Distribution Ray-Tracing

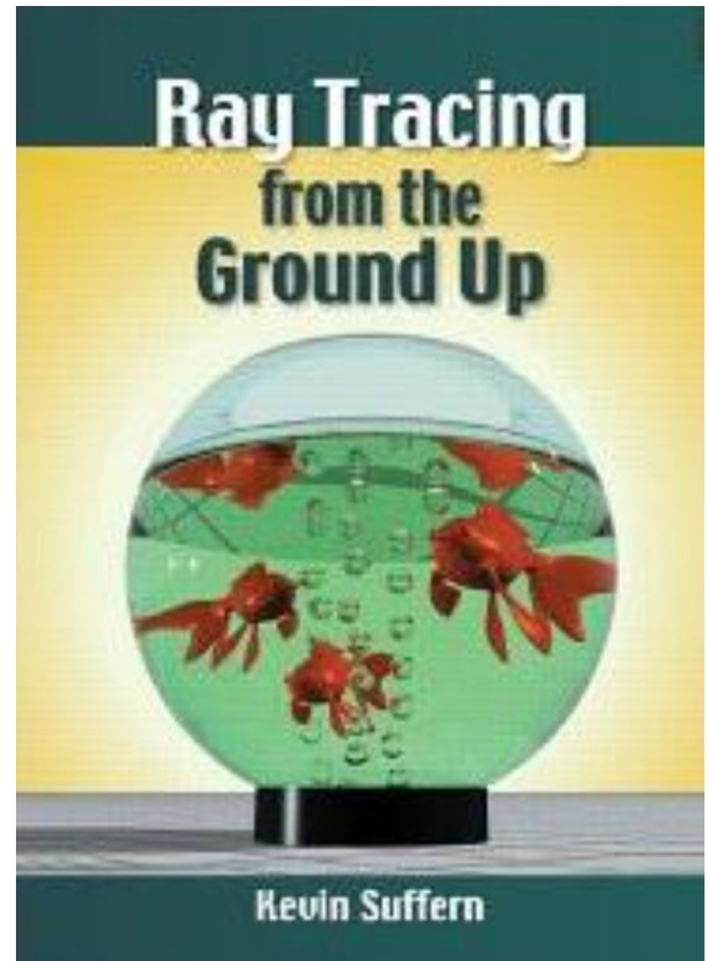
3D Programming Course

João Madeiras Pereira

Bibliography

K. Suffern; “Ray Tracing from the Ground Up”,
http://www.raytracegroundup.com

- Chapter 4, 5 for Anti-Aliasing
- Chapter 6 – for Disc Sampling
- Chapter 10 – for Depth of Field

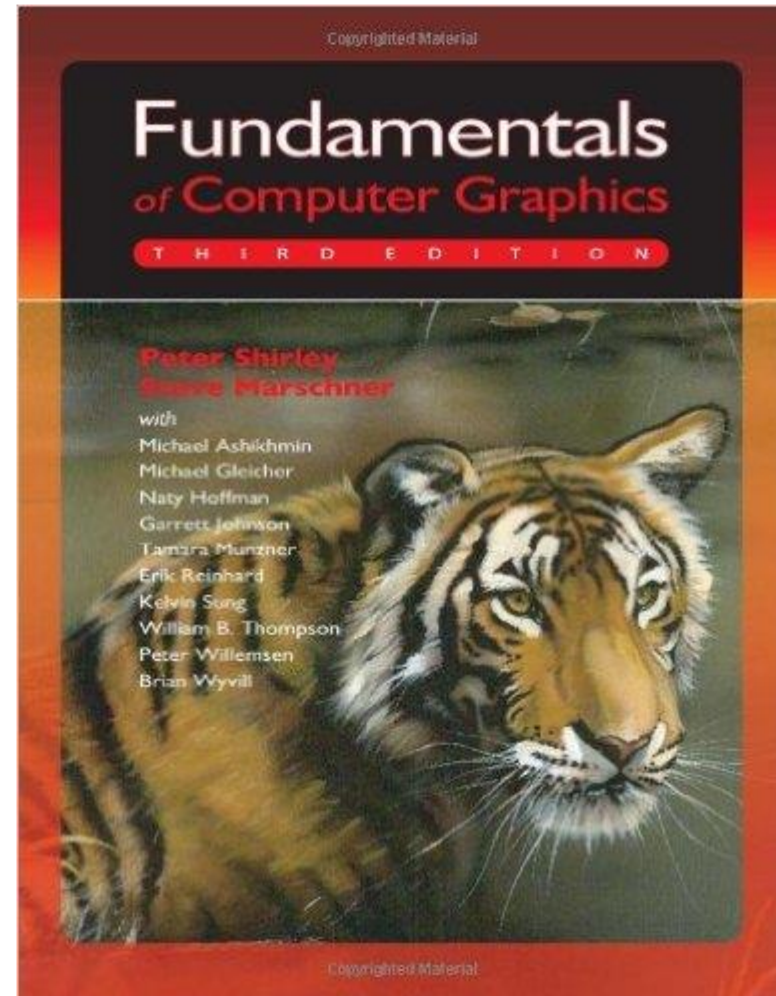


Bibliography

Peter Shirley, Michael Ashikhmin: “Fundamentals of Computer Graphics”

Chapter 10 – Ray-Tracing

- Antialiasing: section 10.11.1
- Soft shadows: section 10.11.2



Problem with Simple Ray Tracing: Aliasing

raytraced images are too "clean"

- soft shadows come from area light
  - raytracing only supports point ligts

[Jason Waltman / jasonwaltman.com]

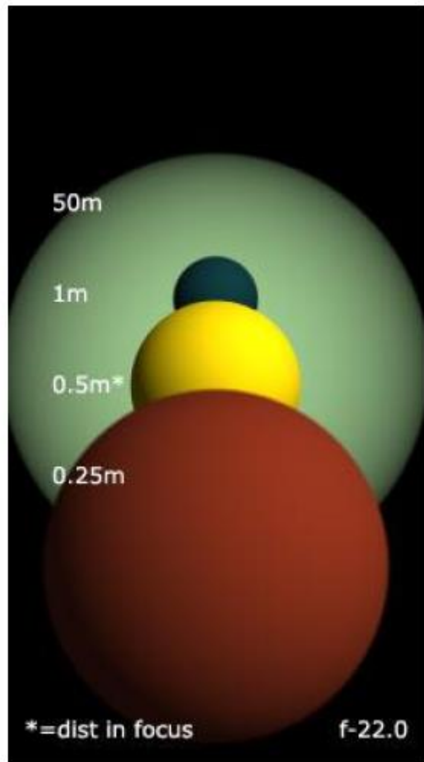
Raytraced images are “too clean”

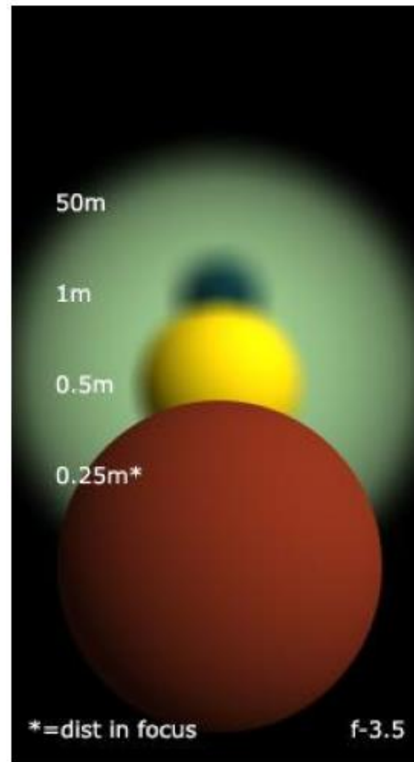
- blurry reflections come from rough materials
  - raytracing only supports perfectly sharp mirror

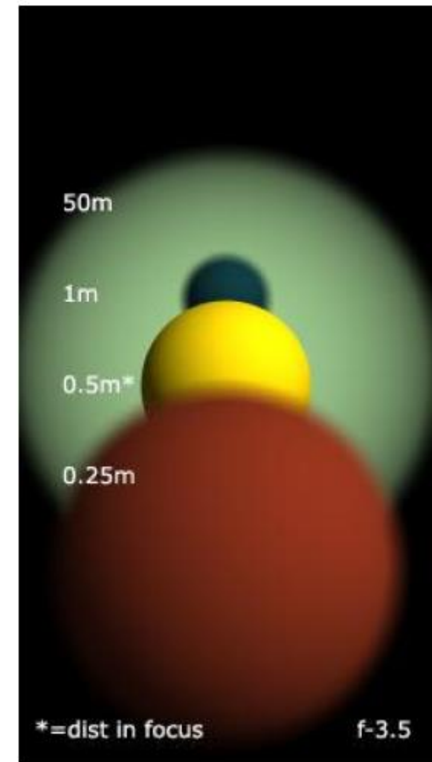
[Jensen]

Raytraced images are "too clean"

- depth of field come from lens system
  - raytracing only support pinhole camera







[Jason Waltman / jasonwaltman.com]

Raytraced images are “too clean”

- motion blur come from shutter speed
  - raytracing only support infinitely fast shutter speed

[Jason Waltman / jasonwaltman.com]

Whitted Raytraced images

Distribution Raytraced images - antialiasing and soft shadows

Distribution Ray –Tracing (DRT)

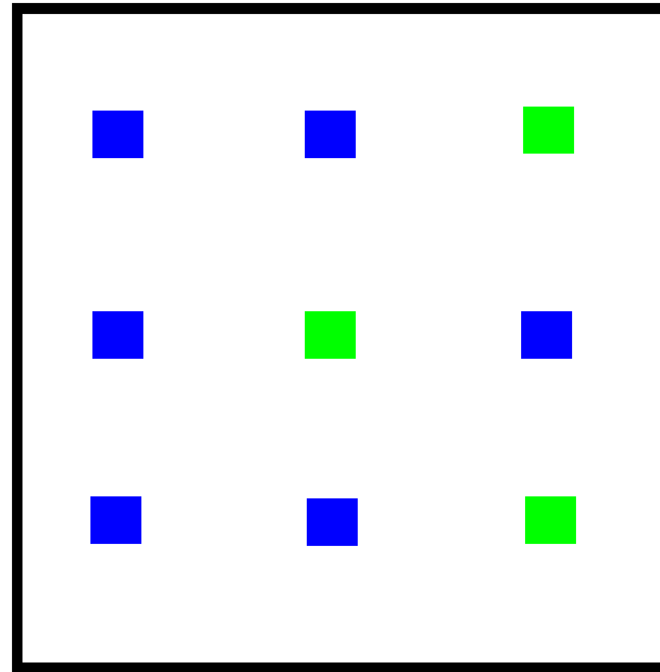
use many rays to compute average values over pixel areas, time, area lights, reflected directions, ...

Distribution RT

- Distributed Ray Tracing, aka Distribution Ray Tracing or Stochastic Ray Tracing, is a refinement of ray tracing that allows for the rendering of "soft" phenomena
- Averaging multiple rays distributed over an interval
- Soft shadows can be rendered by distributing shadow rays over the light source area.
- Spatial anti-aliasing can be rendered by distributing rays over a pixel
- Distribute rays across the eye to simulate depth of field effect
- Blurry reflections and transmissions can be rendered by distributing reflection and transmission rays over a solid angle about the mirror reflection or transmission direction.
- Distribute rays in time to get temporal antialiasing (motion blur)

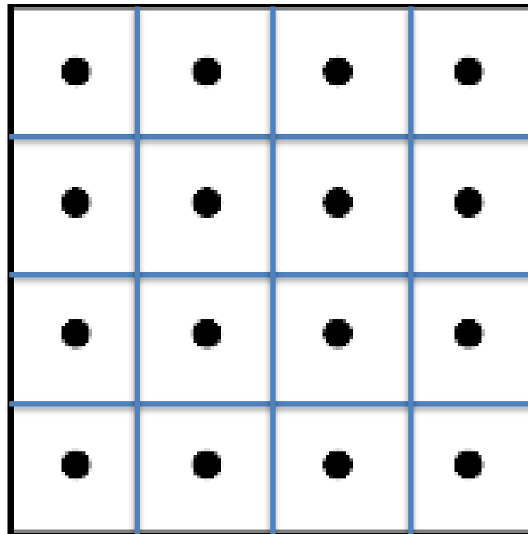
Antialiasing with Supersampling

- attempts to reduce the errors by shooting more than one primary ray into each pixel and averaging the results to determine the pixel's apparent color.



Regular Sampling

- Fire more than one ray for each pixel (e.g., a 4x4 grid of rays
- Average the results (perhaps using a filter)



Antialiasing with Regular Sampling

[Shirley]

- Replace the code

for each pixel (i, j) **do**
  $c_{ij} = \text{ray-color}(i + 0.5, j + 0.5)$

$$\mathbf{d} = -d_f \hat{\mathbf{z}}_e + w\left(\frac{i+0.5}{\mathrm{Re}\,sX} - \frac{1}{2}\right)\hat{\mathbf{x}}_e + h\left(\frac{j+0.5}{\mathrm{Re}\,sY} - \frac{1}{2}\right)\hat{\mathbf{y}}_e$$

- With code that samples on a regular n x n grid:

for each pixel (i, j) **do**
  $c = 0$
  for $p = 0$ to $n - 1$ **do**
    for $q = 0$ to $n - 1$ **do**
      $c = c + \text{ray-color}(i + (p + 0.5)/n, j + (q + 0.5)/n)$
  $c_{ij} = c/n^2$

Regular Sampling issues

- Leads to artefacts like moiré patterns:

- Regular sampling takes 16 times longer to render
- Solutions:
  - Adaptive supersampling

Adaptive supersampling

- If the color of a pixel differs from its neighbors (to the left or below) by at least the set threshold value then the pixel is super-sampled by shooting a given, fixed number of additional rays. A good threshold value is 0.3
- If r1, g1, b1 and r2, g2, b2 are the rgb components of two pixels then the difference between pixels is computed by:

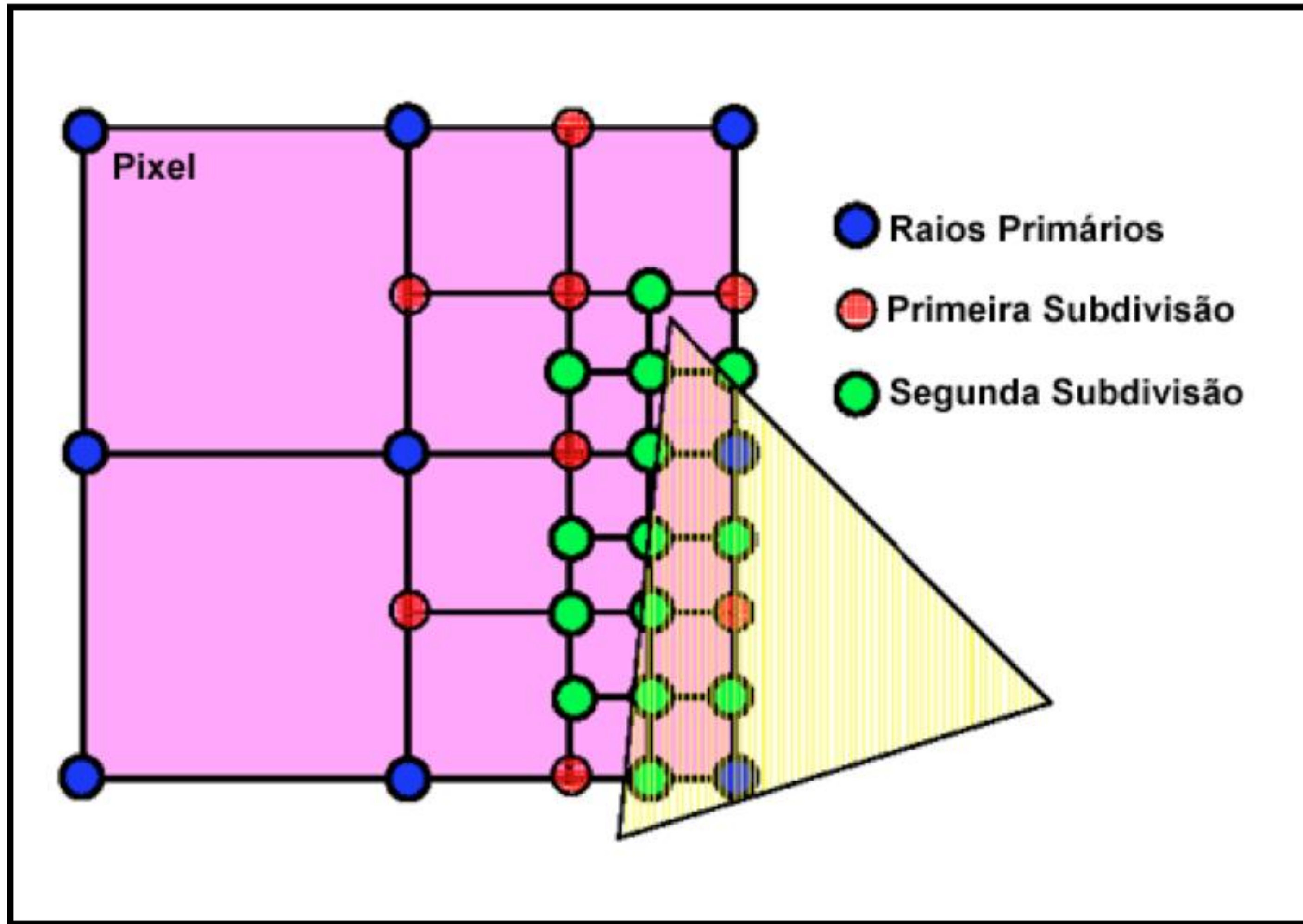
  diff = abs(r1-r2) + abs(g1-g2) + abs(b1-b2)

- If the anti-aliasing threshold is 0.0 then every pixel is super-sampled. If the threshold is 3.0 then no anti-aliasing is done

Adaptive Supersampling - Monte-Carlo Sampling

- It’s a recursive technique
- It starts by tracing four rays at the corners of each pixel.
- If the colors are similar (check the threshold) then just use their average
- Otherwise recursively subdivide each cell of the grid into four sub-pixels: fire additional 5 rays – at the center and at mid of the 4 edges
- Sub-pixels are separately traced and tested for further subdivision
- Keep going until each 2x2 grid is close to uniform or limit is reached
- Filter the result
- The advantage of this method is the reduced number of rays that have to be traced.
- Samples that are common among adjacent pixels and sub-pixels are stored and reused to avoid re-tracing of rays.

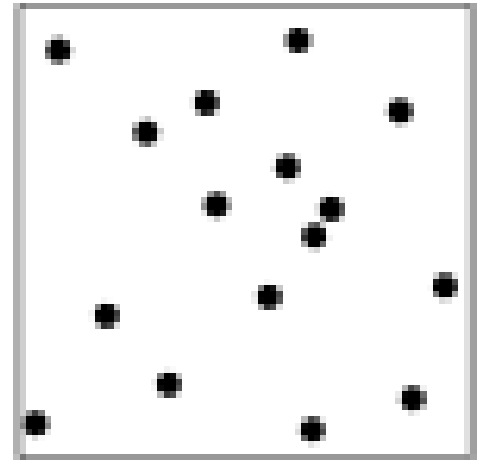
Adaptive Supersampling - Monte-Carlo Sampling



First iteration – 512 x 512 viewport implies 513x513 primary rays

Antialiasing with Stochastic (Random) Sampling [Shirley]

- Adaptive Supersampling still divides pixels into regular patterns of rays, and suffers from aliasing that can occur from regular pixel subdivision – Moiré patterns
- It sends a fixed number of rays into a pixel, but makes sure they are randomly distributed (but more or less evenly cover the area)



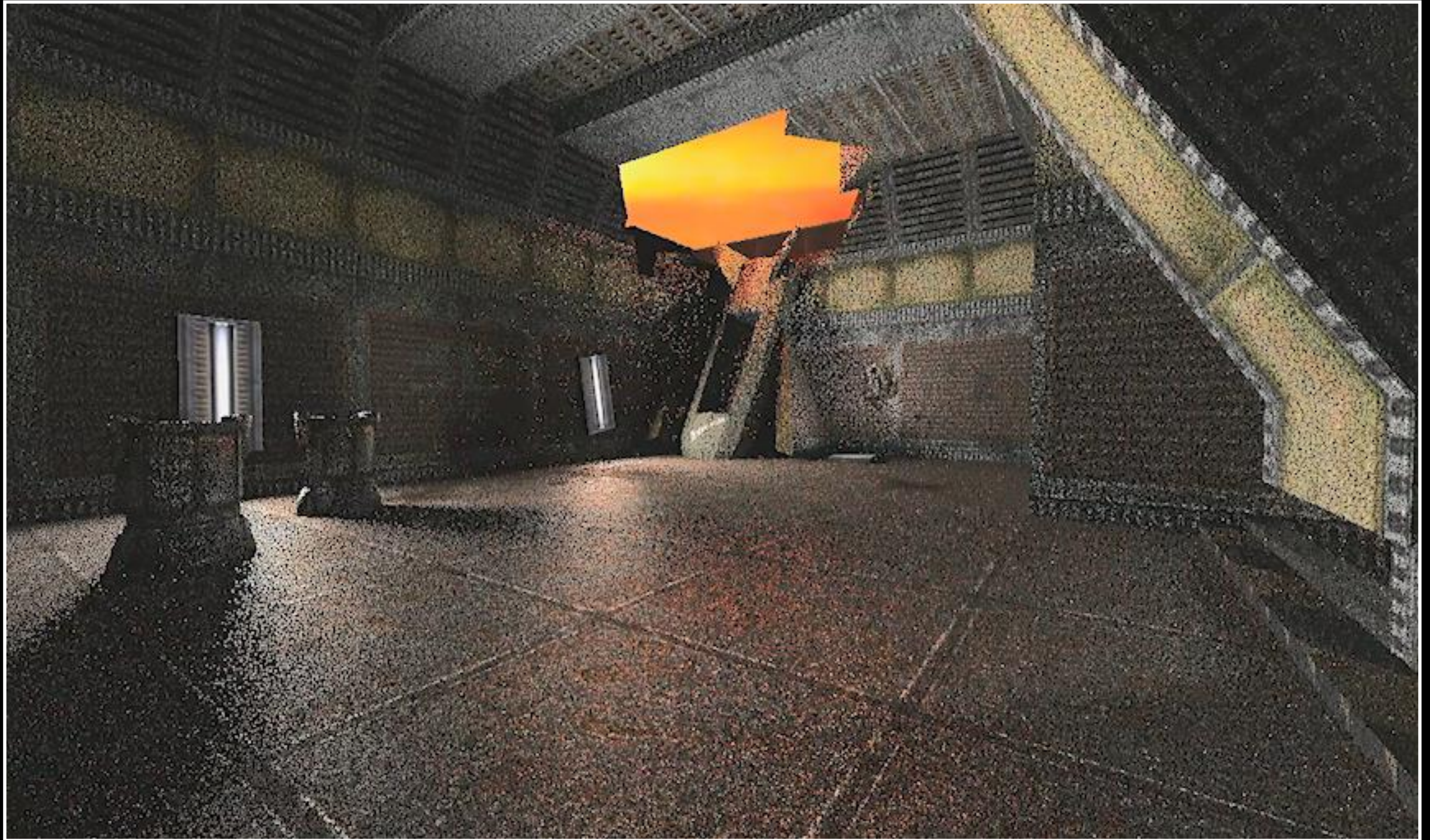
igure 10.28. Sixteen ran
om samples for a singl
xel.

Stochastic (Random) Sampling [Shirley]

- Code:

```
for each pixel (i, j) do
    c = 0
    for p = 1 to n^2 do
        c = c + ray-color(i + ξ, j + ξ)
    c_ij = c/n^2
```

•ξ is a call that returns a uniform random number in the range [0, 1] – see rand_float() and set_rand_seed(seed) in maths.h
•One interesting side effect of the stochastic sampling patterns is that they actually injects noise into the solution (slightly grainier images). This noise tends to be less objectionable than aliasing artifacts.



Stochastic (Random) Sampling issue: Noise

Mitigating noise

Jittering [Shirley]

- With the same number of samples, we can reduce the noise by improving the samples spatial distribution.
- Solution: Hybrid strategy that randomly perturbs a regular grid – **Jittering** or **Stratified Sampling**

Jittering [Shirley]

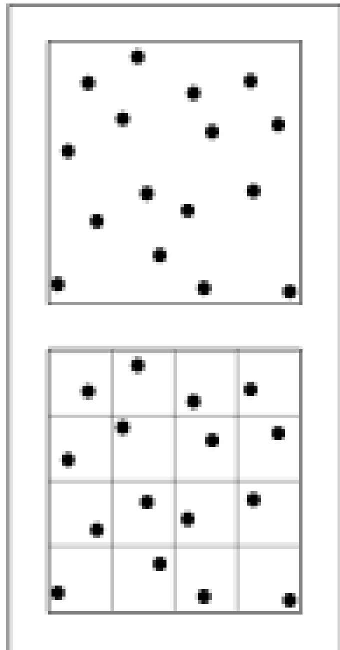
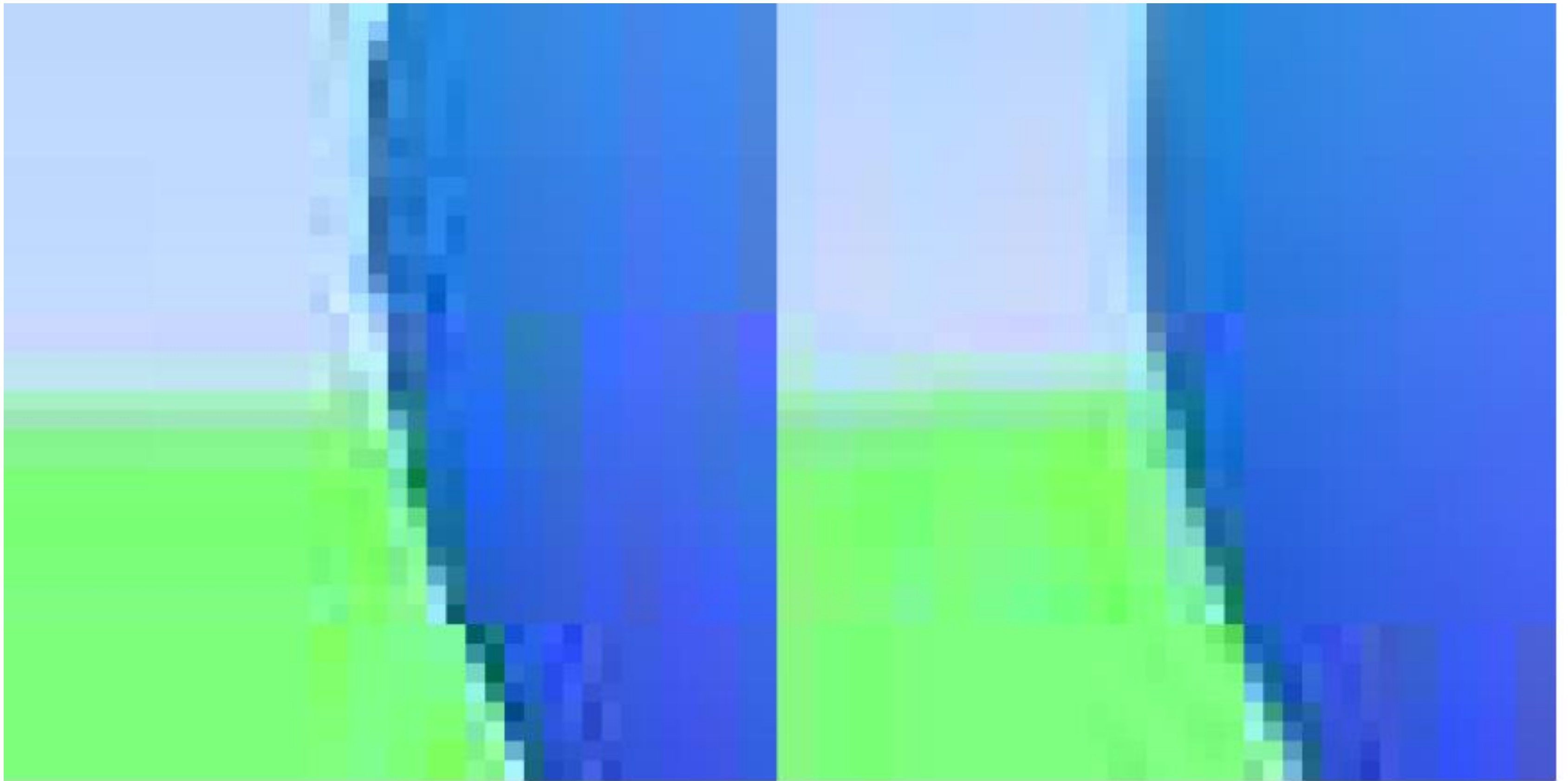


Figure 10.29. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

for each pixel (i, j) **do**
  $c = 0$
  for $p = 0$ **to** $n - 1$ **do**
    for $q = 0$ **to** $n - 1$ **do**
      $c = c + \text{ray-color}(i + (p + \xi)/n, j + (q + \xi)/n)$
  $c_{ij} = c/n^2$

Antialiasing



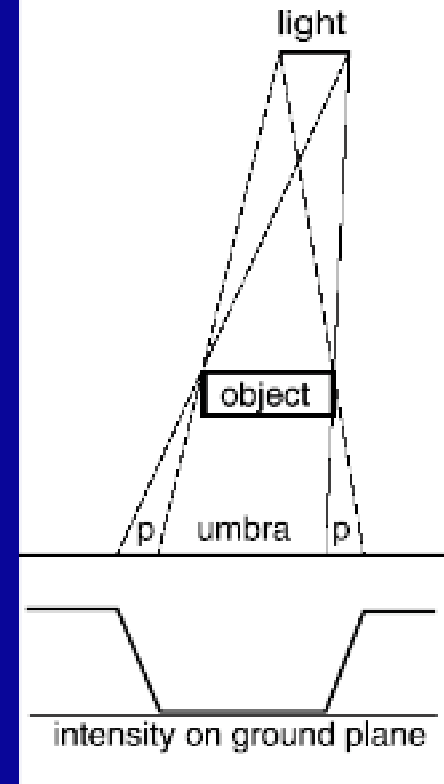
Soft Shadows: distributing over light source areas

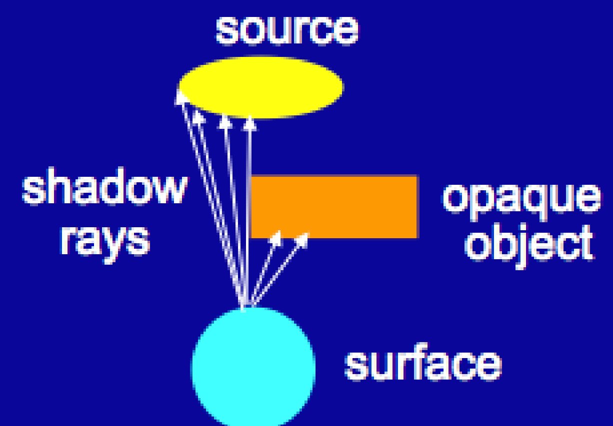
Soft Shadows

Point light sources produce sharp shadow edges
- the point is either shadowed or not
- only one ray is required

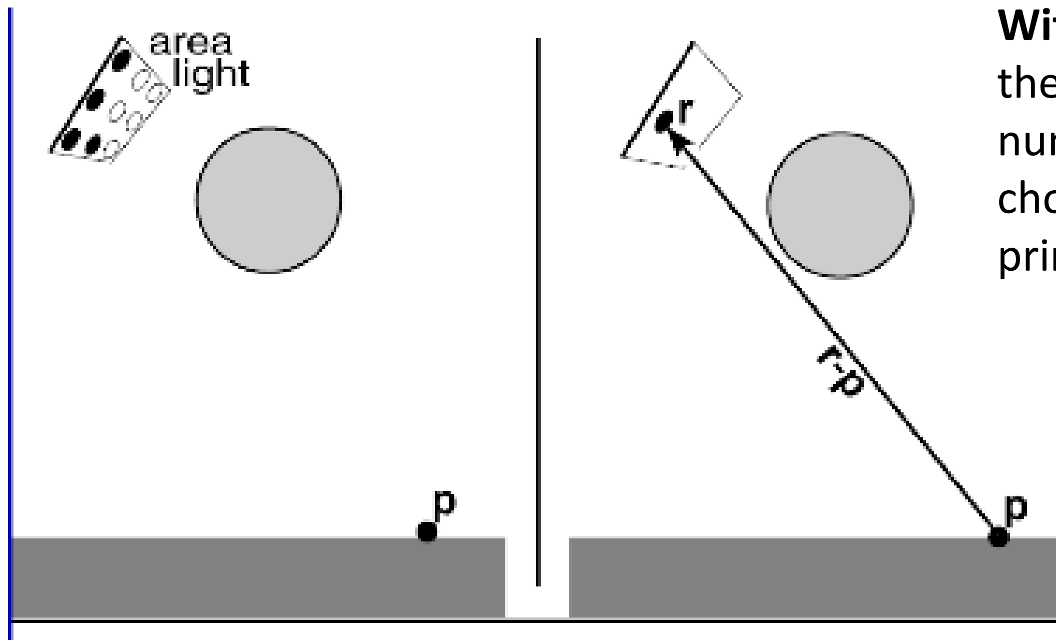
With an extended light source the surface point may be partially visible to it
- only part of the light from the sources reaches the point
- the shadow edges are softer
- the transition region is the *penumbra*

Accomplish this by
- firing shadow rays to multiple points on the light source
- weighting them by the brightness
- the resulting shading depends on the fraction of the obstructed shadow rays





Soft Shadows [Shirley]



With antialiasing: represent the area light as an infinite number of point lights and choose one at random for each primary ray

Without antialiasing: represent the area light as a distributed set of N point lights, each with one Nth of the intensity of the base light

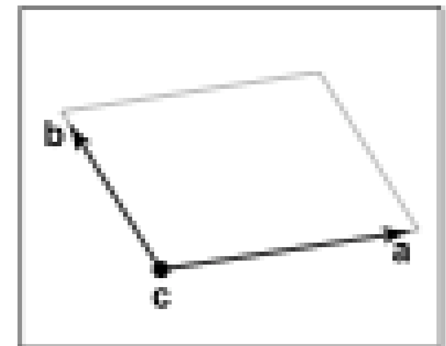
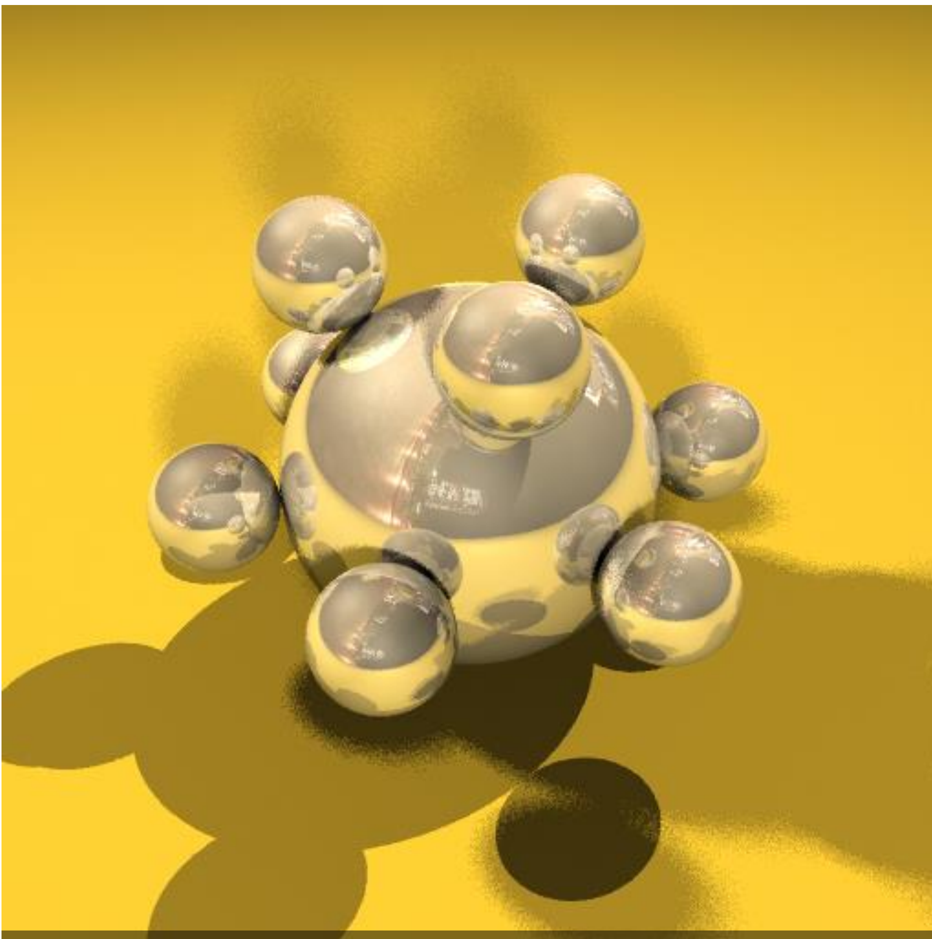


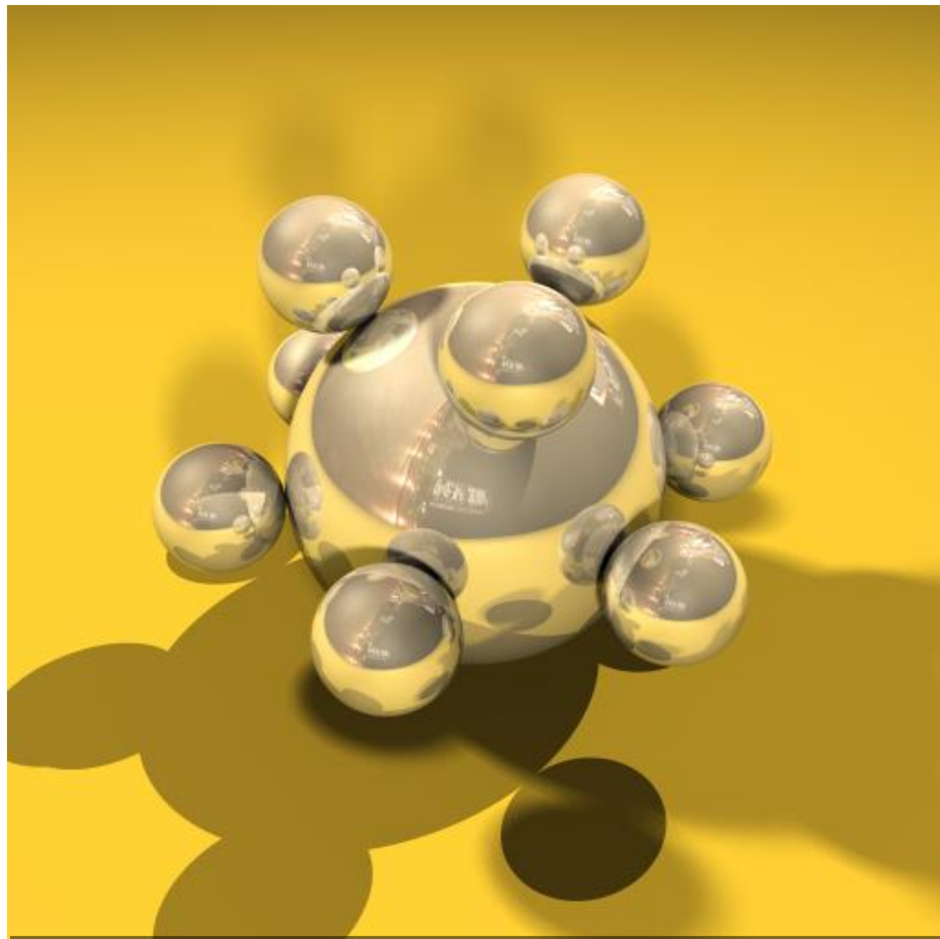
Figure 10.32. The geometry of a parallelogram light specified by a corner point and two edge vectors.

$$\mathbf{r} = \mathbf{c} + \xi_1 \mathbf{a} + \xi_2 \mathbf{b},$$

where ξ_1 and ξ_2 are uniform random numbers in the range $[0, 1)$.



4 samples per pixel (spp)

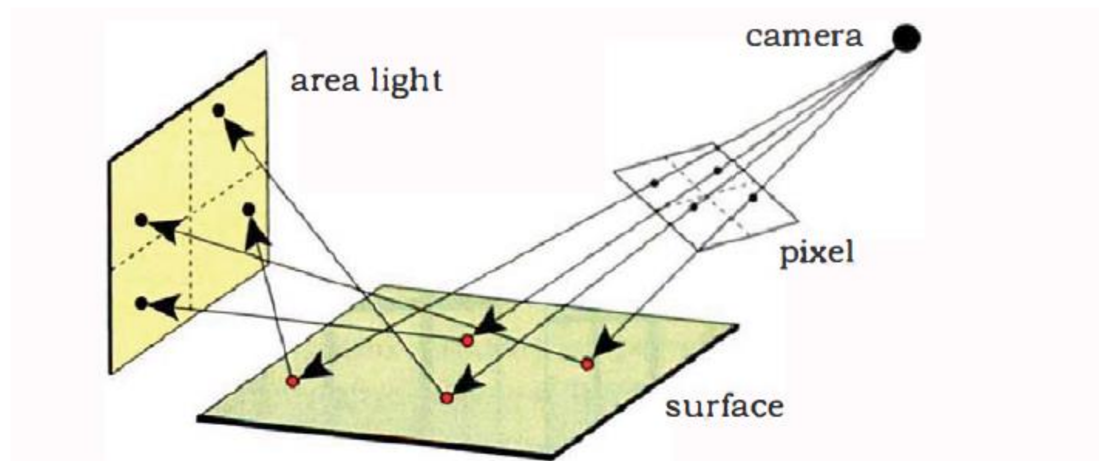


64 spp

Soft Shadows: Assignment 1

Pixel and Light sampling issues

Instead of generating a pure random shadow ray for each primary ray through the pixel sample, we can reduce the noise in the shadow area, by improving the samples spatial distribution, so applying jittering is also a good strategy.



re 5.2. Shading a surface with an area light and four samples per pixel.

Pixel and Light sampling issues

- Jittering samples in the light must be done carefully
  - Number of pixel samples and light samples should be the same
  - We would not want to always have the ray in the upper left-hand corner of the pixel generate a shadow ray to the upper left-hand corner of the light
- Shuffle the samples of light array in order to avoid a correlation with the pixel array

Depth of Field

- Distributing rays over a finite aperture gives:

Virtual and Real Camera [Suffern]

- the virtual camera (a) models the pinhole camera; the eye corresponds to the pinhole of (b)
- In(a) the view plane is between the eye and the objects
- the pinhole in (b) is in the between the objects and the film plane
- The “lens” is infinitely small
- Real cameras have finite-aperture lens with focal distance

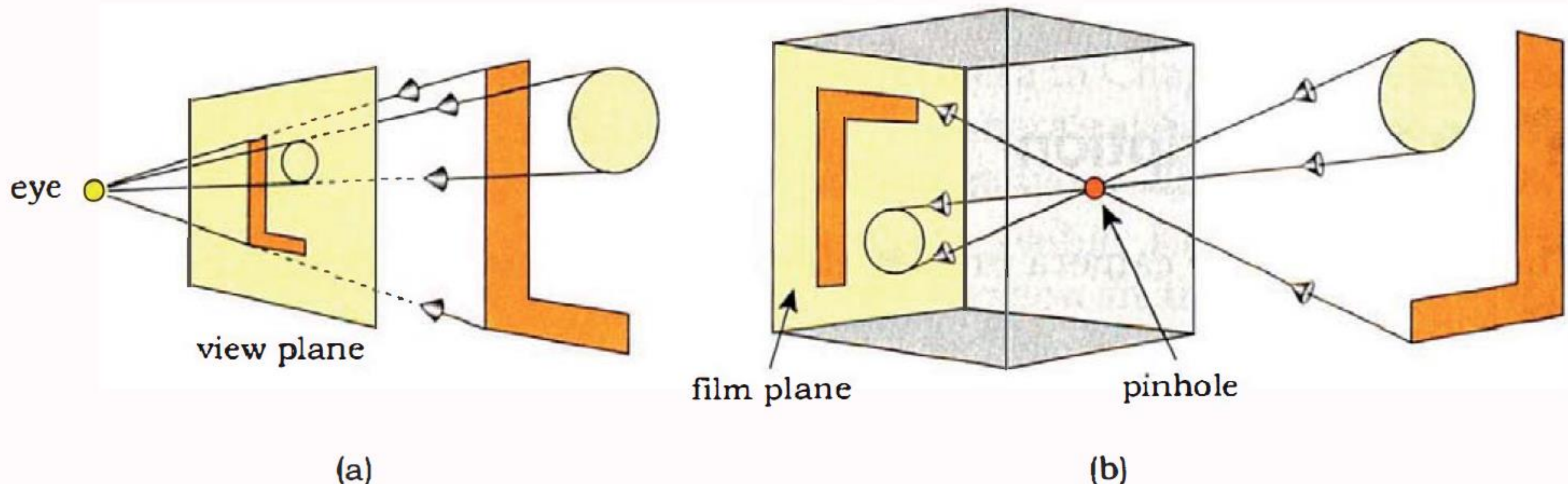
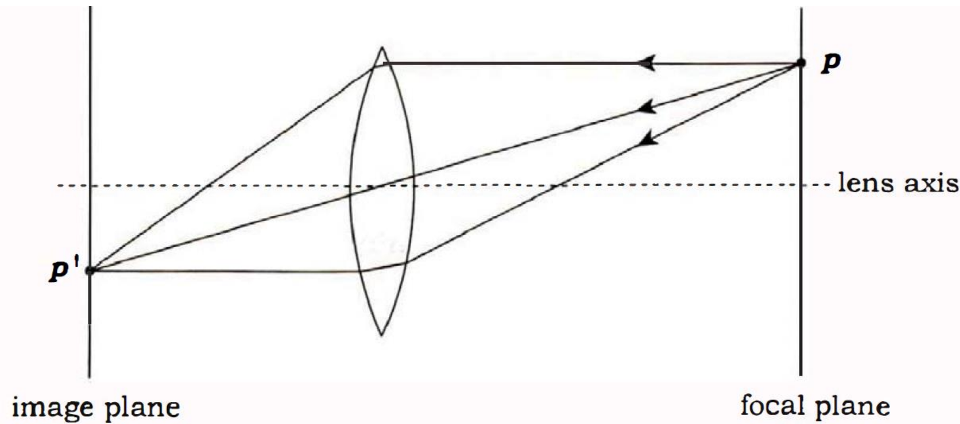
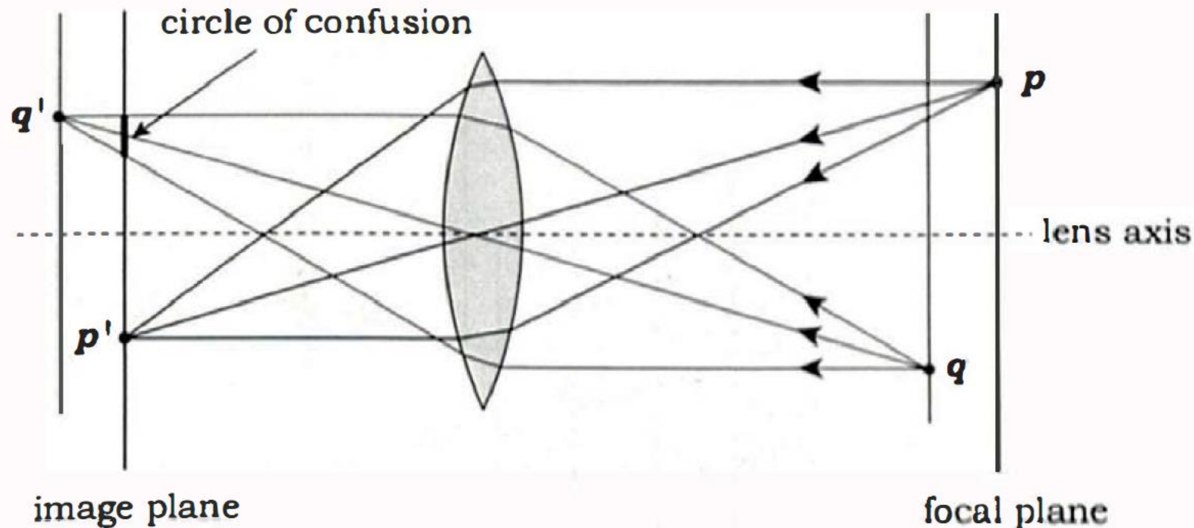


Figure 9.2. (a) Computer perspective viewing; (b) real pinhole camera, where the image on the film plane is inverted.

Depth of Field – Thin Lens [Suffern]





- Depth of field (DOF) is the range of distances parallel to the lens axis in which the scene is in focus
- In RT, the image can appear in focus over the range of distances where the circle of confusion is smaller than a pixel

Thin-Lens Simulation [Suffern]

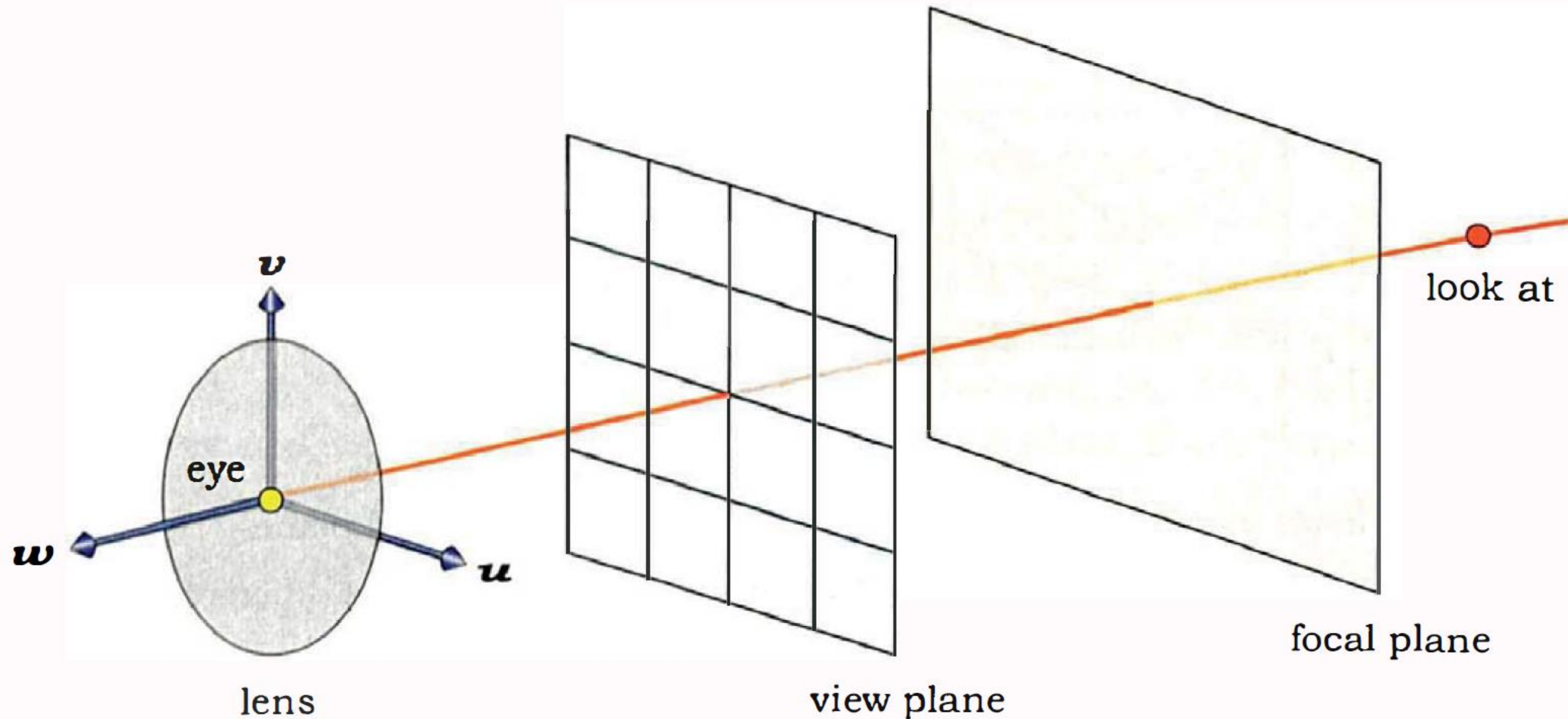
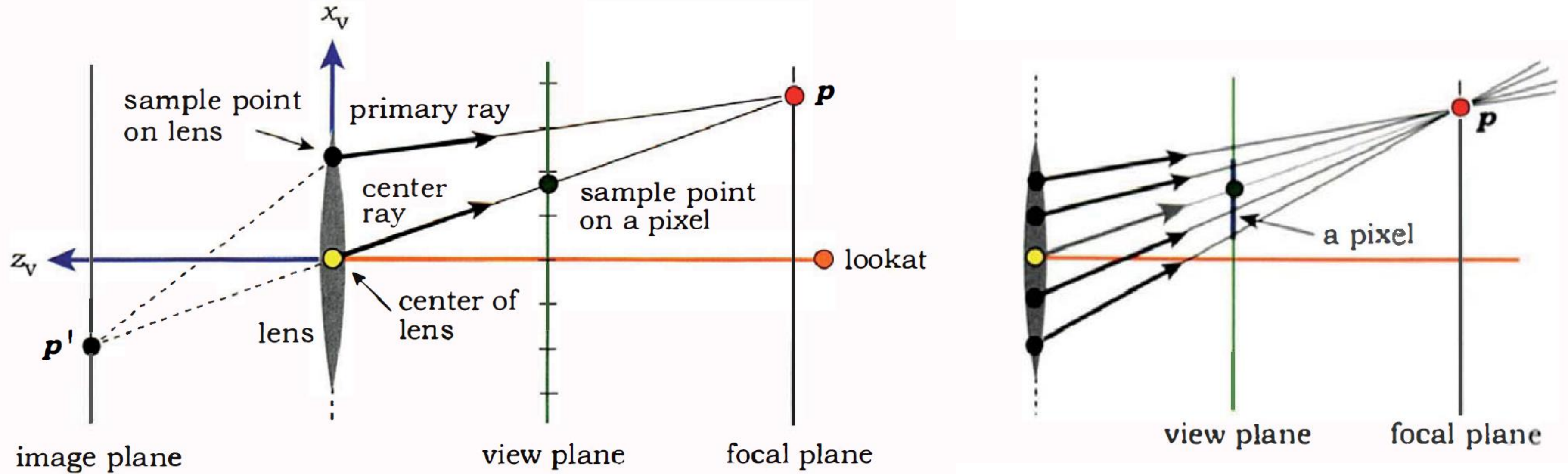


Figure 10.3. A thin-lens camera consists of a disk for the lens, a view plane (as usual), and a focal plane, all perpendicular to the view direction.

The simulation requires a large number of rays/pixel whose origins are distributed over the surface of the lens

Depth of Field [Suffern]



- To simulate DOF:
  - Compute the point **p** where the center ray hits the focal plane;
  - Use **p** and the sample point on the lens to compute the direction of the primary ray so that this ray also goes through **p**;
  - Ray-trace the primary ray into the scene; the center ray does not contribute to the pixel color
- But **p**, although in perfect focus, will not be antialiased; what to do?

DOF + Antialiasing [Suffern]

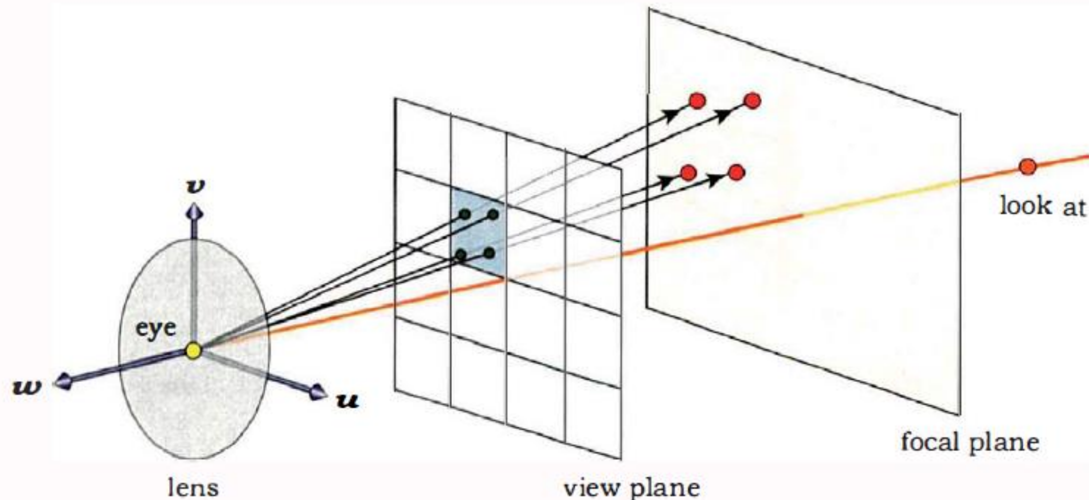


Figure 10.6. Four center rays go through different sample points on a pixel.

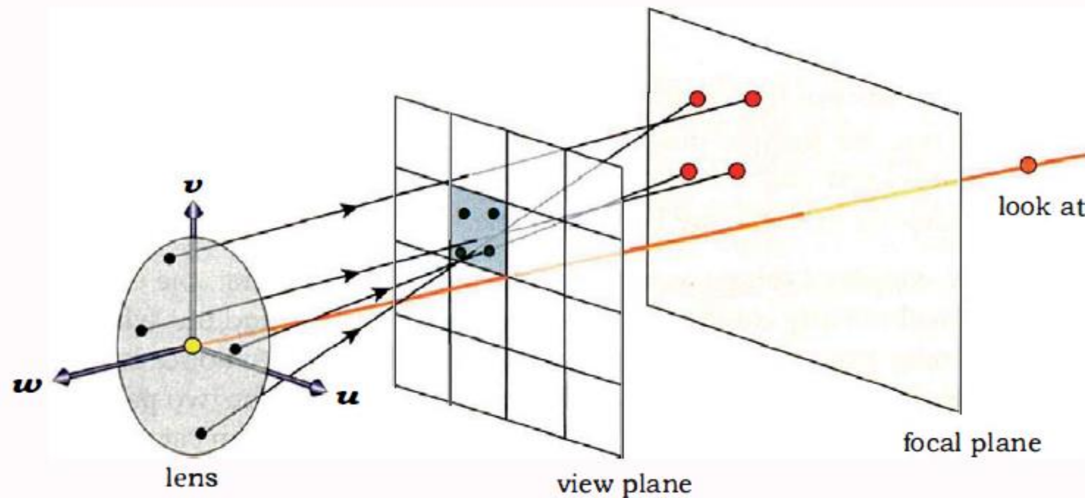
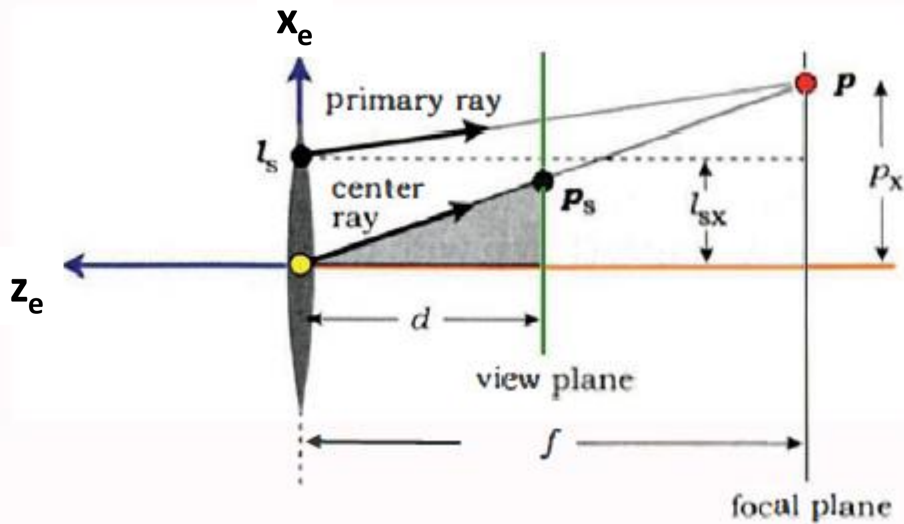


Figure 10.7. Four primary rays that start at sample points on the lens and hit the focal plane at the same points that the center rays in Figure 10.6 hit it.

Primary Rays Calc in WC [Suffern]

Ray PrimaryRay(const Vector& lens_sample, const Vector& pixel_sample) //in camera.h

We don't have to trace the center rays:



in camera coordinates:

$p = (p_x, p_y, -f)$,

$p_s = (p_{sx}, p_{sy}, -d)$,

$l_s = (l_{sx}, l_{sy}, 0)$ = *sample_unit_disk() * aperture*[1]

$p_x = p_{sx}\,(f/d)$

$p_y = p_{sy}(f/d)$.

[1]*aperture = Camera ->GetAperture(),*

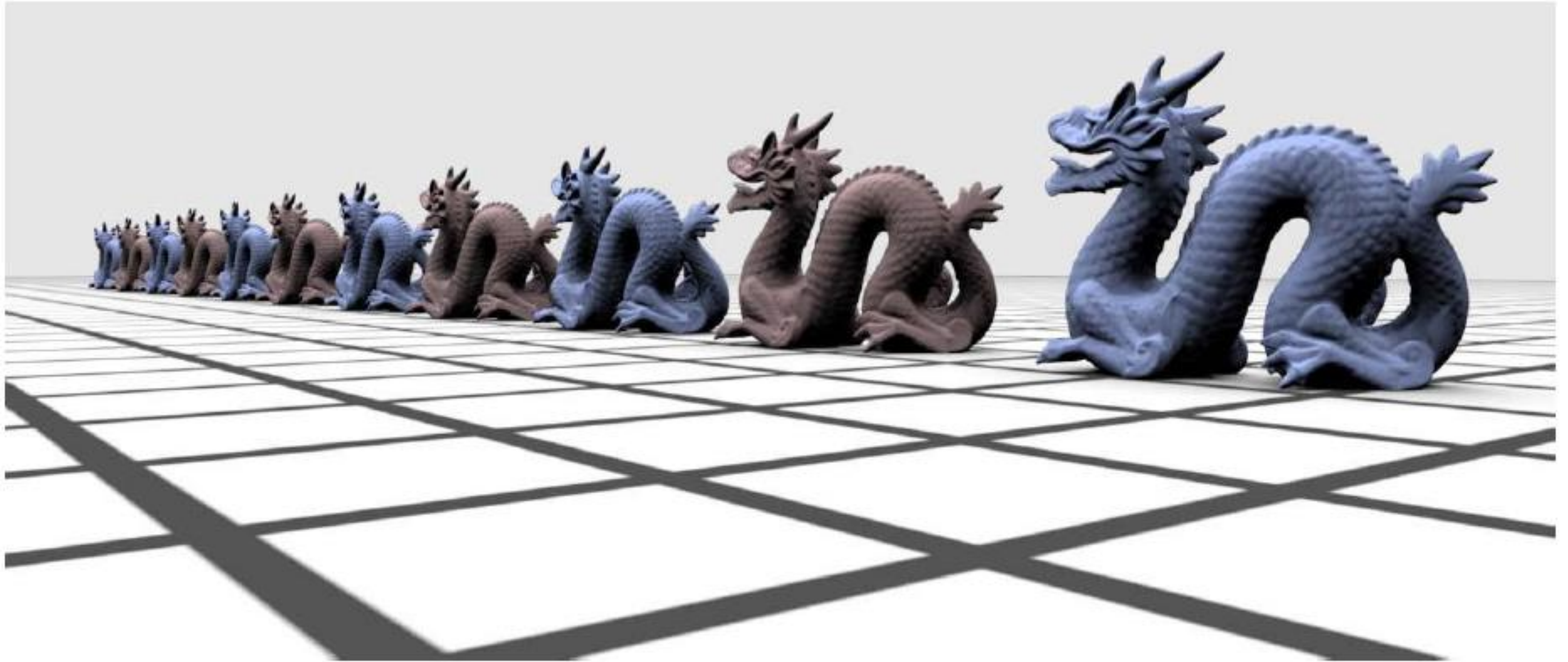
Direction of primary ray:

$$\mathbf{d} = normalize(p - l_s)$$
$$= normalize((p_x - l_{sx})\hat{\mathbf{x}}_e + (p_y - l_{sy})\hat{\mathbf{y}}_e - f\,\hat{\mathbf{z}}_e) \text{ //in WC}$$

Origin of primary ray:

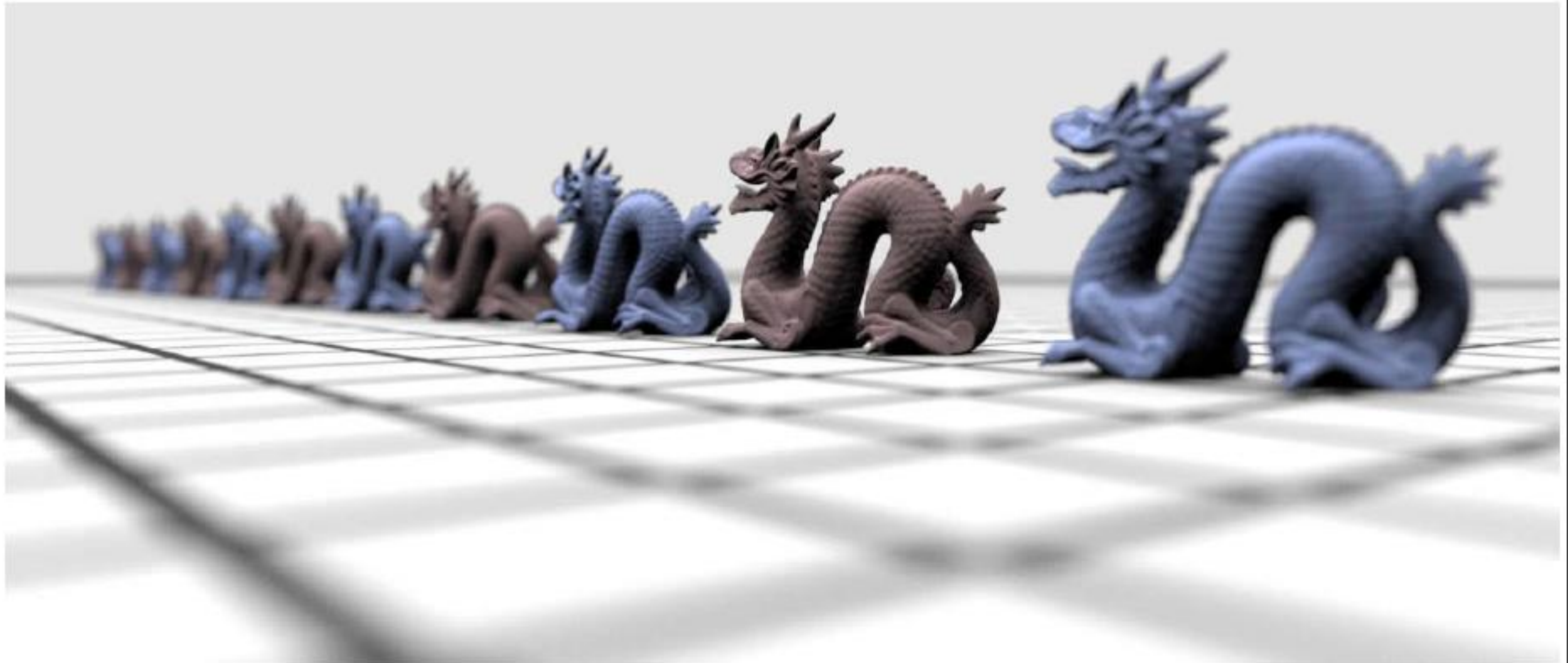
$eye_offset = eye + l_{sx} * \hat{\boldsymbol{x}}_{\boldsymbol{e}} + l_{sy} * \hat{\boldsymbol{y}}_{\boldsymbol{e}};$ //in WC

- **Very Small Aperture**



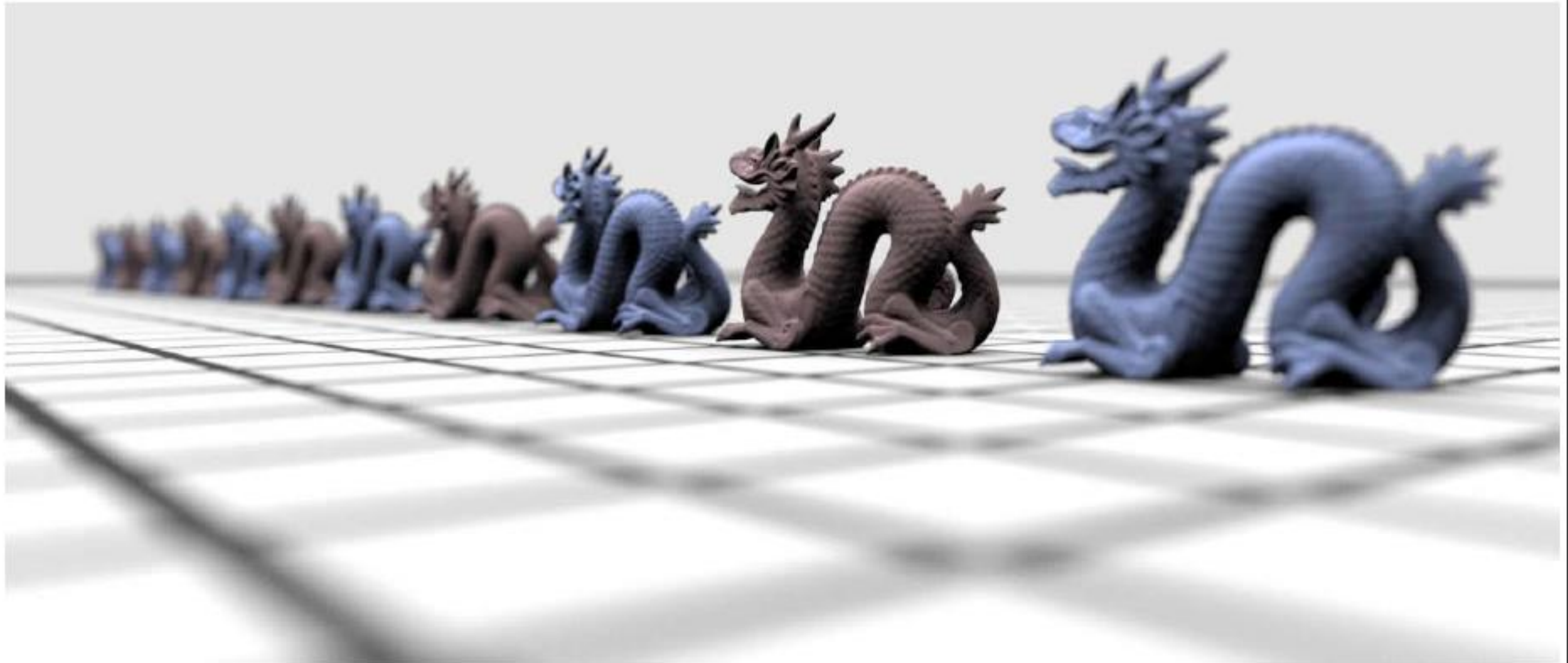
Depth of Field

- **Large Aperture**



Depth of Field

- **Large Aperture**

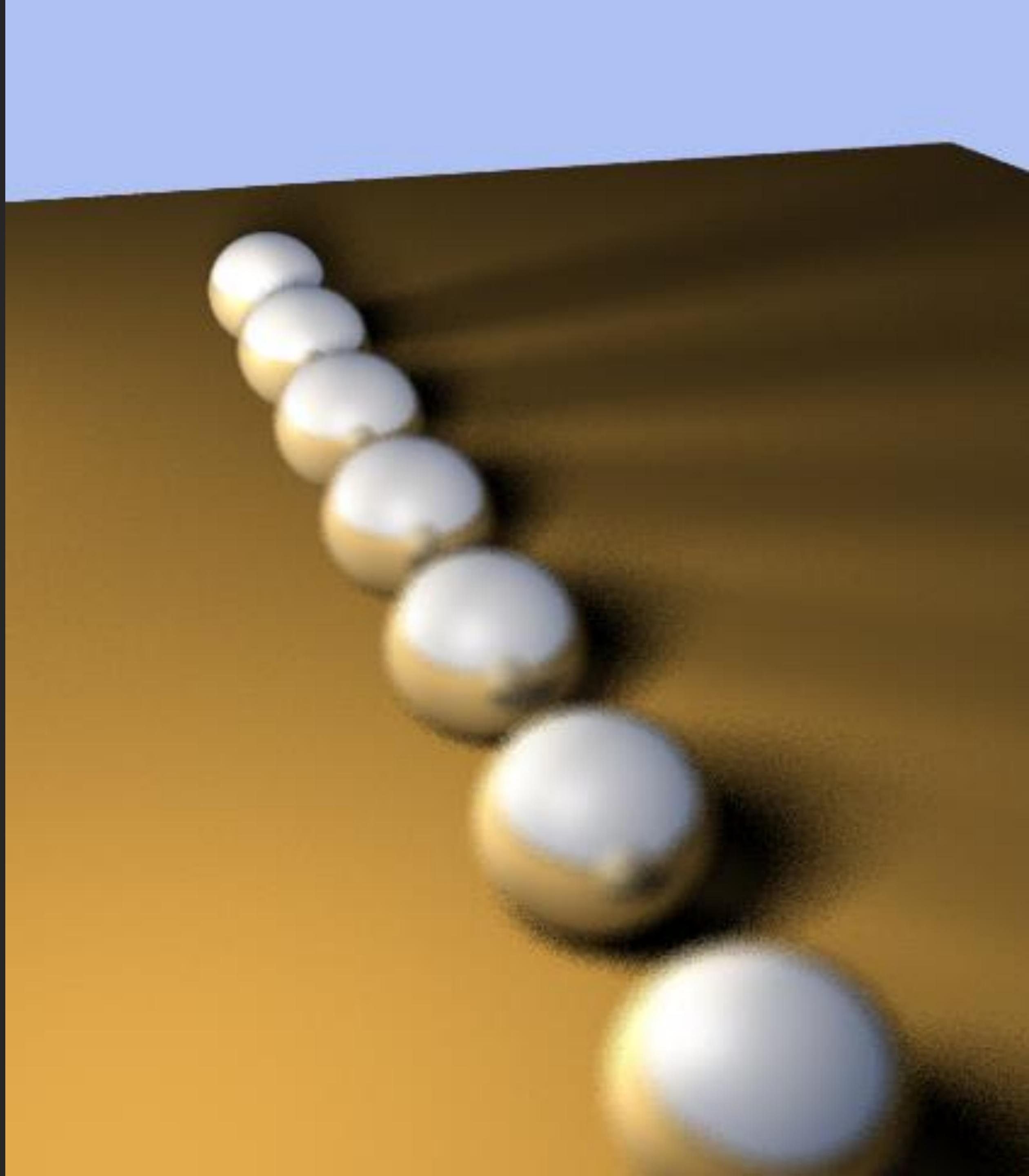


Depth of Field

- **Very Large Aperture**

Depth of Field

Assignment 1
DOF

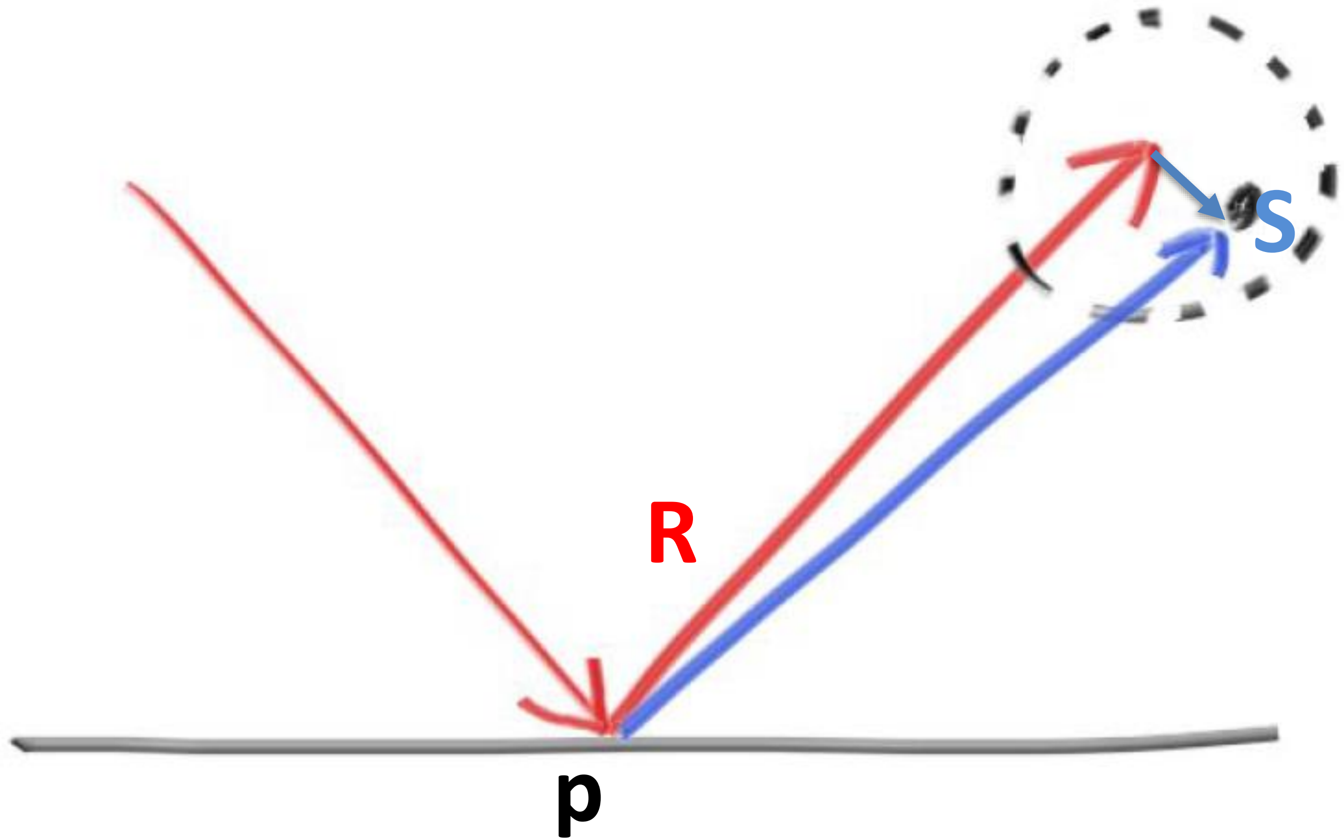


Fuzzy Reflection

- blurry reflections come from rough materials
  - raytracing only supports perfectly sharp mirror

[Jensen]

Fuzzy Reflection



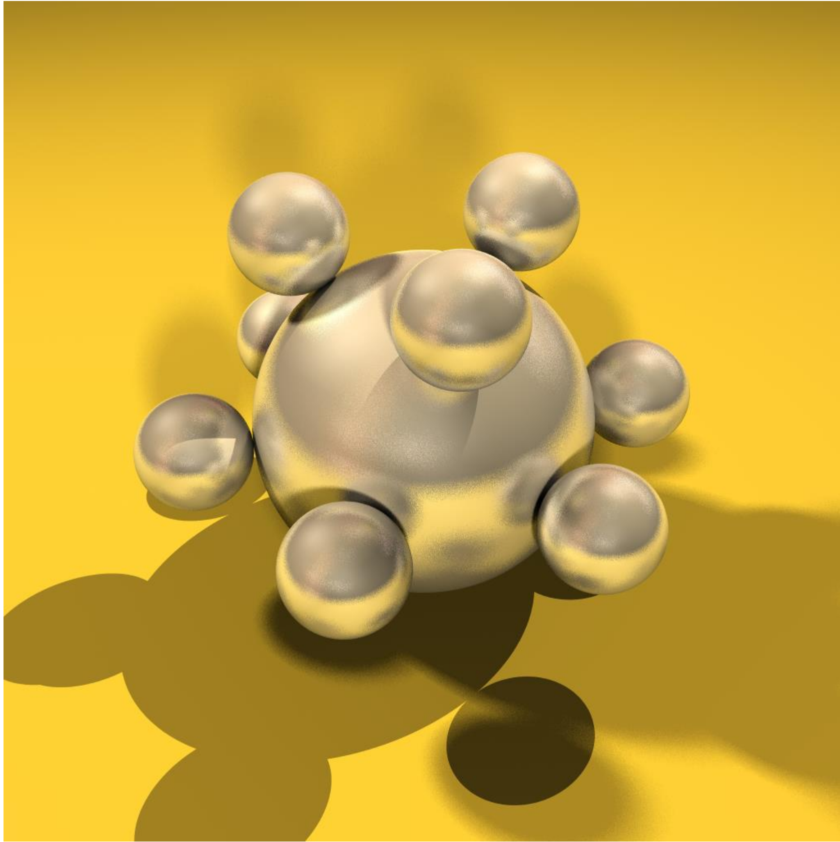
Sphere center = p + R

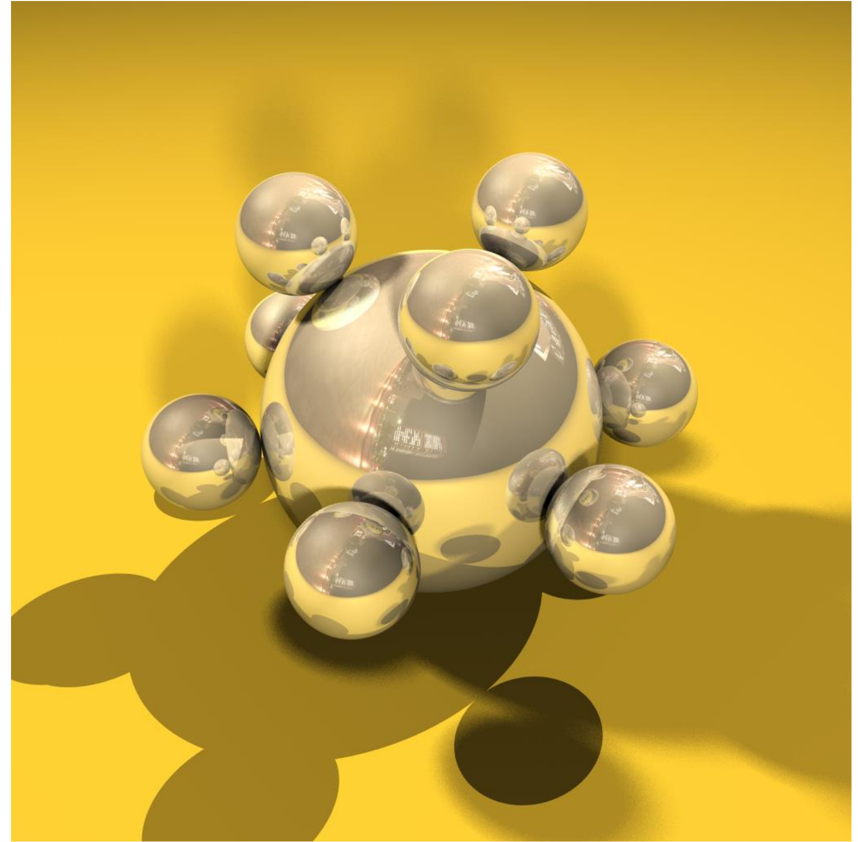
S = p + R + roughness_param * rand_in_unit_sphere()

ray.direction = S – p = normalize(R + roughness_param * rand_in_unit_sphere())

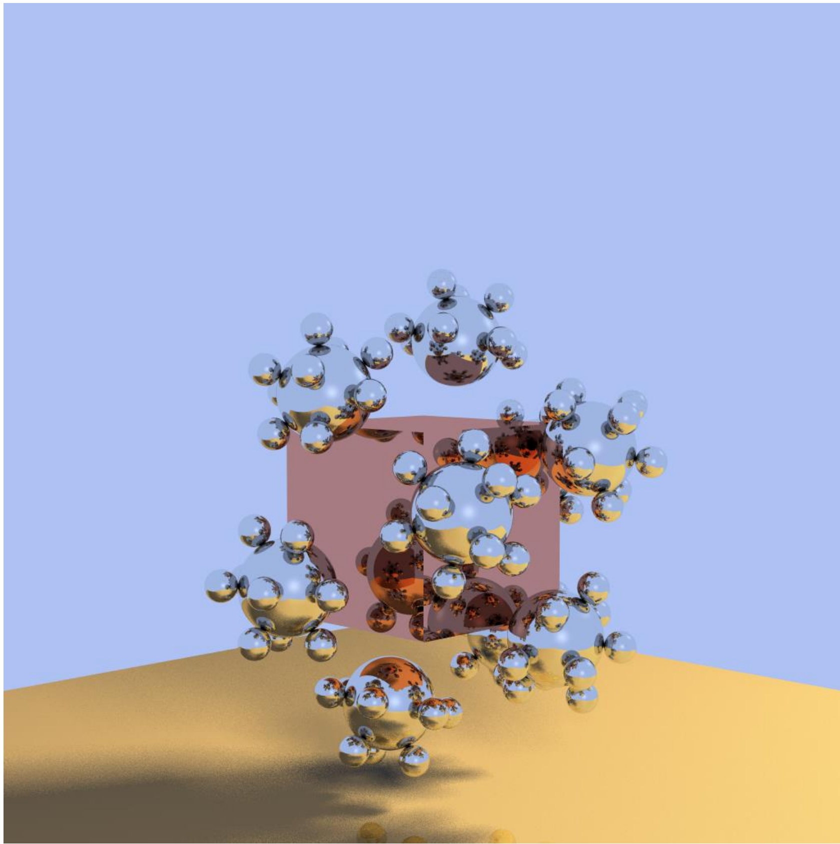
return(Dot(ray.direction, normal) > 0)

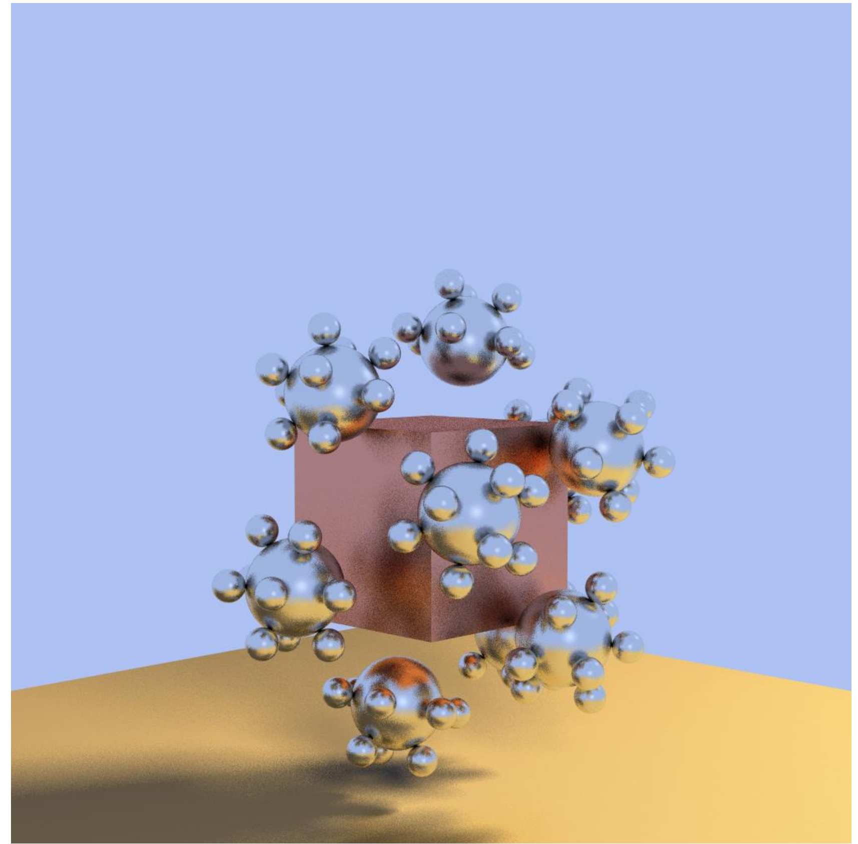
Fuzzy reflection (rougness = 0.3)

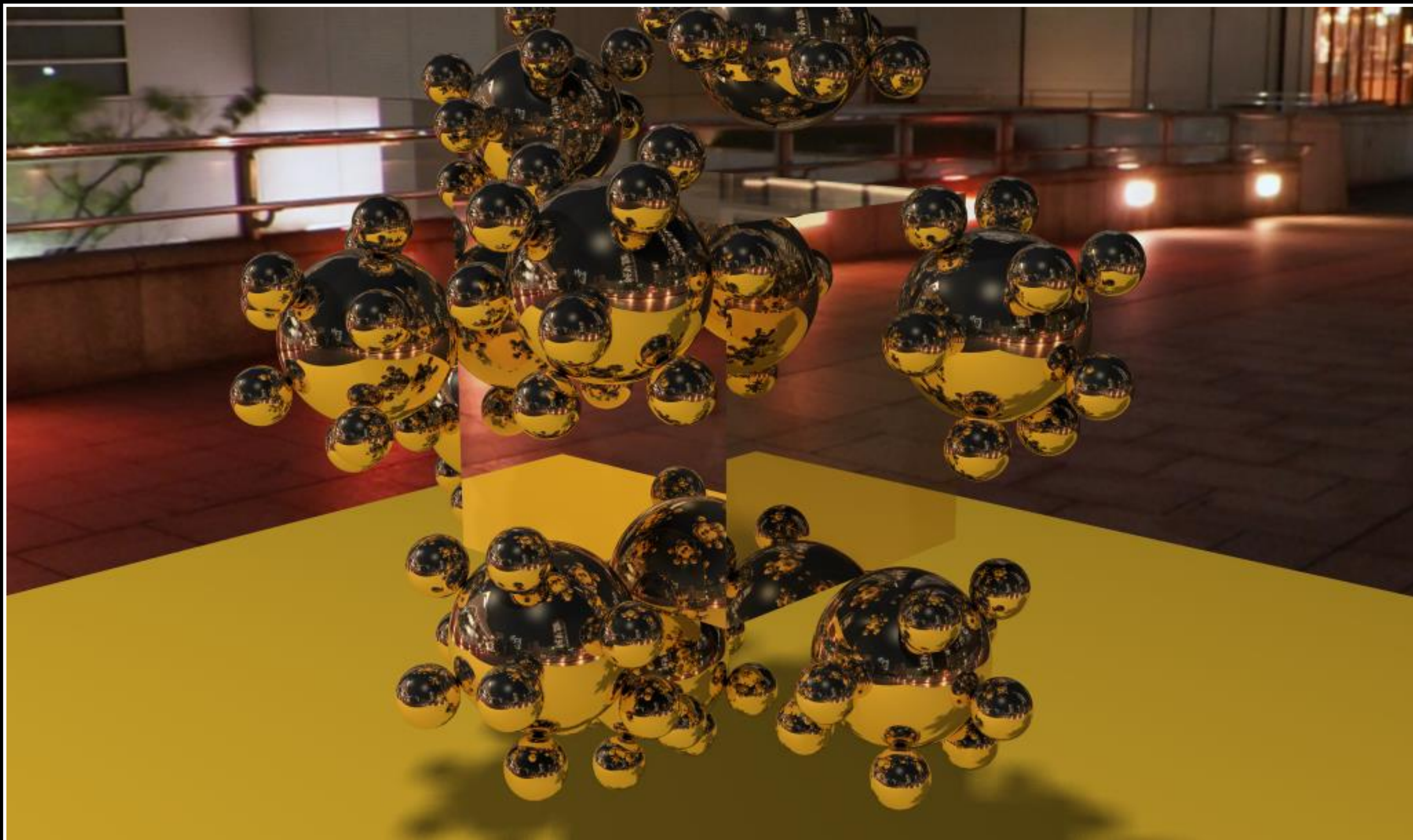




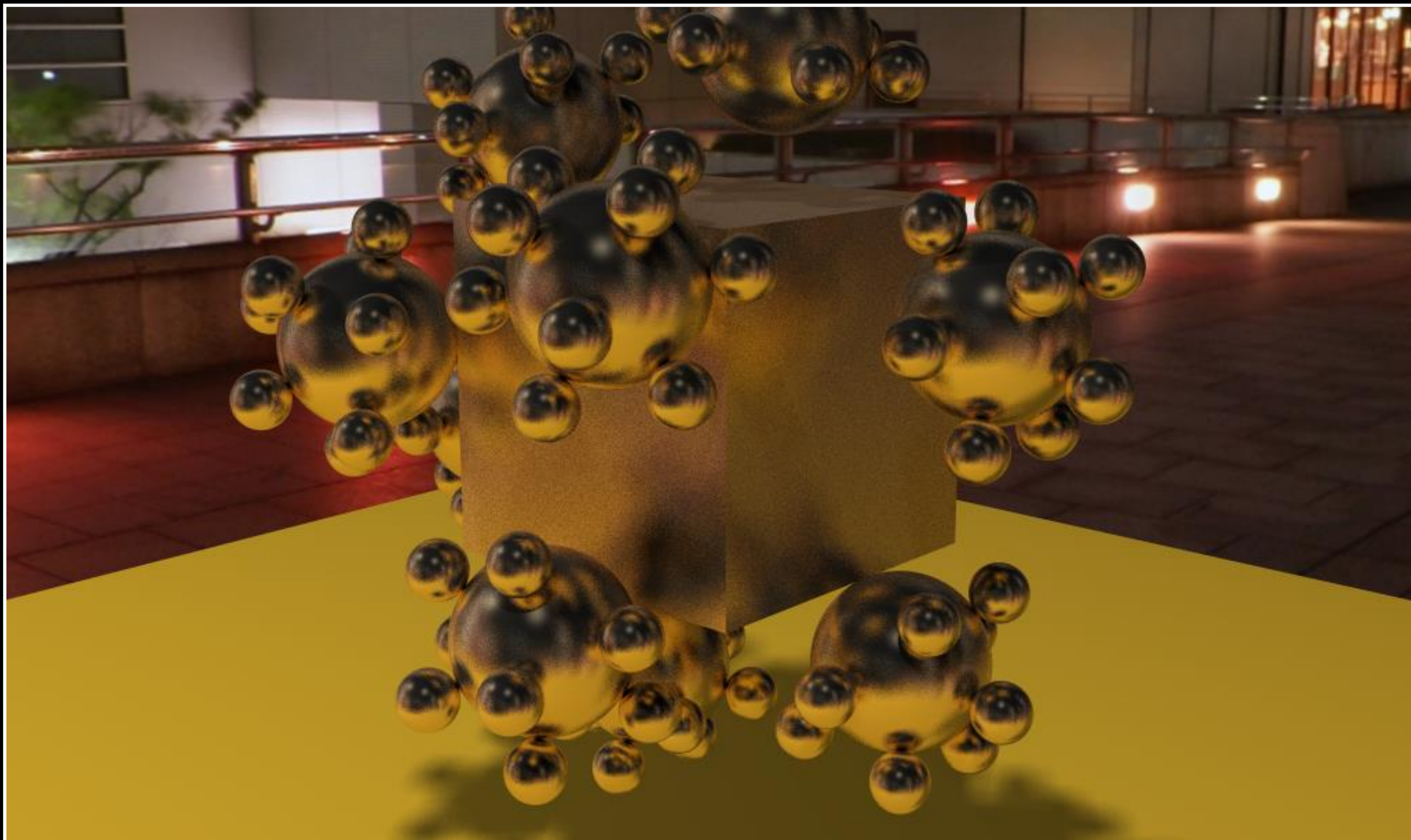
Fuzzy reflection (rougness = 0.3)







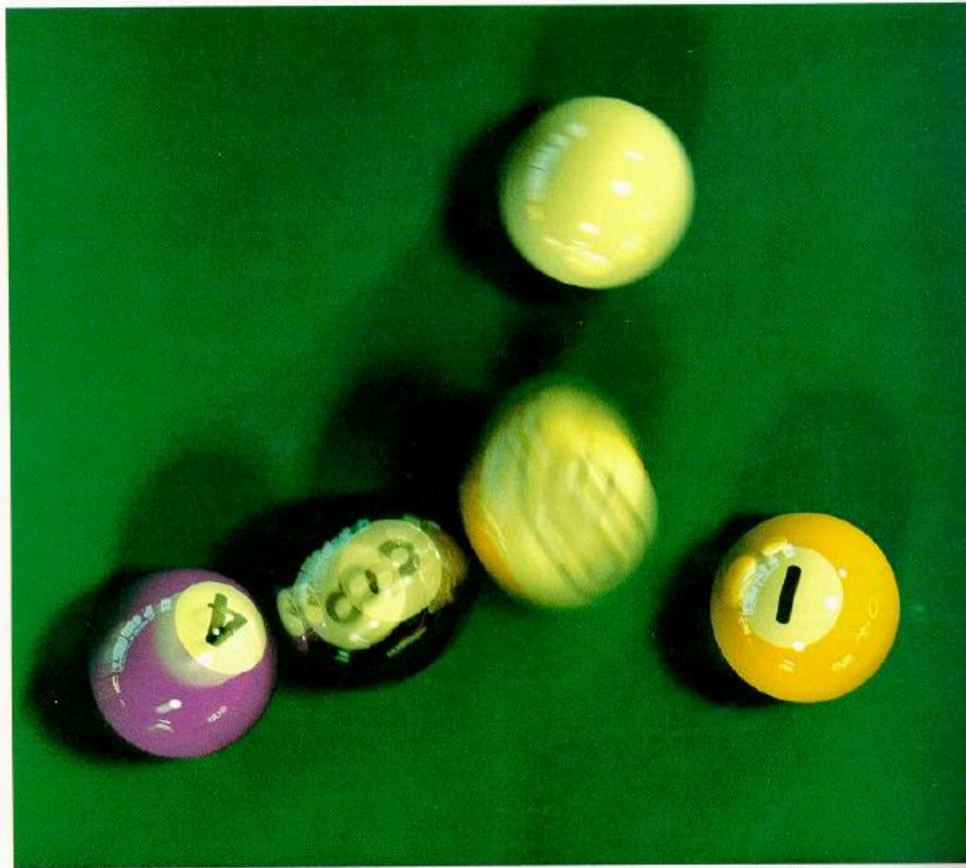
roughness = 0.0 64 spp



roughness = 0.3 64 spp

DRT to simulate motion blur

- Distributing rays over time gives:



DRT to simulate motion blur

```
class ray {
    public:
        ray() {}
        ray(const point3& origin, const vec3& direction, double time = 0.0)
            : orig(origin), dir(direction), tm(time)
        {}

        point3 origin() const  { return orig; }
        vec3 direction() const { return dir; }
        double time() const    { return tm; }

        point3 at(double t) const {
            return orig + t*dir;
        }

    public:
        point3 orig;
        vec3 dir;
        double tm;
};
```

DRT to simulate motion blur

```
class camera {
    public:
        // new:  add t0 and t1
        camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect, float aperture, float focus_dist,
 float t0, float t1) { // vfov is top to bottom in degrees
            time0 = t0;
            time1 = t1;
            lens_radius = aperture / 2;
            float theta = vfov*M_PI/180;
            float half_height = tan(theta/2);
            float half_width = aspect * half_height;
            origin = lookfrom;
            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);
            lower_left_corner = origin  - half_width*focus_dist*u -half_height*focus_dist*v - focus_dist*w;
            horizontal = 2*half_width*focus_dist*u;
            vertical = 2*half_height*focus_dist*v;
        }

        // new: add time to construct ray
        ray get_ray(float s, float t) {
            vec3 rd = lens_radius*random_in_unit_disk();
            vec3 offset = u * rd.x() + v * rd.y();
            float time = time0 + drand48()*(time1-time0);
            return ray(origin + offset, lower_left_corner + s*horizontal + t*vertical - origin - offset, time);
        }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        float time0, time1;  // new variables for shutter open/close times
        float lens_radius;
};
```

DRT to simulate motion blur

```
class moving_sphere: public hitable  {
    public:
        moving_sphere() {}
        moving_sphere(vec3 cen0, vec3 cen1, float t0, float t1, float r, material *m) :
 center0(cen0), center1(cen1), time0(t0),time1(t1), radius(r), mat_ptr(m)  {};
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        vec3 center(float time) const;
        vec3 center0, center1;
        float time0, time1;
        float radius;
        material *mat_ptr;
};

vec3 moving_sphere::center(float time) const{
    return center0 + ((time - time0) / (time1 - time0))*(center1 - center0);
}
```

Distribution Ray Tracing

distribute rays throughout a pixel to get spatial antialiasing

distribute rays in time to get temporal antialiasing (motion blur)

distribute rays in reflected ray direction to simulate gloss

distribute rays across area light source to simulate penumbras (soft shadows)

distribute rays across eye to simulate depth of field

distribute rays across hemisphere to simulate diffuse interreflection

also called: “distributed ray tracing” or stochastic ray tracing

aliasing is replaced by less visually annoying noise.

powerful idea! (but requires significantly more computation)