Lecture #20

Distribution Ray Tracing

Computer Graphics

Winter Term 2016/17

Marc Stamminger / Roberto Grosso

Last Lecture

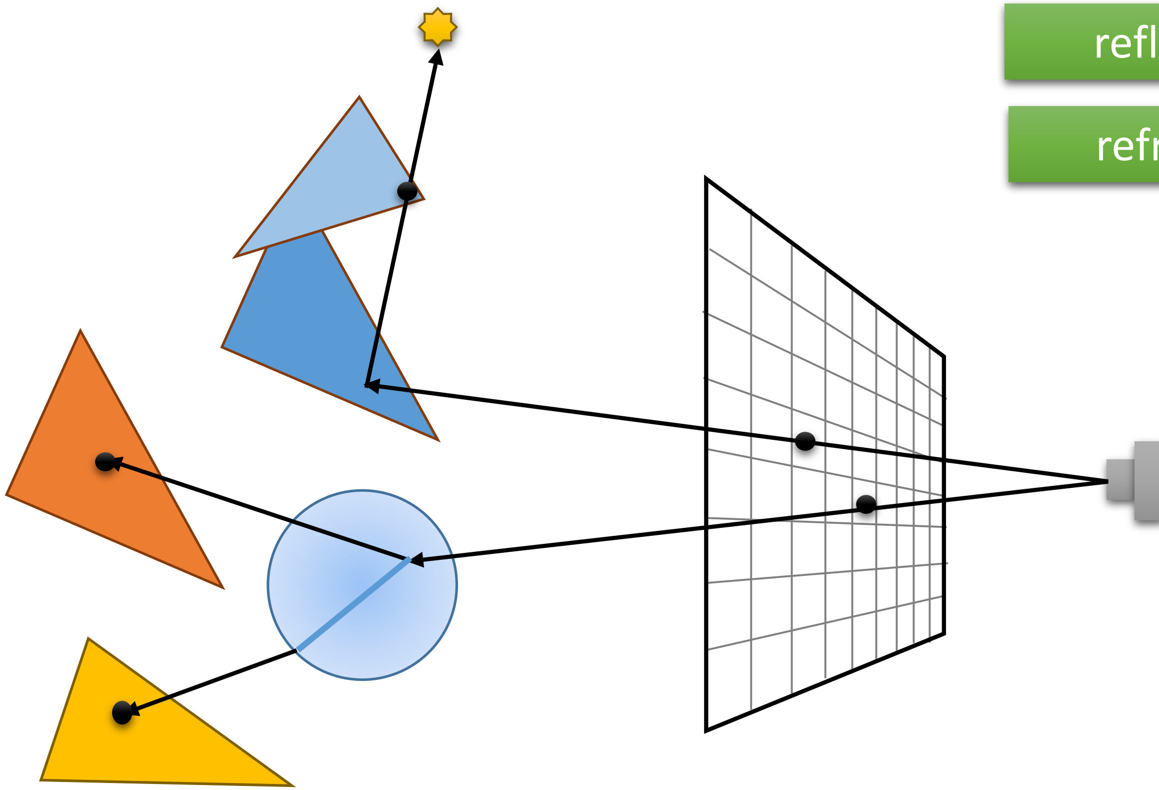
- Ray Casting → Ray Tracing

eye rays

shadow rays

reflection rays

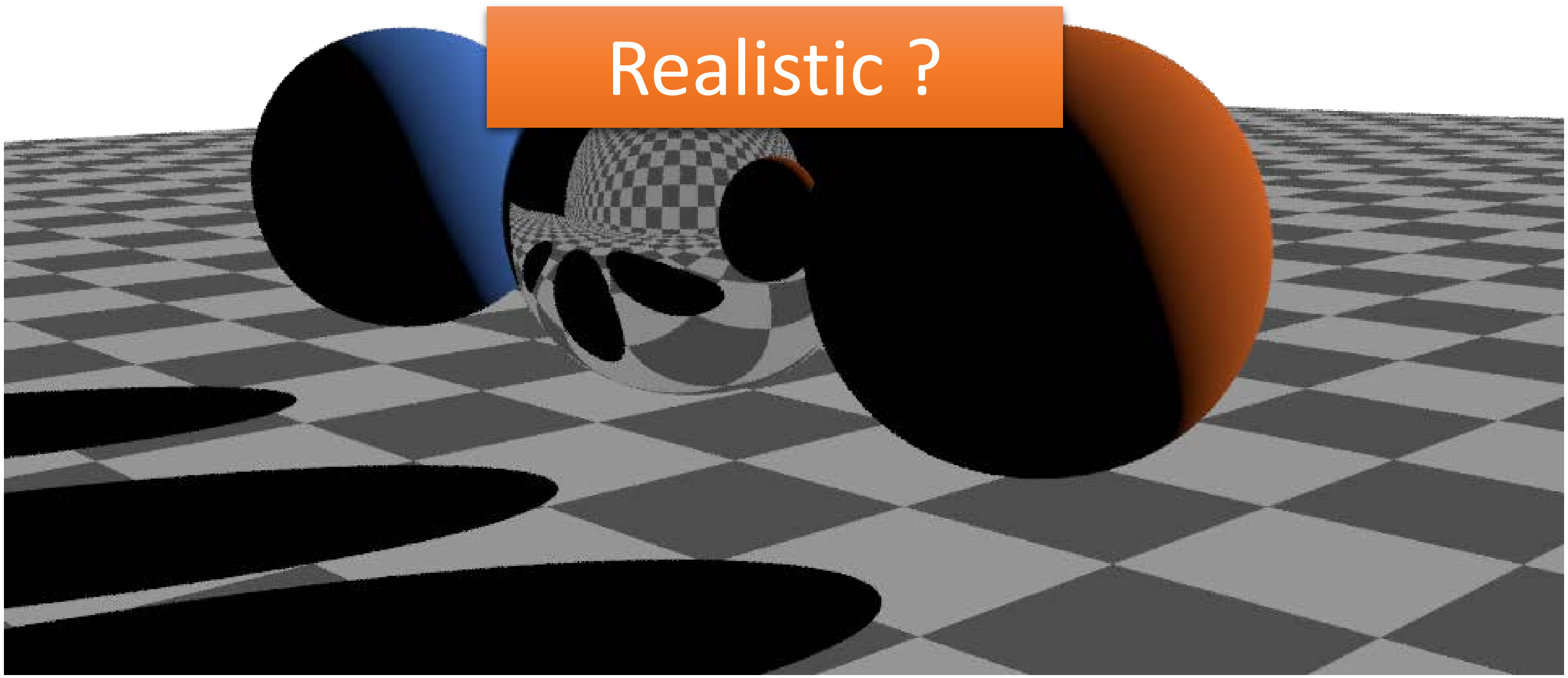
refraction rays



- **Recursive Process !**

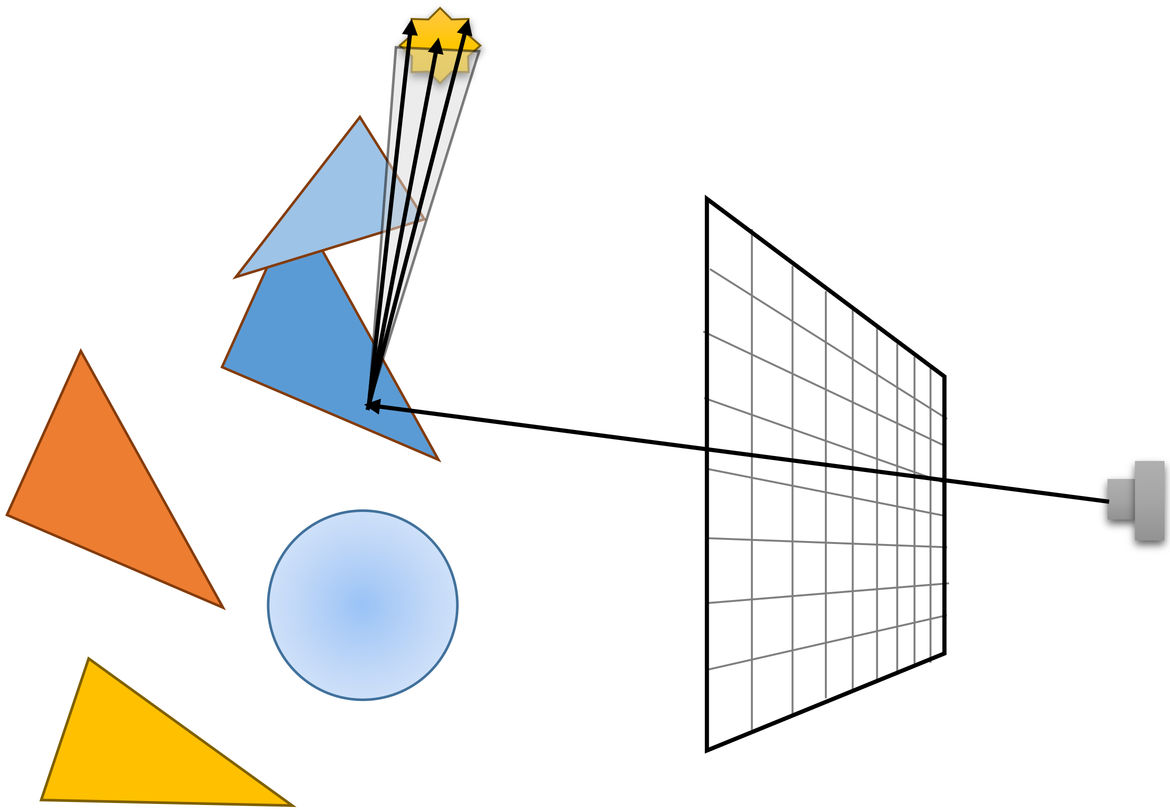
Ray Tracing

- Simple result



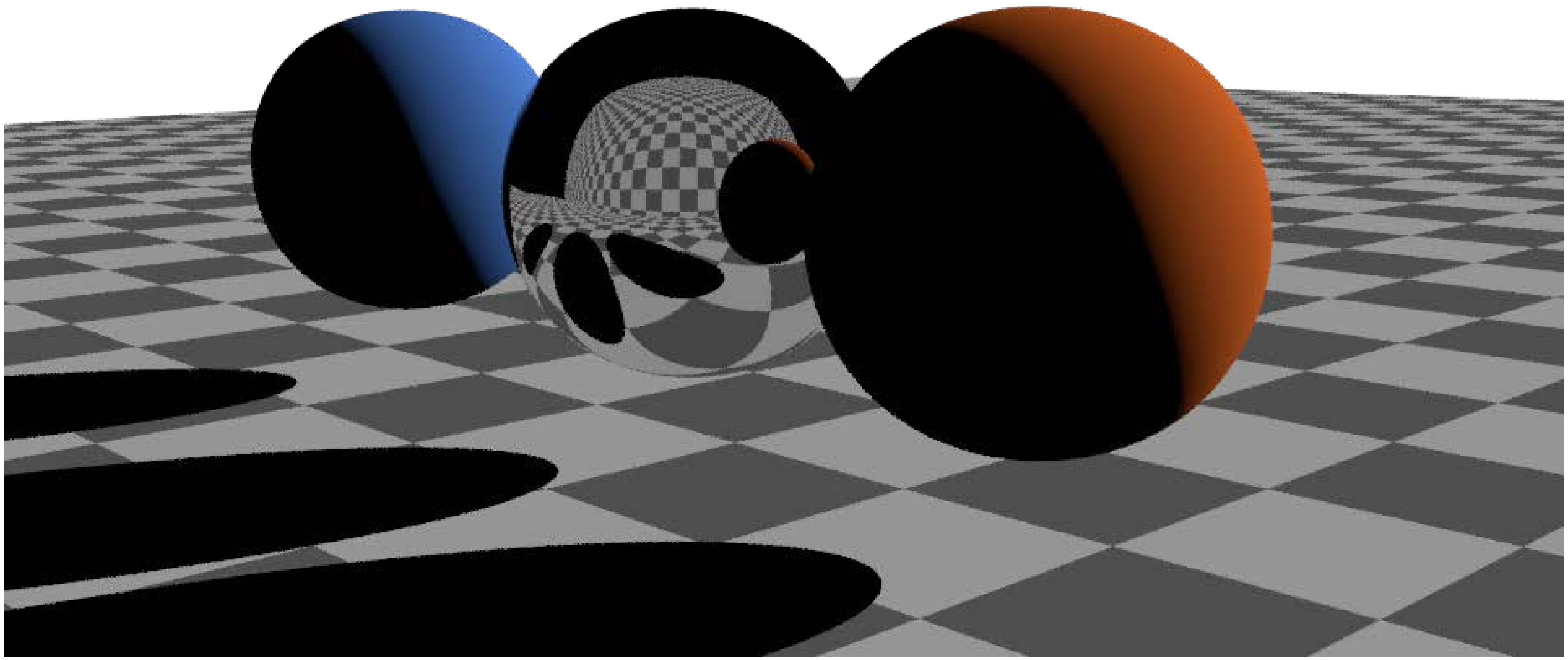
Sampling the light source

- Real light sources are not points, but have some extent
  → shadow rays should be distributed over light source



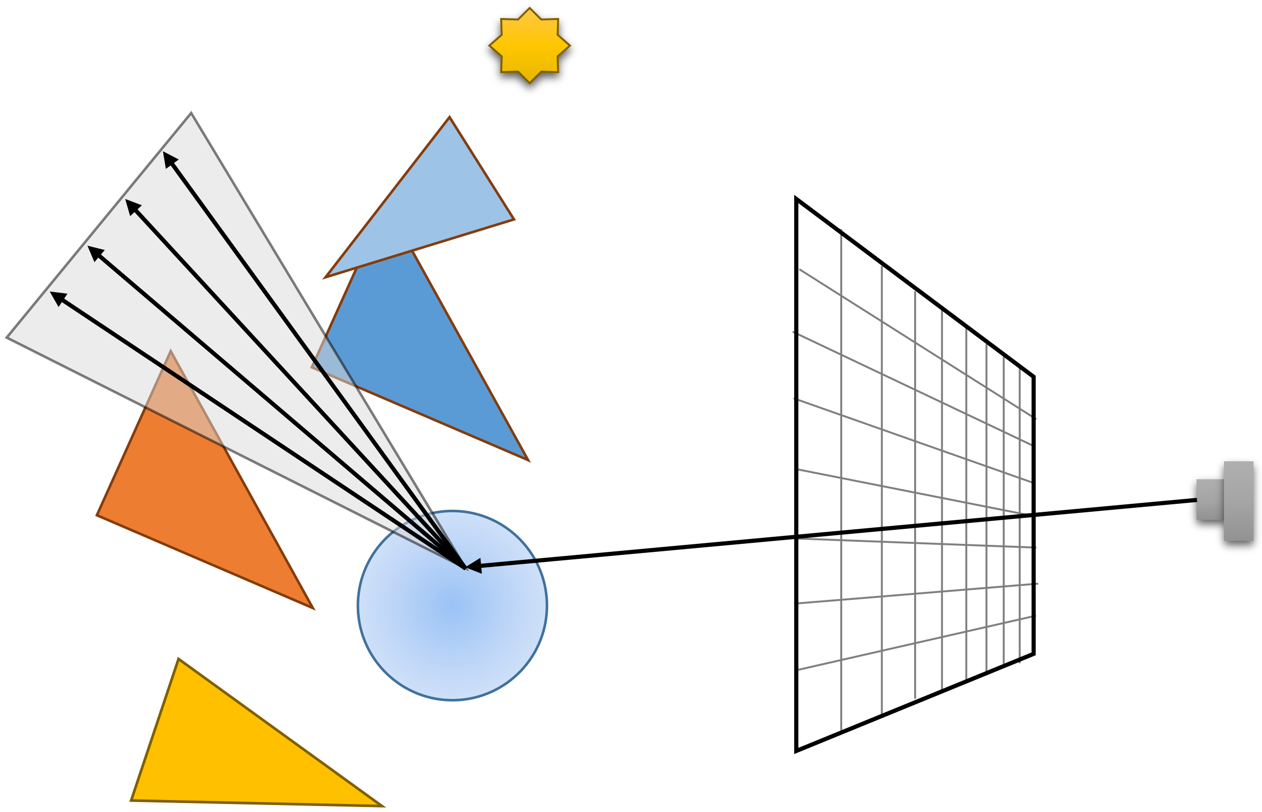
Sampling the light source

- Result



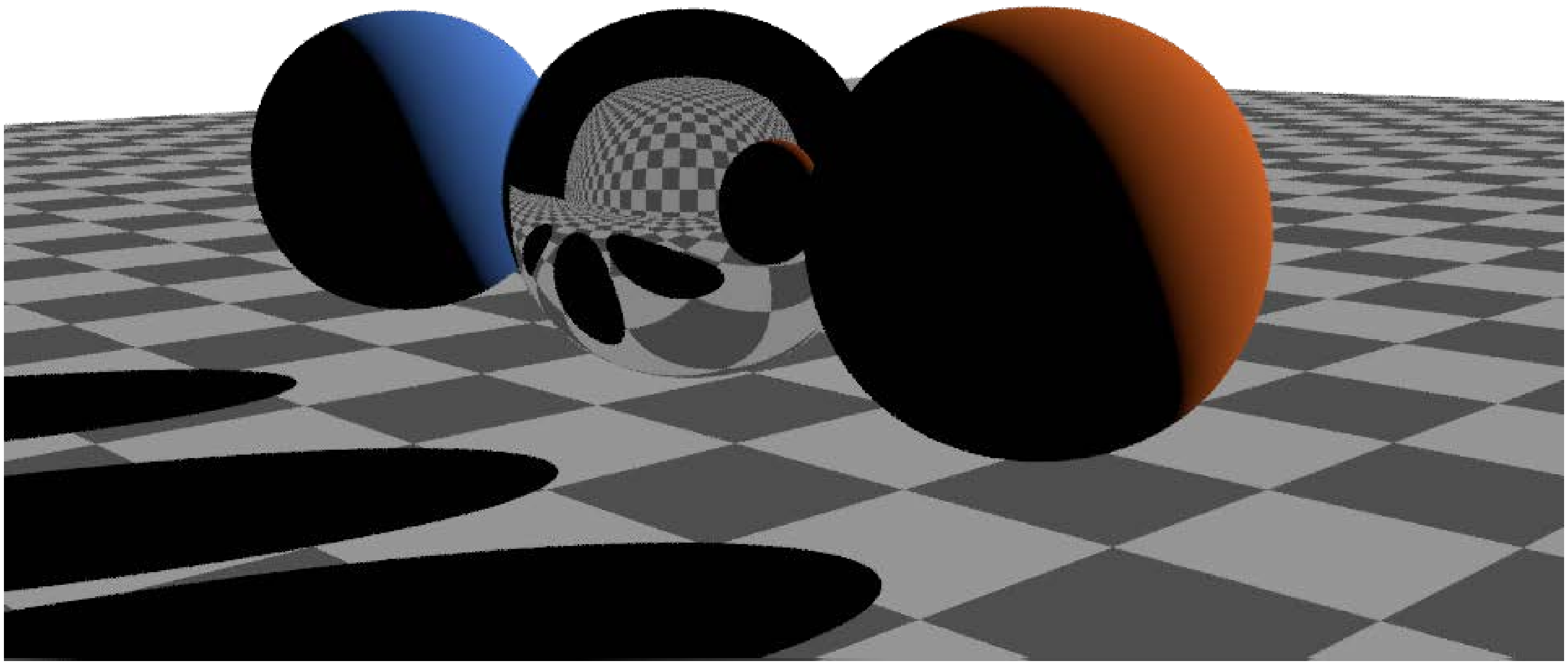
Sampling reflection directions

- For rough surfaces, we should consider a cone of reflection directions
  → reflection rays should be distributed around reflection direction



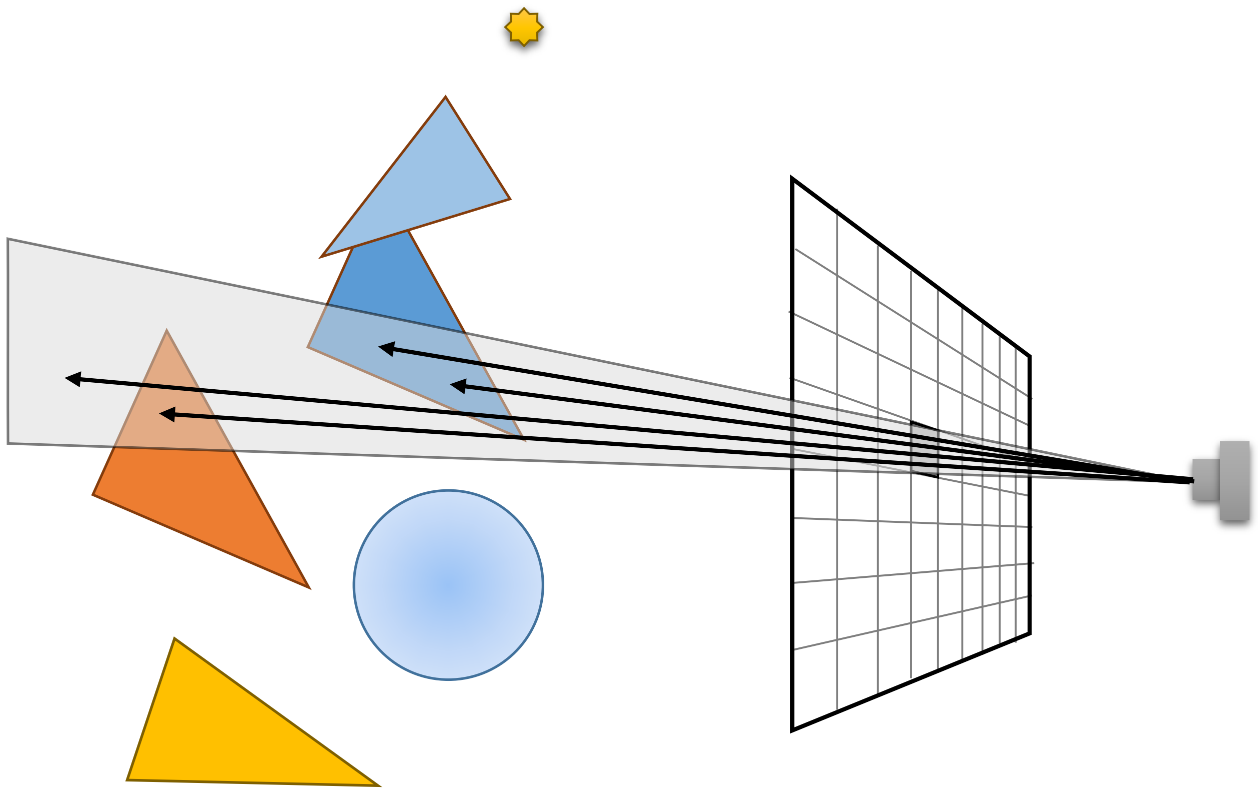
Sampling reflection directions

- Result



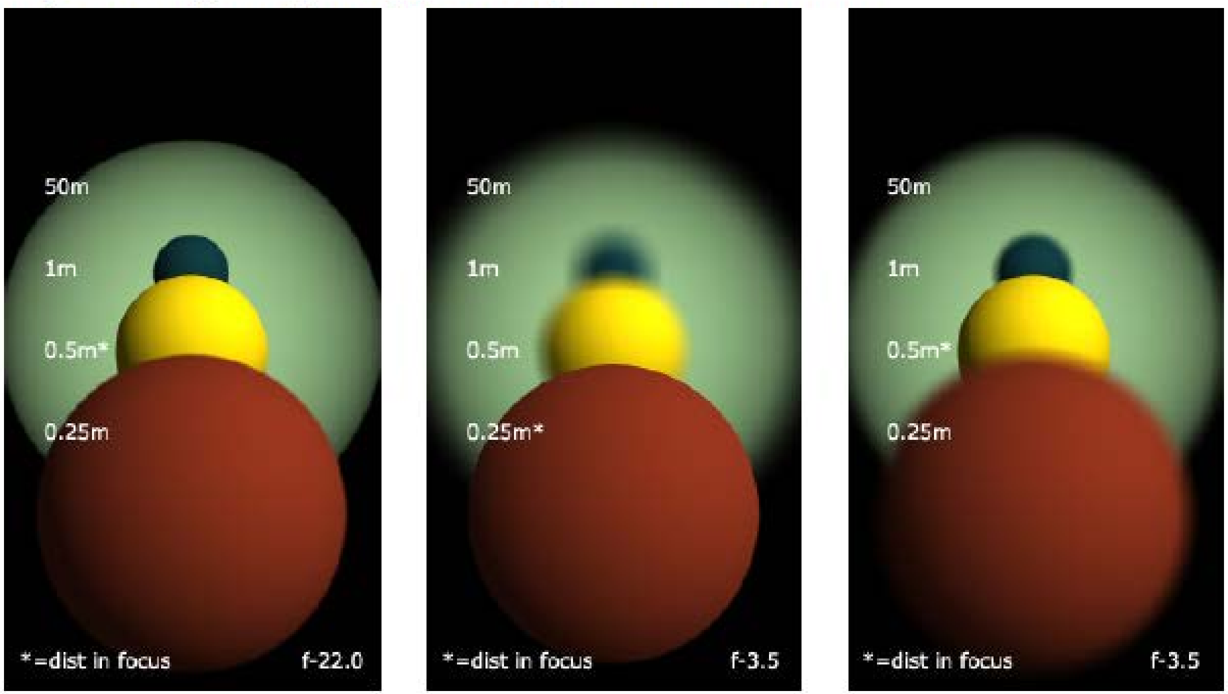
Sampling the pixel

- The color of a pixel is the average over the entire pixel
  → eye rays should be distributed over pixel
  → see Lectures **#3 "Rasterization" and #15 "Texture Aliasing"**



Sampling the Lens

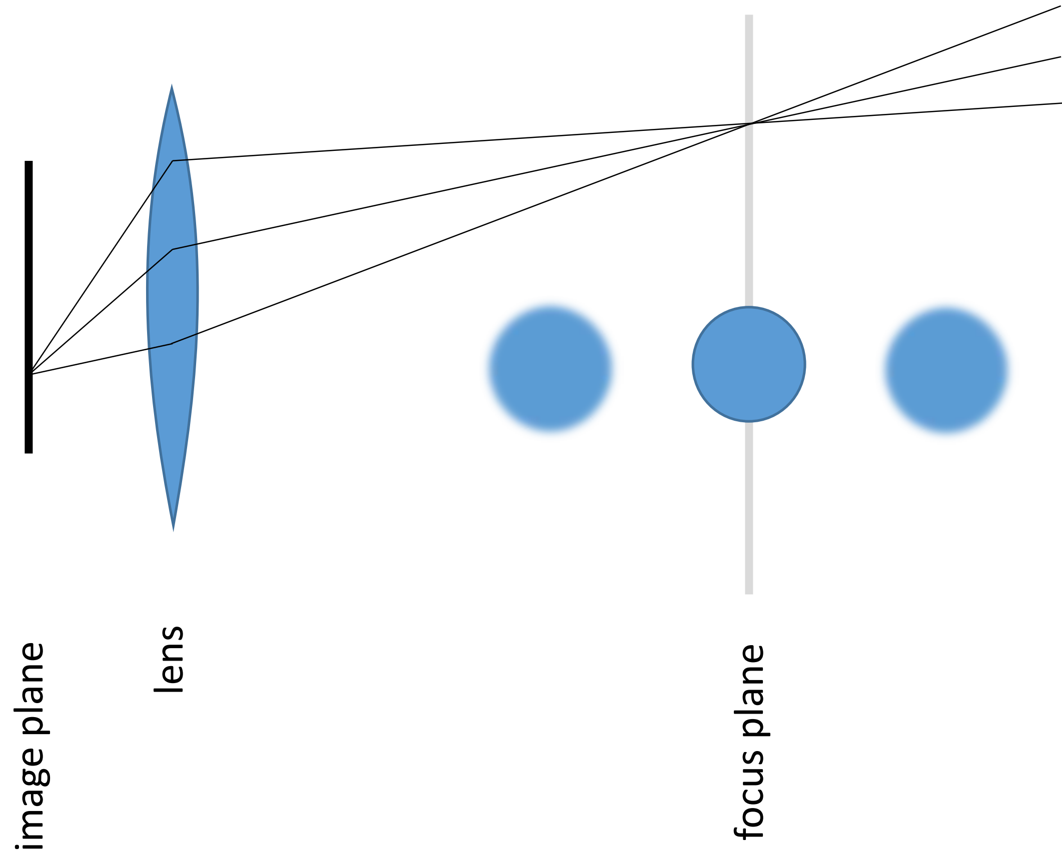
- Depth-of-Field
  - Pinhole model not realistic
  - Real lens has focus plane, only objects on this plane are sharp



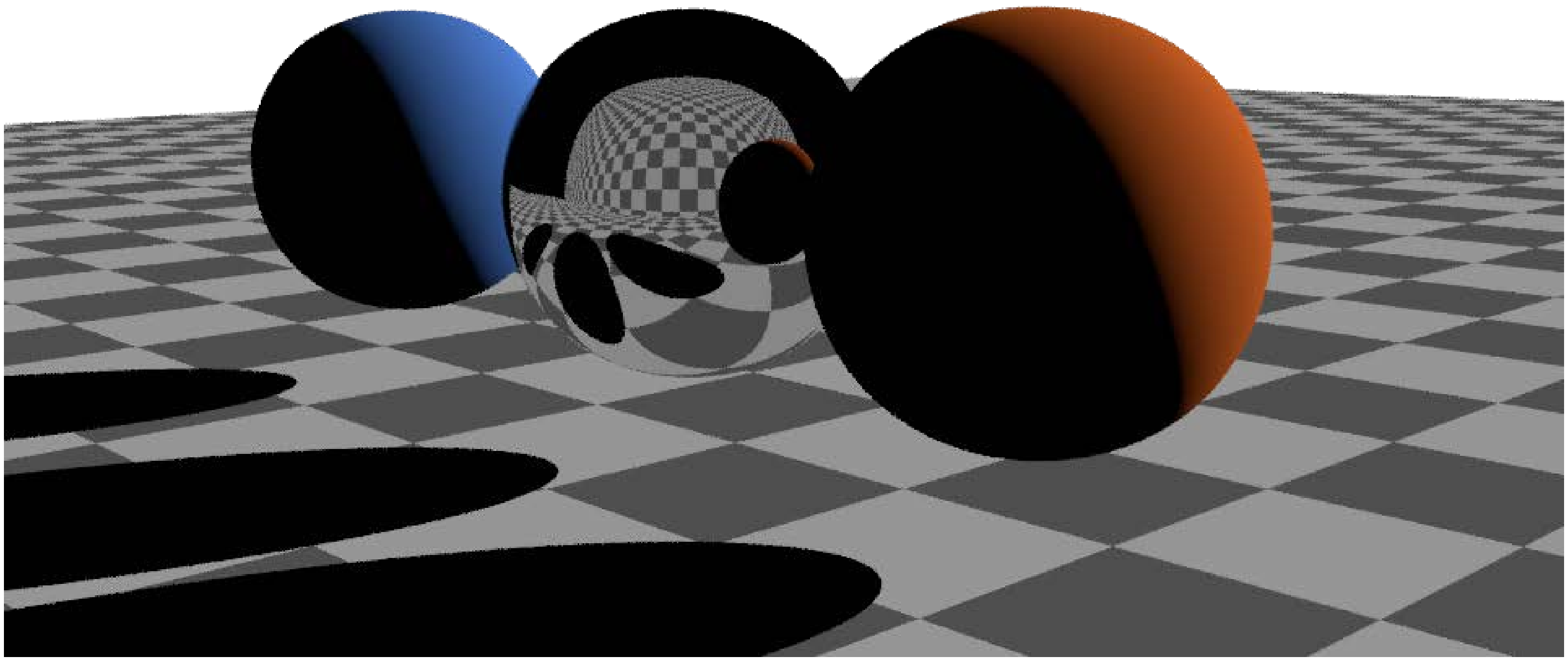
[Jason Waltman / jasonwaltman.com]

Distribution Ray Tracing

- Depth-of-Field
  - No pinhole, but lens
    (size varies with aperture)
  - Eye rays from a point on the image plane converge at focus plane
  - Objects on focus plane appear sharp, those before and behind get unsharp
- Can be simulated by casting multiple rays from lens to point on focus plane



Depth of Field



Distribution Raytracing

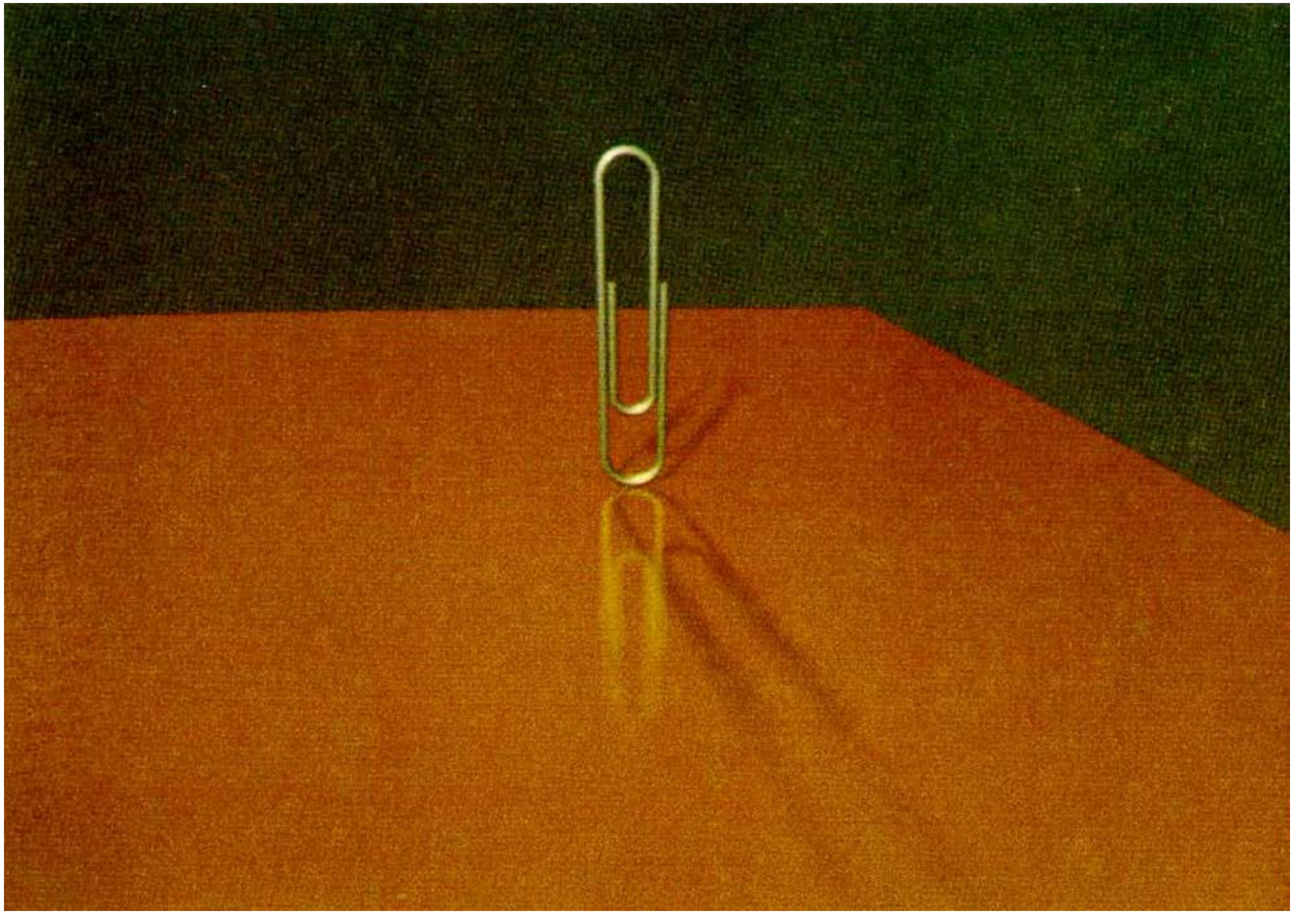
- Eye rays distributed over pixel and over lens
- Shadow rays distributed over light source
- Reflection rays distributed over reflection cone
- other effects: motion blur, frequency sampling, …

→ **Distribution Raytracing**

- Simple to integrate:
  - instead of one eye ray, we cast n_{AA} eye rays (n_{AA}: nr. of antialiasing-Samples)
  - for each eye ray, we compute n_{sh} shadow rays (n_{sh}: nr. of shadow samples)
  - for each eye ray, we cast n_{refl} reflection rays (n_{refl}: nr. of reflection-Samples)

Distribution Raytracing

- Original paper: Cook, Porter, Carpenter (Lucasfilm): "Distributed Ray Tracing", Siggraph 1984



rough glossy reflection

Distribution Raytracing

- Original pap[er] "Distributed Ray Tracing", Sig[graph]

Distributed Ray Tracing

Robert L. Cook
Thomas Porter
Loren Carpenter

Computer Division
Lucasfilm Ltd.

Abstract

Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light. Ray directions, however, have been determined precisely, and this has limited the capabilities of ray tracing. By distributing the directions of the rays according to the analytic function they sample, ray tracing can incorporate fuzzy phenomena. This provides correct and easy solutions to some previously unsolved or partially solved problems, including motion blur, depth of field, penumbras, translucency, and fuzzy reflections. Motion blur and depth of field calculations can be integrated with the visible surface calculations, avoiding the problems found in previous methods.

CR CATEGORIES AND SUBJECT DESCRIPTORS:
I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism;

ADDITIONAL KEY WORDS AND PHRASES: camera, constructive solid geometry, depth of field, focus, gloss, motion blur, penumbras, ray tracing, shadows, translucency, transparency

1. Introduction

Ray tracing algorithms are elegant, simple, and powerful. They can render shadows, reflections, and refracted light, phenomena that are difficult or impossible with other techniques[11]. But ray tracing is currently limited to sharp shadows, sharp reflections, and sharp refraction.

Ray traced images are sharp because ray directions are determined precisely from geometry. Fuzzy phenomenon would seem to require large numbers of additional samples per ray. By distributing the rays rather than adding more of them, however, fuzzy phenomena can be rendered with no additional rays beyond those required for spatially oversampled ray tracing. This approach provides correct and easy solutions to some previously unsolved problems.

This approach has not been possible before because of aliasing. Ray tracing is a form of point sampling and, as such, has been subject to aliasing artifacts. This aliasing is not inherent, however, and ray tracing can be filtered as effectively as any analytic method[4]. The filtering does incur the expense of additional rays, but it is not merely oversampling or adaptive oversampling, which in themselves cannot solve the aliasing problem. This antialiasing is based on an approach proposed by Rodney Stock. It is the subject of a forthcoming paper.

Antialiasing opens up new possibilities for ray tracing. Ray tracing need not be restricted to spatial sampling. If done with proper antialiasing, the rays can sample motion, the camera lens, and the entire shading function. This is called ***distributed ray tracing***.

Distributed ray tracing is a new approach to image synthesis. The key is that no extra rays are needed beyond those used for oversampling in space. For example, rather than taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time. Once we accept the expense of oversampling in space, distributing the rays offers substantial benefits at little additional cost.

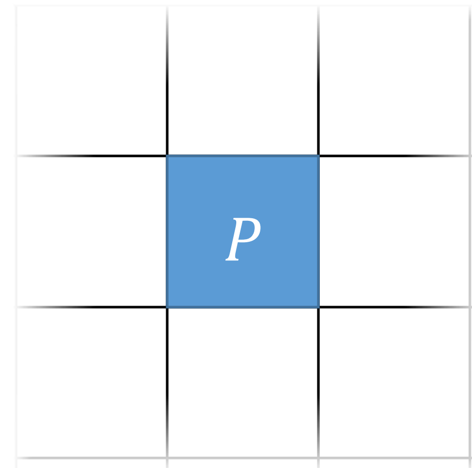
- Sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection).
- Sampling the transmitted ray produces translucency (blurred transparency).
- Sampling the solid angle of the light sources produces penumbras.

Distribution Raytracing = Numeric Integration

- Mathematically:

  color of pixel P is integral over pixel:

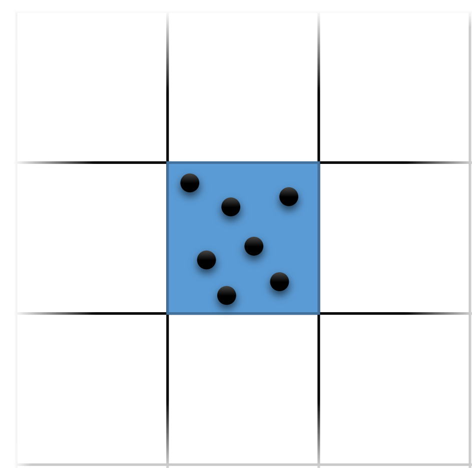
$$C_p = \frac{1}{|P|} \iint_P C(x, y) dx\, dy$$



- Numerically:

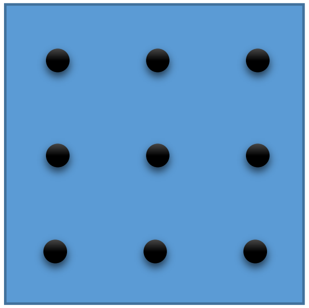
  approximate integral by averaging samples:

$$C_P \approx \frac{1}{n} \sum_{i=1 \ldots n} C(x_i, y_i)$$

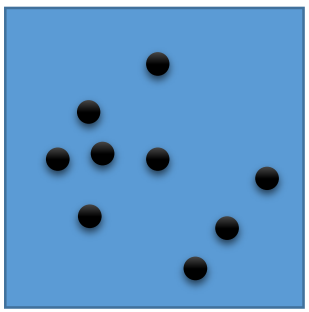


Low-Dimensional Numerical Integration

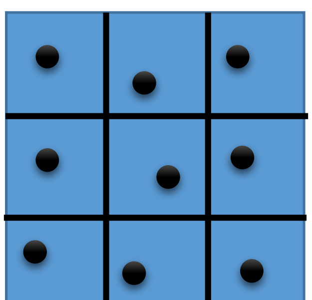
- Pixel example: how to distribute samples over pixel



uniform:
- very uniform distribution
- good numerical properties
- only square numbers as sample numbers



random:
- non-uniform distribution
- worse numerical properties
- arbitrary sample numbers, incrementable

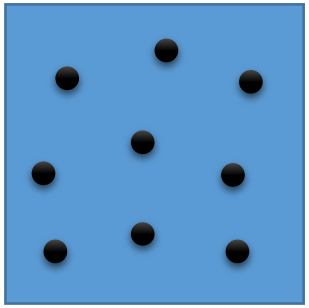


random stratified:
- uniform distribution
- good numerical properties
- only square numbers as sample numbers

Low-Dimensional Numerical Integration

- Poisson-Disk Sampling (aka dart throwing):

```
for desired number of samples
    do
        generate random sample
    until sample far enough from all previous ones
```



Poisson:
- very uniform distribution
- good numerical properties
- arbitrary sample numbers
- last samples costly: $O(n^2)$

High-Dimensional Numerical Integration

- Same is true for other effects
  → Pixel Color is high dimensional integral

$$C_p = \iint\limits_{pixel} \iint\limits_{lens} \iint\limits_{reflection} \iint\limits_{area\ light} \dots \ d \dots$$

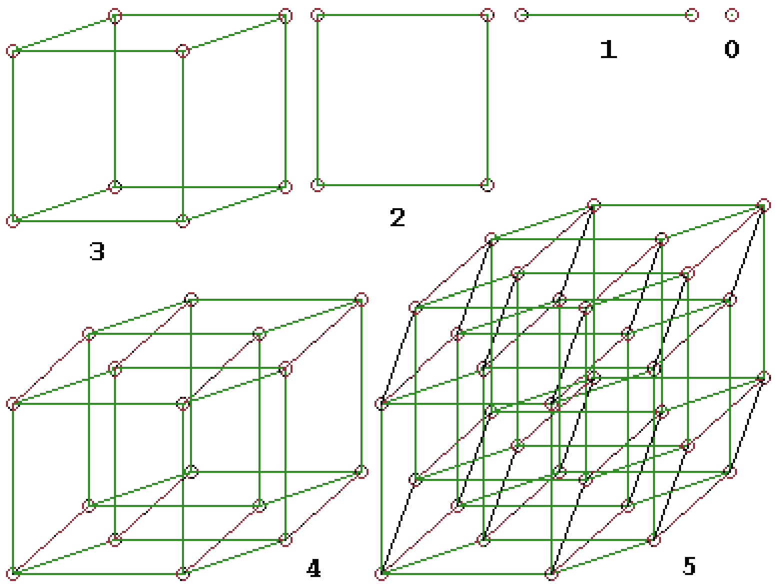
- **→ next lecture "The Rendering Equation"**

High-Dimensional Numerical Integration

- Let's say 10 dimensions
  →We have to sample a 10D cube

- **Uniform / Stratified**
  - n samples in one dimension
  - n^{10} samples overall
  - not practical
  - **→ curse of dimensionality**

- **Random**
  - works, but uneven distribution



hypercubes up to 5D
(wikipedia.de)

Random Sampling for Distribution Raytracing

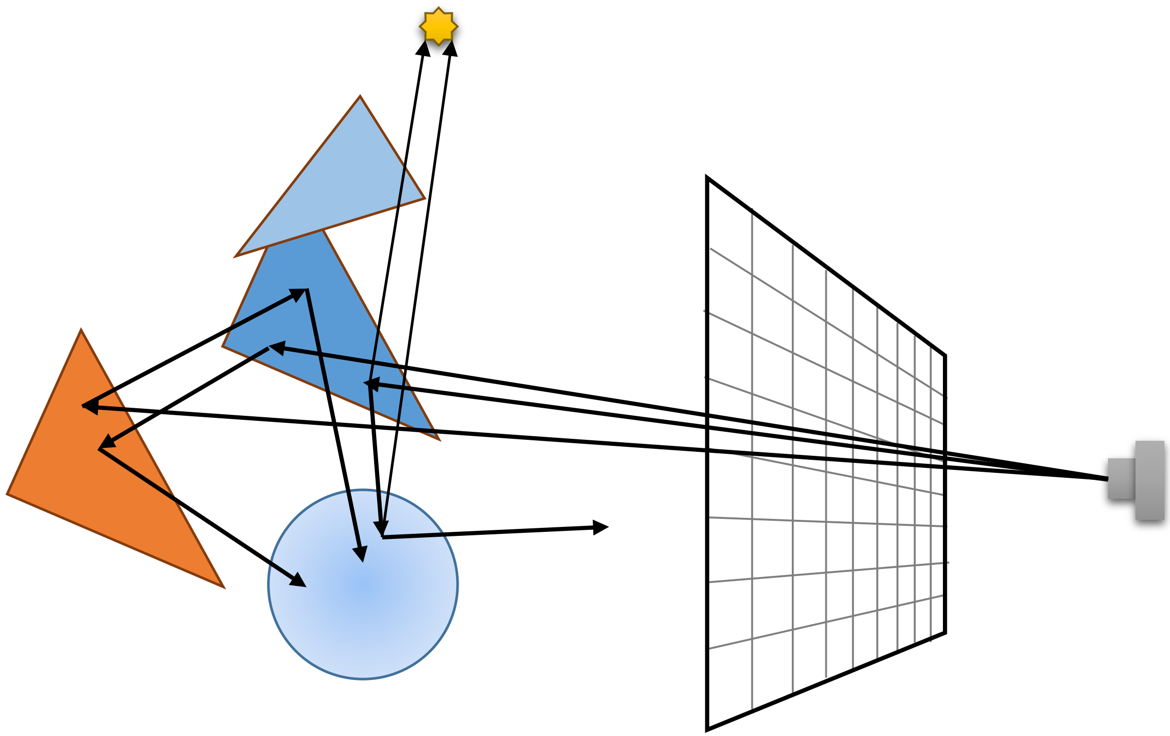
- Practical solution: random sampling → **“Monte-Carlo-Raytracing”**
- What is a random sample?

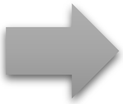
$$C_p = \iint_{pixel} \iint_{lens} \iint_{reflection} \iint_{area\ light} \dots \ d \dots$$

choose random pixel position	choose random lens position	choose random reflection direction	choose random position on area light

Random Sampling for Distribution Raytracing

- Each sample is a random path through the pixel



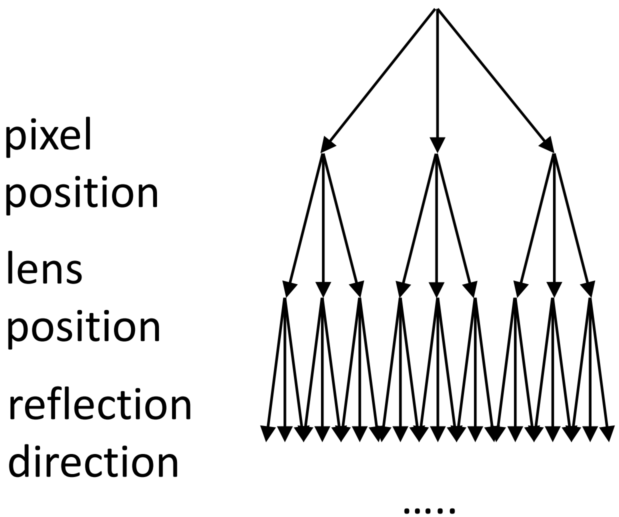


Path Tracing

Path Tracing

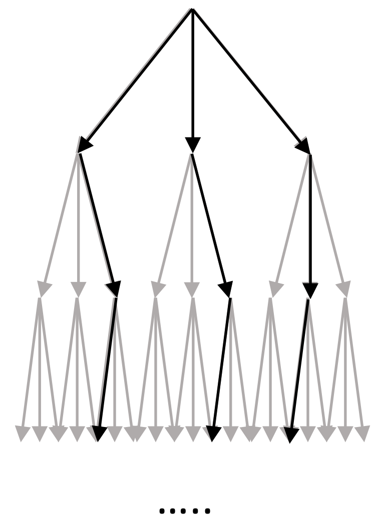
- Uniform Sampling

`computePixelColor()`



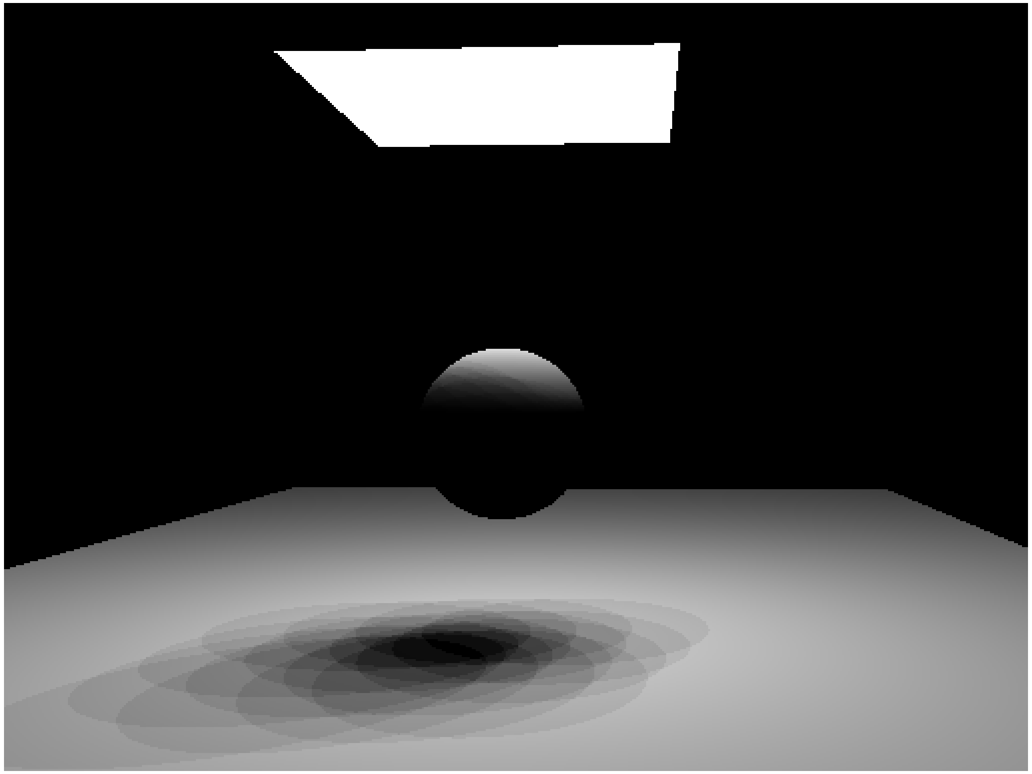
- Random Sampling = Path Tracing

`computePixelColor()`



Path Tracing

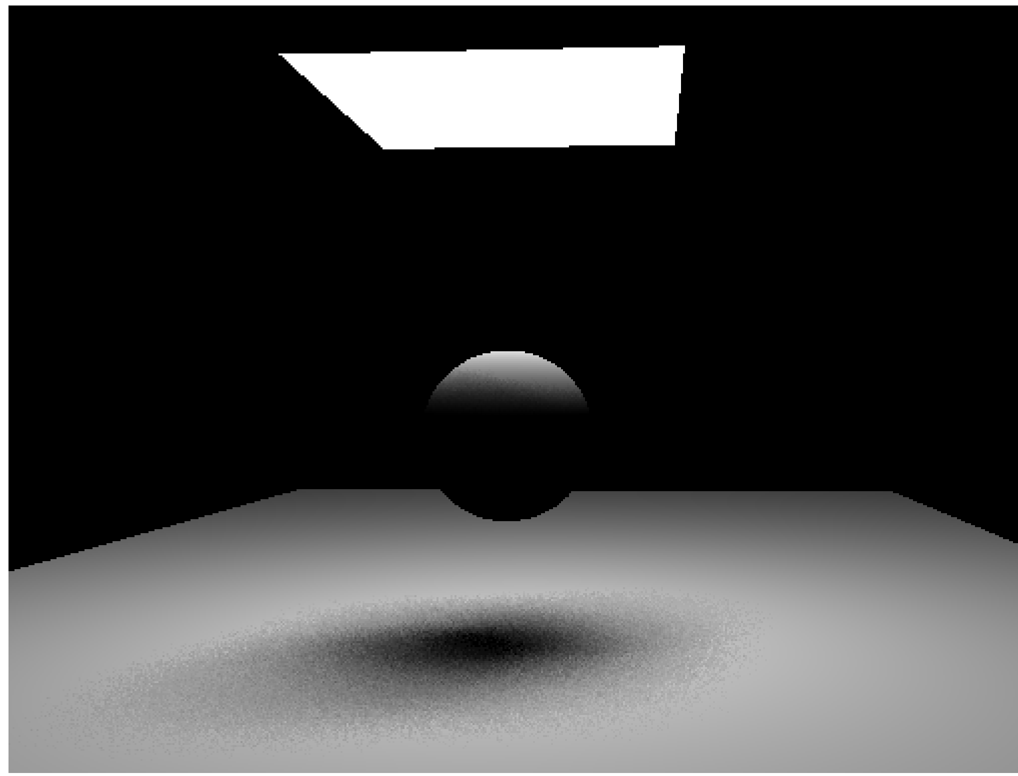
- Using the same samples for all pixels:
  Banding from multiple fixed point lights



Path Tracing

- Use varying set of 32 random samples → noise → visually more pleasing

- vary sample pattern from pixel to pixel !

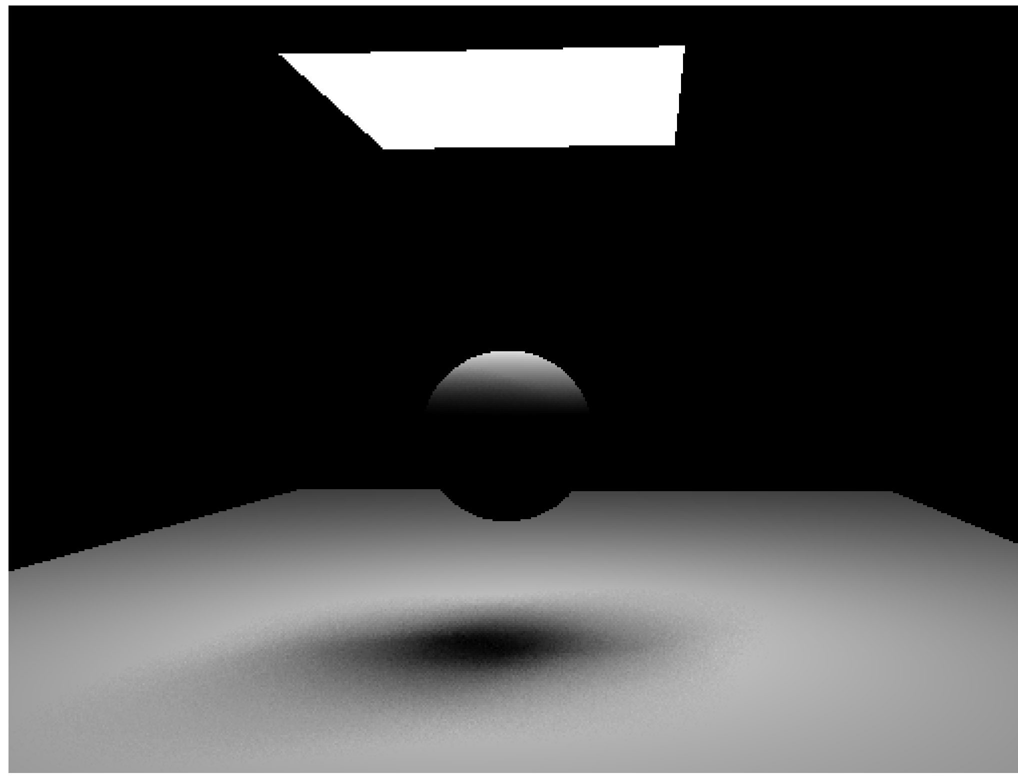


Path Tracing

- Noise from 128 random samples

- Rule: error of Monte-Carlo-Integration decreases by $\frac{1}{\sqrt{n}}$

  → halving the error requires four times as many samples !



Next Lecture

- How does this integral look like
  → **Rendering Equation**

$$C_p = \iint\limits_{pixel} \iint\limits_{reflection} \iint\limits_{area\,light} \dots \; d \dots \qquad ?$$