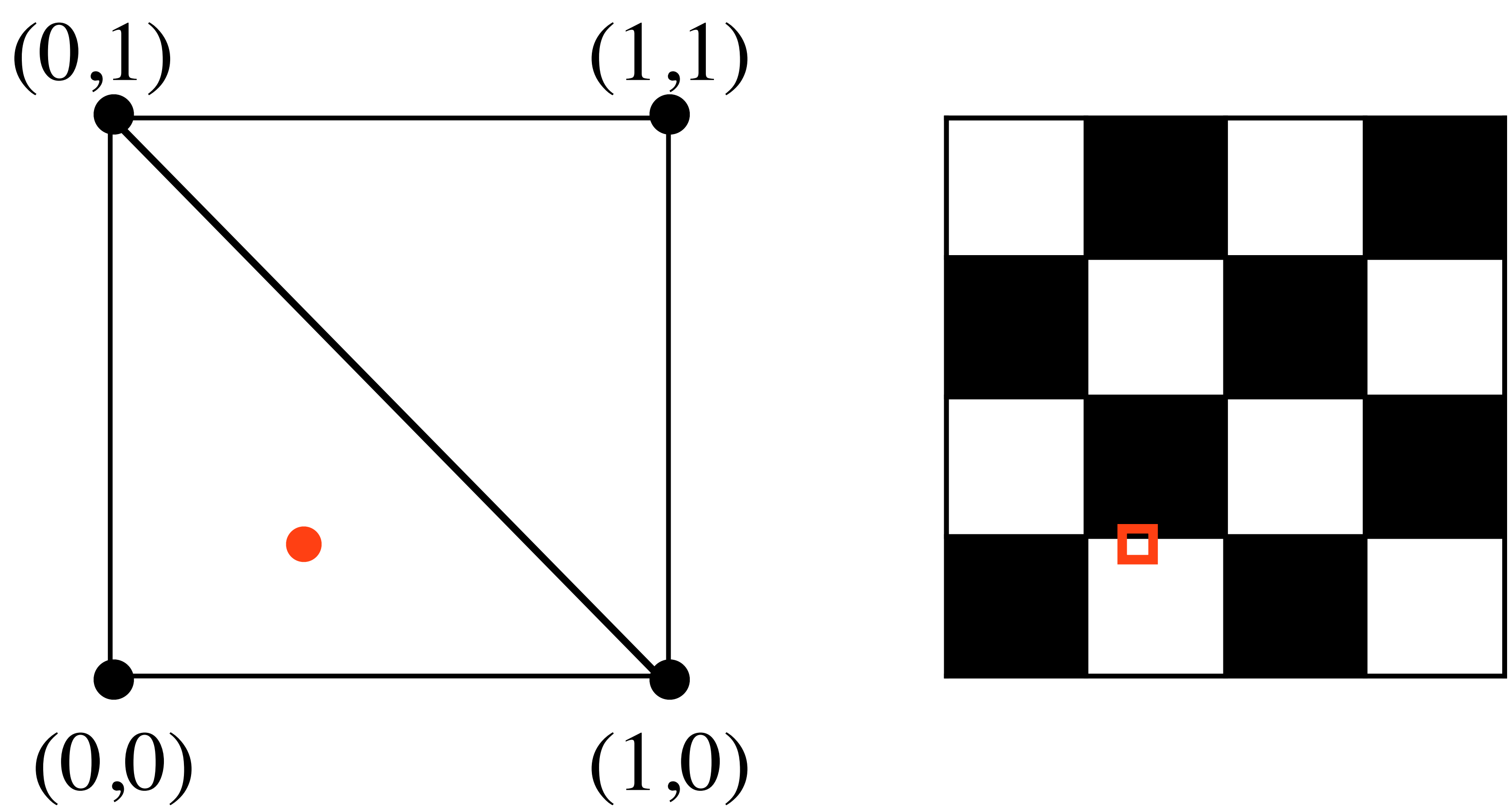
- Specify tex coordinate at each vertex
  - Hardware interpolates texCoord for each pixel
- Use coordinate to lookup in texture





Procedural Texturing

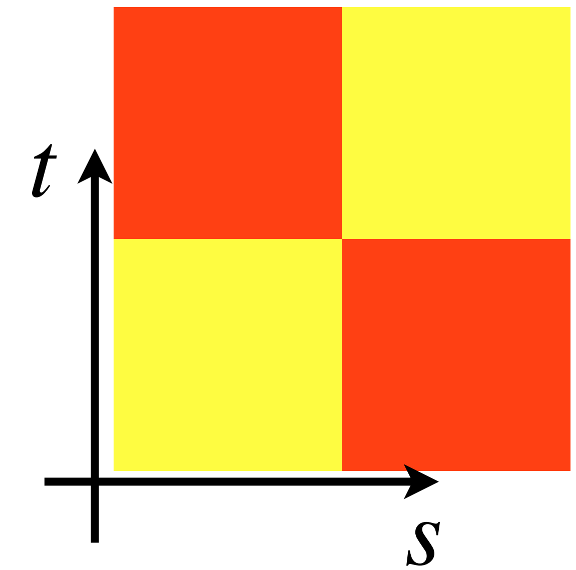
- Use coordinate to define a function that computes the color
  - No need to store (large) texture in memory!



Example: Checkerboard

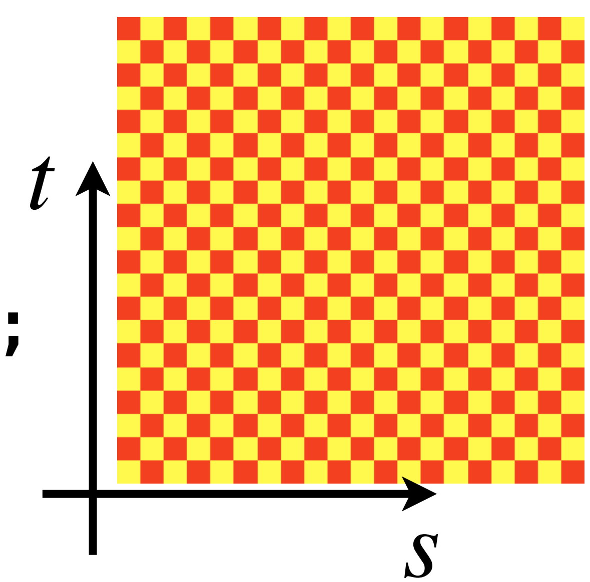
```
float s = texCoord.x;
float t = texCoord.y;
vec3 red    = vec3(1,0,0);
vec3 yellow = vec3(1,1,0);
vec3 c = (s < 0.5) ^^ (t < 0.5) ? red : yellow;
fColor = vec4(c,1.0);
```

A	B	A^^B (XOR)
0	0	0
1	0	1
0	1	1
1	1	0



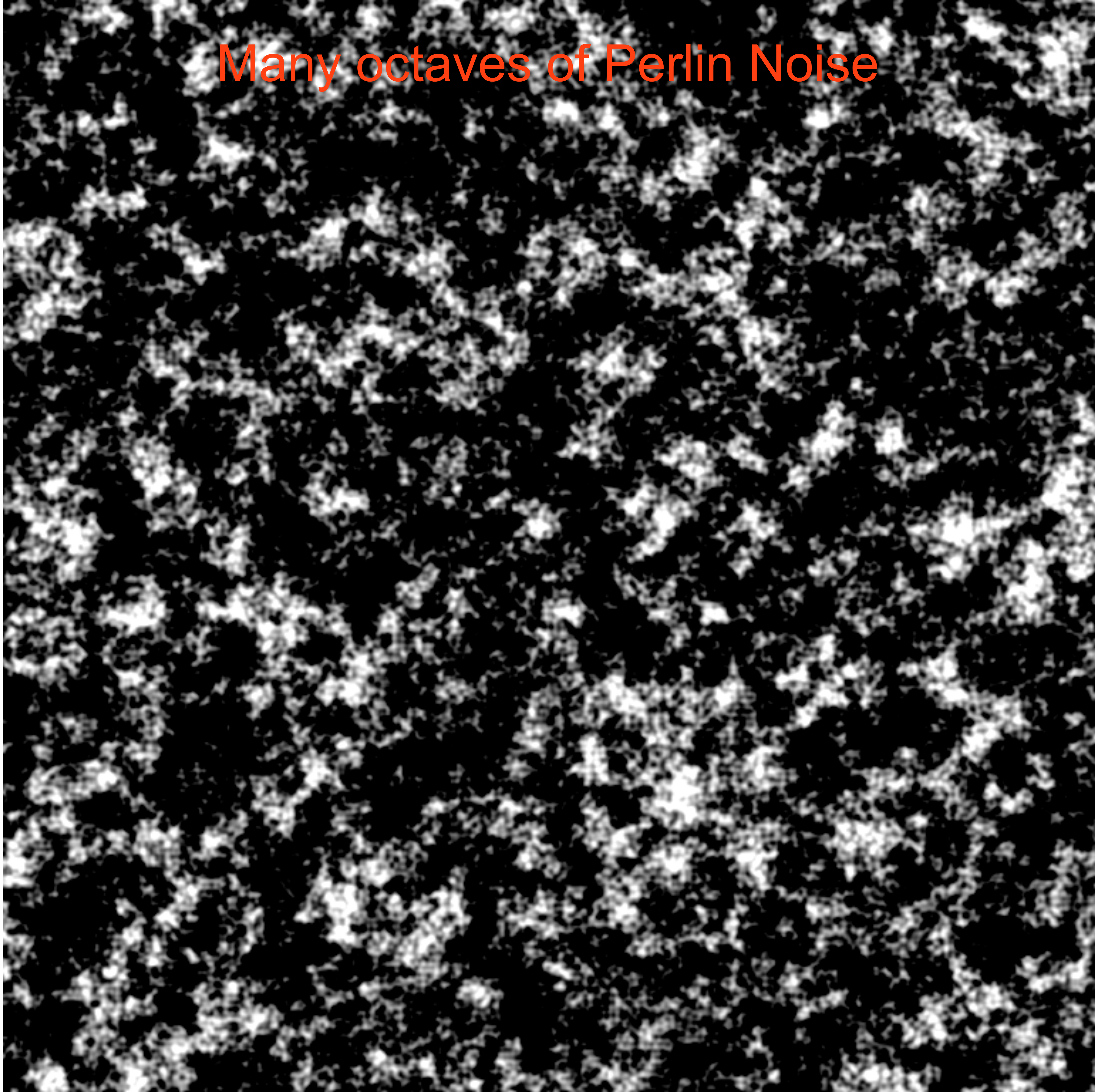
Generalize to higher frequencies

```
float freq = 10.0;
vec3 c =    (mod(freq*s,1) < 0.5)
         ^^ (mod(freq*t,1) < 0.5) ? red : yellow;
```



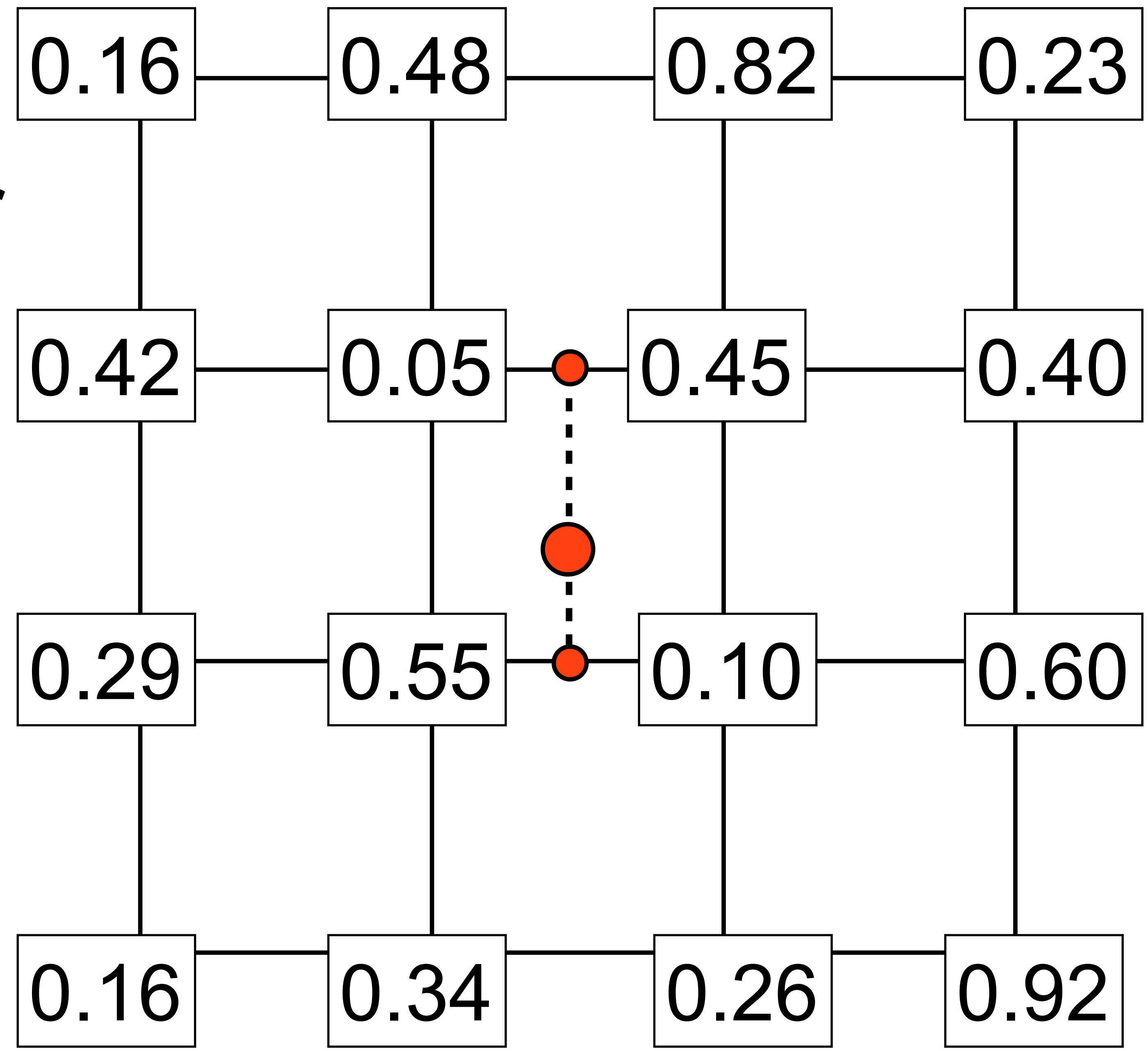
Noise

- A controlled random primitive
  - Bandlimited
  - Repeatable (same input value gives same output)
- Smooth interpolation between random values
  - Assign a value to each integer
  - Interpolate between integer values


Many octaves of Perlin Noise

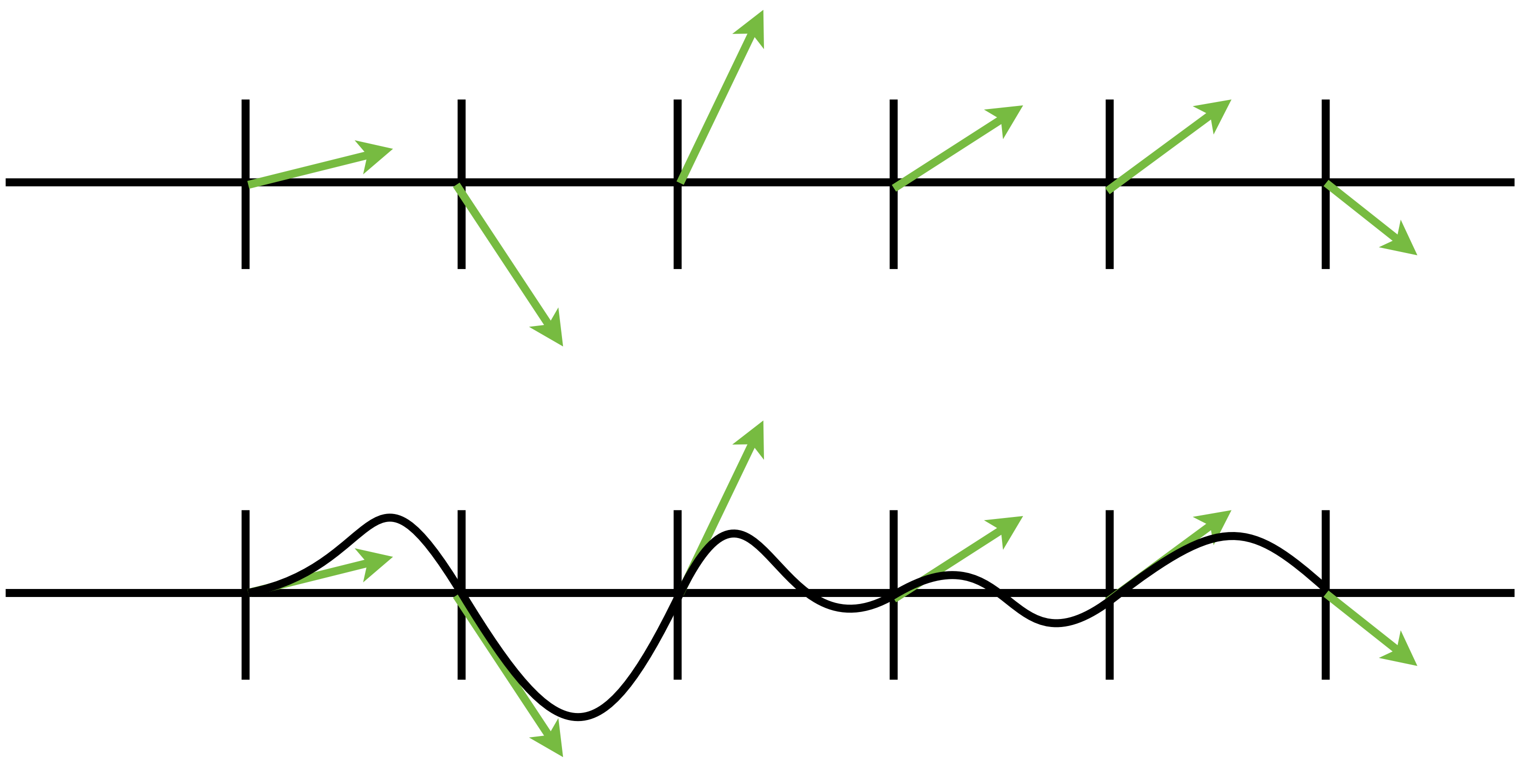
Example: 2D Value Noise

- Assign random value to each vertex in integer grid
- Bi-linear interpolation between values
- Smooths out noise

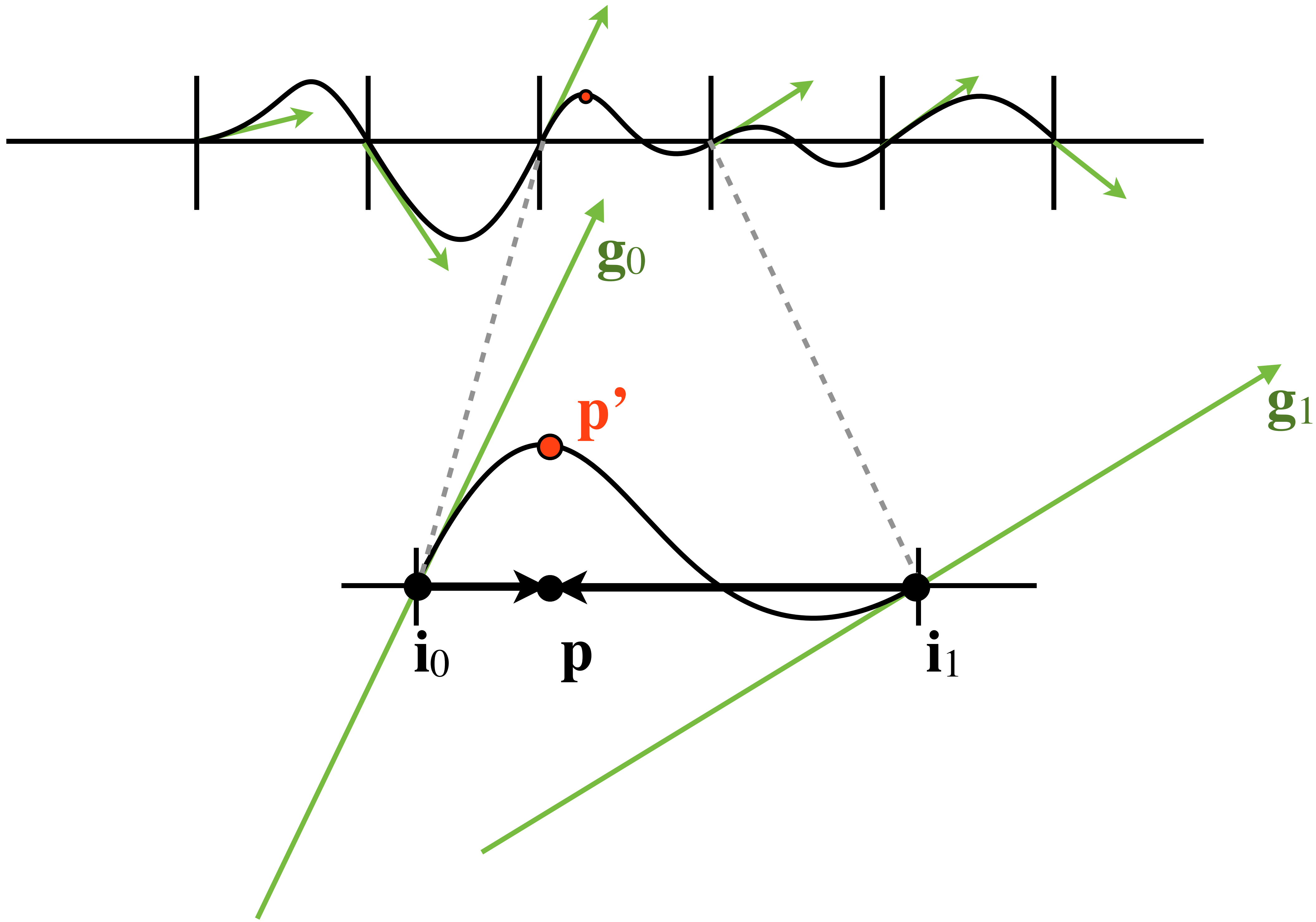


Perlin Noise

- Assign random gradient to each grid point
- Smooth interpolation between gradients



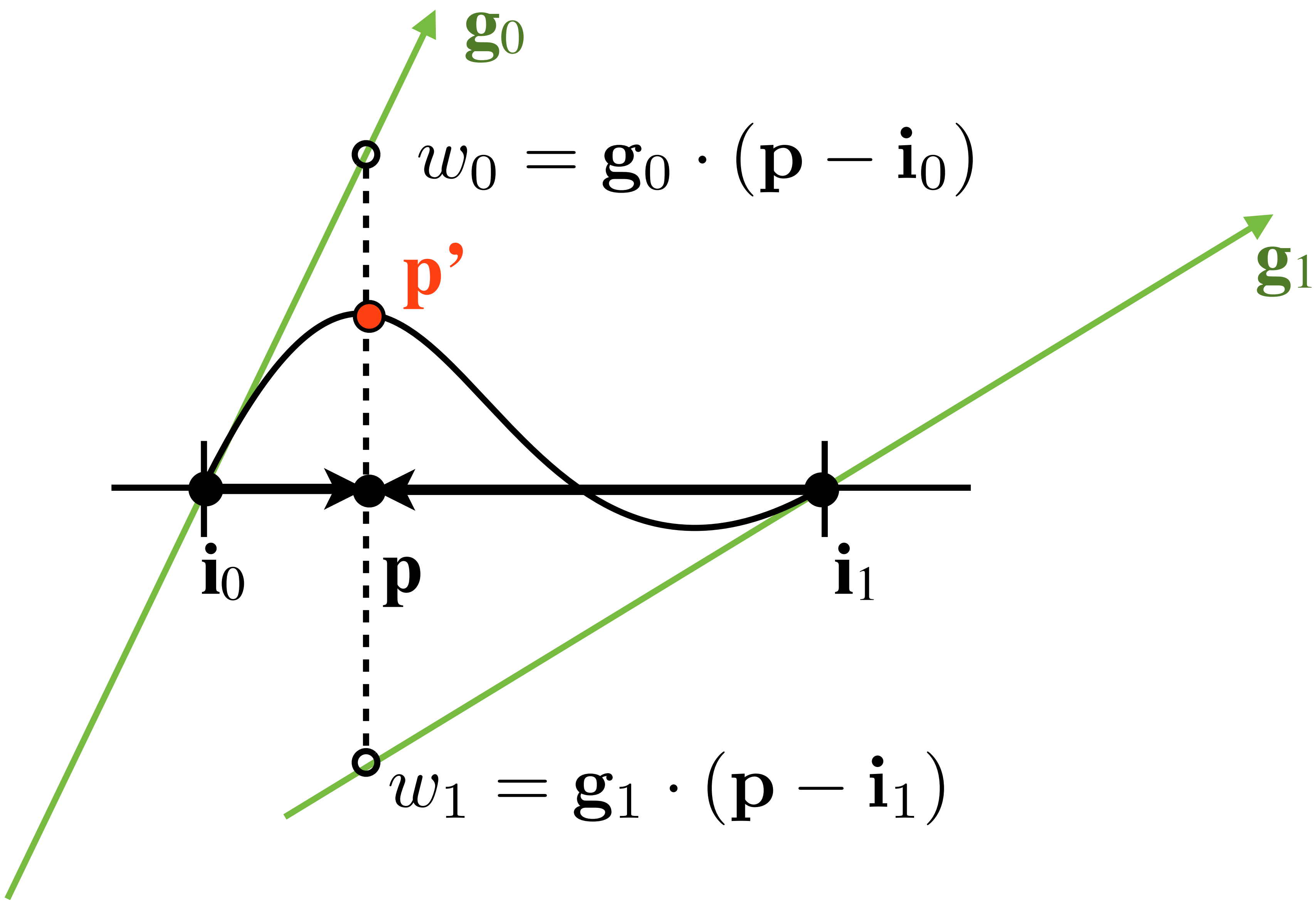
Perlin Noise



What is the value at **p'** ?

Perlin Noise

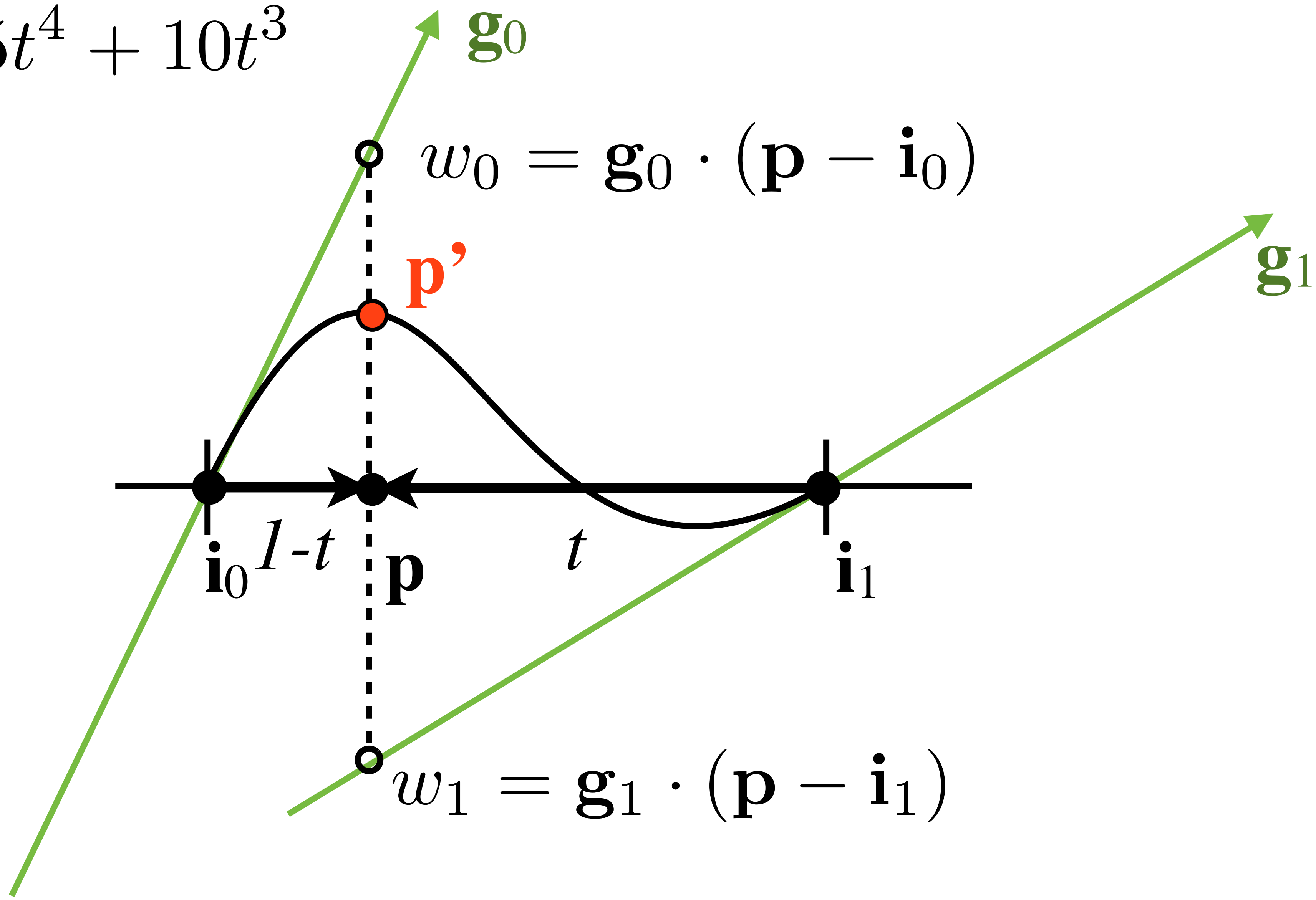
Find weights (scalar product between gradient $\mathbf{g}$ and $\mathbf{p}$-$\mathbf{i}$)



Perlin Noise

Smooth interpolation between weights

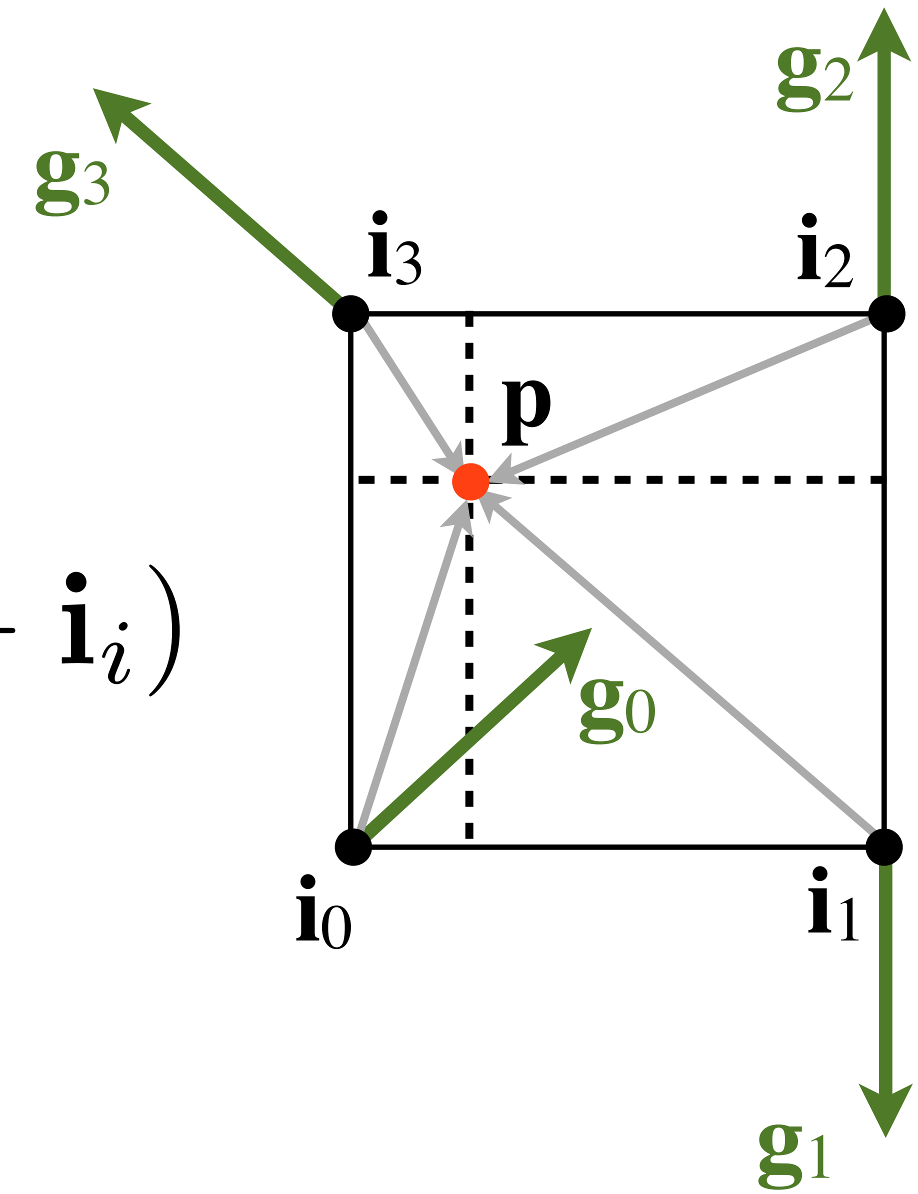
$$f(t) = 6t^5 - 15t^4 + 10t^3$$



$$\boxed{\mathbf{p}' = f(|\mathbf{p} - \mathbf{i}_0|)w_0 + f(|\mathbf{p} - \mathbf{i}_1|)w_1}$$

Perlin Noise

- Assign gradient vector, $\mathbf{g}$, to each lattice point $\mathbf{i}$
- Compute gradients weights: $w_i = \mathbf{g}_i \cdot (\mathbf{p} - \mathbf{i}_i)$
- Blend between weights
$$\sum_i f(|\mathbf{p} - \mathbf{i}_i|) w_i$$
  - Use smooth interpolation between the weights



$$f(t) = 6t^5 - 15t^4 + 10t^3$$

read more at http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

Perlin Noise

- Perlin noise can be efficiently implemented
  - No need to store large grid of noise values
- Extends to 3D & 4D

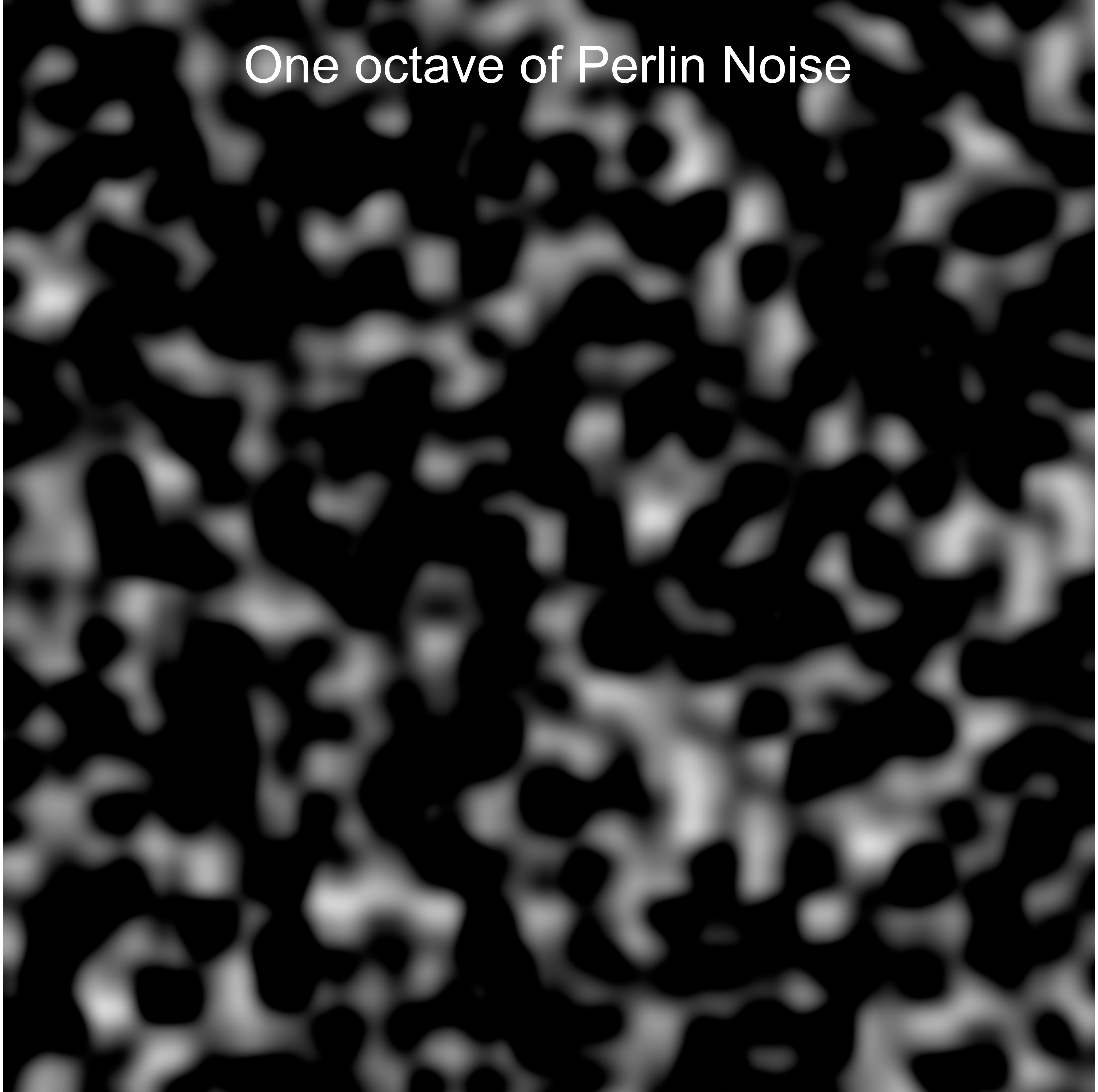
Paper: http://mrl.nyu.edu/~perlin/paper445.pdf

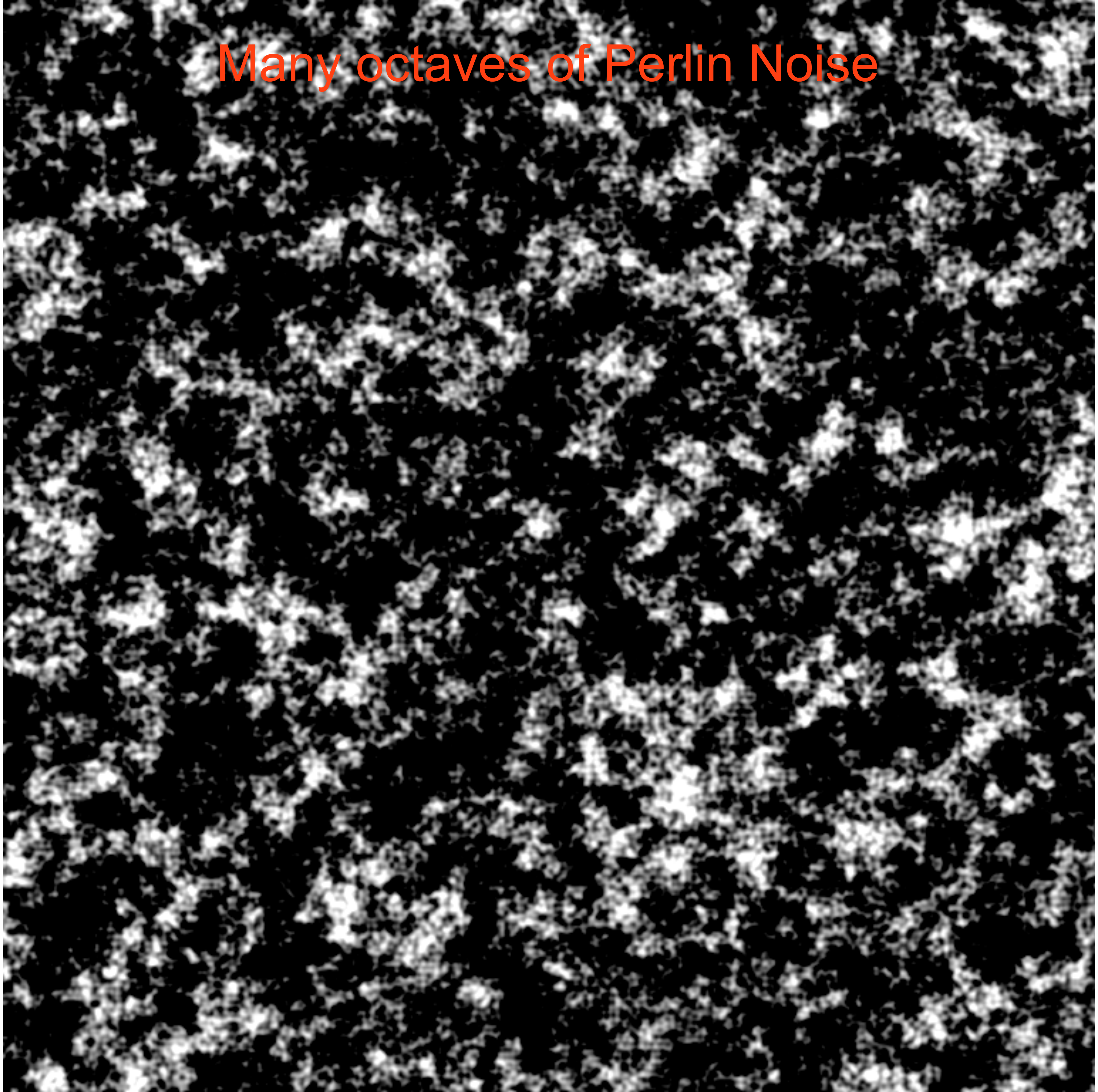
Example code: http://ispc.github.com/downloads.html
Noise example - shows Perlin noise and turbulence

Turbulence

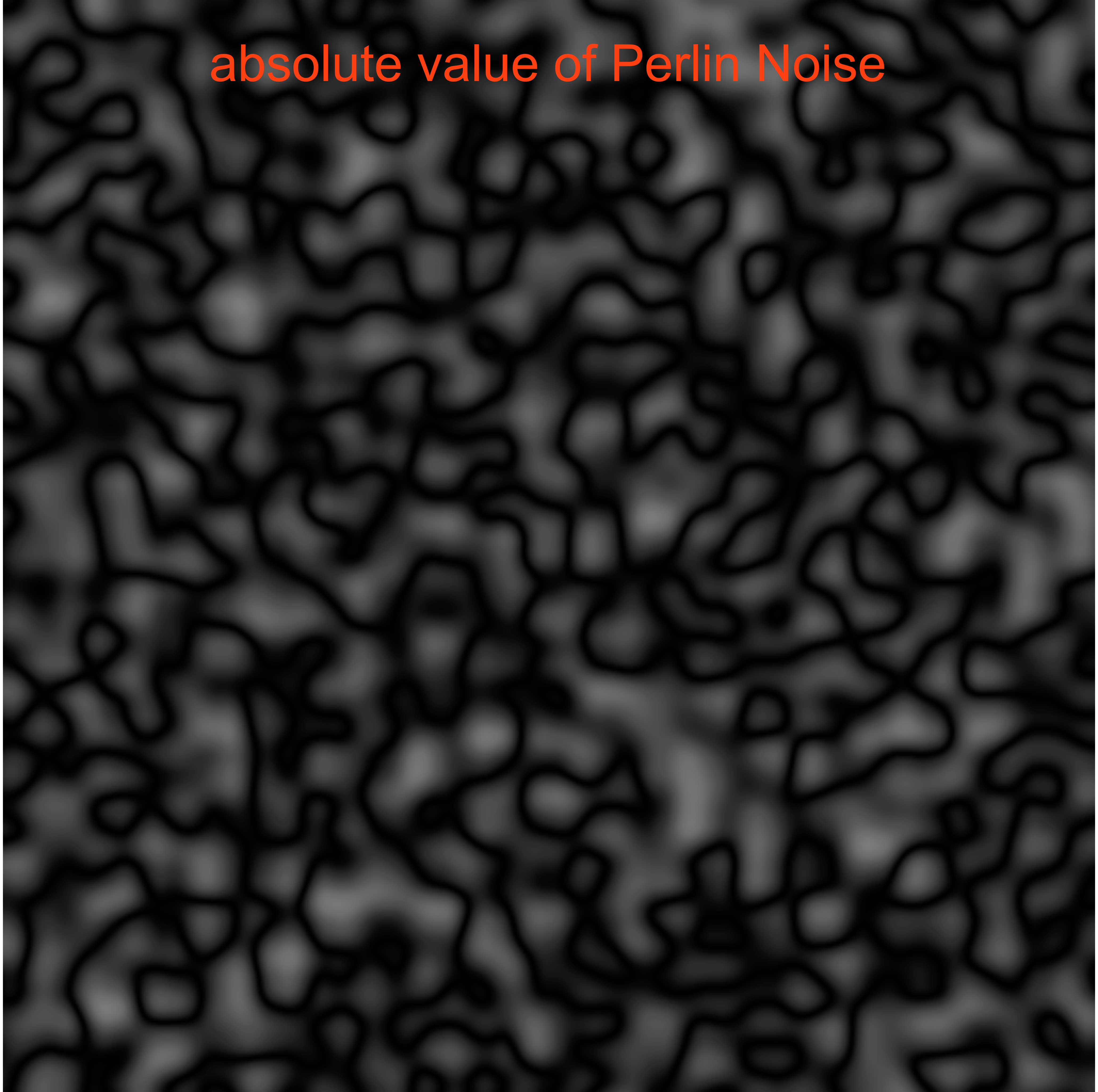
- Sum several octaves of Perlin noise

```
float turbulence(vec3 pos, const int octaves)
{
    float sum = 0;
    float omega = 0.6;
    float lambda = 1.0;
    float o = 1.0;
    for (int i=0; i<octaves; ++i)
    {
        sum += abs(o * noise(pos*lambda));
        lambda *= 1.99f;
        o *= omega;
    }
    return sum;
}
```

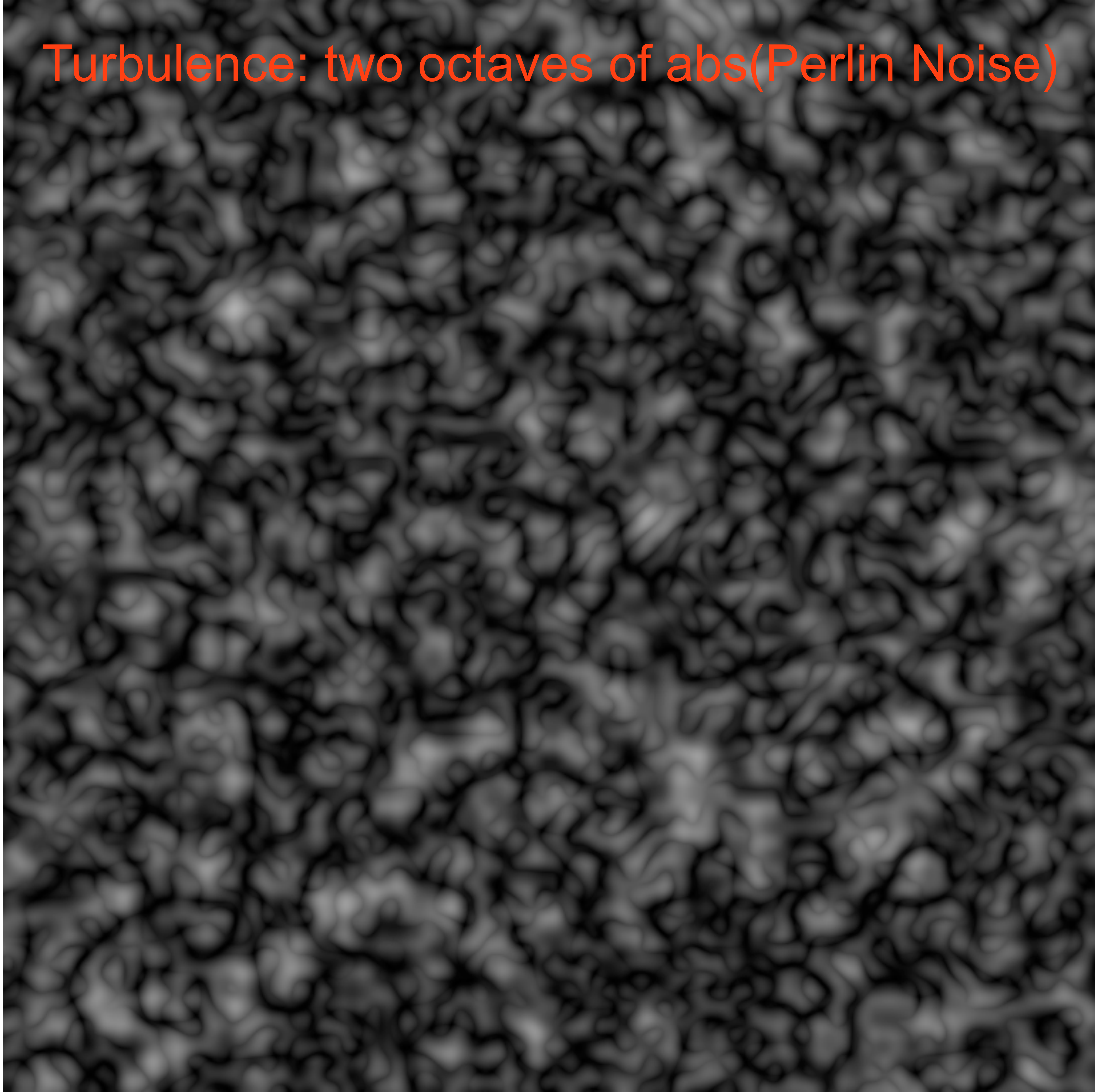

One octave of Perlin Noise


Many octaves of Perlin Noise

absolute value of Perlin Noise



Turbulence: two octaves of abs(Perlin Noise)



Turbulence: many octaves of abs(Perlin Noise)