# CS 300 : Advanced Computer Graphics I
# Ray Tracing

Pushpak Karnick

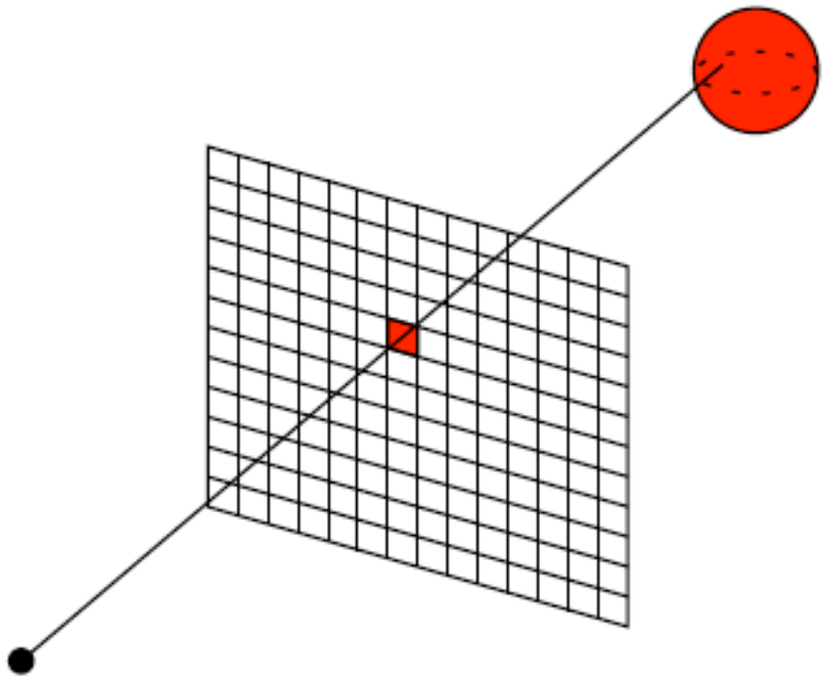
# Quick Recap

- **Rendering:** Visualizing a 3D scene on a 2D screen of pixels
- Two types of rendering methods
  - **Projective Rendering**
    - Computer graphics pipeline (CS 200, CS 250, CS 300)
  - **Ray-traced Rendering**
    - Ray casting / ray tracing / path tracing algorithms

# Ray Tracing Algorithm

For photo-realistic rendering, usually ray tracing algorithms are used: for every pixel

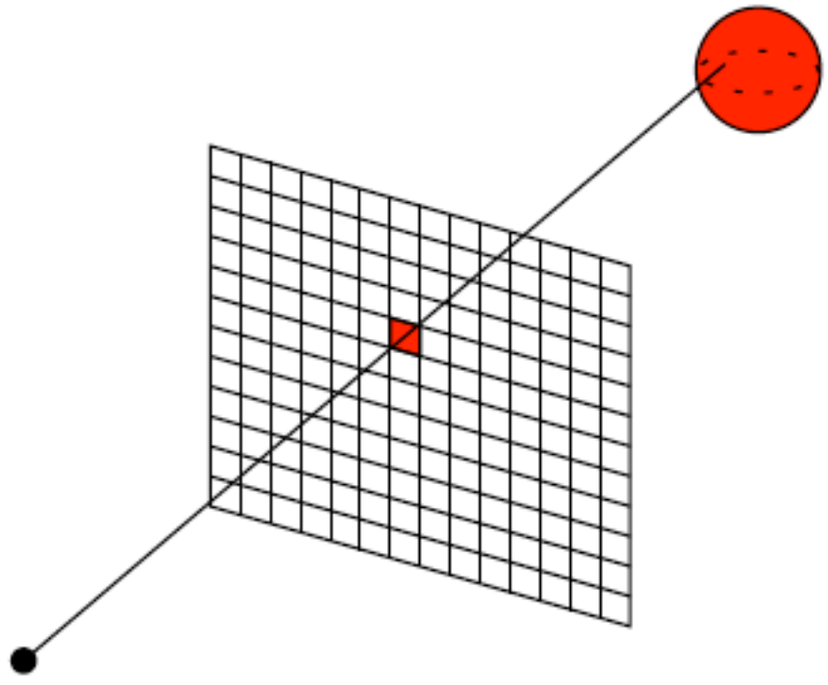
- Compute ray from viewpoint through pixel center
- Determine intersection point with first object hit by ray
- Calculate shading for the pixel (possibly with recursion)



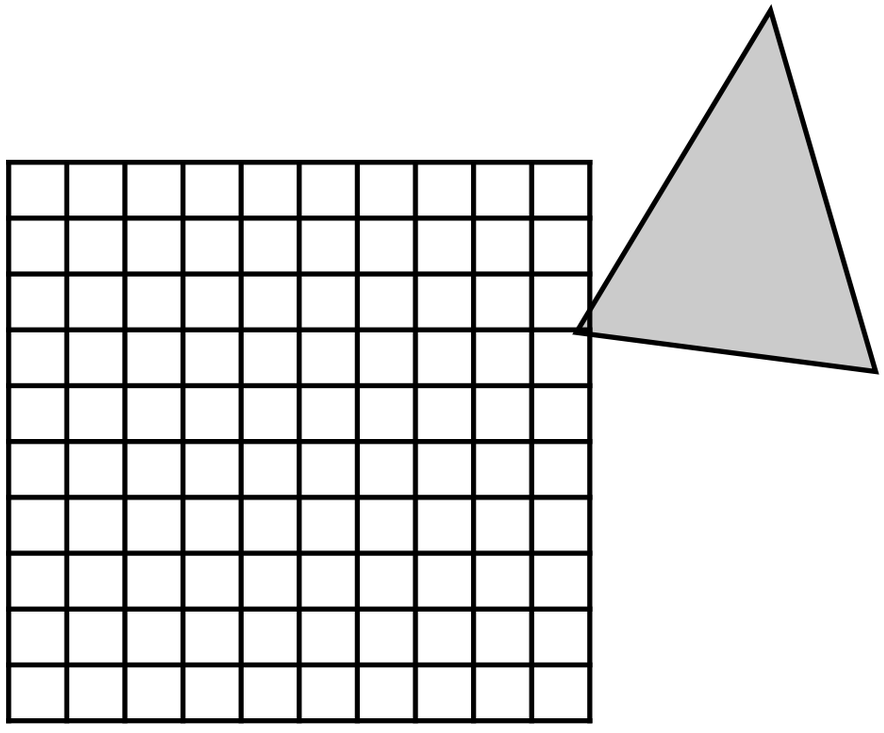
# Algorithm (2)

FOR each pixel DO

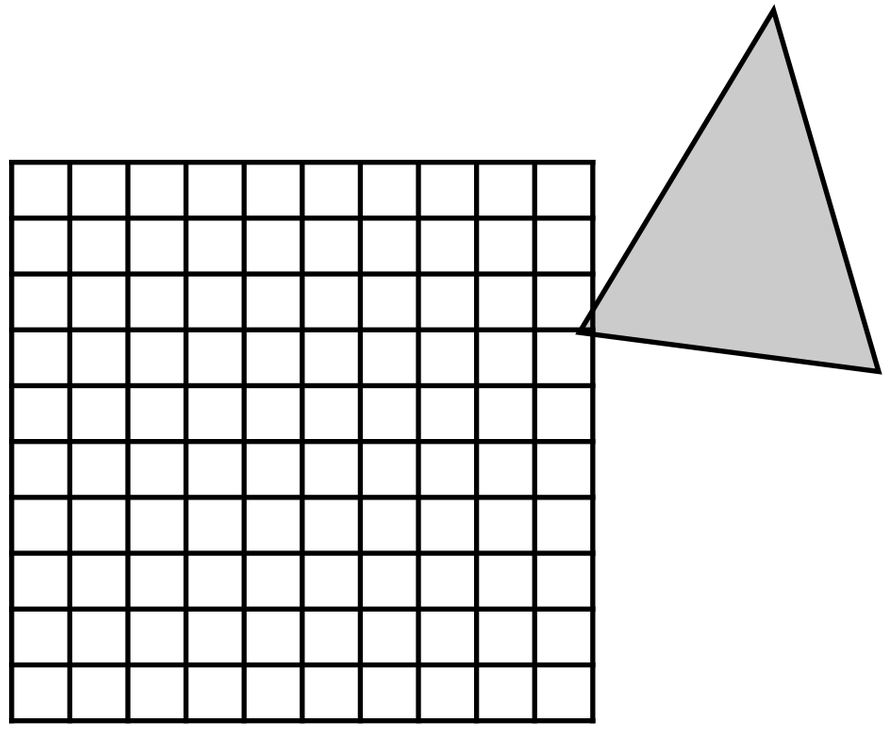
- find 1st object hit by ray and surface normal $\vec{n}$
- set pixel color to value computed from hit point, light, and $\vec{n}$



# Defining the Ray-casting Process

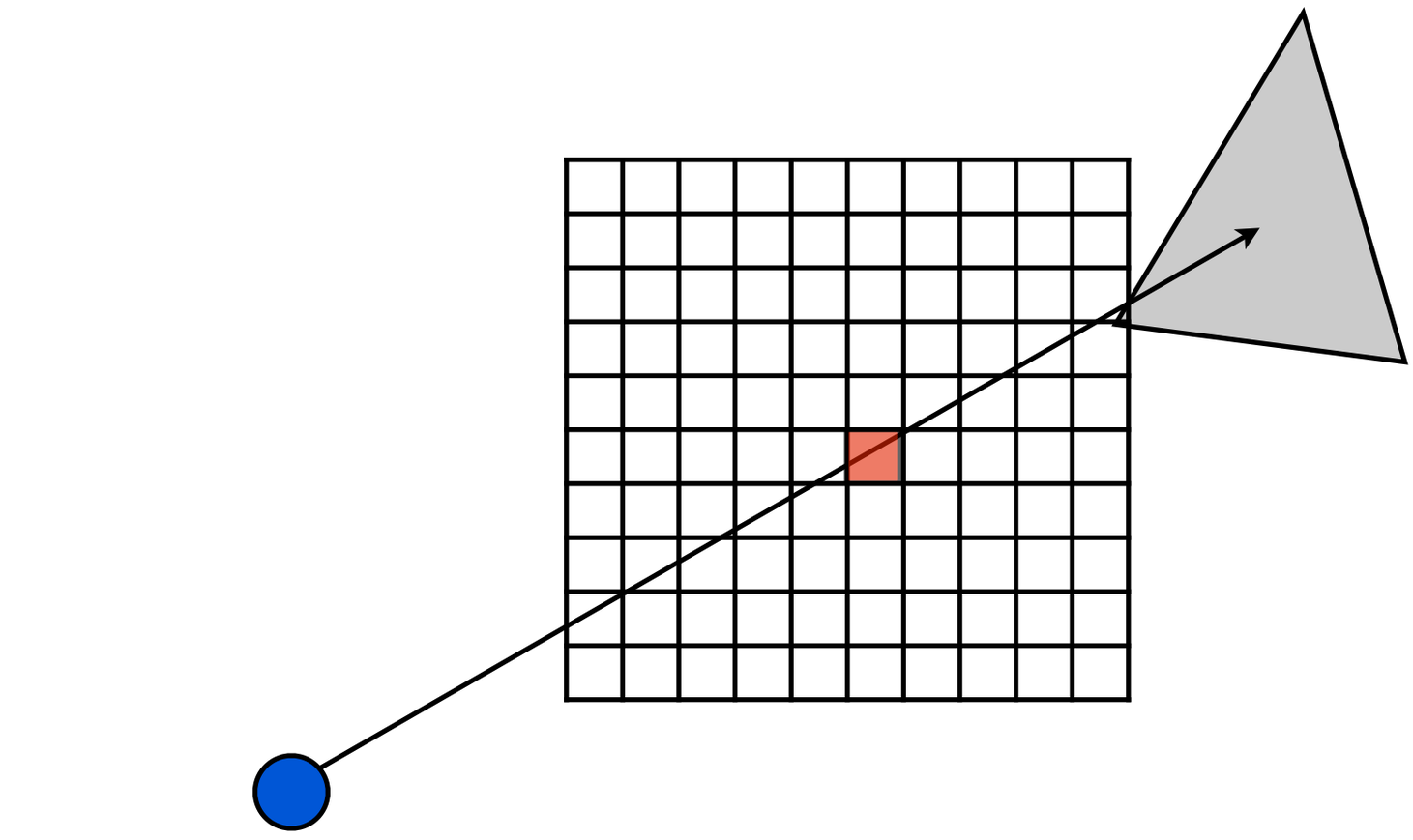


# Defining the Ray-casting Process

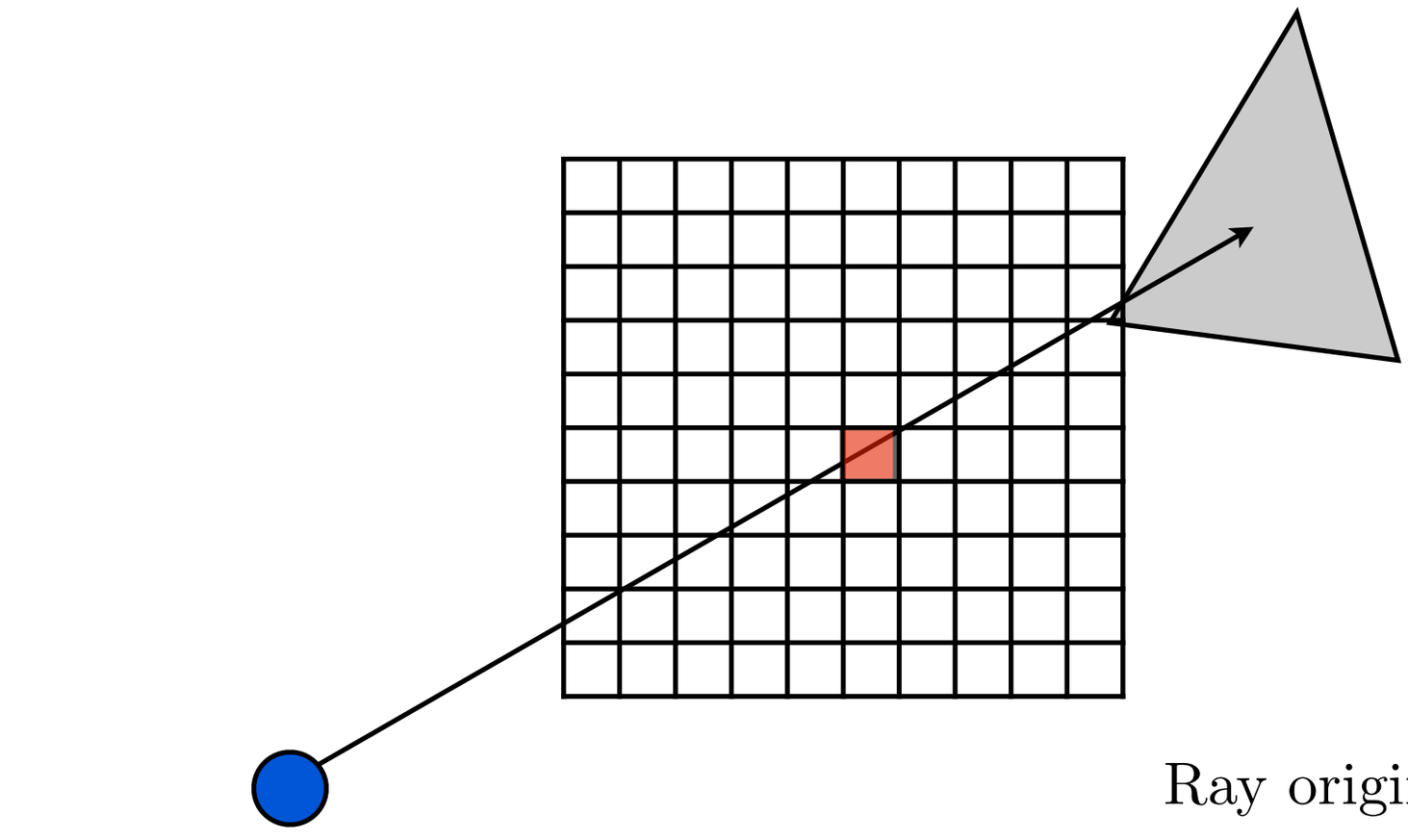


Camera

# Defining the Ray-casting Process



# Defining the Ray-casting Process




Ray origin : $c$

Pixel location (on near plane) : $p(x, y)$

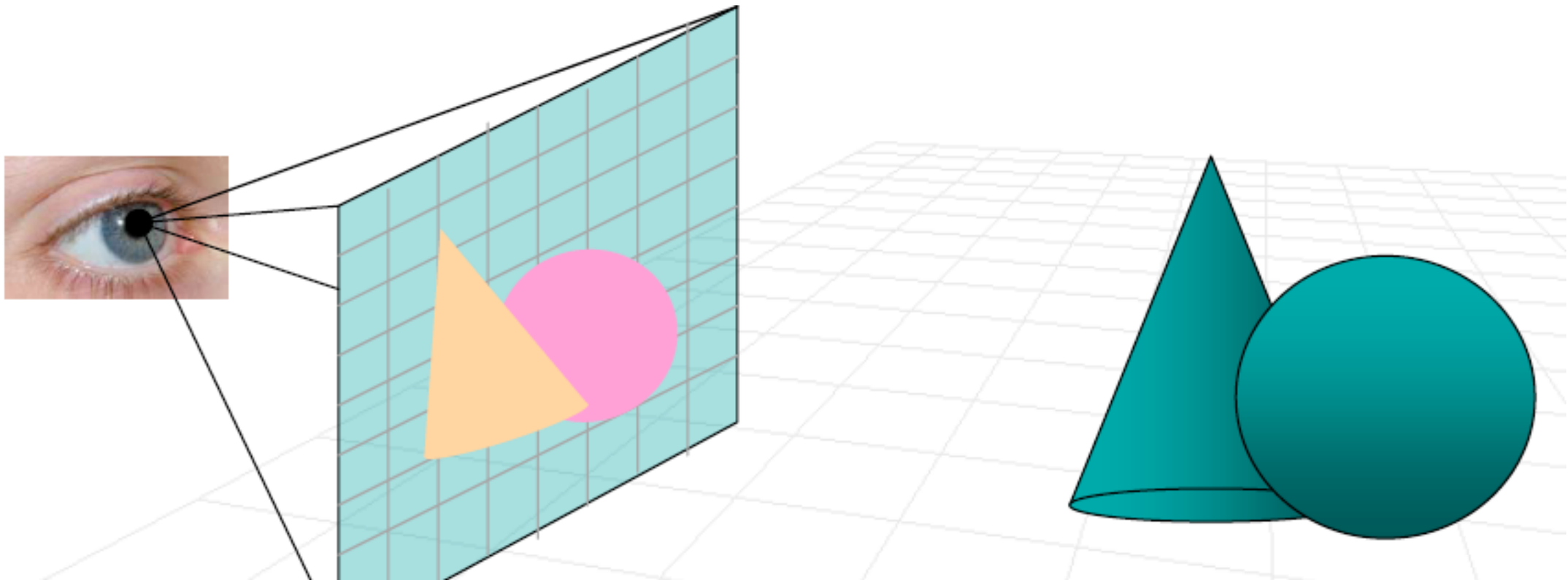
Ray direction : $\overrightarrow{p(x, y) - c}$

# Pseudo-code

- Given camera location (in world space), **c**
- For every pixel location on the near plane **p(x,y)**
  - Calculate ray, r = p(x,y) - c
  - Traverse the ray in the environment to check first object hit, X
  - Use the surface characteristics at point X to calculate the color of the pixel

# Hardware based implementation

- Ray can be cast per fragment
- Very easily integrated into Deferred shading
  - Output unique object id as a buffer in the Geometry stage
  - Z-ordering ensures that the nearest object id is stored in the buffer



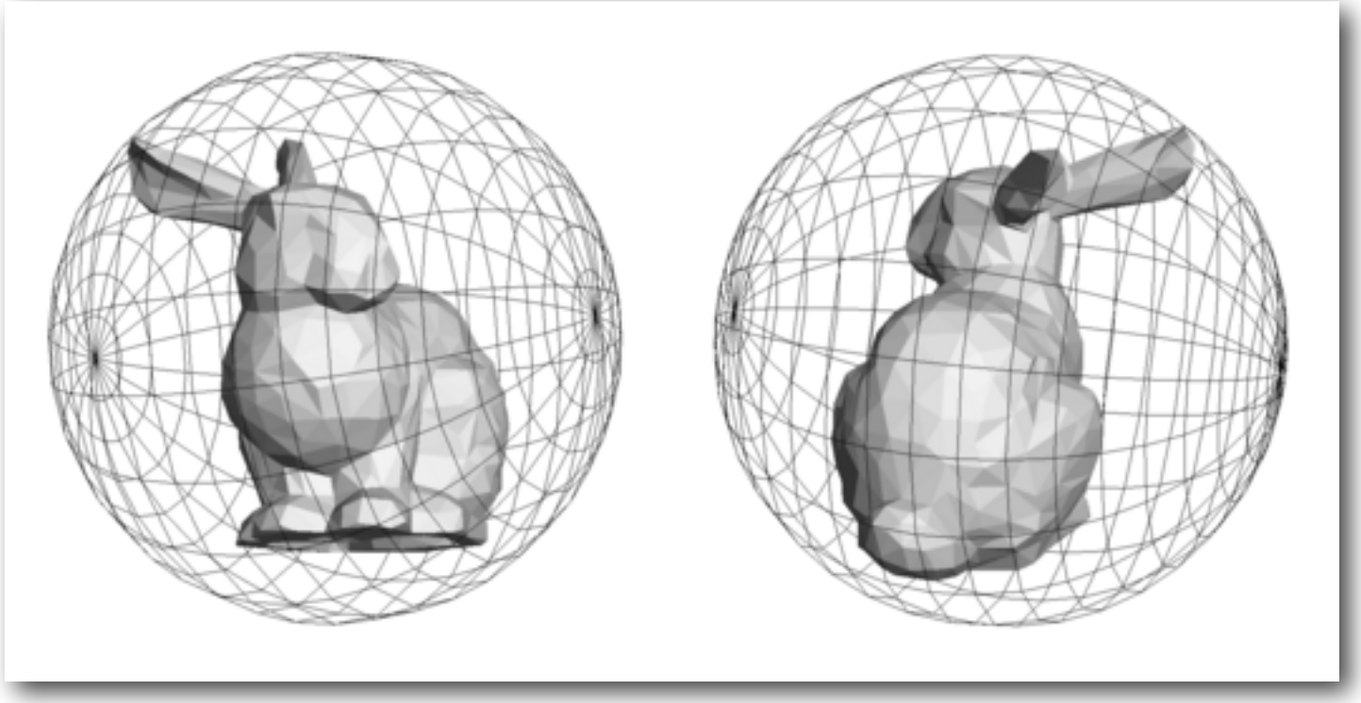
# Hit Checking the Rays

# Hit checking

- Very expensive to check ray-triangle intersection for ALL triangles
  - Use bounding volumes (BSphere, AABB, OBB)
  - Use spatial partitioning to reject objects “far” from the ray (Octree, K-d tree, BSP tree)
- Both topics are focus of CS 350!

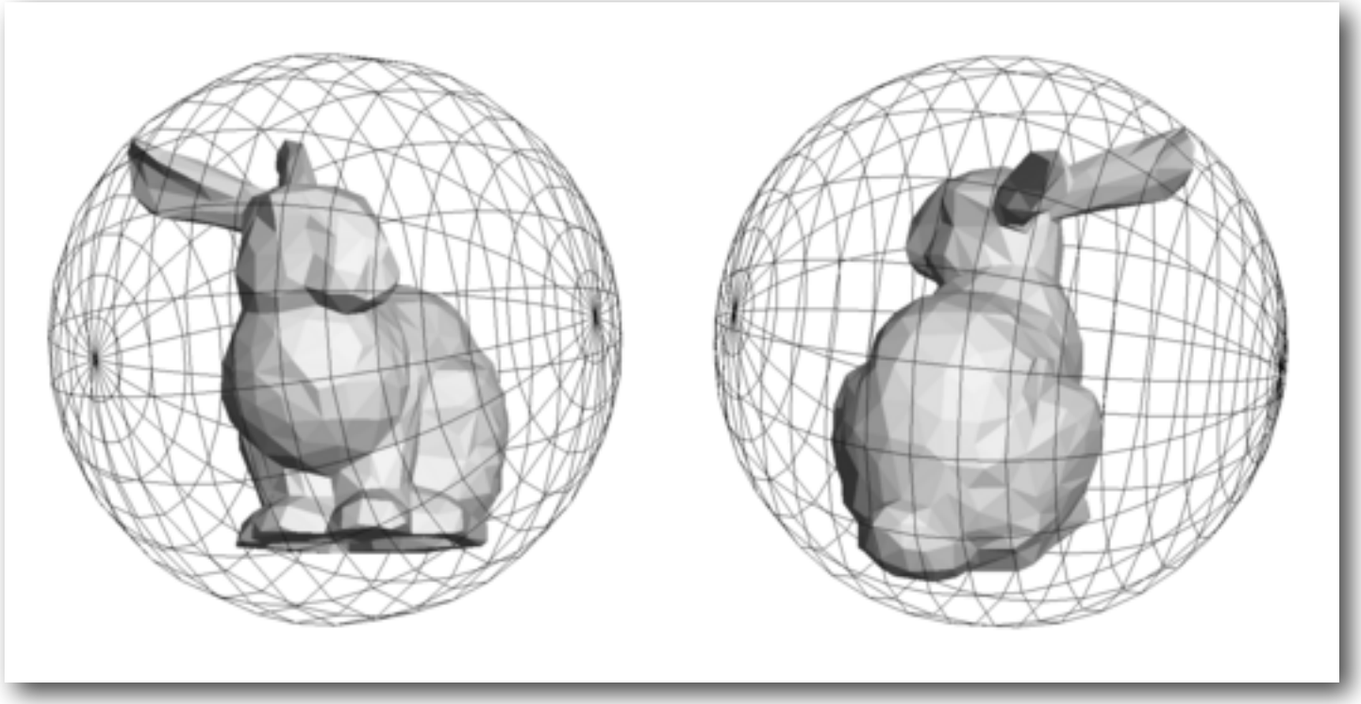
# Hit checking

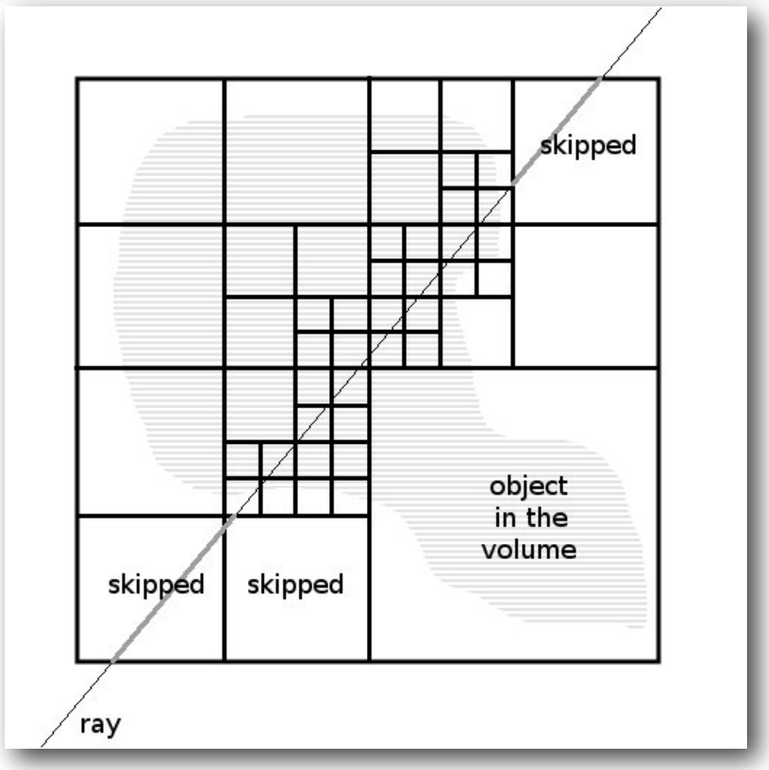
- Very expensive to check ray-triangle intersection for ALL triangles
  - Use bounding volumes (BSphere, AABB, OBB)
  - Use spatial partitioning to reject objects “far” from the ray (Octree, K-d tree, BSP tree)
- Both topics are focus of CS 350!



# Hit checking

- Very expensive to check ray-triangle intersection for ALL triangles
  - Use bounding volumes (BSphere, AABB, OBB)
  - Use spatial partitioning to reject objects “far” from the ray (Octree, K-d tree, BSP tree)
- Both topics are focus of CS 350!





# Efficiency Issues

- Easy to compute ray intersection with **implicit surfaces**

$$f(x, y, z) = 0$$

- Spheres, ellipsoids, planes, parametric surfaces

- Can find analytical solutions to the problem of intersection in fairly quick time

# Color calculation

- Use the normal to the surface at hit point (X) to calculate the color using Phong model

# Color calculation

- Use the normal to the surface at hit point (X) to calculate the color using Phong model

$$I_{total} = I_{diffuse} + I_{specular}$$

# Color calculation

- Use the normal to the surface at hit point (X) to calculate the color using Phong model

$$I_{total} = I_{diffuse} + I_{specular}$$

Local illumination model

# Color calculation

- Use the normal to the surface at hit point (X) to calculate the color using Phong model

$$I_{total} = I_{diffuse} + I_{specular} + I_{reflectance}$$

Local illumination model

# Color calculation

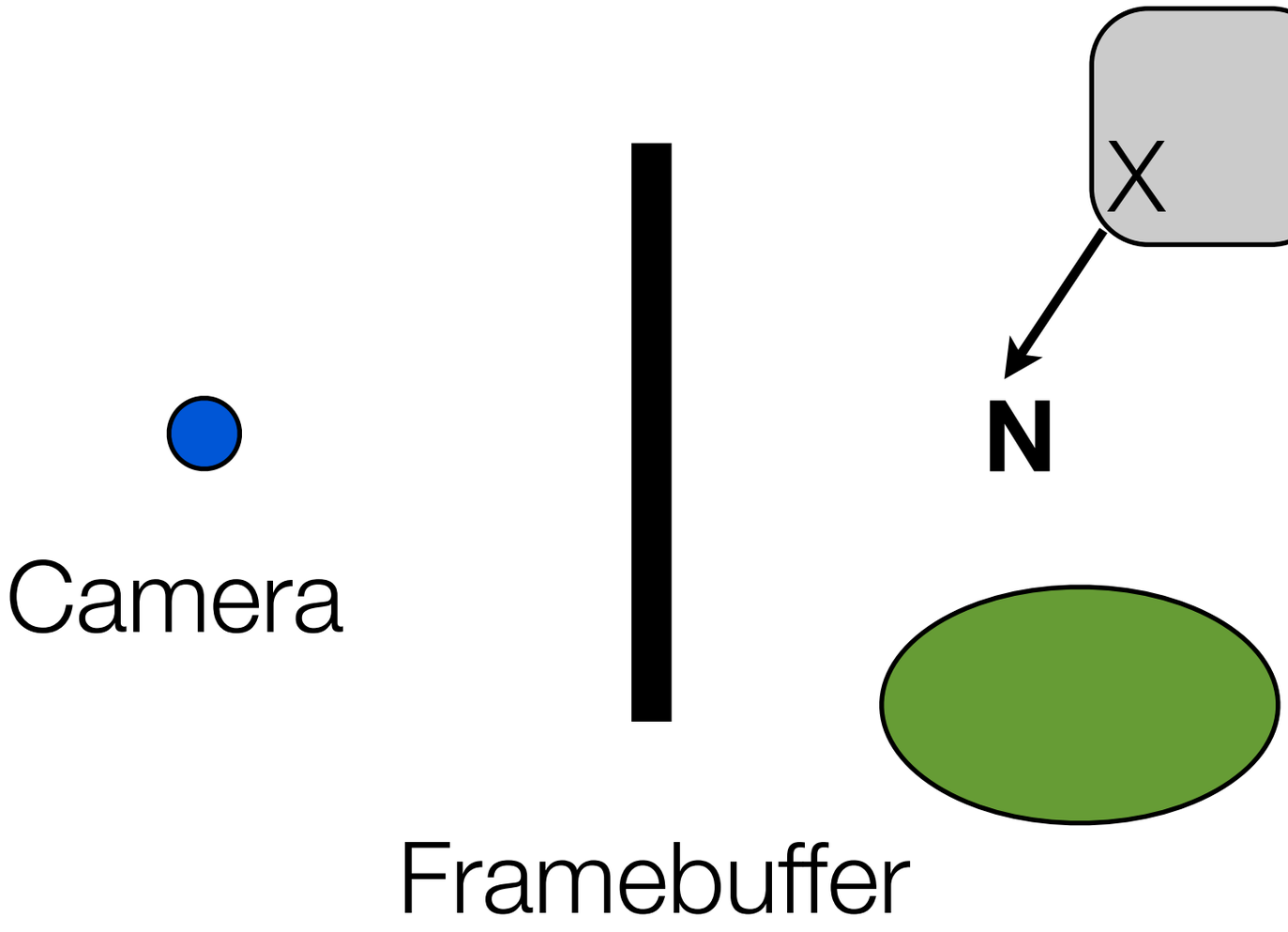
- Use the normal to the surface at hit point (X) to calculate the color using Phong model

$$I_{total} = I_{diffuse} + I_{specular} + I_{reflectance}$$

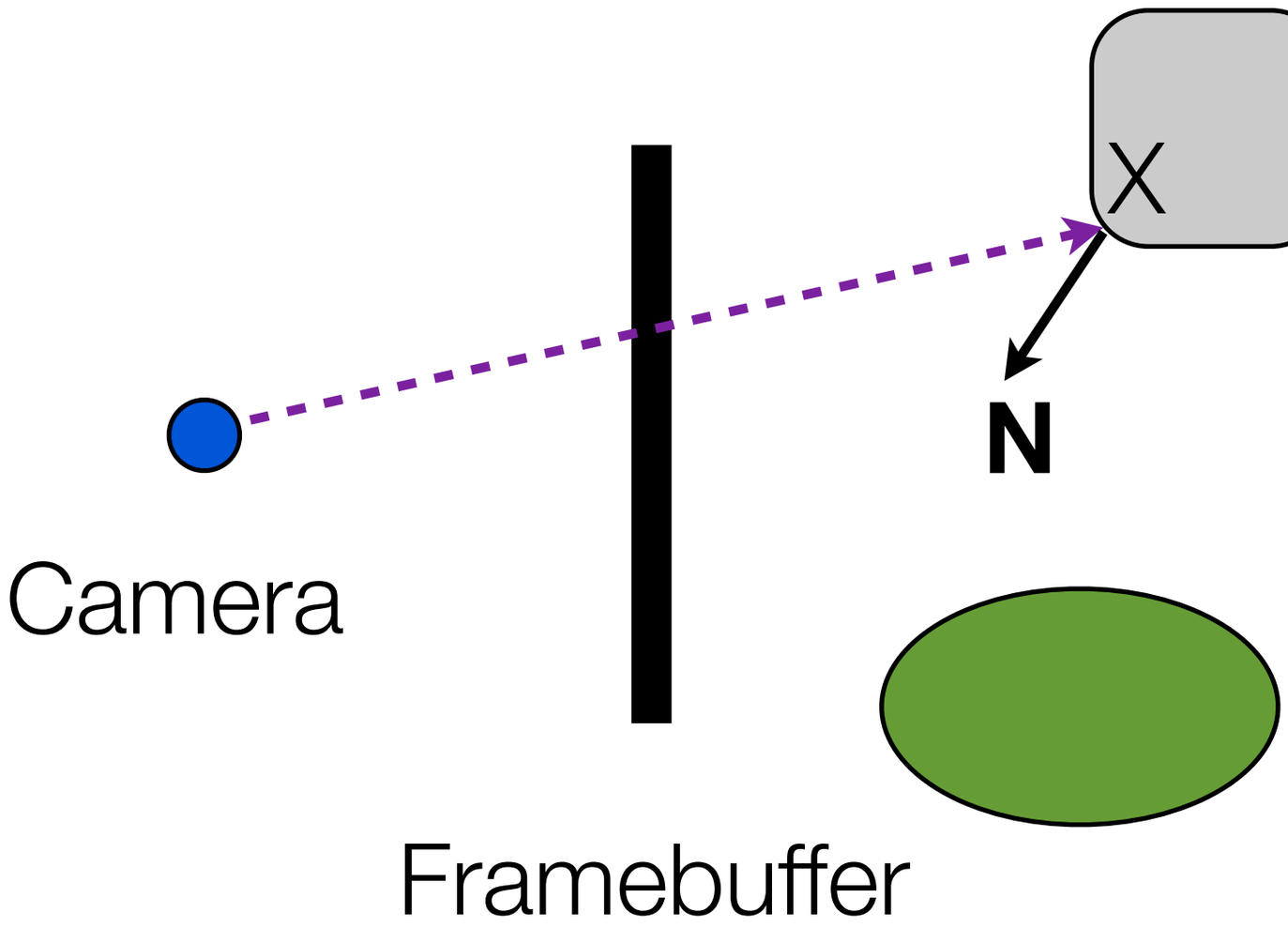
Local illumination model

Global ray-traced solution

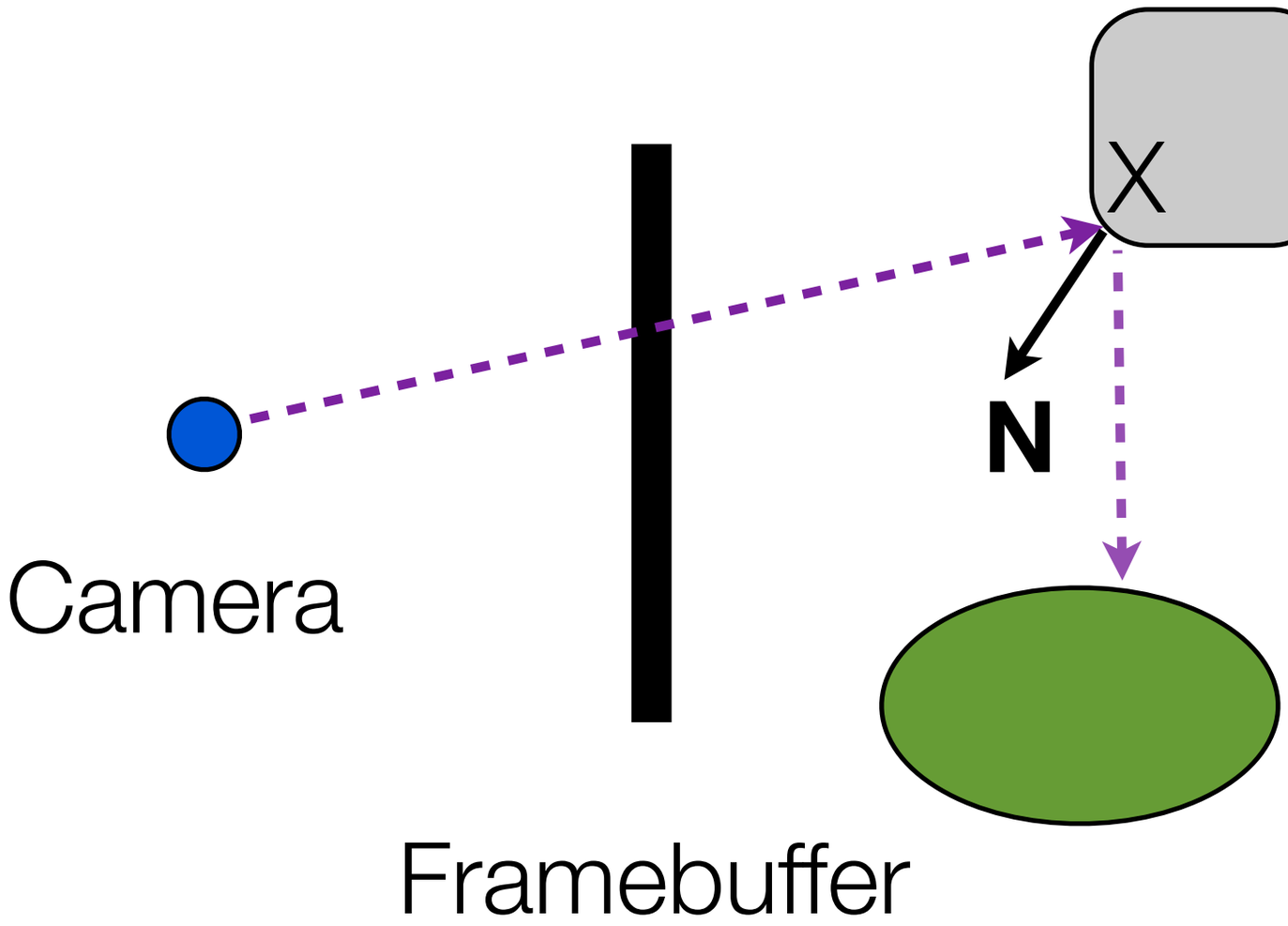
# Example - 1 “bounce”



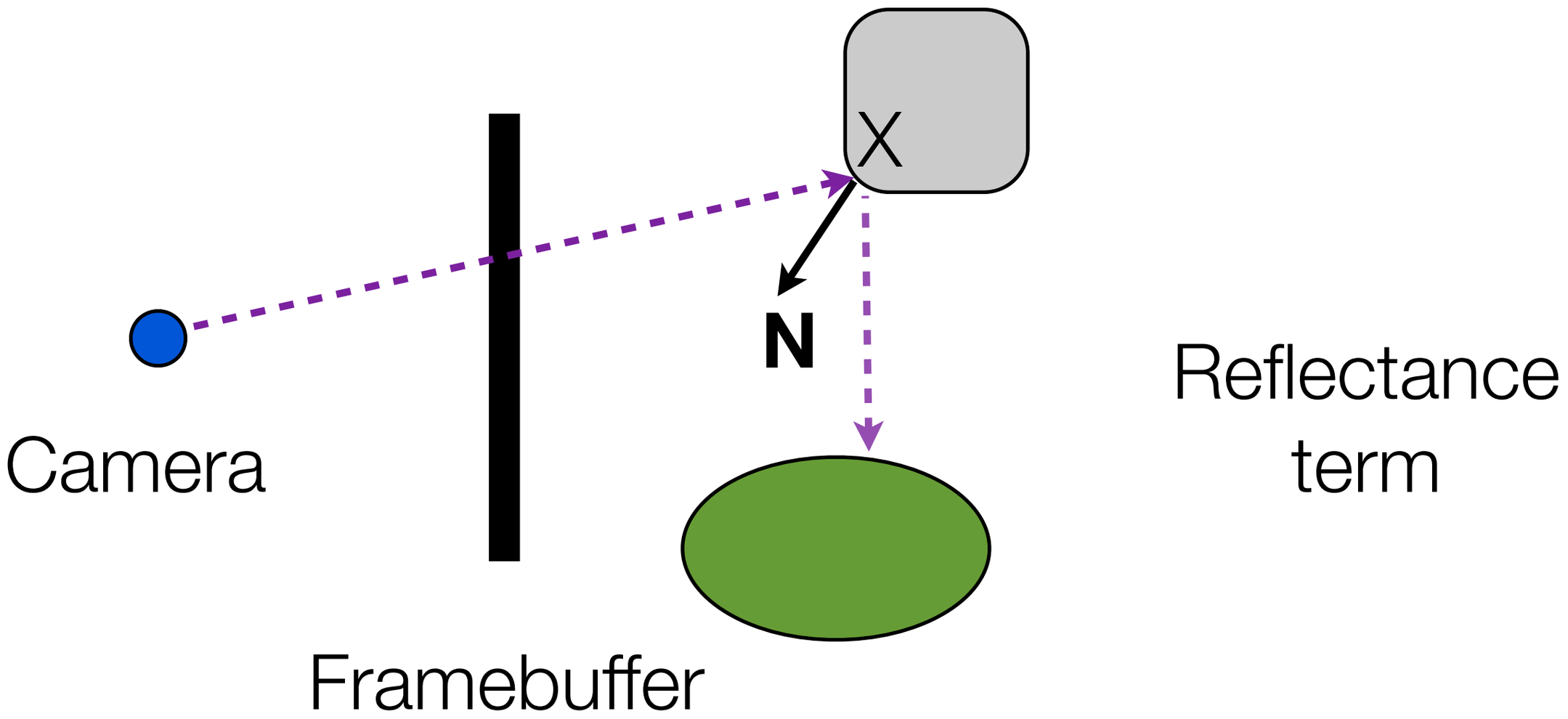
# Example - 1 “bounce”



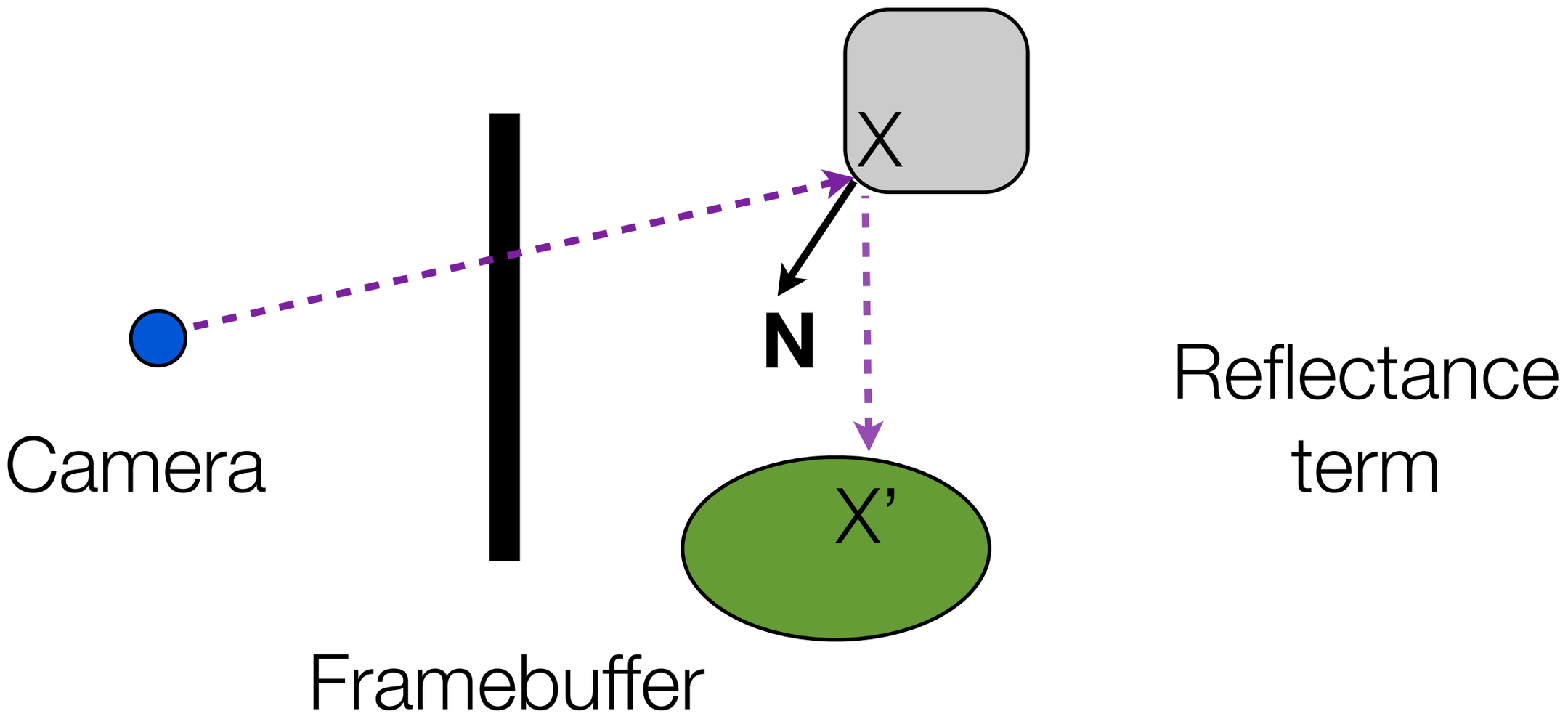
# Example - 1 “bounce”



# Example - 1 “bounce”

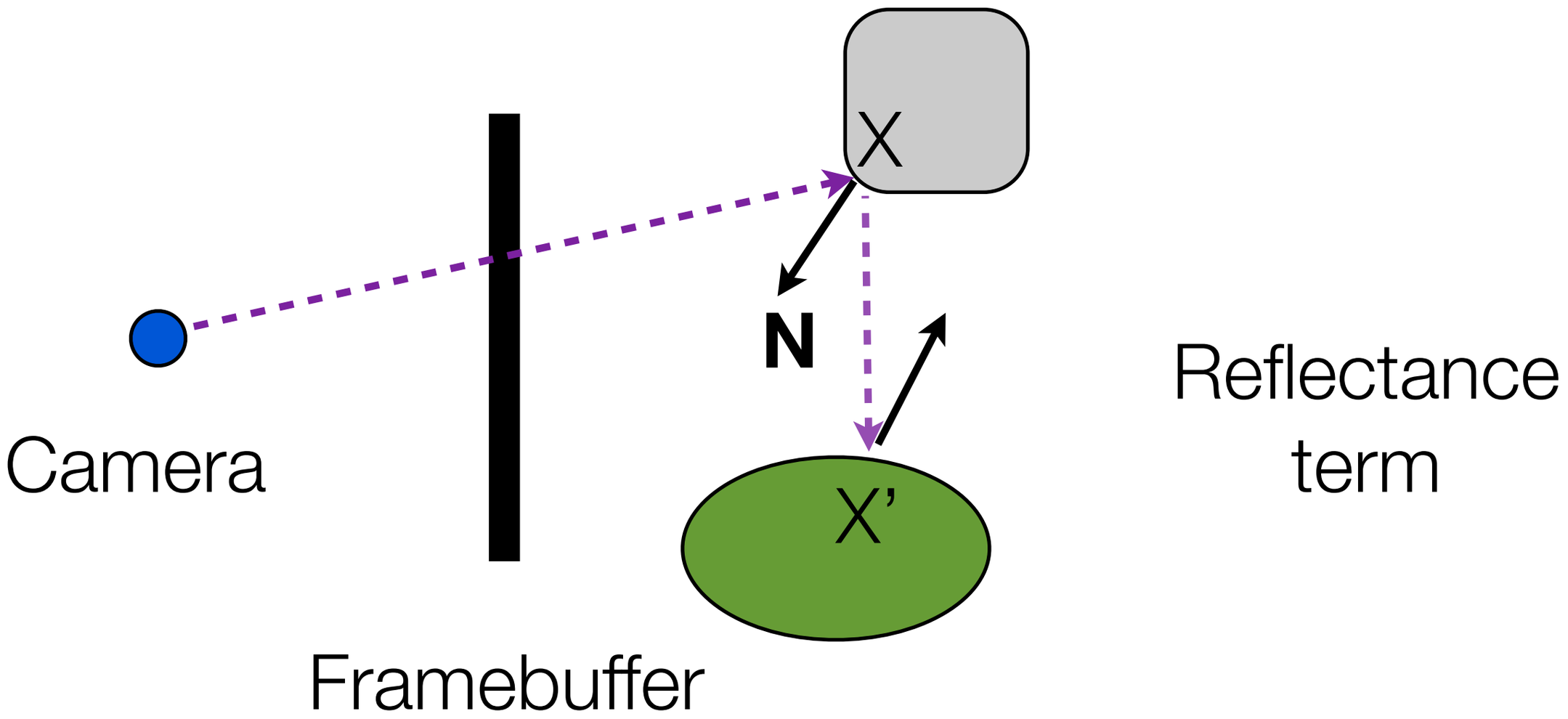


# Example - 1 “bounce”



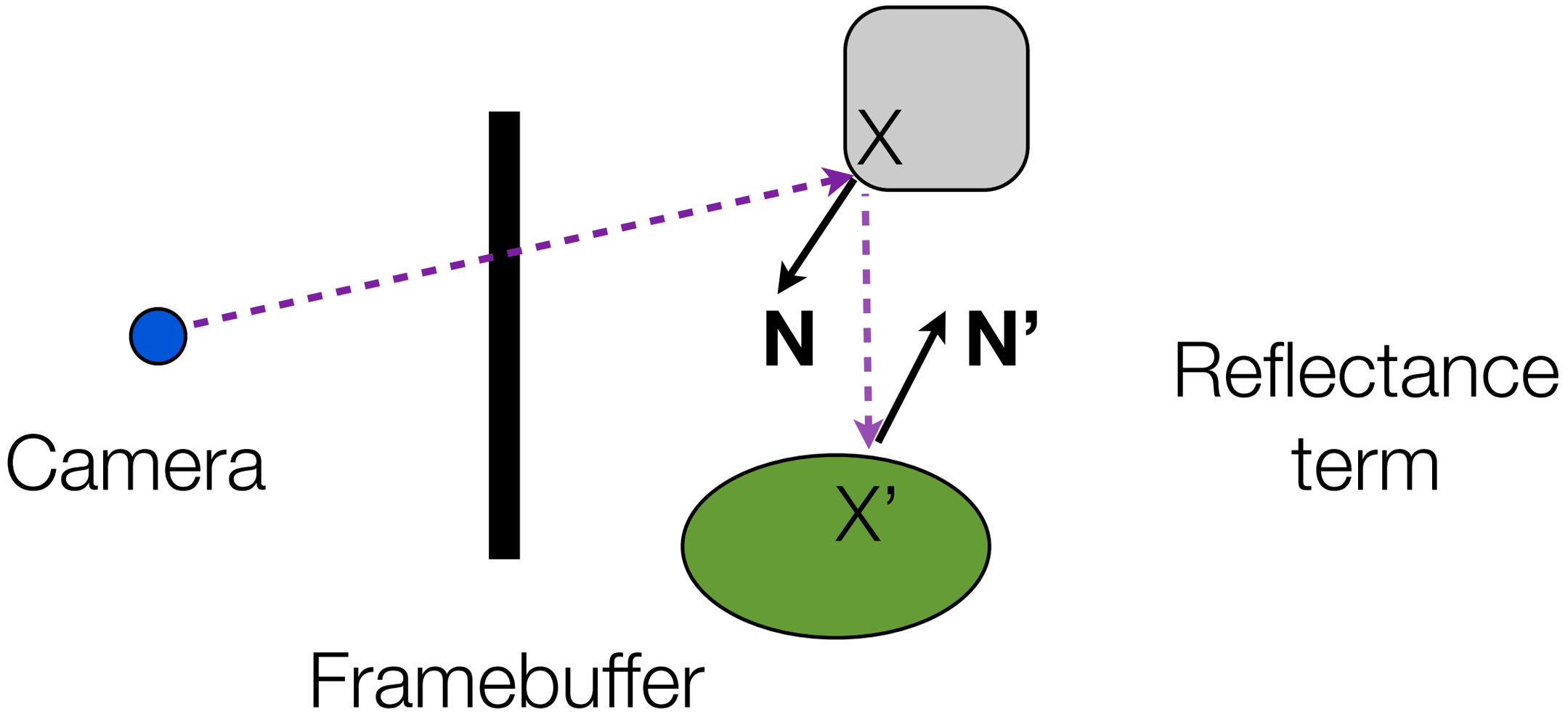

Reflectance term

# Example - 1 “bounce”

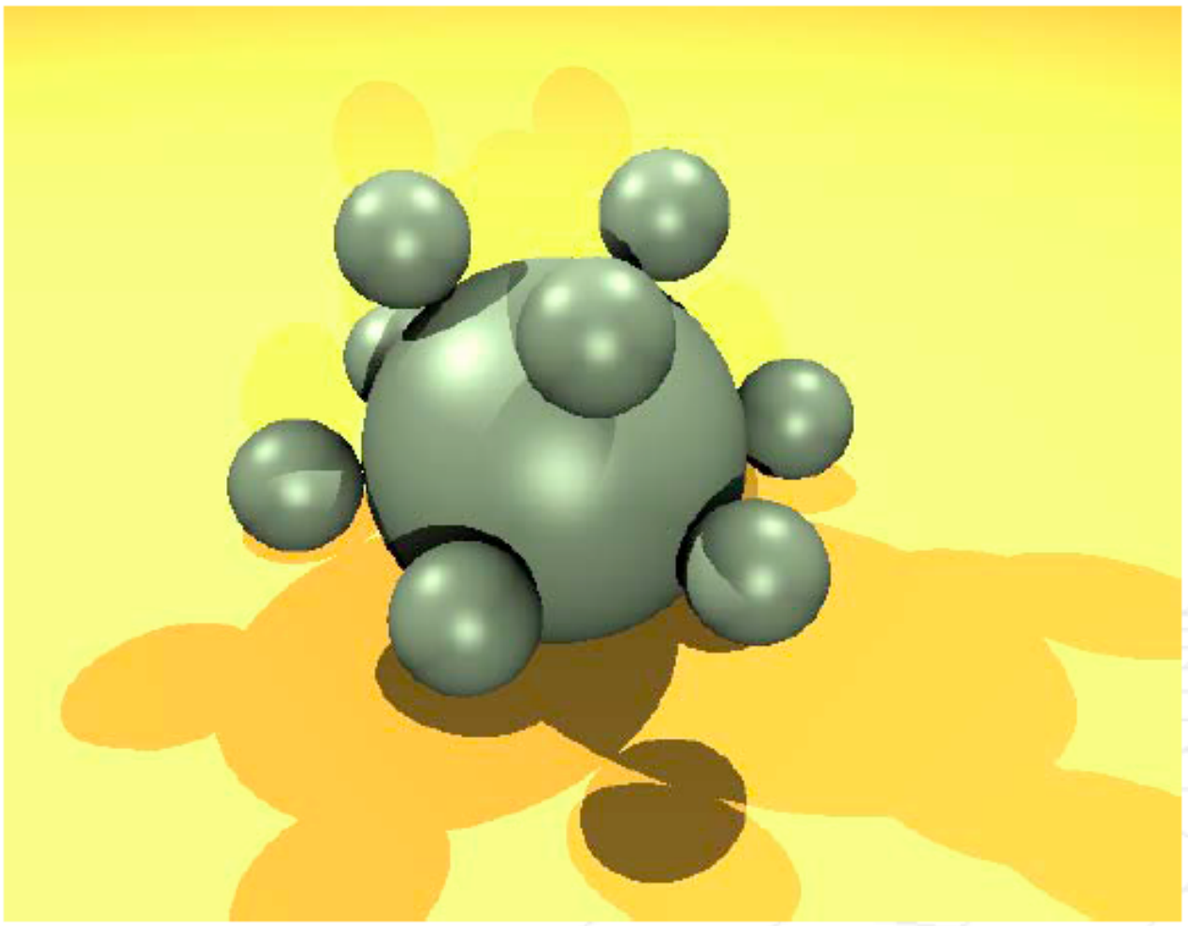



Reflectance term

# Example - 1 “bounce”

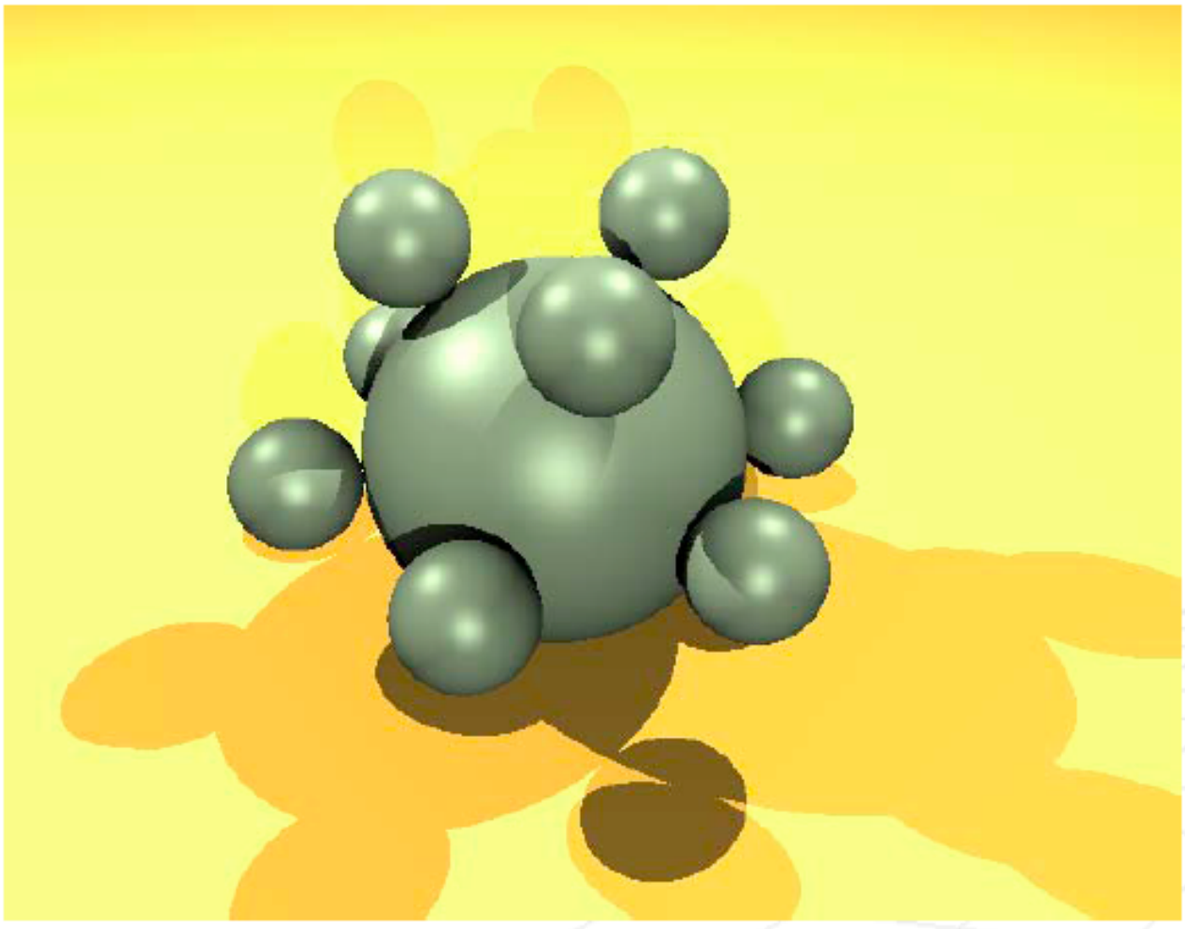


# Examples

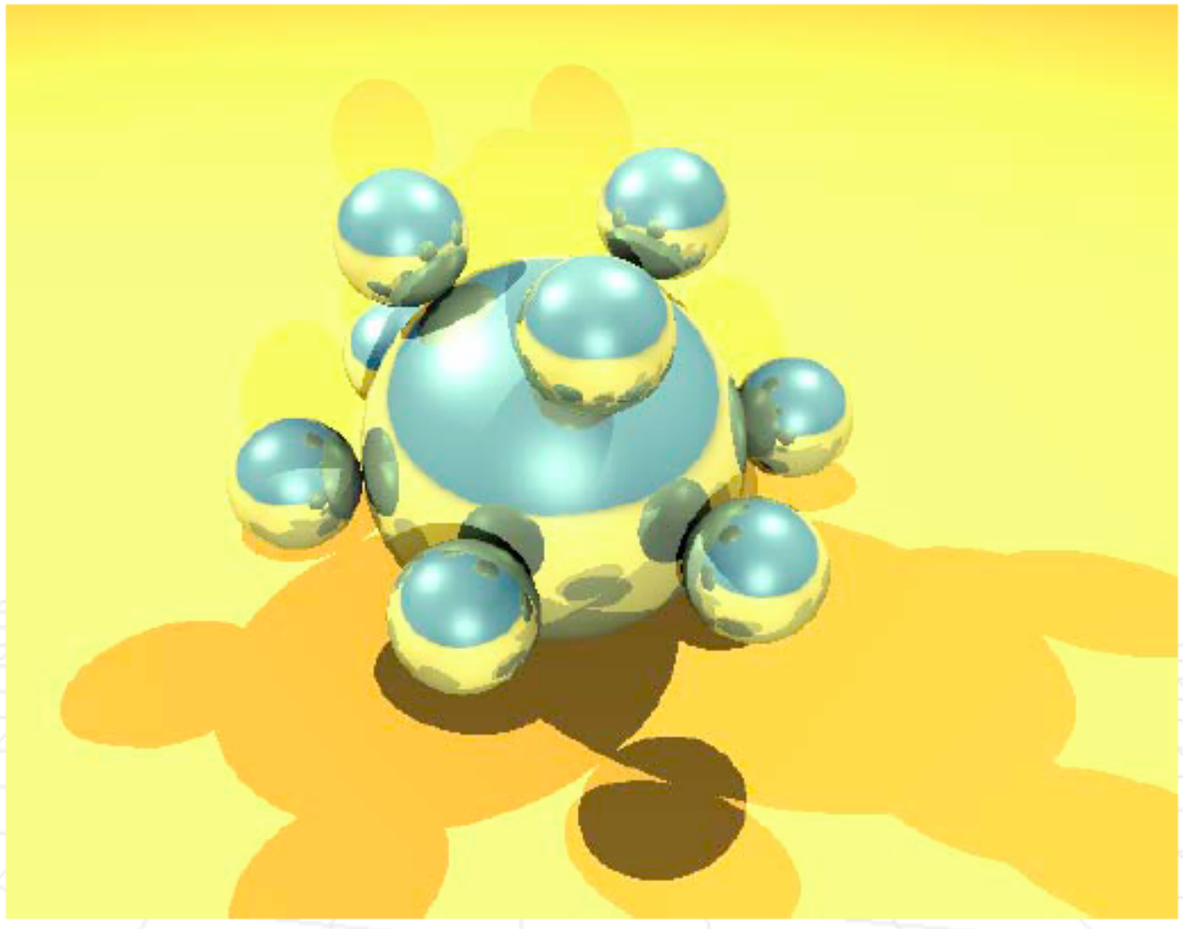


**No recursive rays (local lighting)**

# Examples



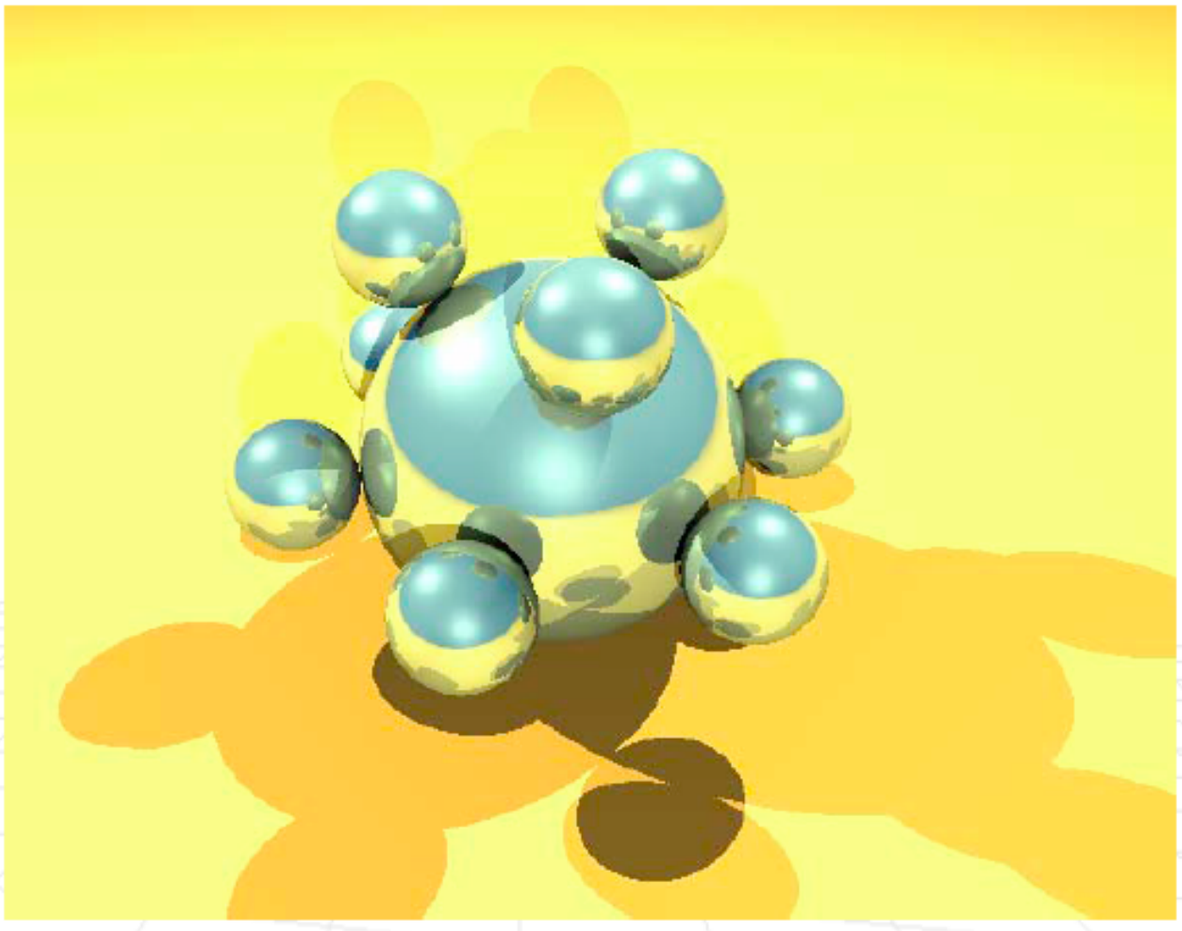
No recursive rays (local lighting)



1 Level of recursive reflection

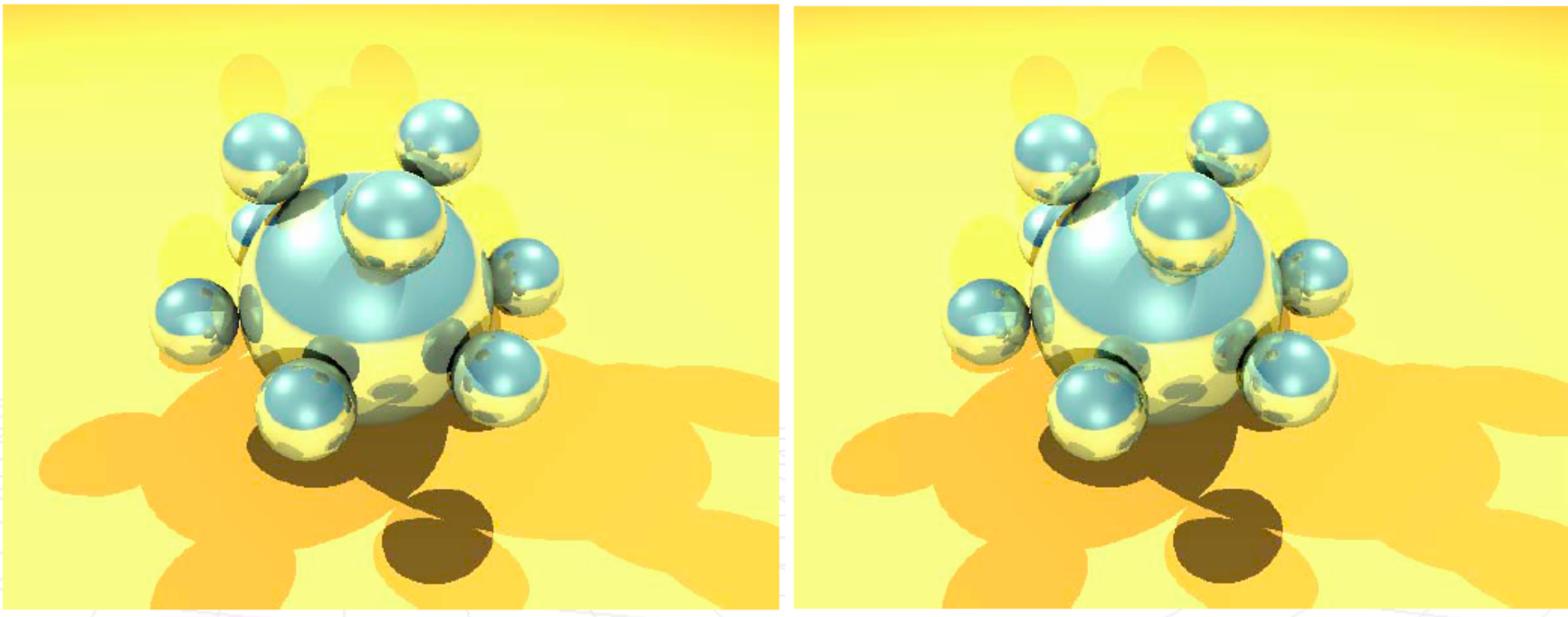
# 2-bounce

# 2-bounce



1 Level of recursive reflection

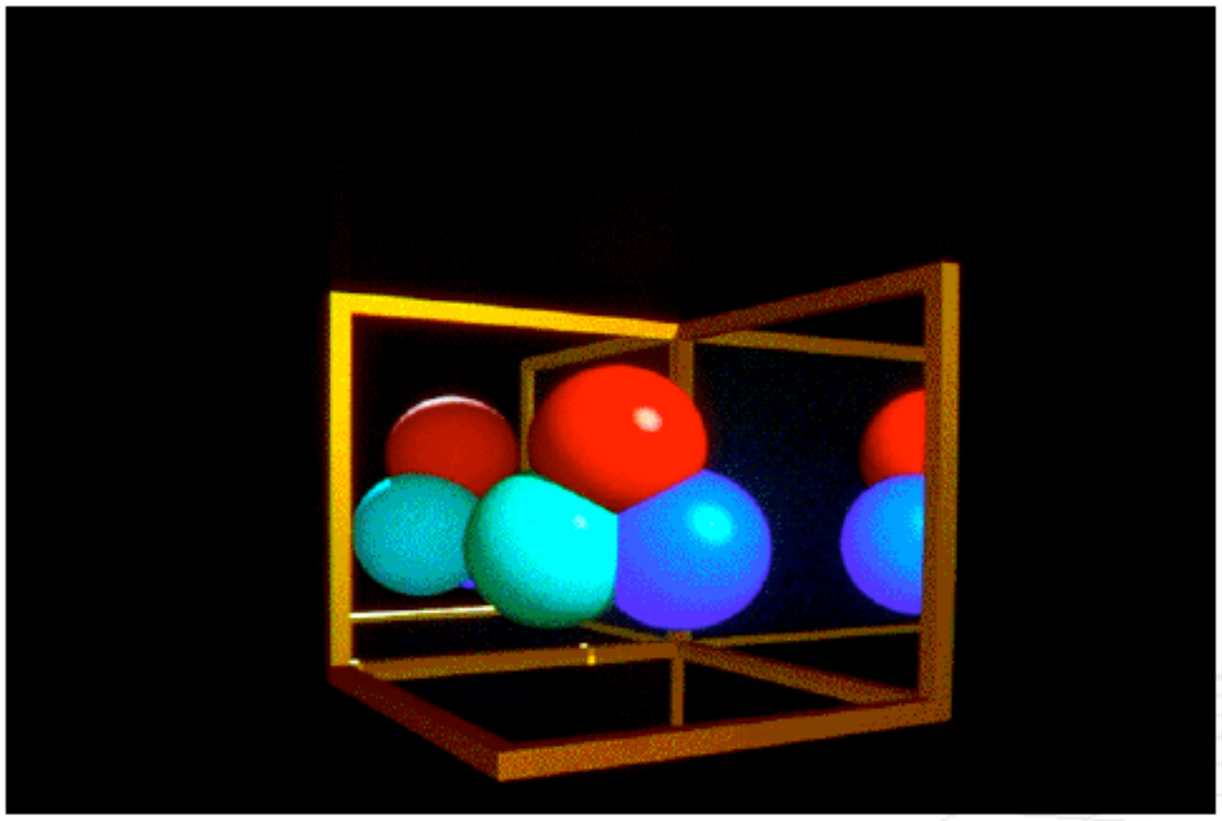
# 2-bounce



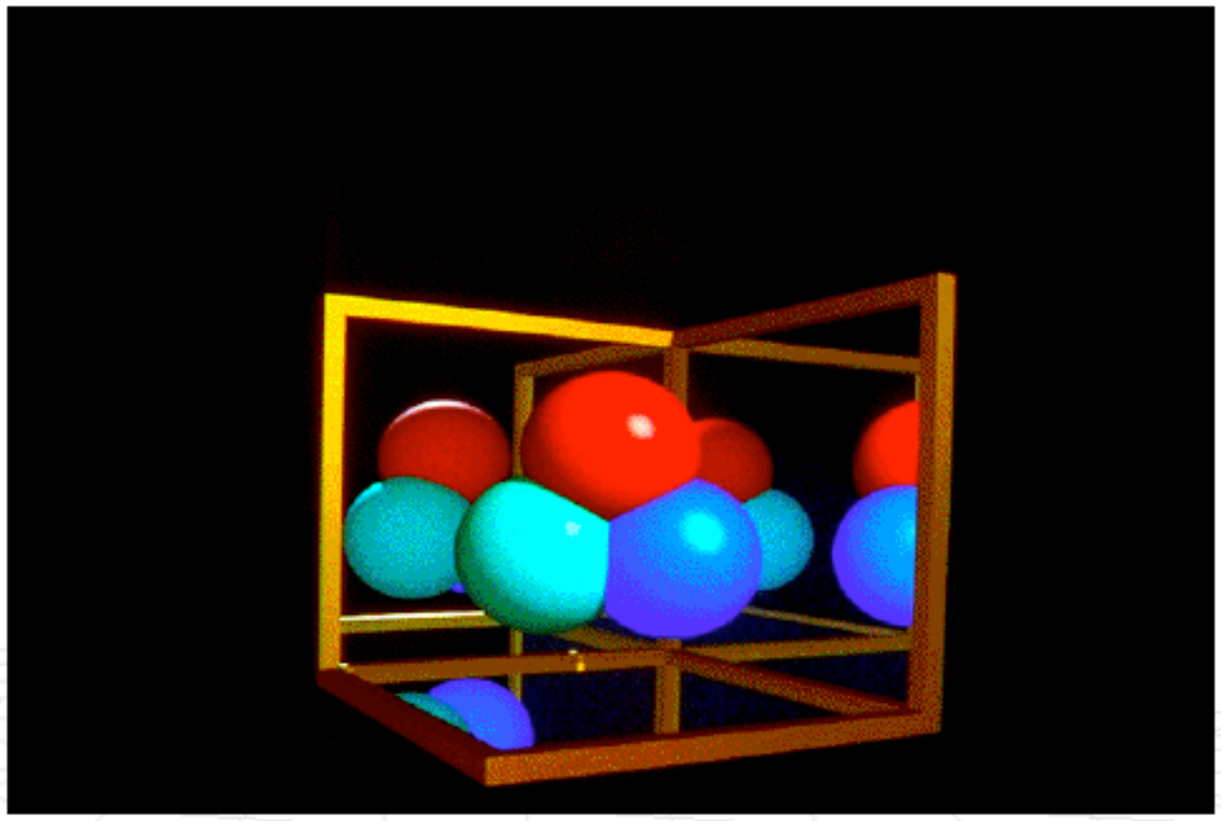
1 Level of recursive reflection

2 Levels of recursive reflection

# More examples



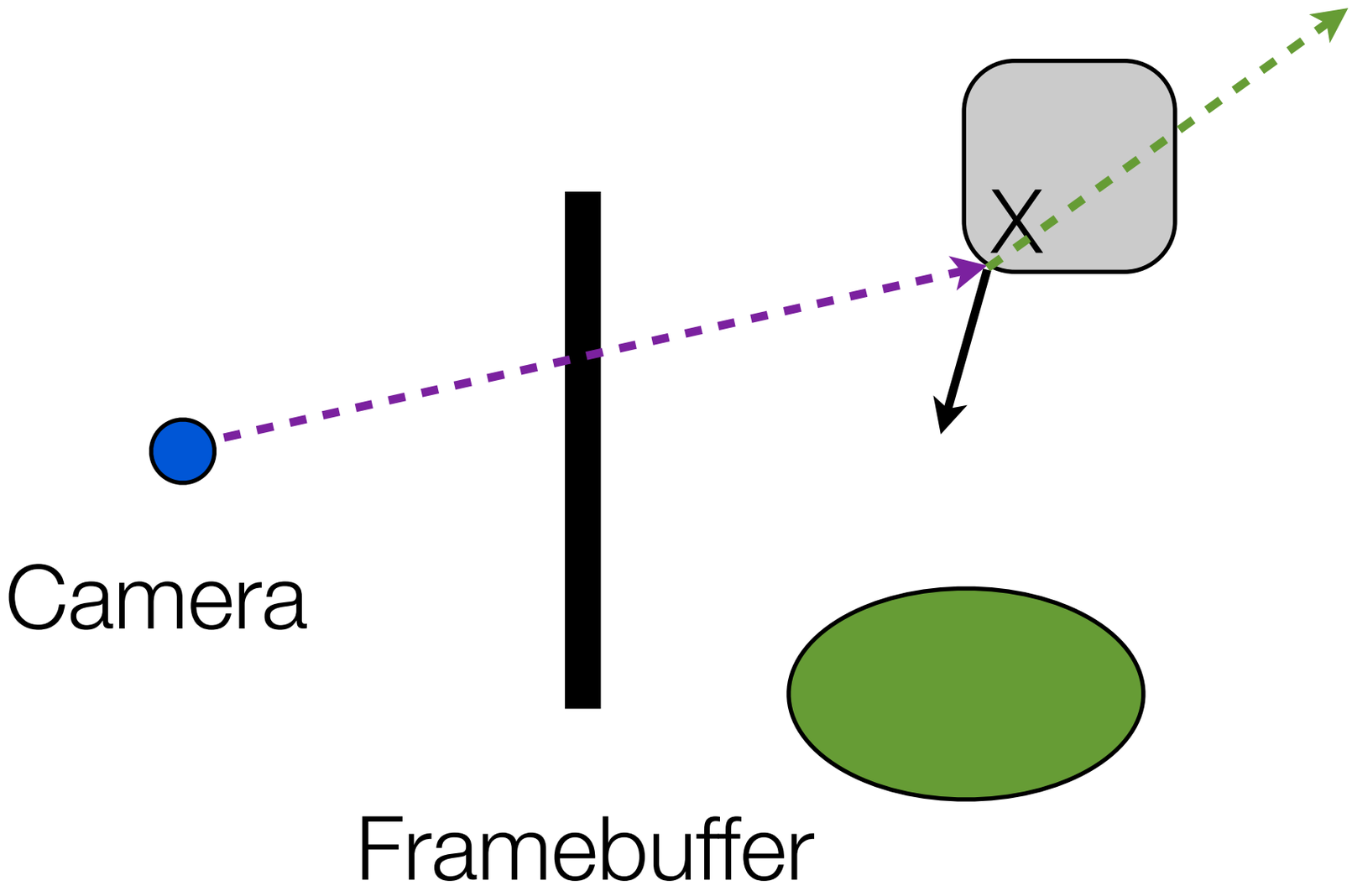
1 Level of recursive reflection



2 Levels of recursive reflection

# Refraction

- Easily integrated with the refracted ray as a new recursion branch



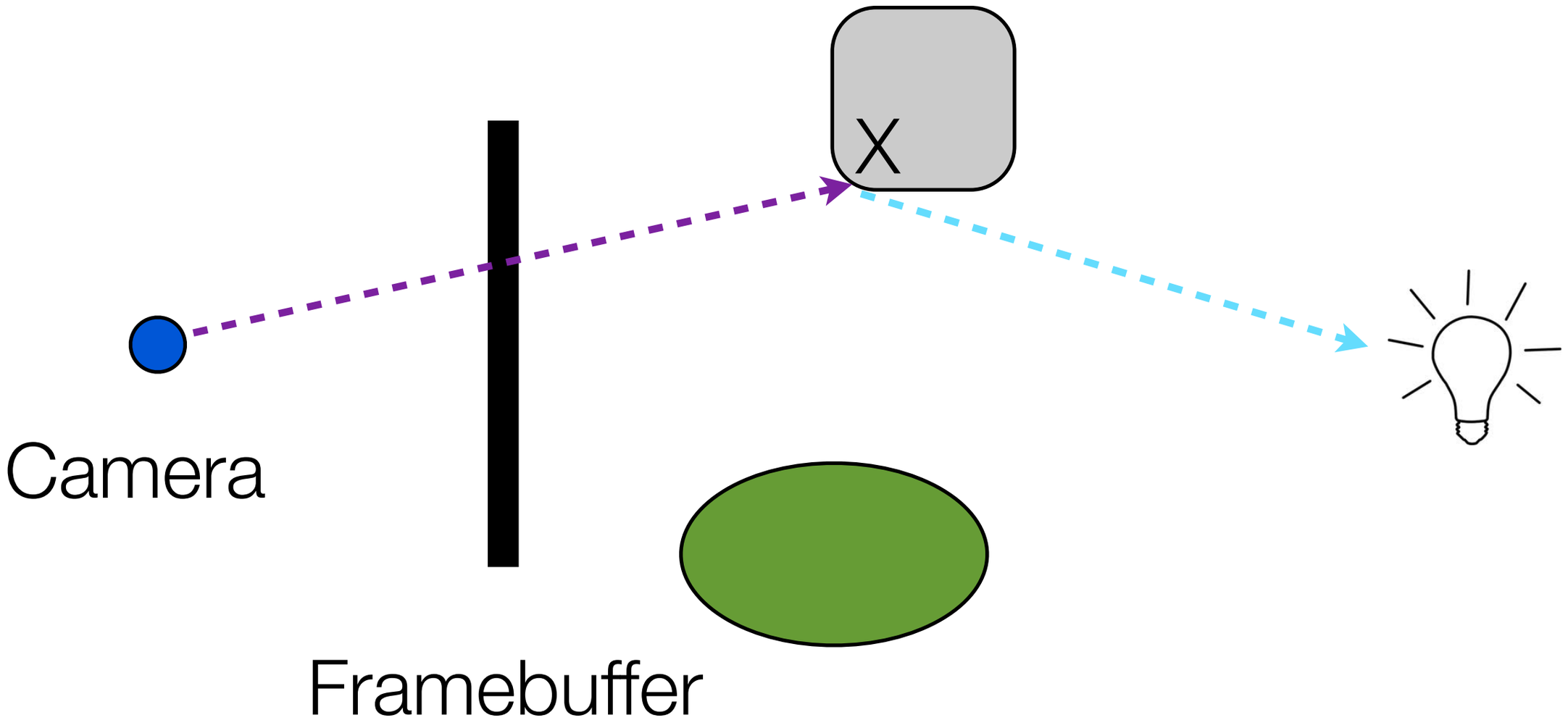
# Example

| Material | Index of Refraction |
|---|---|
| *vacuum* | 1.0 |
| *ice* | 1.309 |
| *water* | 1.333 |
| *ethyl alcohol* | 1.36 |
| *glass* | 1.5–1.6 |
| *diamond* | 2.417 |

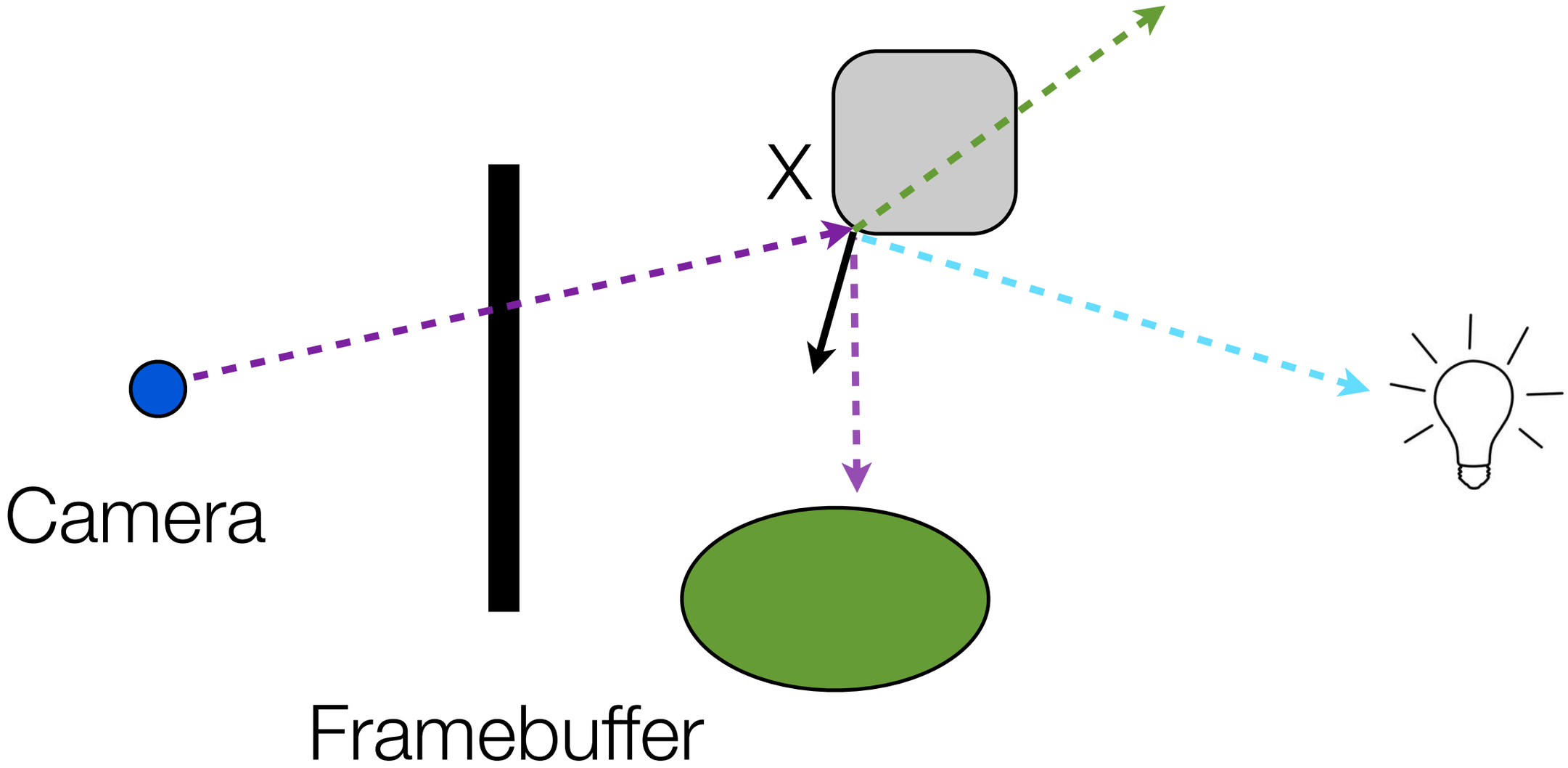
# Example - Refraction

# Shadows??

- Cast a third ray from hit point X, towards the light source
- If no intersection along the way, point is illuminated by that light source



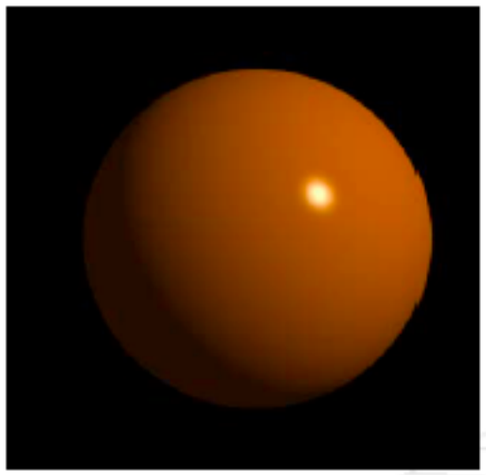
# Total Color Contribution



# All 3 terms together

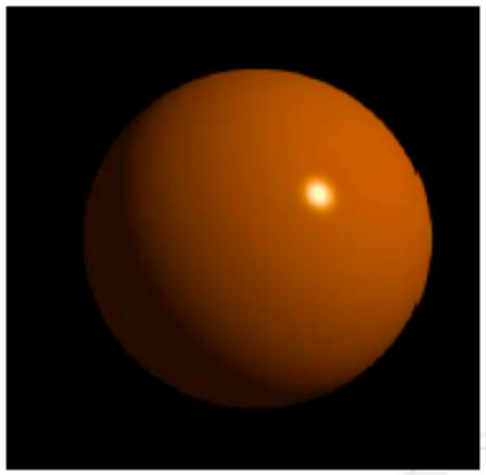
# All 3 terms together

**Local**
**Lighting**
**Phong**

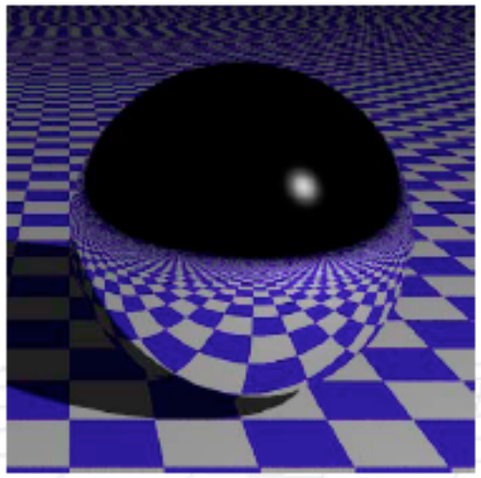


# All 3 terms together

**Local Lighting Phong**

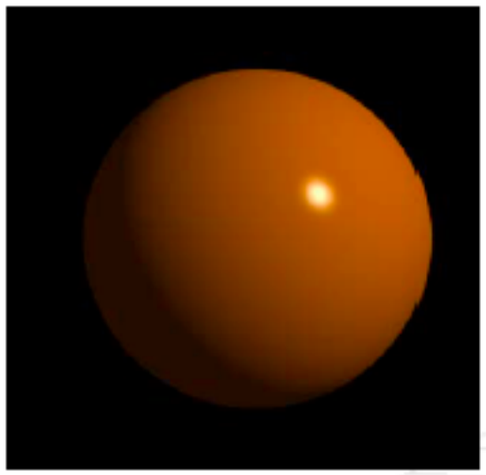


**Global Reflections (and shadows)**

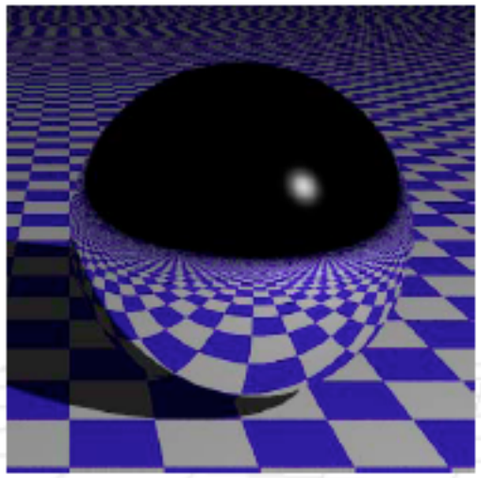


# All 3 terms together

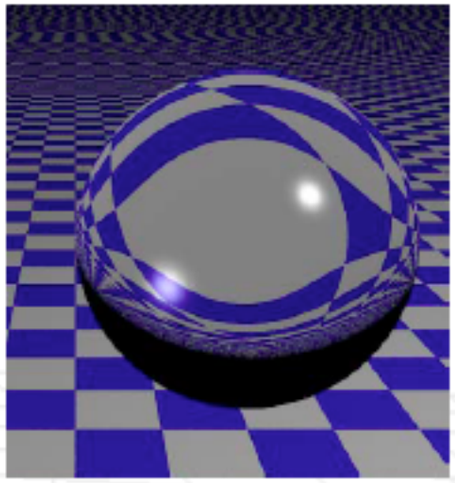
**Local Lighting Phong**



**Global Reflections (and shadows)**

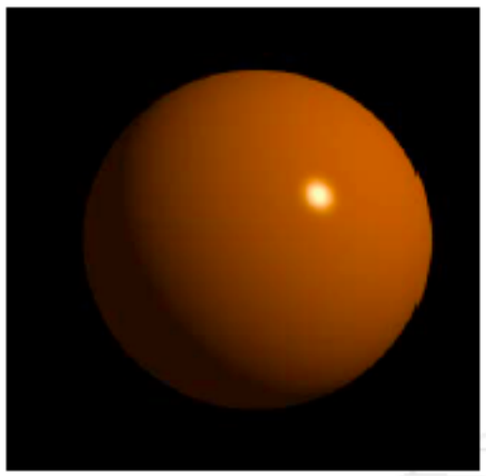


**Refractions**

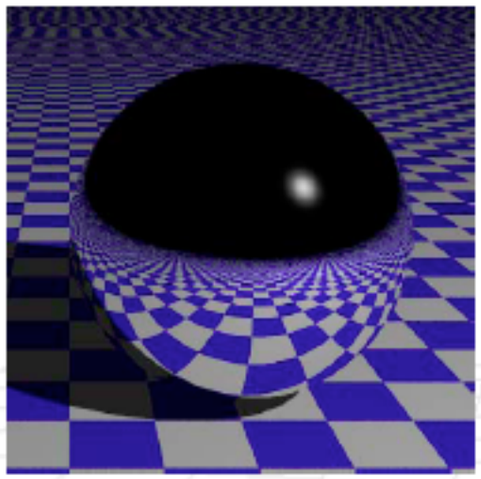


# All 3 terms together

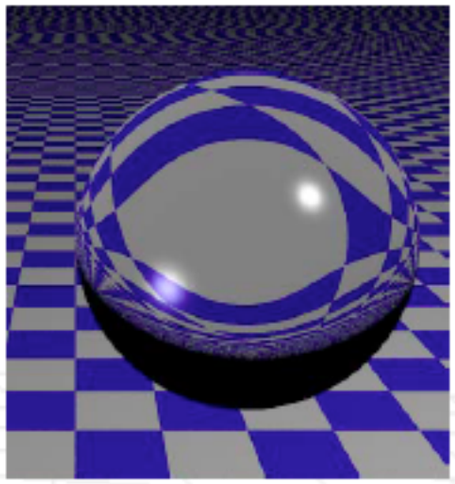
**Local Lighting Phong**



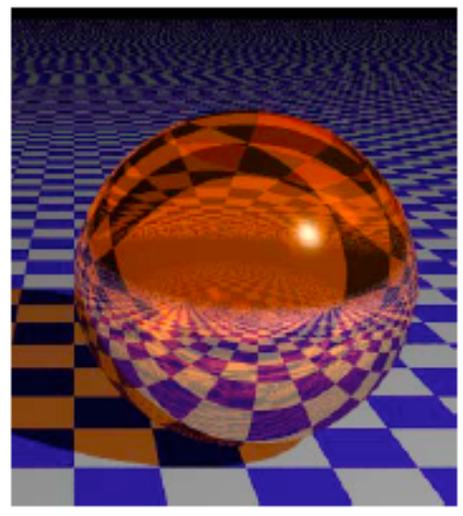
**Global Reflections (and shadows)**



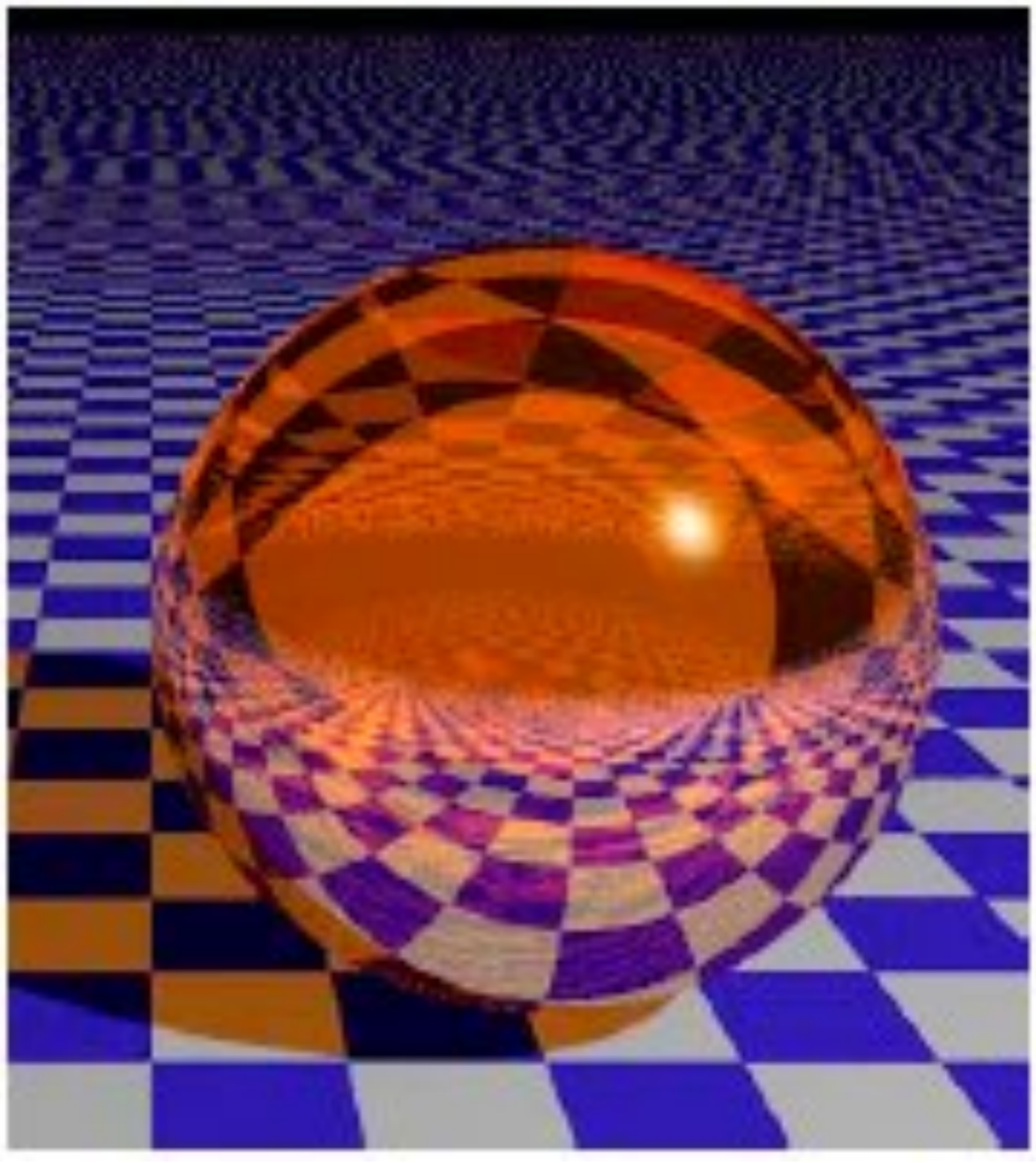
**Refractions**



**Final Image**



# Output



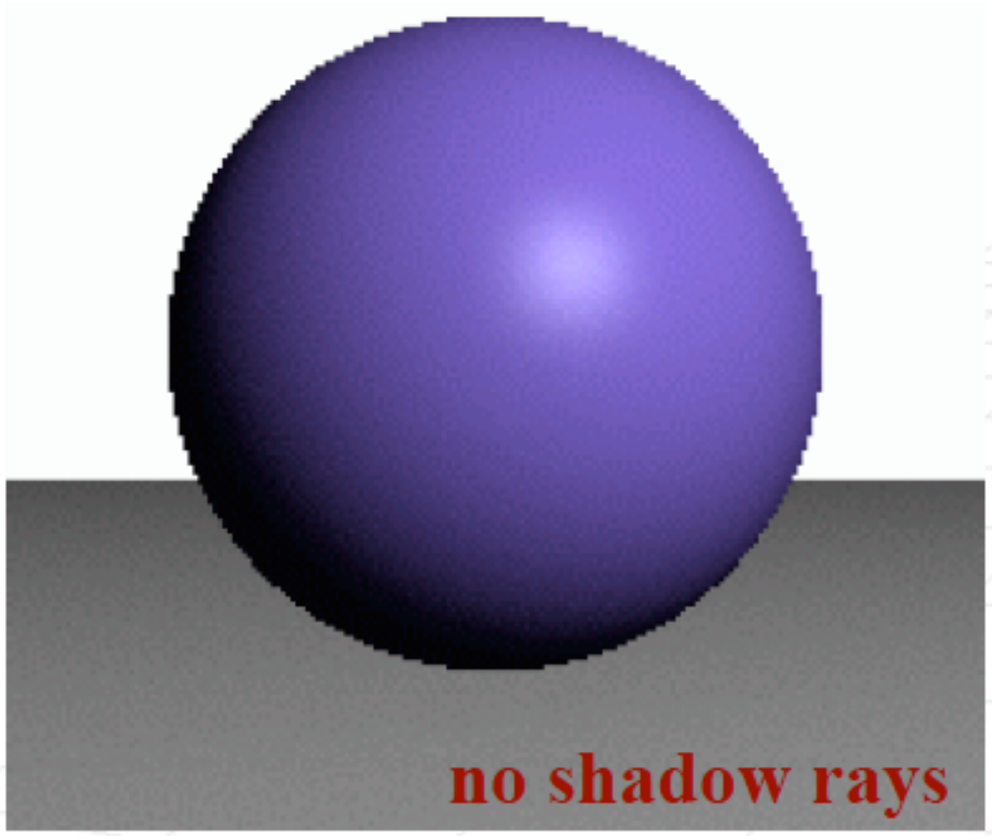
# Anti Aliasing & Monte Carlo Sampling

# Aliasing Artifacts

- We shoot 1 ray per pixel
  - Shadow rays, refraction rays can result in blocky rendering

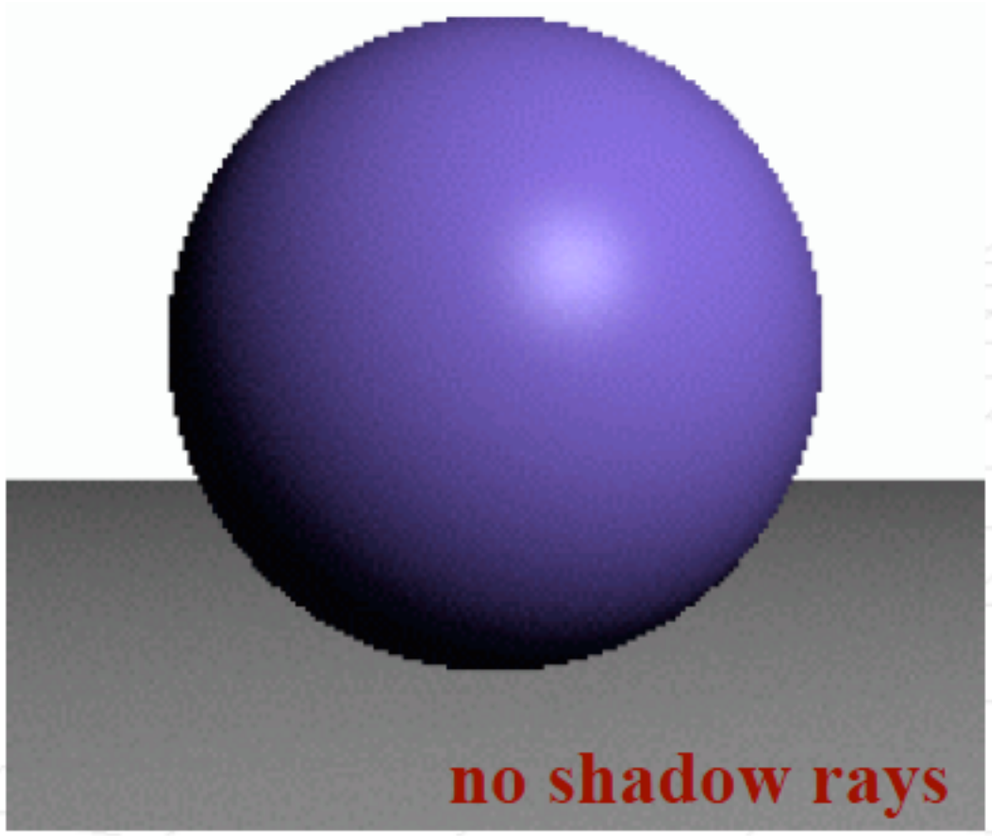
# Aliasing Artifacts

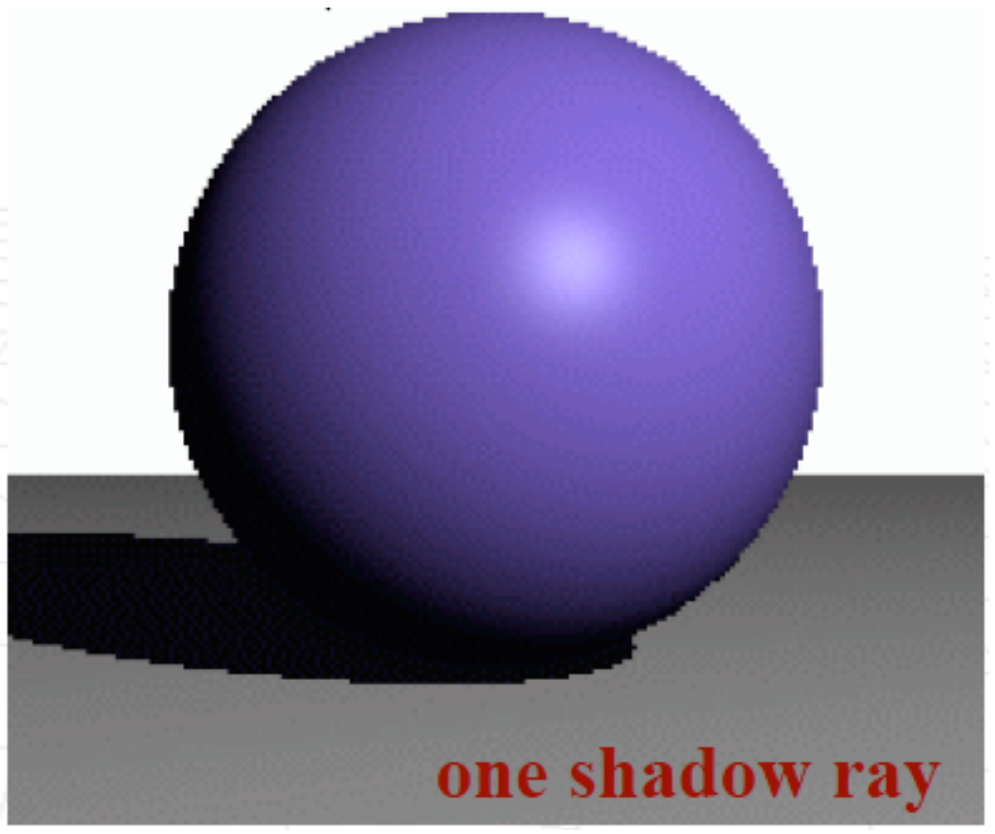
- We shoot 1 ray per pixel
  - Shadow rays, refraction rays can result in blocky rendering



# Aliasing Artifacts

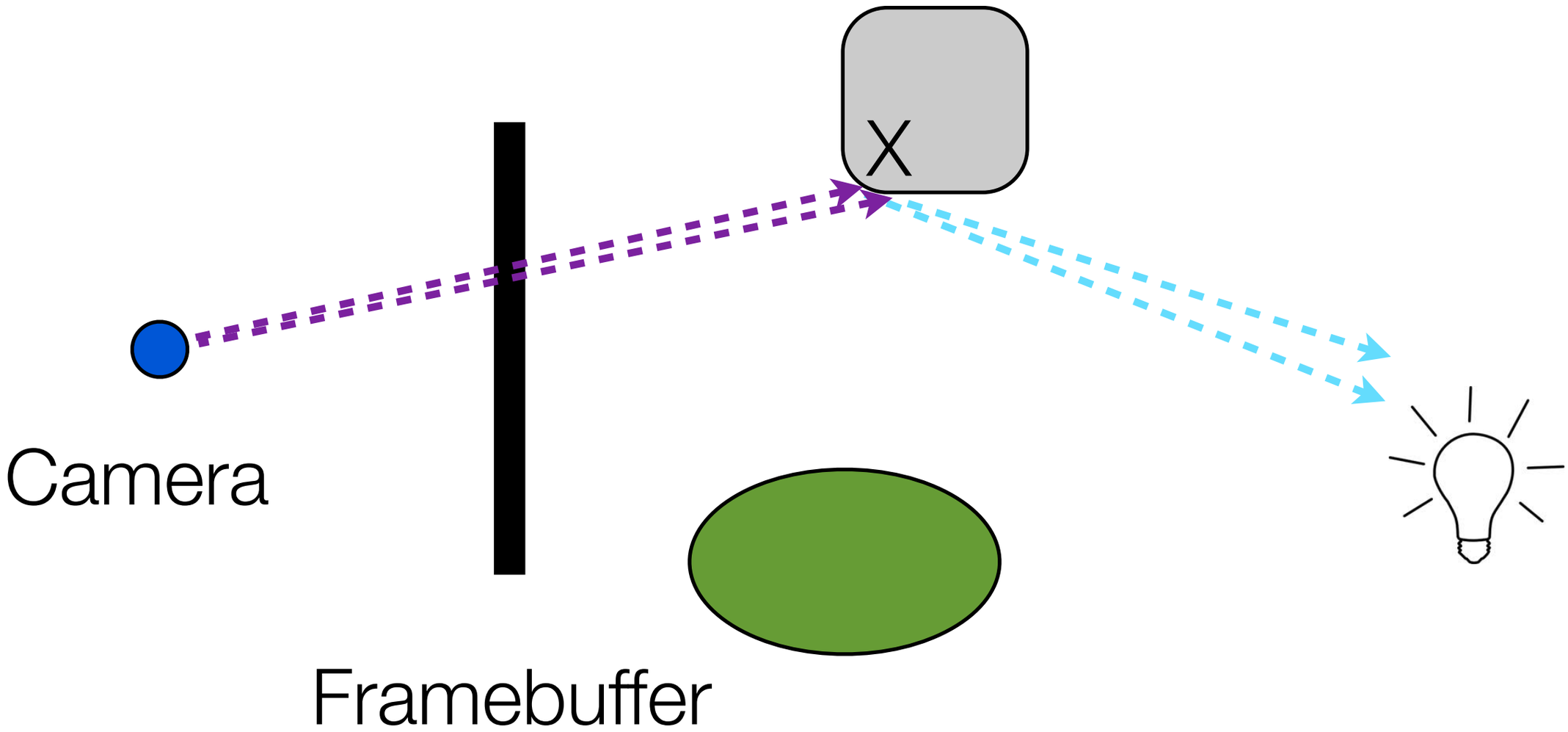
- We shoot 1 ray per pixel
  - Shadow rays, refraction rays can result in blocky rendering



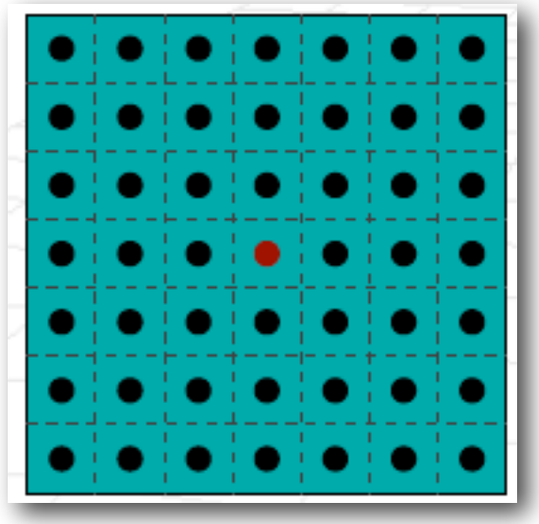
# Anti-aliasing

- Multiple rays from the same point
  - Accumulate the shadow solution for all rays as a weight



# Approach 1 - Deterministic Integration

- Subdivide a pixel into uniform blocks
- Cast rays at the center of each block
- Average the result by number of sub-pixels

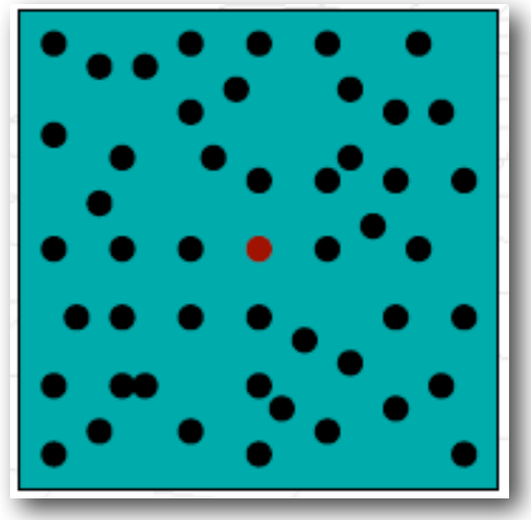


One Pixel

- Structured artifacts (noise), repeating textures

# Approach 2 - Monte Carlo Integration

- Sample randomly inside the pixel “area”
- Cast rays for every sample
- Calculate average of the color values for all samples



One Pixel

- Does not suffer from periodic artifacts

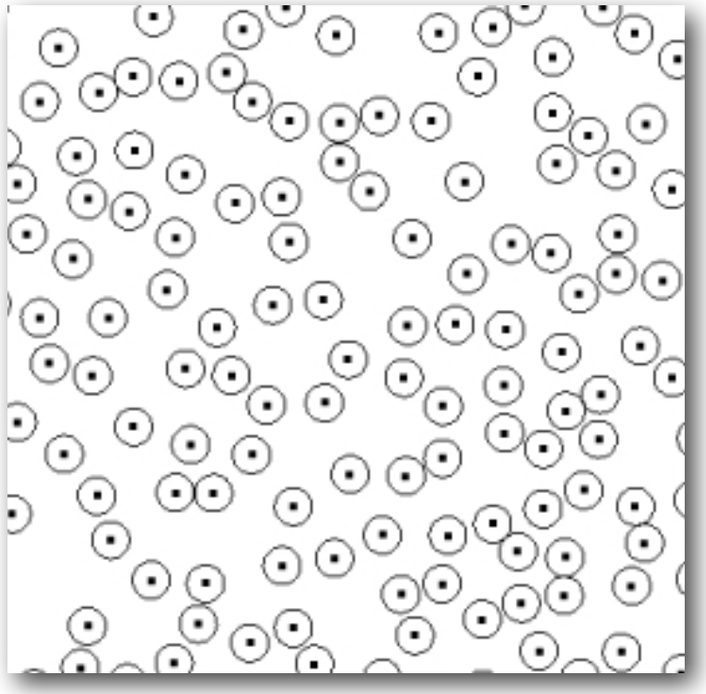
# Monte-Carlo Integration -- a very (very) simple* introduction

- Sample **randomly** inside the pixel “area”
- Most commonly provided random number generators are NOT random
  - *“rand()” is NOT a programmer’s friend*
- Need something more sophisticated than simple C function calls
  - General approach - Monte Carlo approach
  - Can be implemented using various *techniques*

* Yes, it is very simplified

# Poisson Distribution

- A technique under Monte Carlo methods
- Also known as d-distance sampling
- Basic idea
  - Every sample that is chosen is separated from all other samples by at least a distance “d”



# Stratified Sampling

- Uniform sampling can result in a very bad distribution of points
- Divide the pixel into stratified regions (grid)
- Sample randomly inside these regions
- Used as approximation to Poisson Sampling

# Stratified Sampling

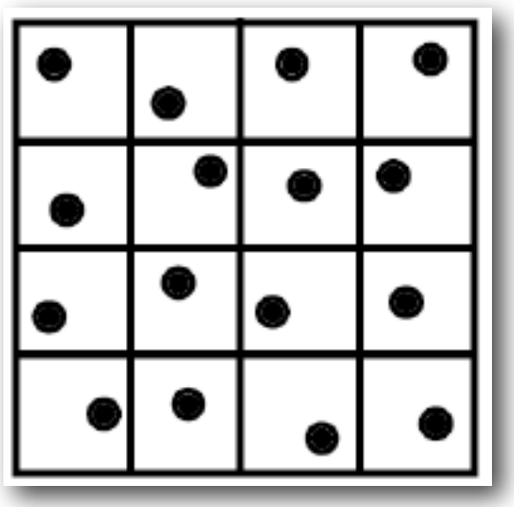
- Uniform sampling can result in a very bad distribution of points

- Divide the pixel into stratified regions (grid)
- Sample randomly inside these regions
- Used as approximation to Poisson Sampling

# Stratified Sampling

- Uniform sampling can result in a very bad distribution of points

- Divide the pixel into stratified regions (grid)
- Sample randomly inside these regions
- Used as approximation to Poisson Sampling



# Importance Sampling

# Importance Sampling

- Choose samples in areas that are more “important”

# Importance Sampling

- Choose samples in areas that are more “important”
  - more likely to occur

# Importance Sampling

- Choose samples in areas that are more “important”
  - more likely to occur

# Importance Sampling

- Choose samples in areas that are more “important”
  - more likely to occur
- Chicken-and-egg problem

# Importance Sampling

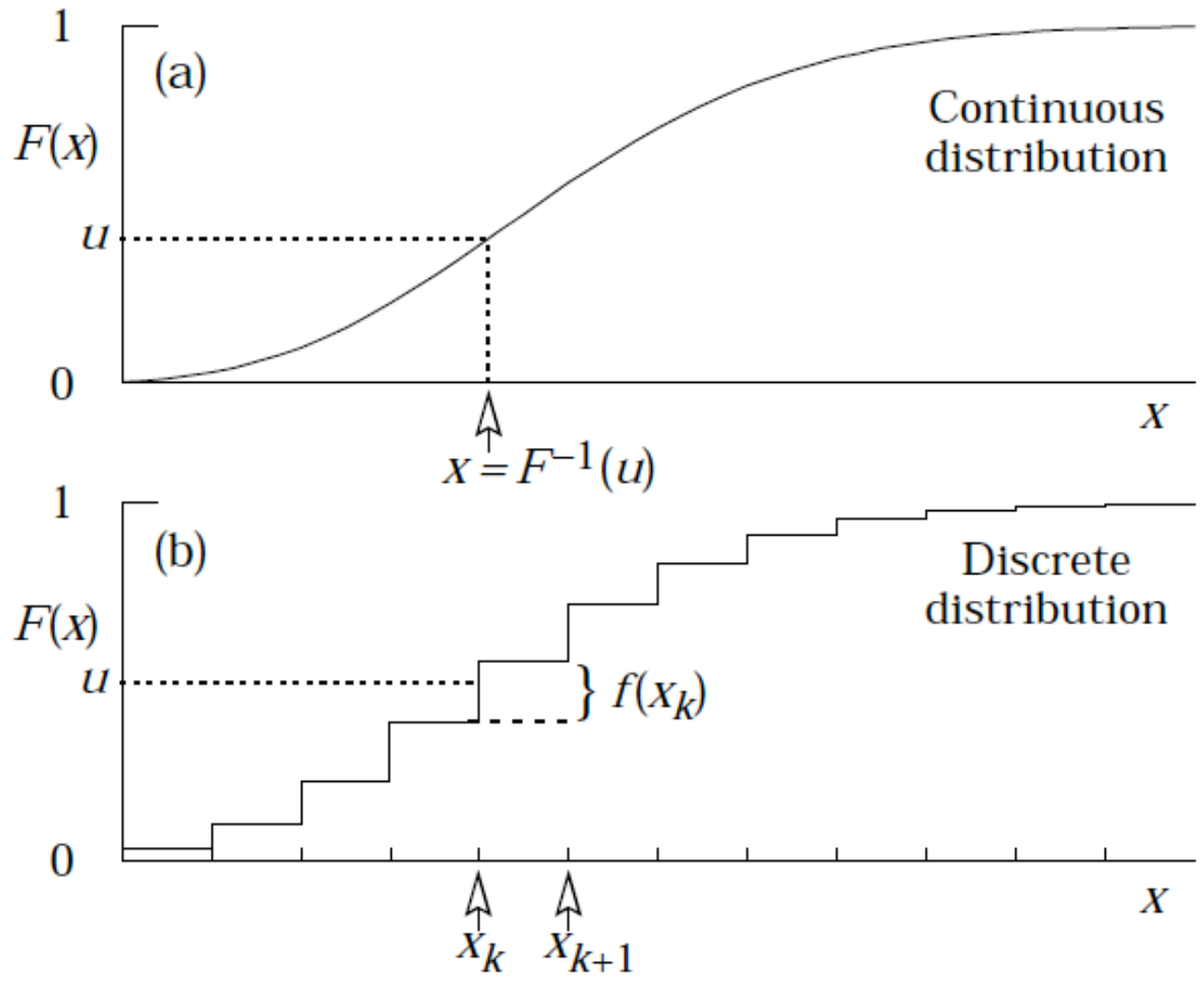
- Choose samples in areas that are more “important”
  - more likely to occur
- Chicken-and-egg problem
  - How do we know if a value for a random variable is most likely to occur??

# Importance Sampling

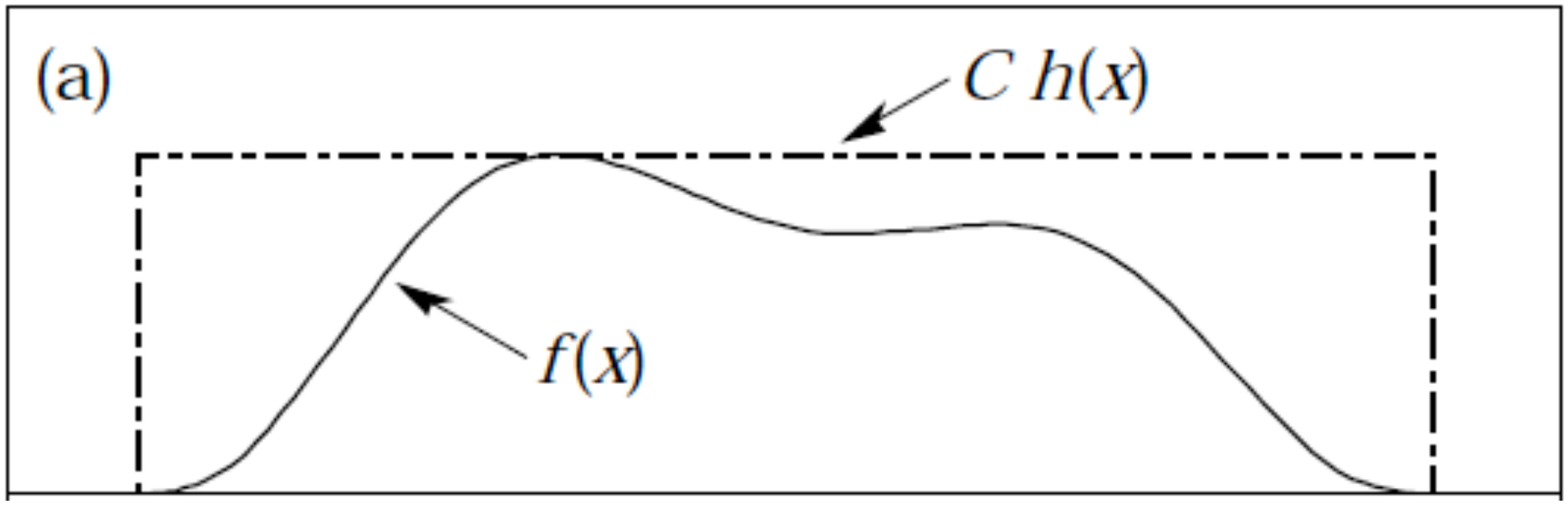
- Choose samples in areas that are more “important”
  - more likely to occur
- Chicken-and-egg problem
  - How do we know if a value for a random variable is most likely to occur??
  - Estimate the Probability distribution function!

# Importance Sampling

- Choose samples in areas that are more “important”
  - more likely to occur
- Chicken-and-egg problem
  - How do we know if a value for a random variable is most likely to occur??
  - Estimate the Probability distribution function!

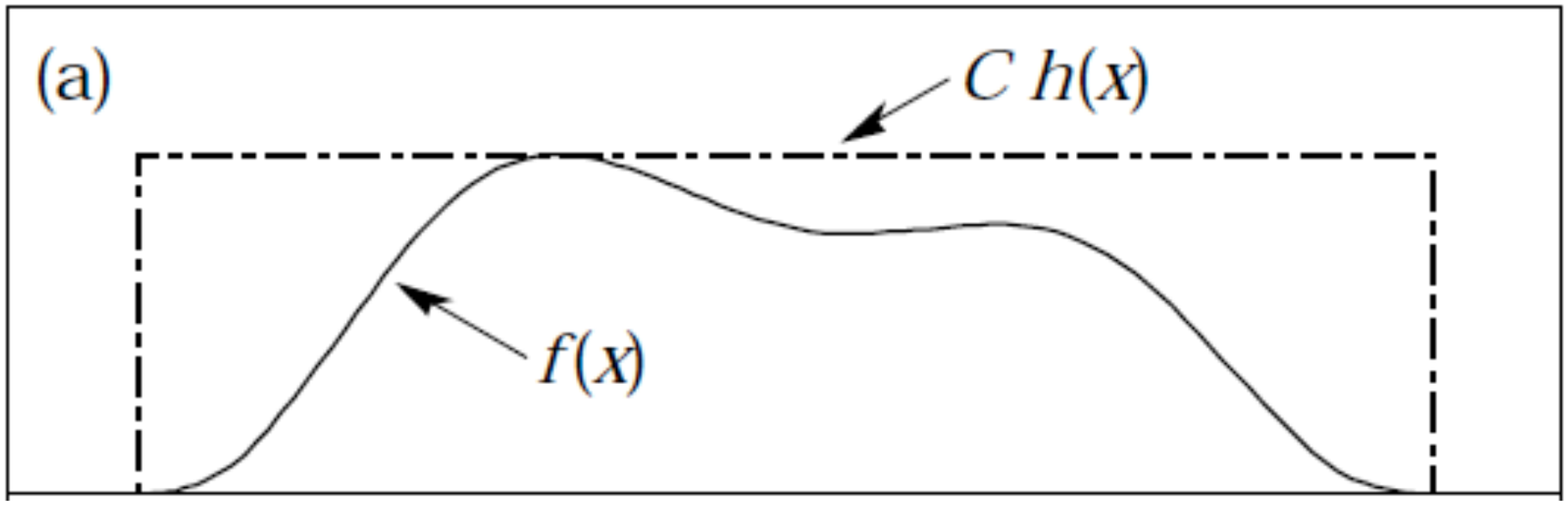


# Accept/Reject Sampling




h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

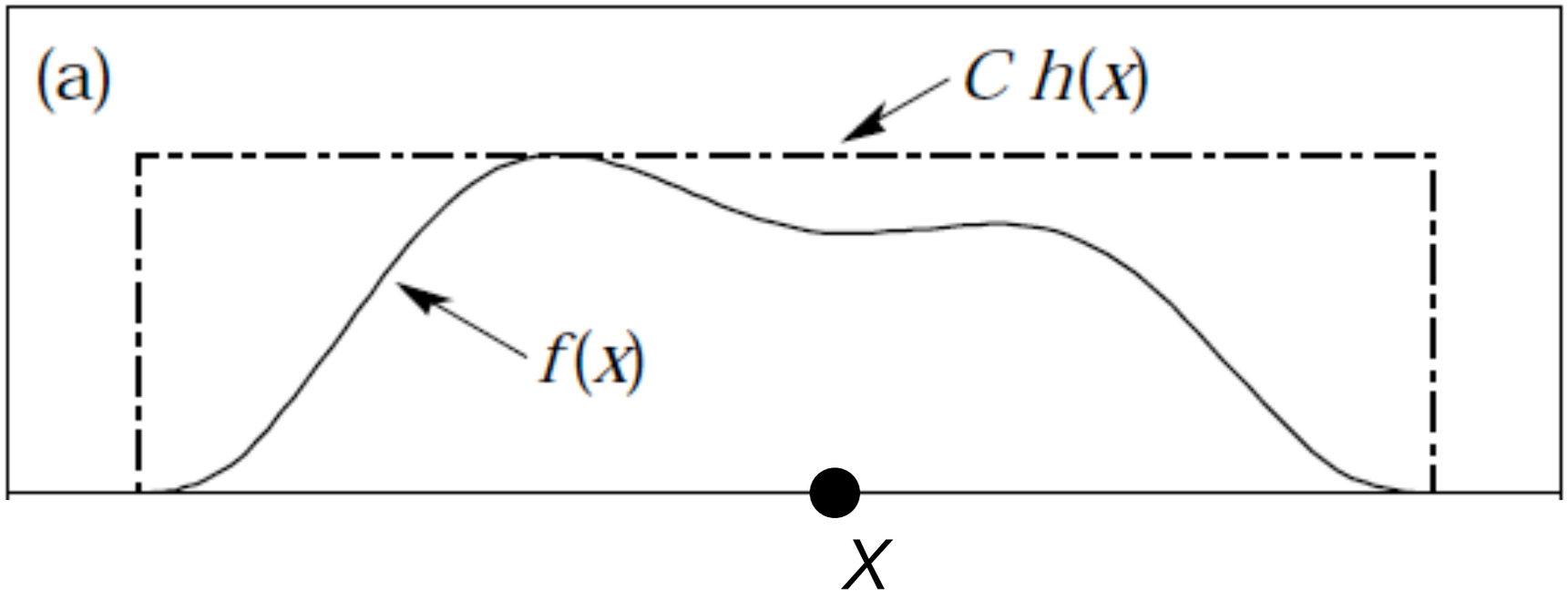
# Accept/Reject Sampling (2)




Step 1: Generate x based on h(x)

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

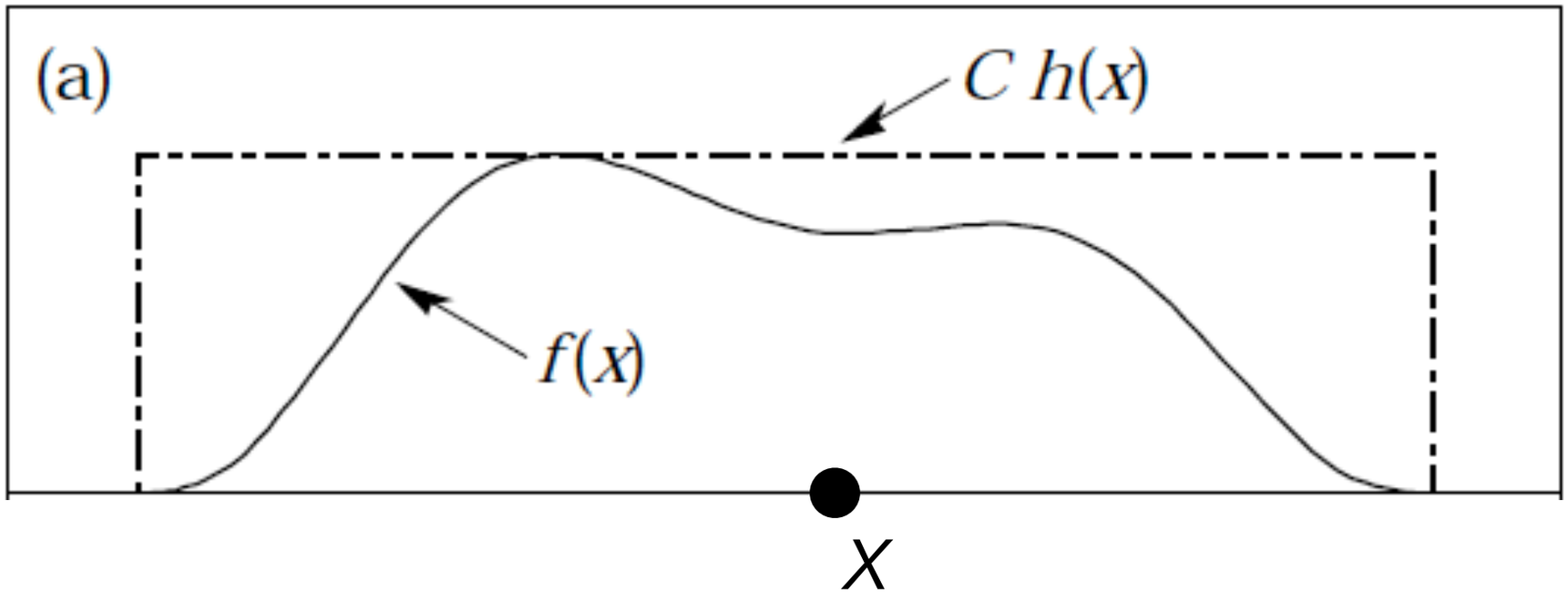
# Accept/Reject Sampling (2)




Step 1: Generate x based on h(x)

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

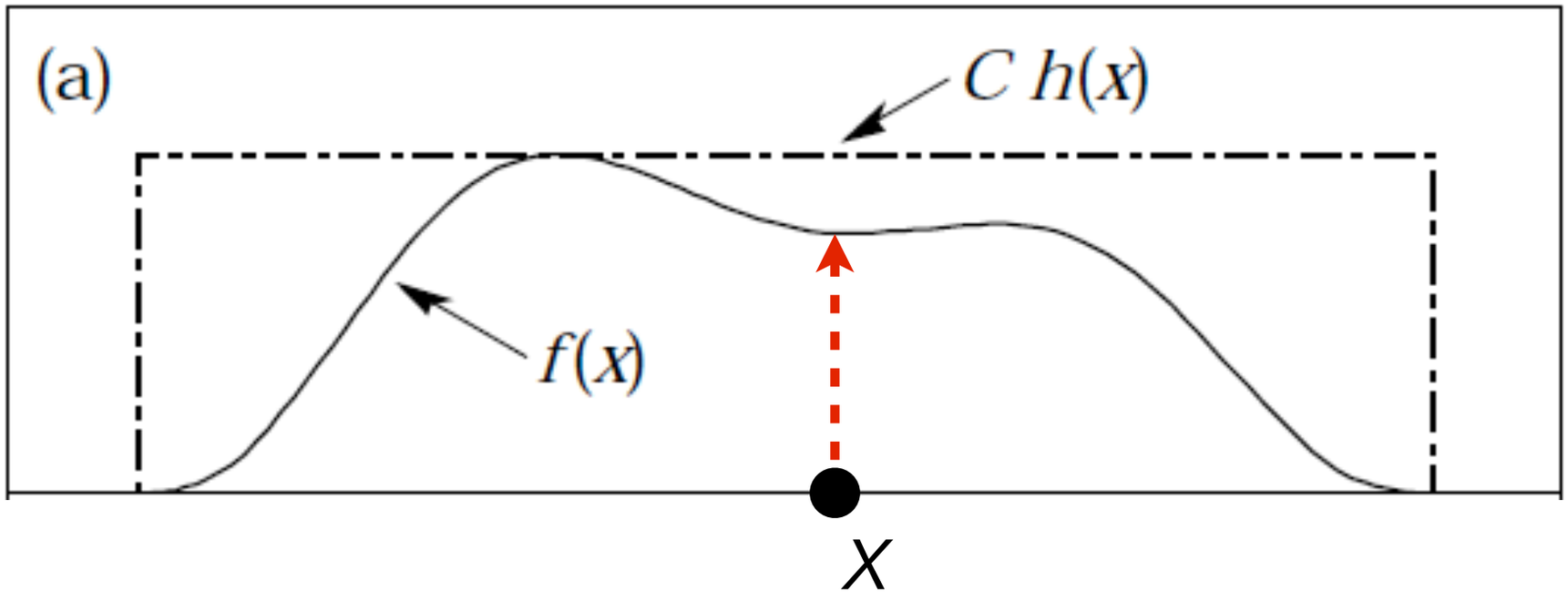
# Accept/Reject Sampling (3)




Step 2: Calculate f(x), C*h(x)

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

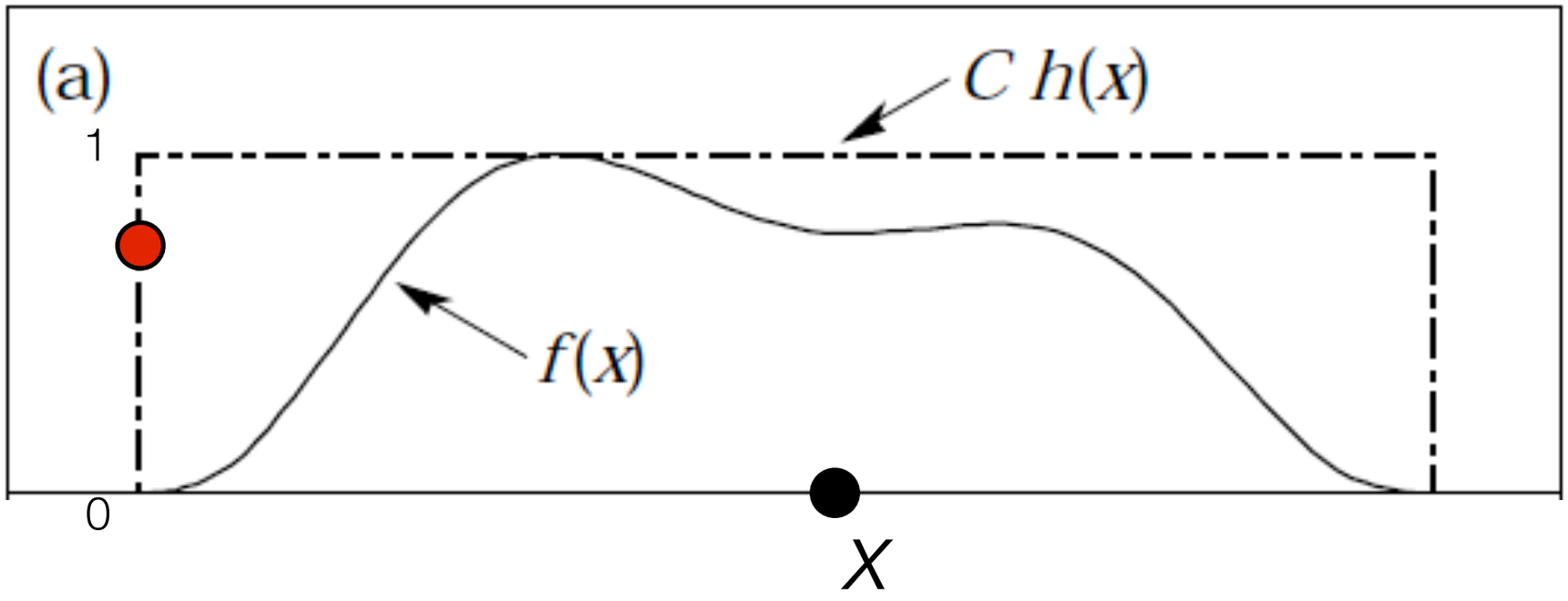
# Accept/Reject Sampling (3)




Step 2: Calculate f(x), C*h(x)

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

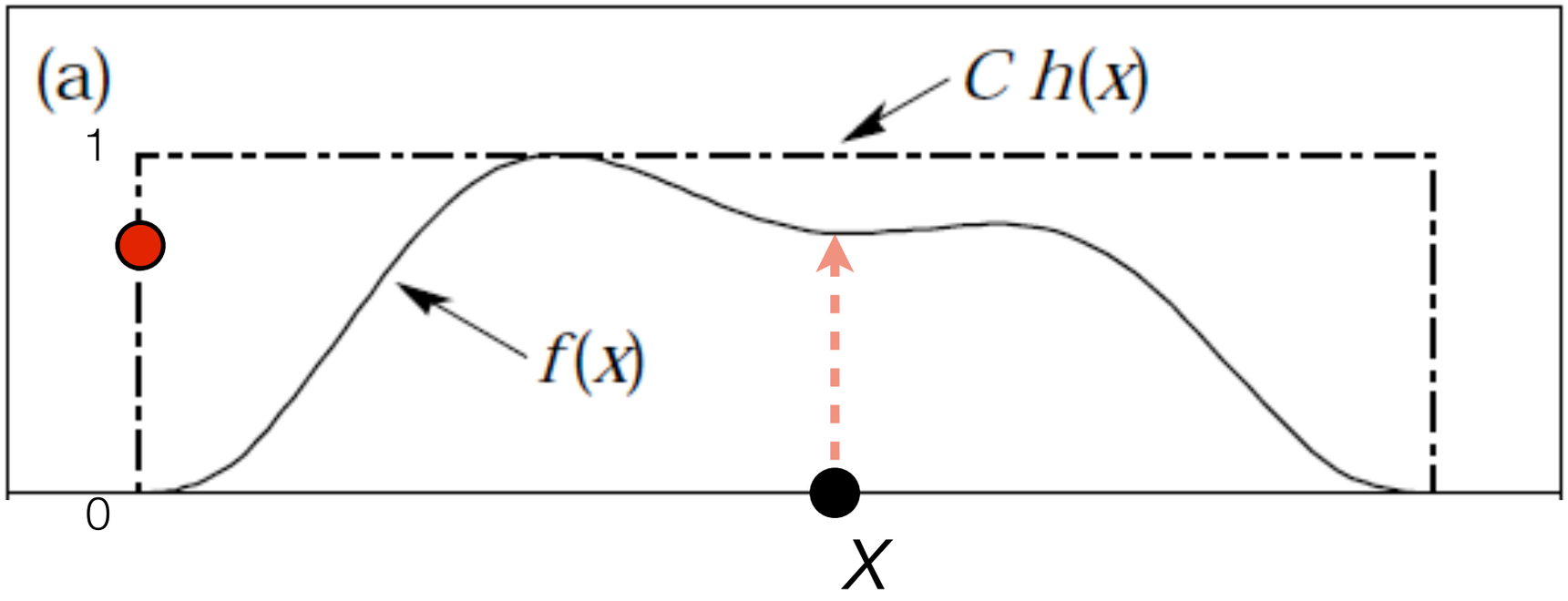
# Accept/Reject Sampling (4)




Step 3: Generate u

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

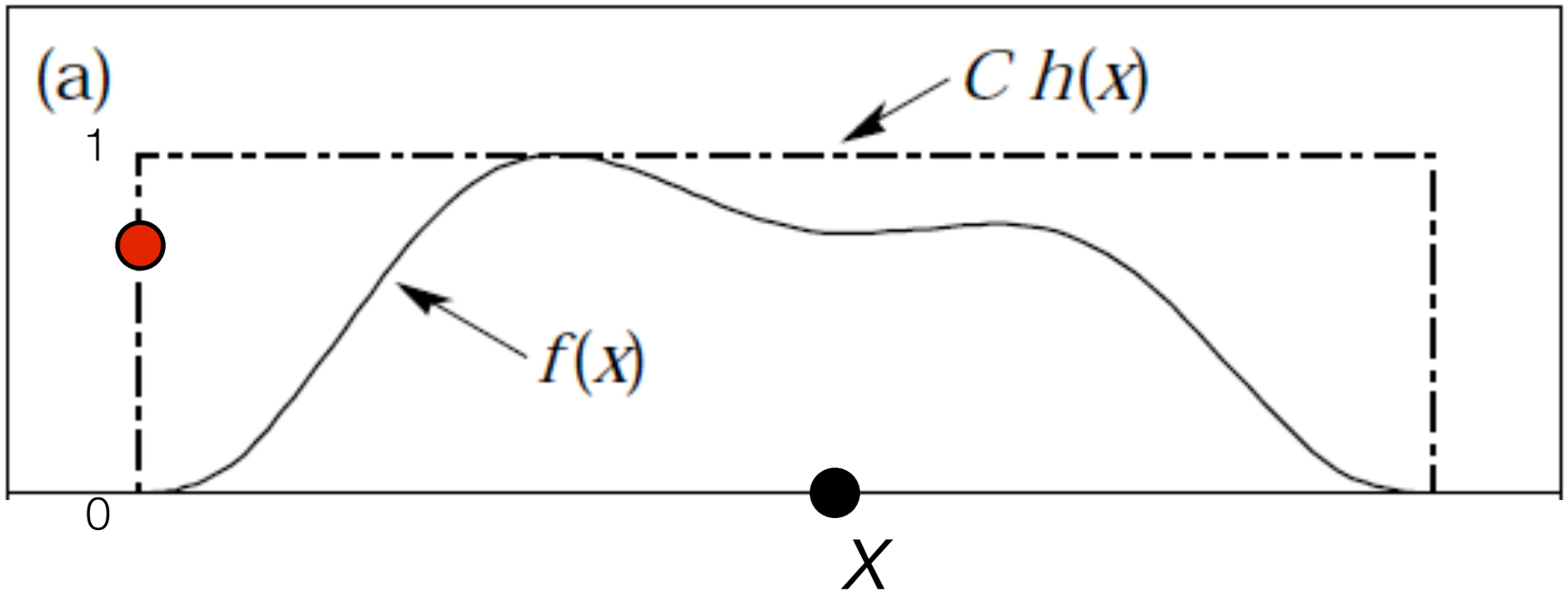
# Accept/Reject Sampling (4)




Step 3: Generate u

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

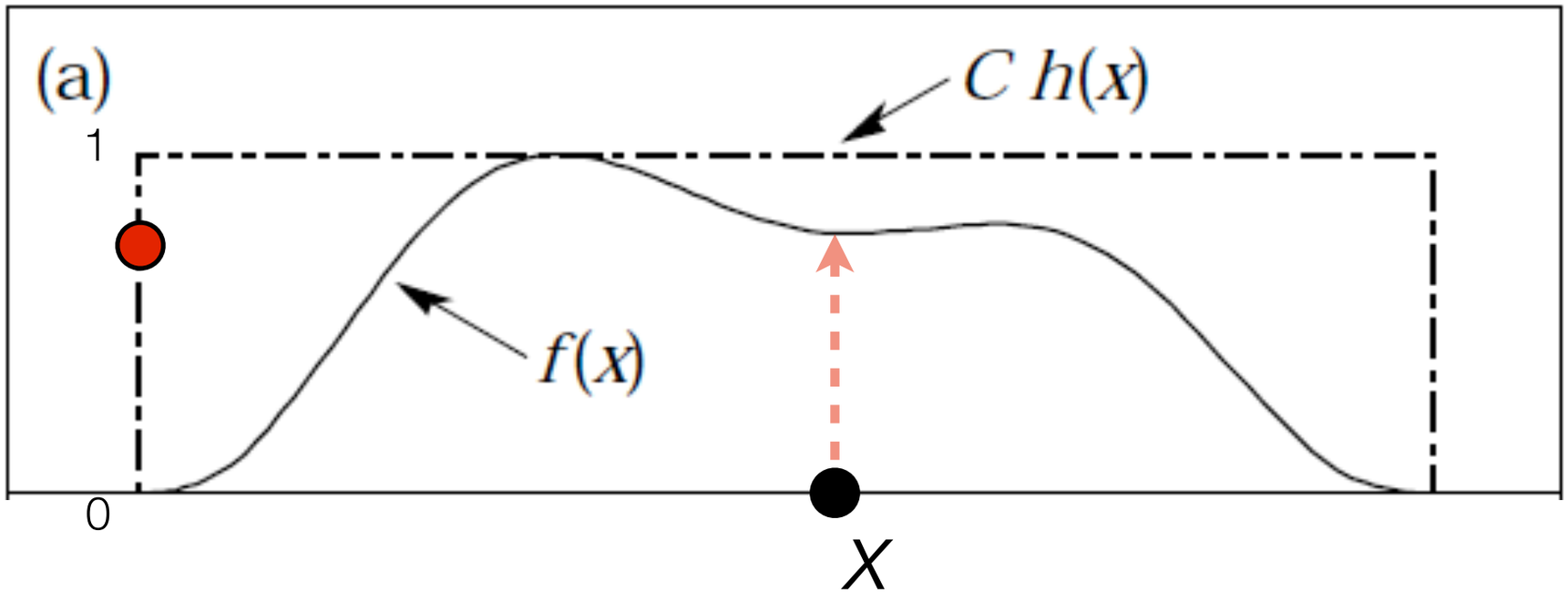
# Accept/Reject Sampling (5)




Step 4: If u*C*h(x) <= f(x), accept x

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

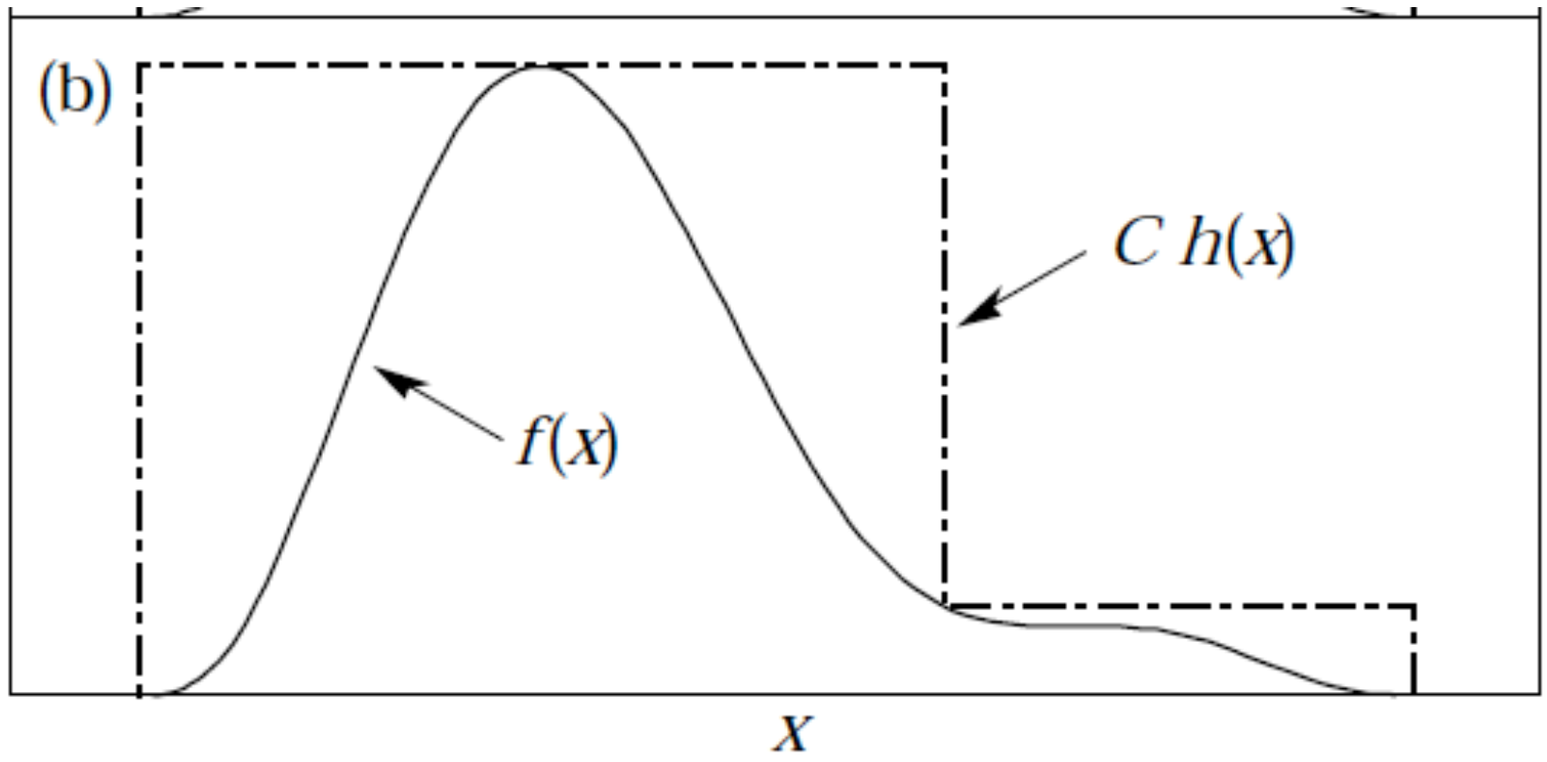
# Accept/Reject Sampling (5)




Step 4: If u*C*h(x) <= f(x), accept x

h(x) : Uniform sampling distribution
C : scalar used as a scale
f(x) : Unknown PDF

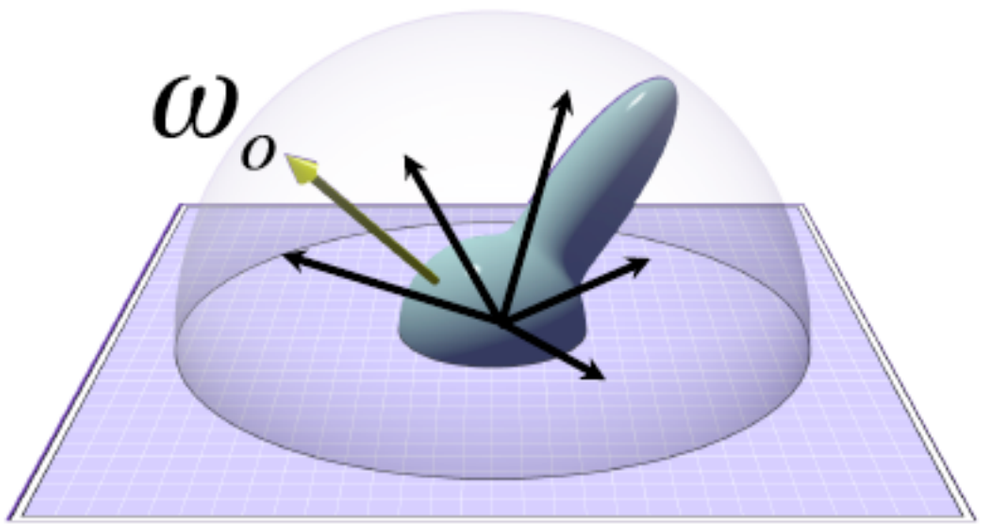
# Importance Sampling



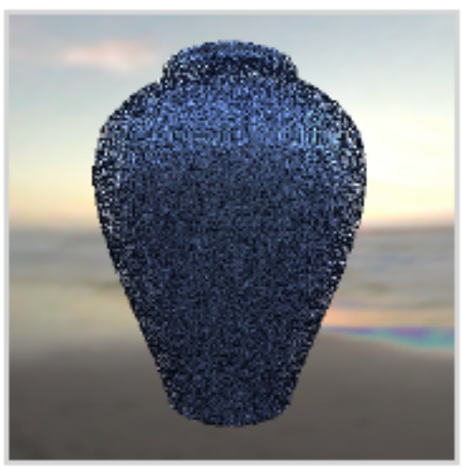

More likely to generate (and accept) x values that are closer to f(x)

# Uniform Vs Importance Sampling in Ray Tracing

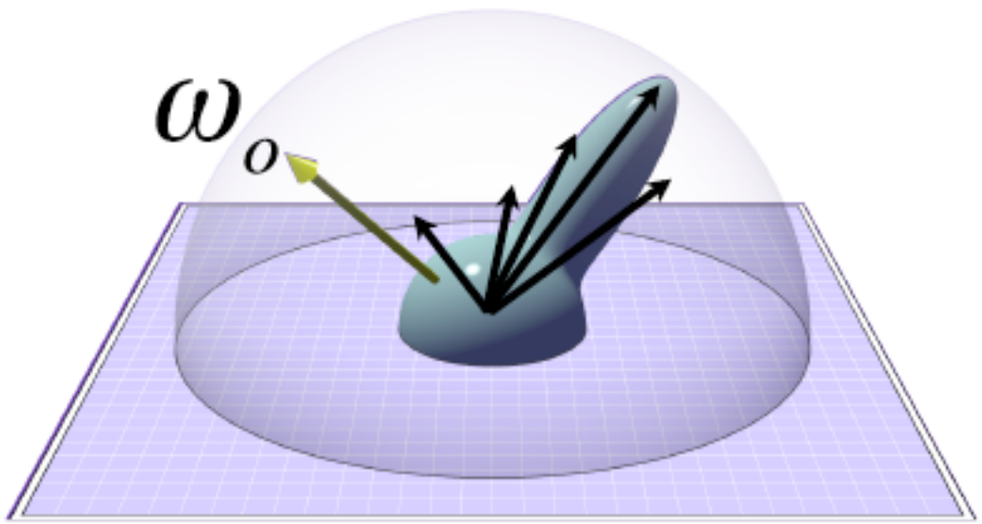
$U(\omega_i)$

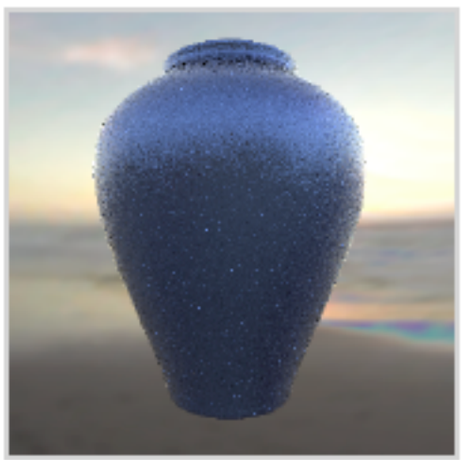



5 Samples/Pixel



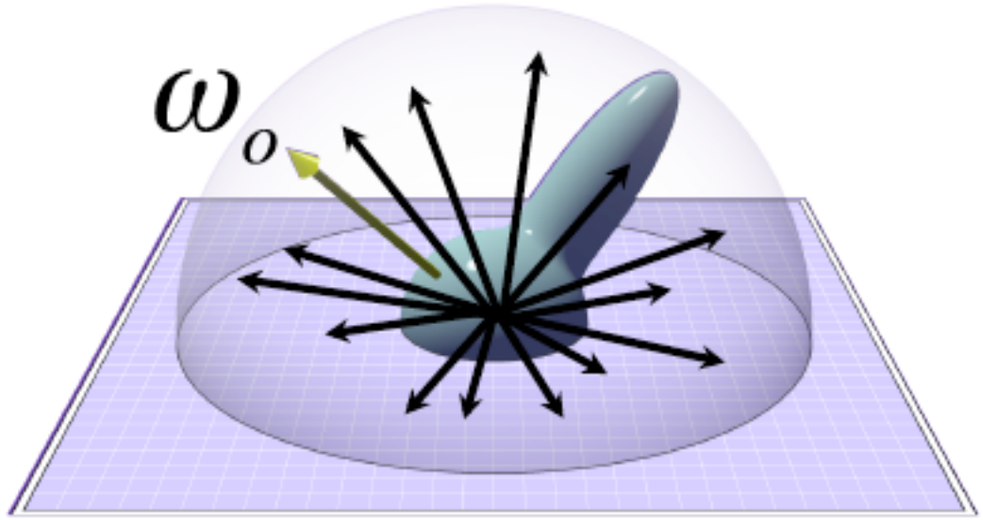
$P(\omega_i)$



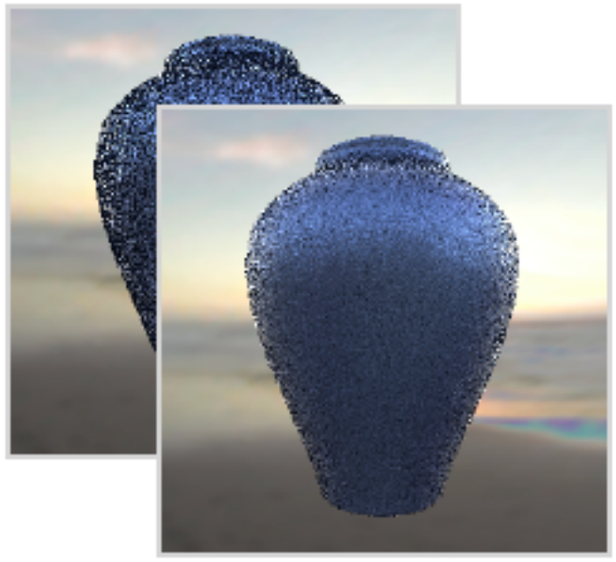



# Increasing samples

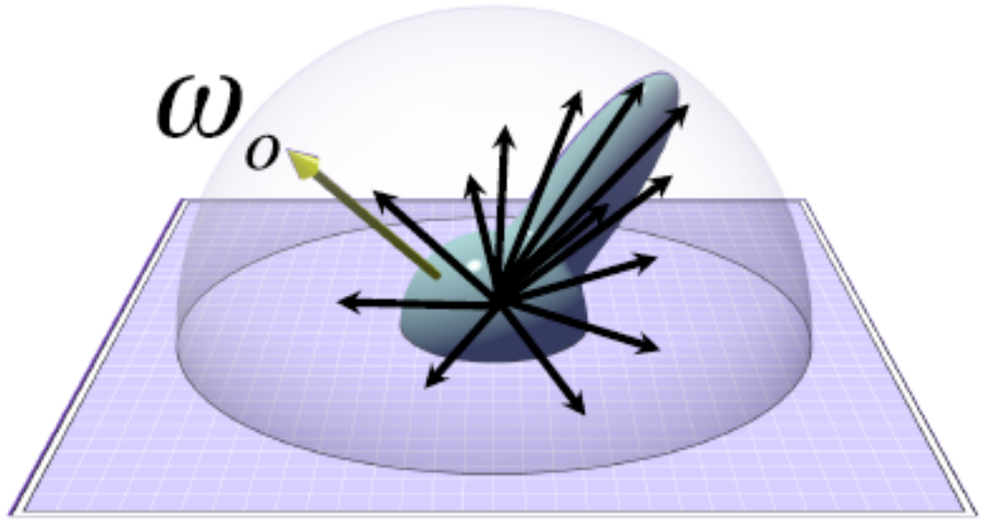
$U(\omega_i)$

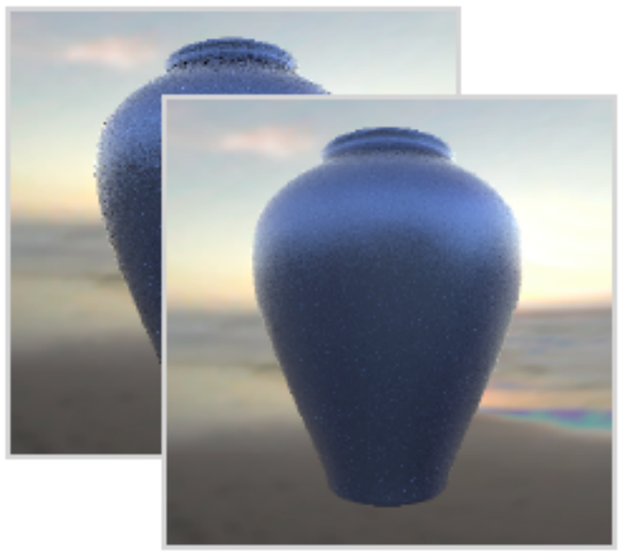



25 Samples/Pixel



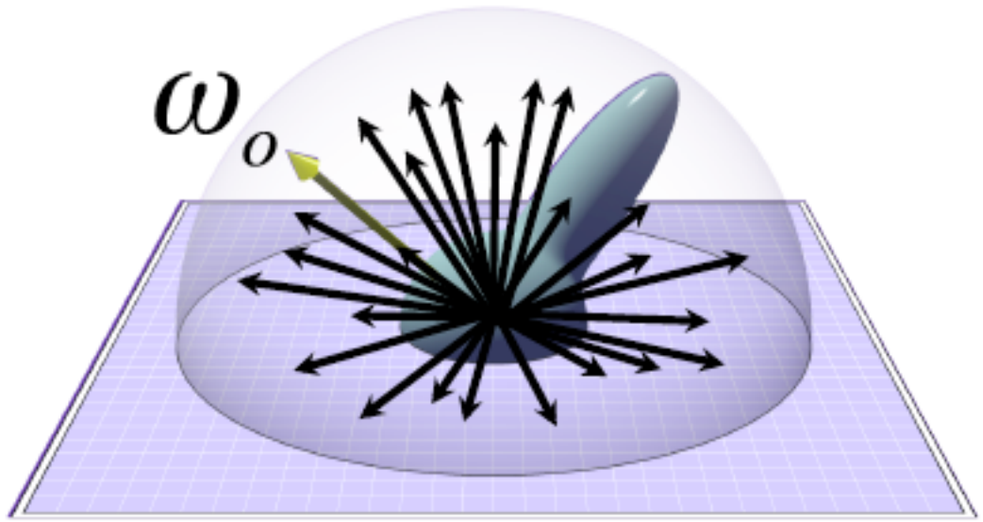
$P(\omega_i)$



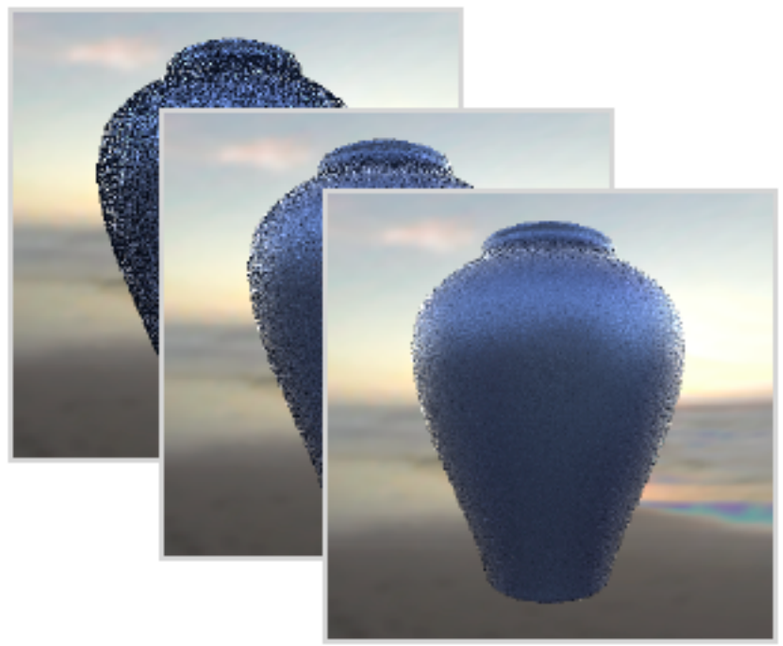



# More samples ....

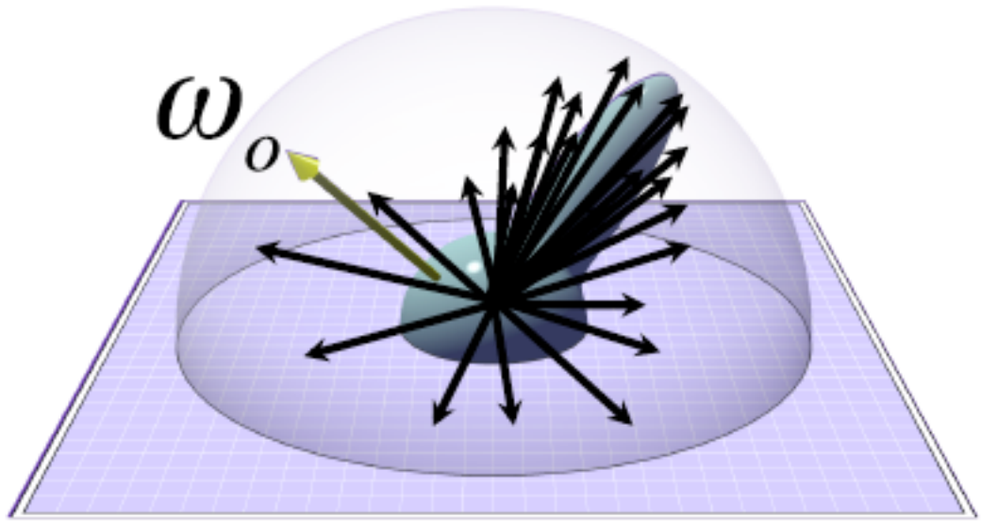
$U(\omega_i)$

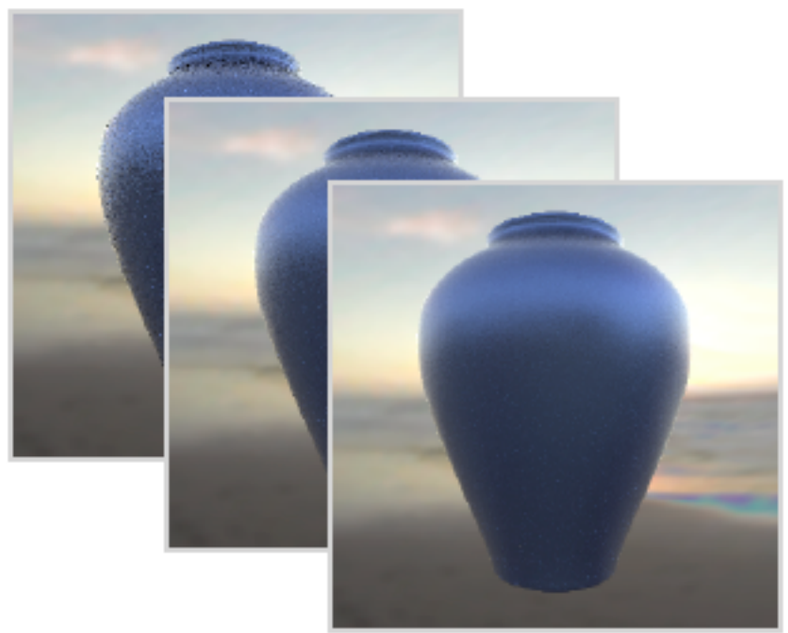



75 Samples/Pixel



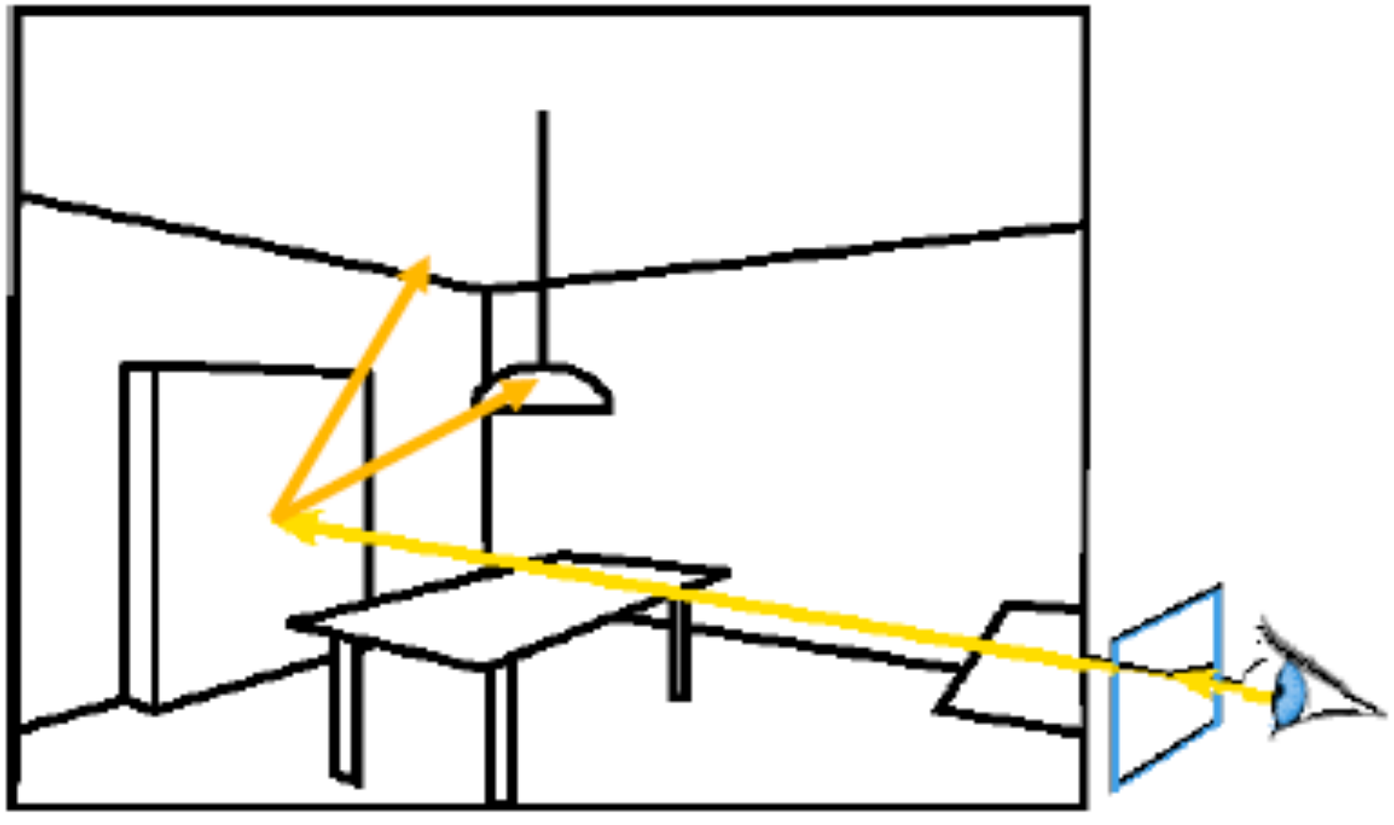
$P(\omega_i)$



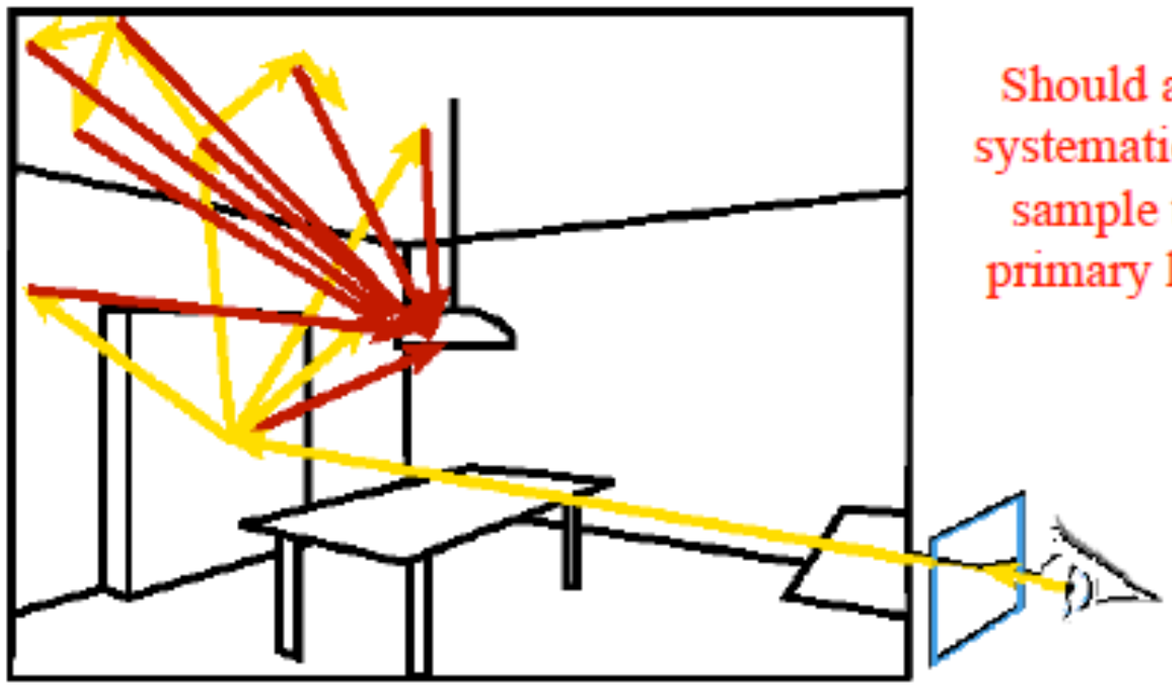
# Monte Carlo Ray Tracing

Cast primary and secondary rays for every pixel

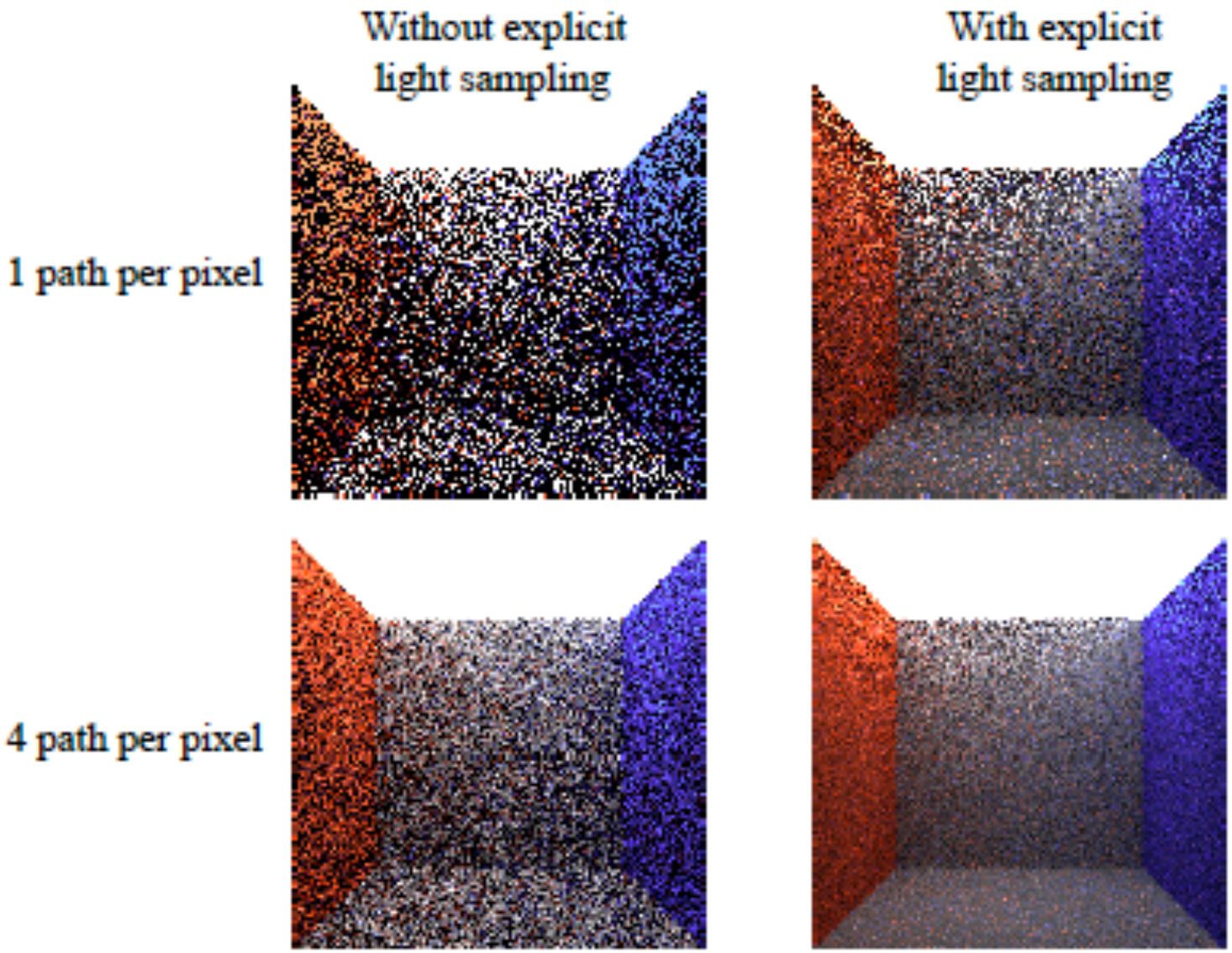


# Monte Carlo Ray Tracing

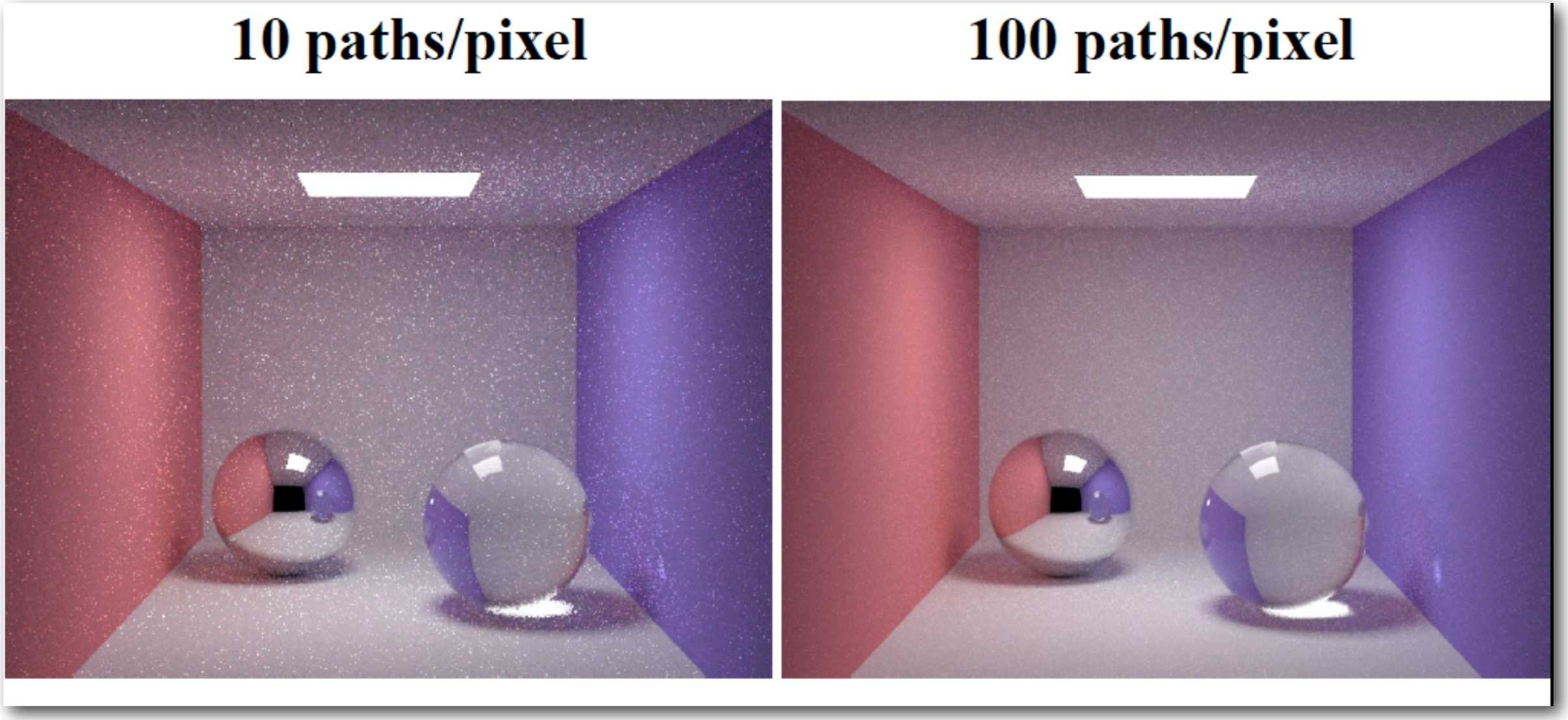
- Cast random secondary rays from one primary ray into the environment



# Importance Sampling of Light Rays



# Final Rendering



# Thank You.