# Global Illumination I
# Whitted-Style Ray Tracing

# Why Ray Tracing?

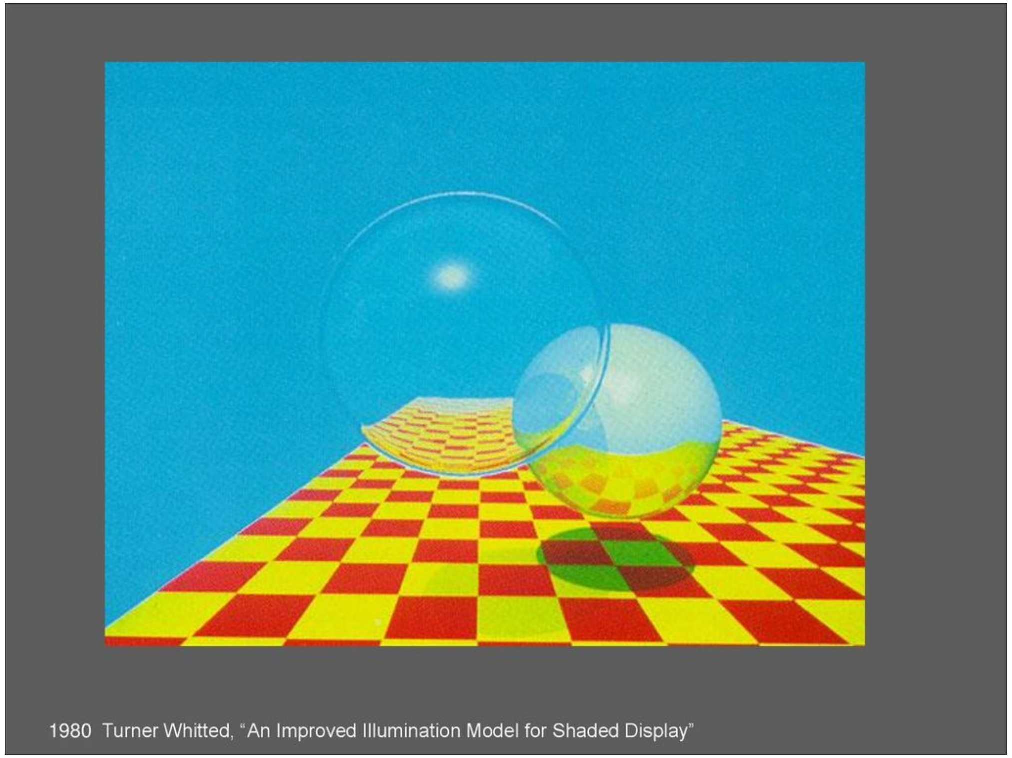
- Rasterization couldn't handle **global effects** well
  - (Soft) shadows
  - Light bounces more than once

Soft shadows

Glossy reflection

Indirect illumination

# Ray Tracing by Turner Whitted




The first scene through ray tracing

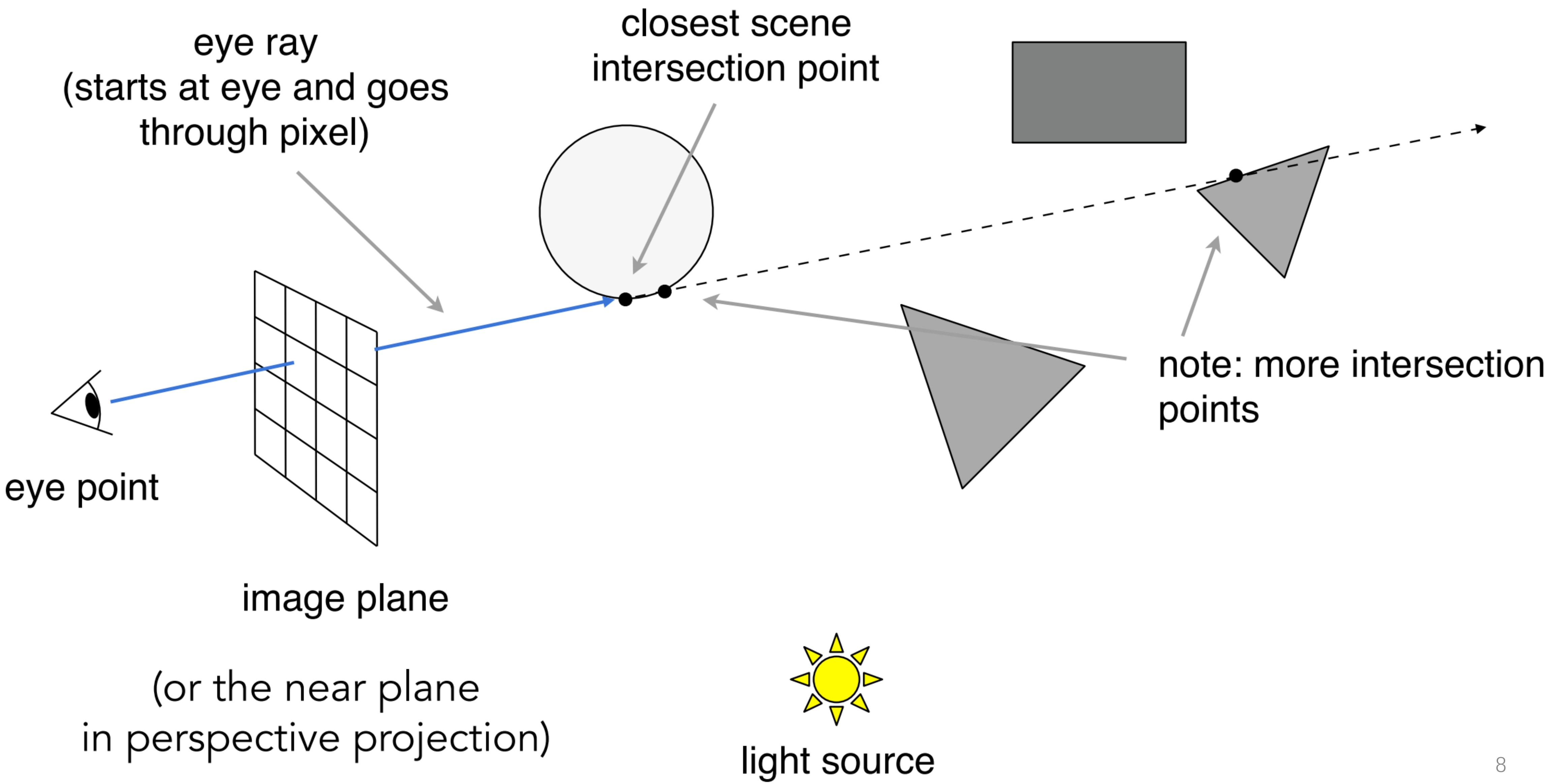
Turner Whitted (right)

# From Rasterization to Ray Tracing

- Simple shading (typified by OpenGL, z-buffering, and Phong illumination model) assumes:
  - direct illumination (light leaves source, bounces at most once, enters eye)
  - no shadows (except using shadow buffer)
  - opaque surfaces
  - point light sources (otherwise integration for area lights)
  - sometimes fog
- (Whitted-style) ray tracing relaxes that, simulating:
  - specular reflection
  - shadows
  - transparent surfaces (transmission with refraction)
  - sometimes indirect illumination (a.k.a. global illumination)
  - sometimes area light sources
  - sometimes fog

# Ray Casting

# Let's start from: Ray Casting




(or the near plane
in perspective projection)

# Ray Casting

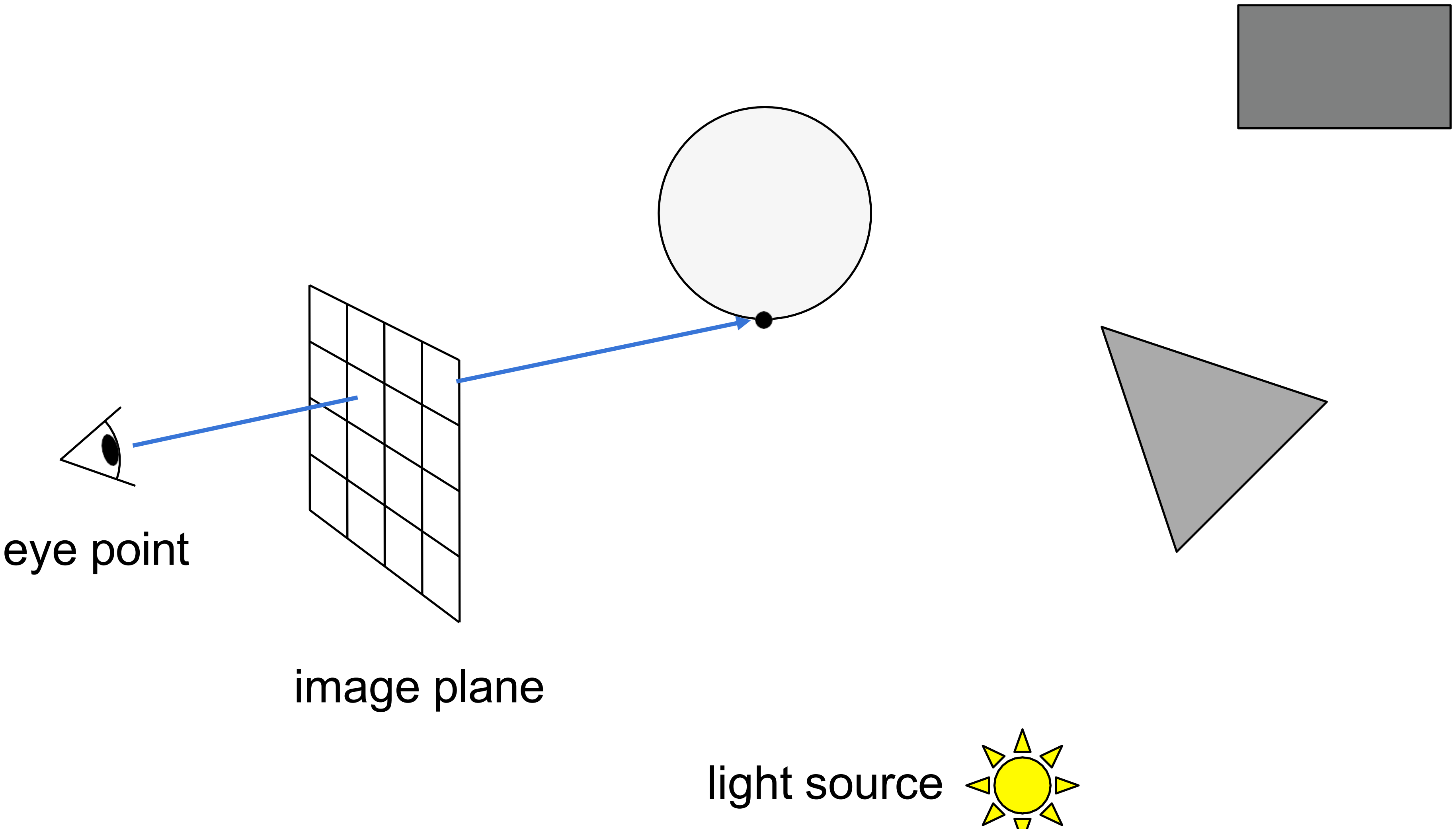
- A very flexible visibility algorithm
  - loop y, loop x
    - shoot ray from eye point through pixel (x,y) into scene
    - intersect with all surfaces, find first one the ray hits
    - shade that surface point to compute pixel (x,y)'s color

```
Raycast()                          // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)                         // fire a ray, return RGB radiance
                                   // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface
  normal, material properties, etc.)

Shade(point, ray)                 // return radiance of light leaving
                                  // point in opposite of ray direction
    calculate surface normal vector
    check shadow map for light visibility
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

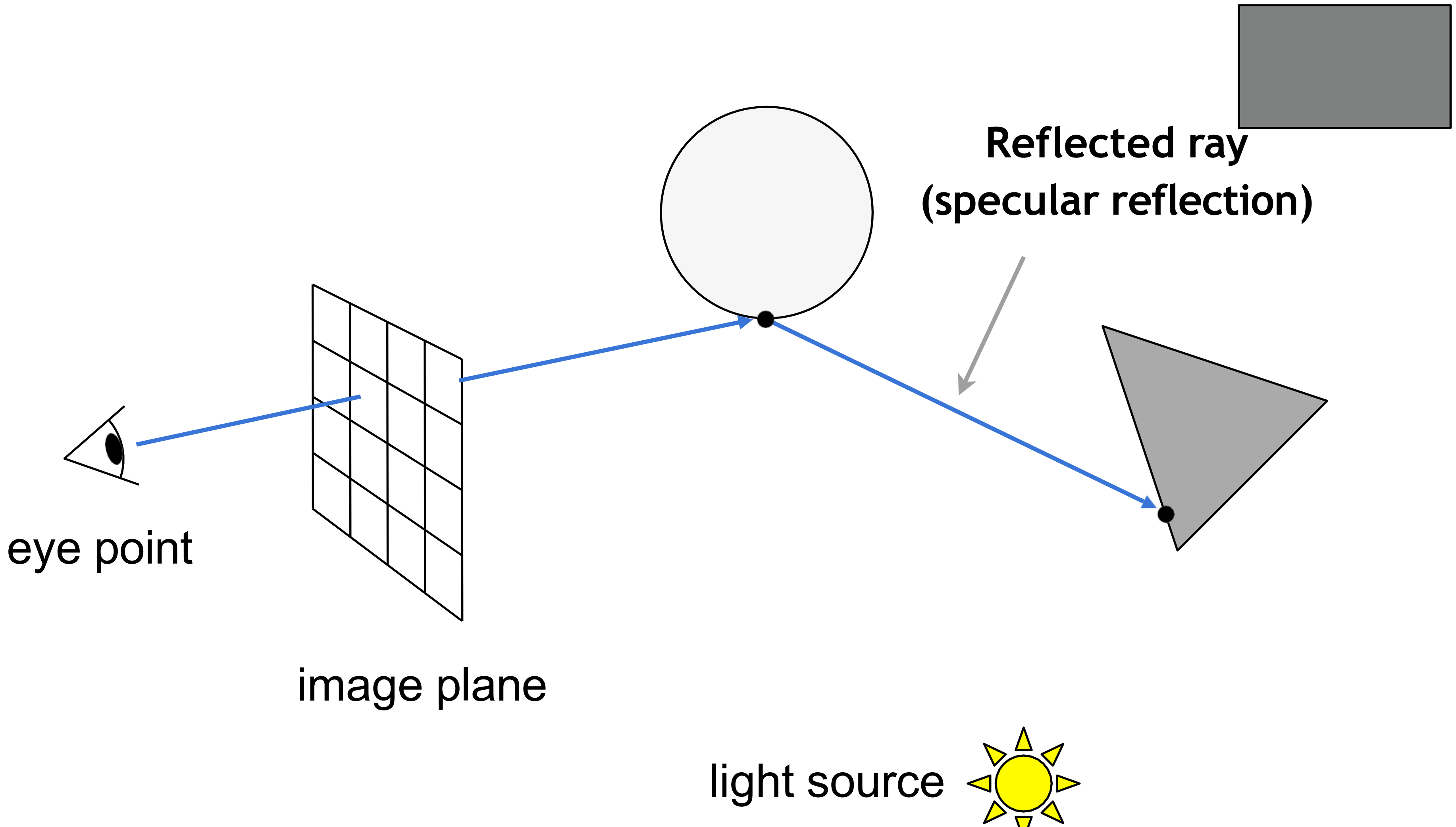
# Ray Casting

- This can be easily generalized to give recursive ray tracing, that will be discussed later
- Can handle translucency (which rasterization cannot!)
- calc_intersection (ray, surface) is the most important operation
  - compute not only coordinates, but also geometric or appearance attributes at the intersection point

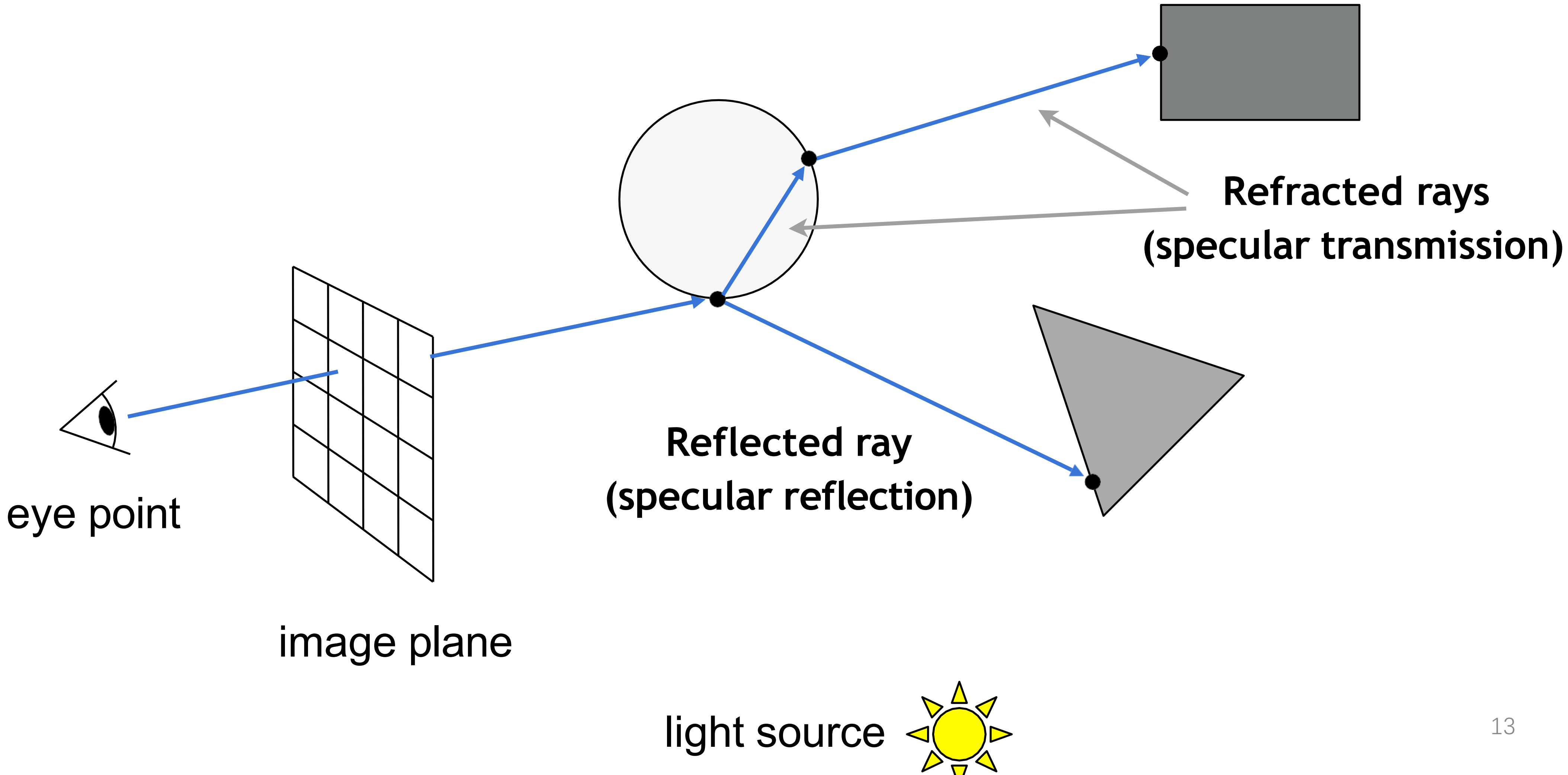
# Recursive ray tracing



# Recursive ray tracing

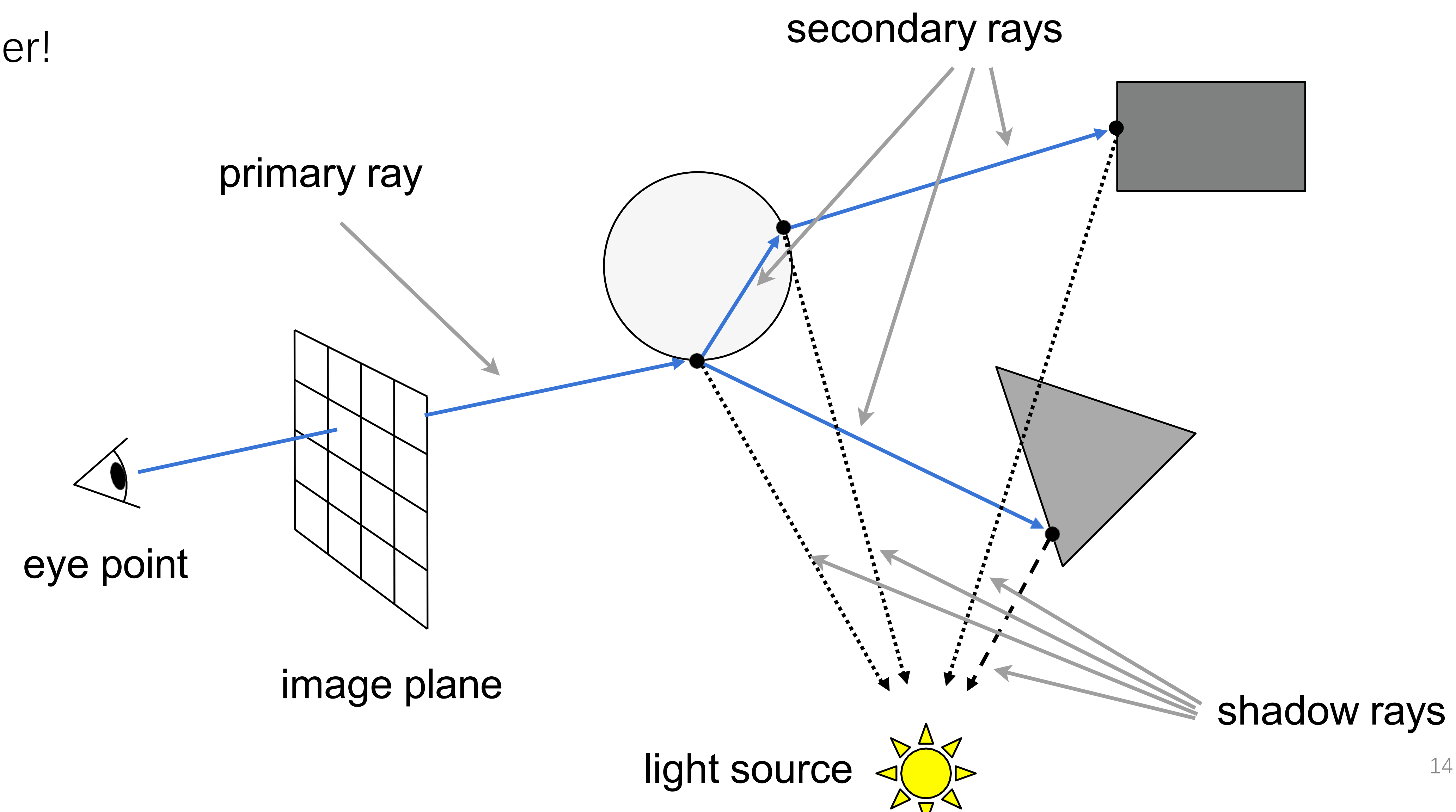


# Recursive ray tracing



# Recursive ray tracing

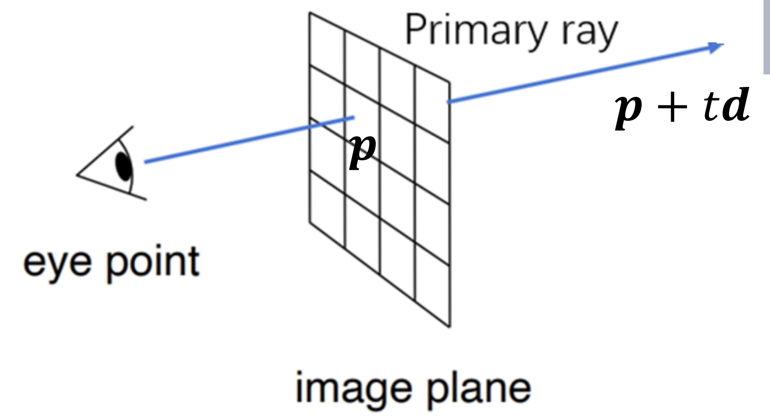
Until Later!



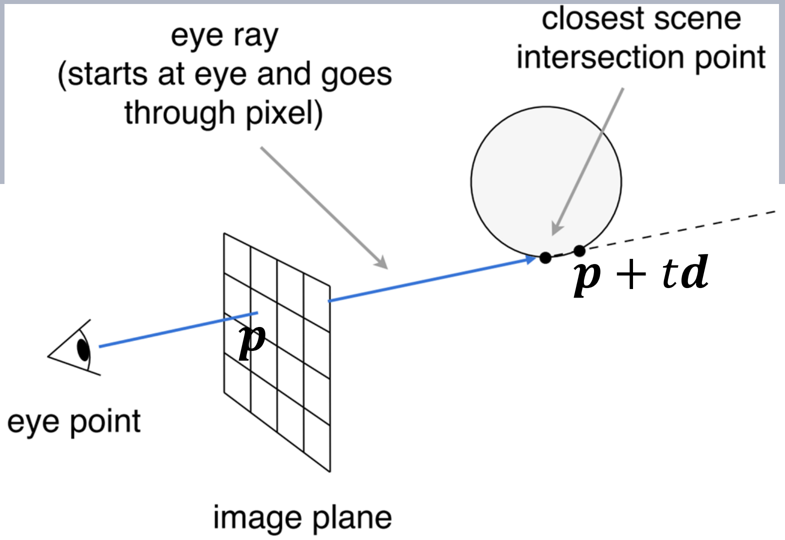
# Ray-Surface Intersection

# Ray Equation

- How to represent a ray?
  - A ray is $\boldsymbol{p} + t\boldsymbol{d}$: $\boldsymbol{p}$ is ray origin, $\boldsymbol{d}$ the direction
  - $t = 0$ at origin of ray, $t > 0$ in positive direction of ray
  - typically assume $||\boldsymbol{d}|| = 1$
  - $\boldsymbol{p}$ and $\boldsymbol{d}$ are typically computed in world space



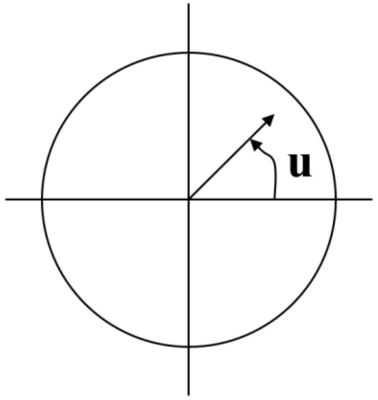
# Ray-Surface Intersections




- How to represent a ray?
  - A ray is $\boldsymbol{p} + t\boldsymbol{d}$: $\boldsymbol{p}$ is ray origin, $\boldsymbol{d}$ the direction
  - $t = 0$ at origin of ray, $t > 0$ in positive direction of ray
  - typically assume $||\boldsymbol{d}|| = 1$
  - $\boldsymbol{p}$ and $\boldsymbol{d}$ are typically computed in world space
- Recap: how to represent a surface?
  - Implicit functions: $f(\boldsymbol{x}) = 0$
  - Parametric functions: $\boldsymbol{x} = \boldsymbol{g}(u, v)$

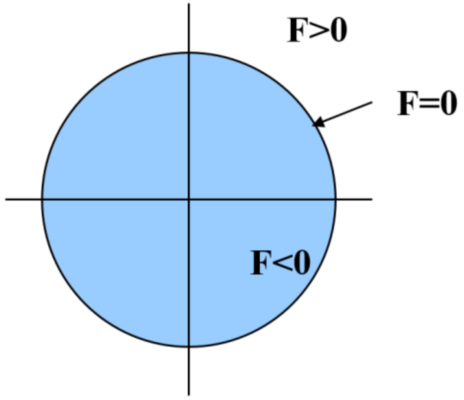
Solve the x and t for

$$\begin{aligned} \mathbf{x} &= \boldsymbol{p} + t\boldsymbol{d} \\ f(\boldsymbol{x}) &= 0 \end{aligned}$$




**Parametric**

$x(u) = r \cos(u)$
$y(u) = r \sin(u)$




**Implicit**

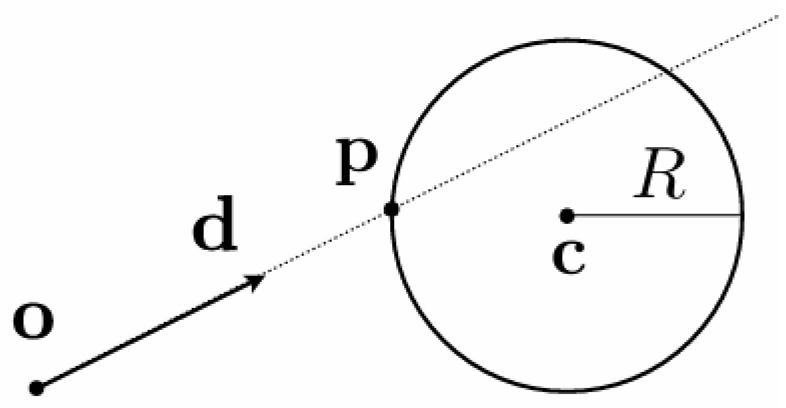
$F(x,y) = x^2 + y^2 - r^2$

# Ray-Surface Intersections

- Compute Intersections:
  - Substitute ray equation for x= $\boldsymbol{p} + t\boldsymbol{d}$
  - Find roots
  - Implicit: $f(\boldsymbol{p} + t\boldsymbol{d}) = 0$
    - one equation in one unknown – univariate root finding
  - Parametric: $\boldsymbol{p} + t\boldsymbol{d} - \boldsymbol{g}(u, v) = 0$
    - three equations in three unknowns (t,u,v) – multivariate root finding
  - For univariate polynomials, use closed form solution; otherwise, use numerical root finder

# Ray-Sphere Intersection

- Ray-sphere intersection is an easy case
- A sphere's implicit function is: $x^2 + y^2 + z^2 - r^2 = 0$ if sphere at origin
- The ray equation is: $x = p_x + td_x$
$$y = p_y + td_y$$
$$z = p_z + td_z$$
- Substitution gives: $(p_x + td_x)^2 + \left(p_y + td_y\right)^2 + \left(p_y + td_y\right)^2 - r^2 = 0$
- A quadratic equation in $t$.
- Solve the standard way: $A = d_x^2 + d_y^2 + d_z^2 = 1$ (unit vector)
$$B = 2\left(p_x d_x + p_y d_y + p_z d_z\right)$$
$$C = p_x^2 + p_y^2 + p_z^2 - r^2$$
- Quadratic formula has two roots: $t = (-B \pm \sqrt{B^2 - 4C})/2$
    - which correspond to the two intersection points
    - We take the smaller t (the first intersection)
    - negative discriminant means ray misses sphere

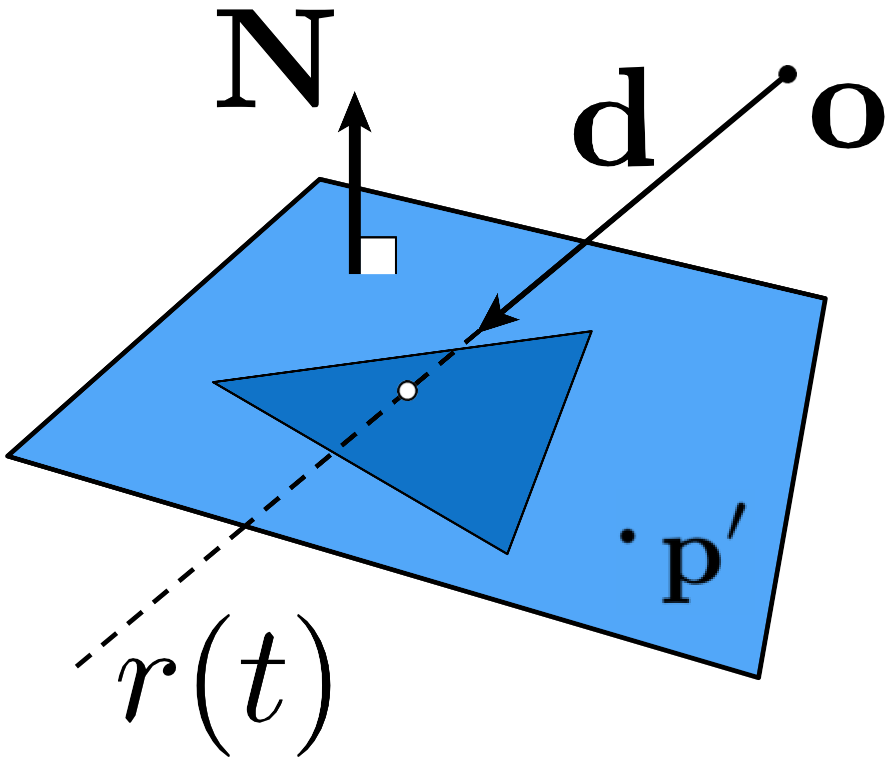


# Ray Intersection With Triangle

Triangle is in a plane

- Ray-plane intersection
- Test if hit point is inside triangle

Many ways to optimize…




$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$

all points on plane

one point on plane

normal vector

$$ax + by + cz + d = 0$$

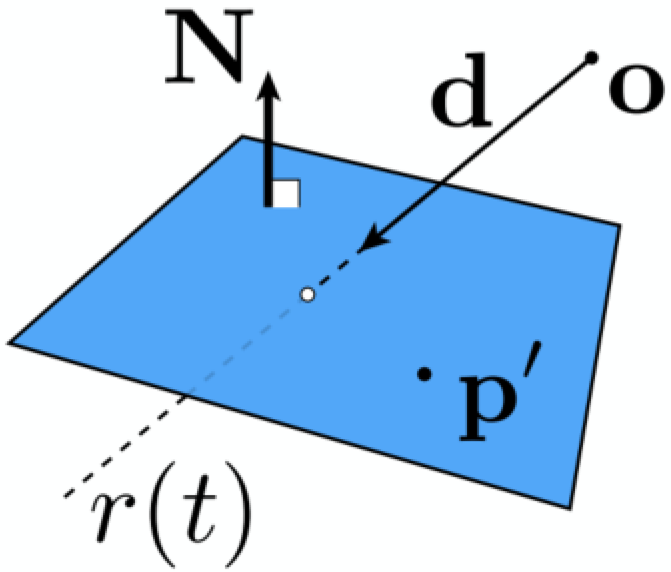
# Ray Intersection With Plane

Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d},\ 0 \leq t < \infty$$

Plane equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$




Solve for intersection

Set $\mathbf{p} = \mathbf{r}(t)$ and solve for $t$

$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = (\mathbf{o} + t\,\mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0$$

$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$$

**Check:** $0 \leq t < \infty$

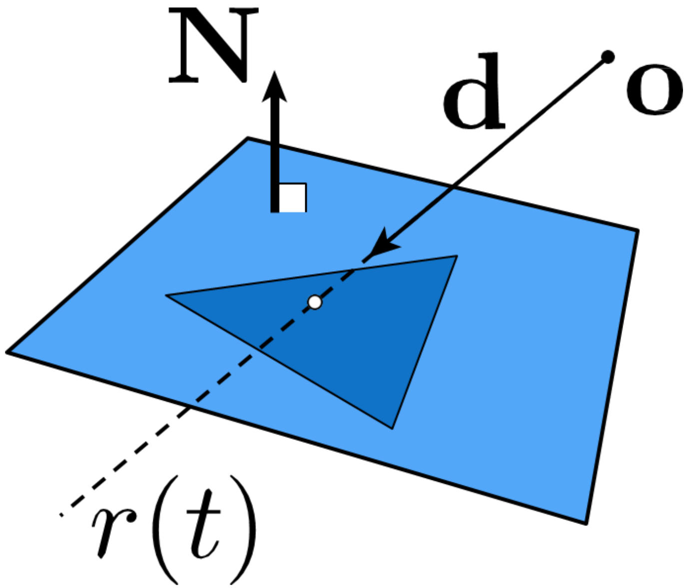
# Ray Intersection With Plane

Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d},\ \ 0 \leq t < \infty$$

Plane equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$



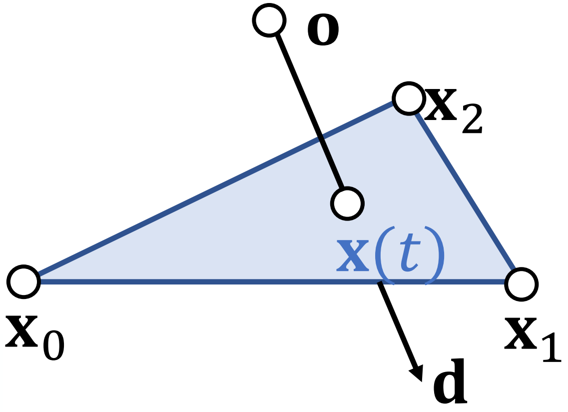

Solve for intersection

Set $\mathbf{p} = \mathbf{r}(t)$ and solve for $t$

$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = (\mathbf{o} + t\,\mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0$$

$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$$

**Check:** $0 \leq t < \infty$



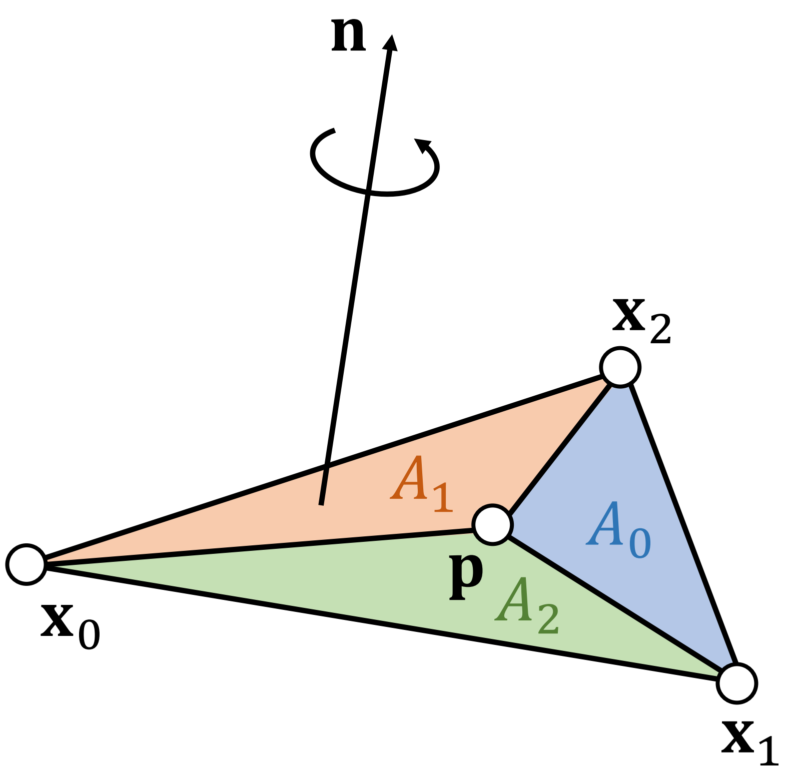

If $t > 0$ and $\mathbf{x}(t)$ *inside*:
return Intersection point, $\mathbf{x}(t)$

# Barycentric Coordinates

$$\boxed{\mathbf{p} = b_0\mathbf{x}_0 + b_1\mathbf{x}_1 + b_2\mathbf{x}_2}$$

$$\left.\begin{aligned} b_0 &= A_0/A \\ b_1 &= A_1/A \\ b_2 &= A_2/A \end{aligned}\right\} \quad b_0 + b_1 + b_2 = 1$$

$$A_0 = \tfrac{1}{2}(\mathbf{x}_1 - \mathbf{p}) \times (\mathbf{x}_2 - \mathbf{p}) \cdot \mathbf{n}$$
$$A_1 = \tfrac{1}{2}(\mathbf{x}_2 - \mathbf{p}) \times (\mathbf{x}_0 - \mathbf{p}) \cdot \mathbf{n}$$
$$A_2 = \tfrac{1}{2}(\mathbf{x}_0 - \mathbf{p}) \times (\mathbf{x}_1 - \mathbf{p}) \cdot \mathbf{n}$$

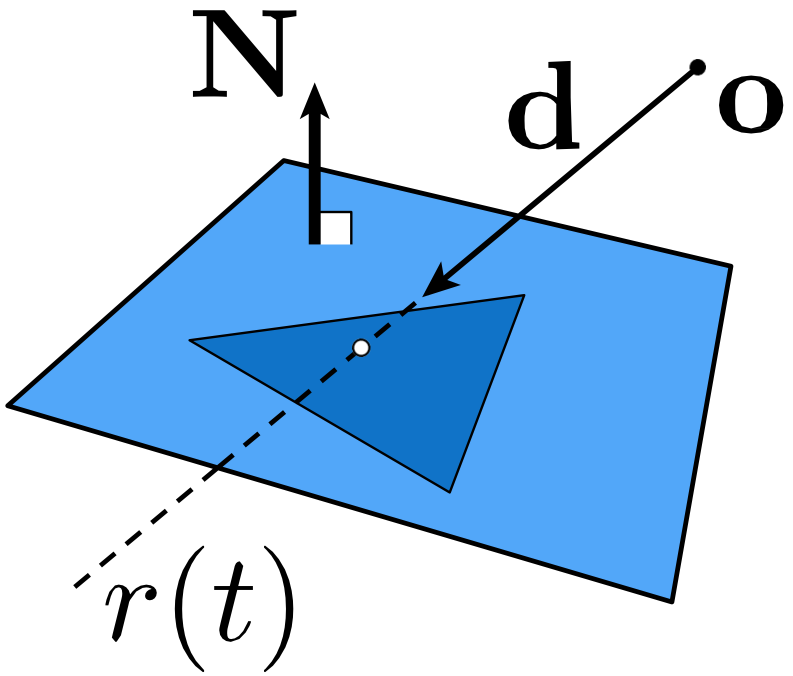



Inside: $0 < b_i < 1 \ \ (i = 0,1,2)$, and coplanar

Outside: otherwise

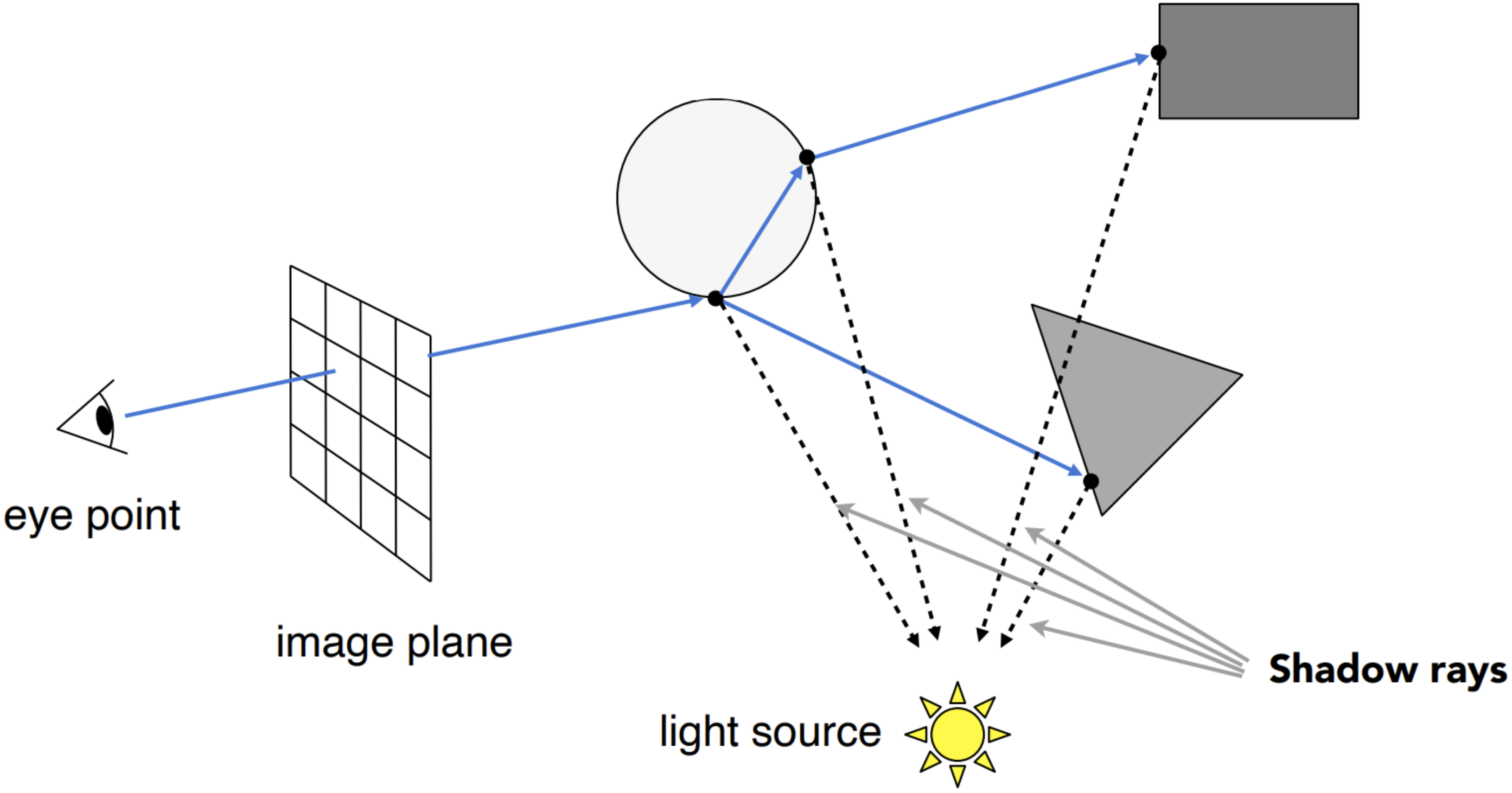
# Ray-Polygon Intersection

- Assuming we have a planar polygon
  - first, find intersection point of ray with plane
  - then check if that point is inside the polygon
- Latter step is a point-in-polygon test in 3-D:
  - inputs: a point x in 3-D and the vertices of a polygon in 3D
  - output: INSIDE or OUTSIDE
  - problem can be reduced to point-in-polygon test in 2-D (**how?**)
- Point-in-polygon test in 2-D:
  - easiest for triangles
  - easy for convex n-gons
  - harder for concave polygons
  - most common approach: subdivide all polygons into triangles
  - for optimization tips, see article by Haines in the book **Graphics Gems IV**



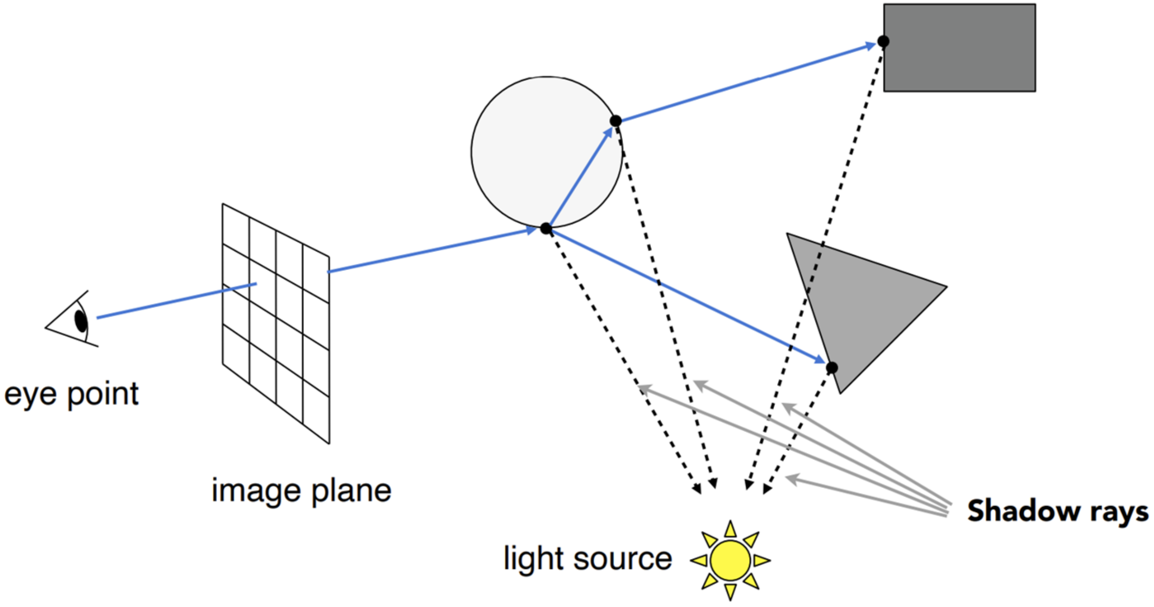
# Whitted-Style Ray Tracing

# Whitted-Style Ray Tracing



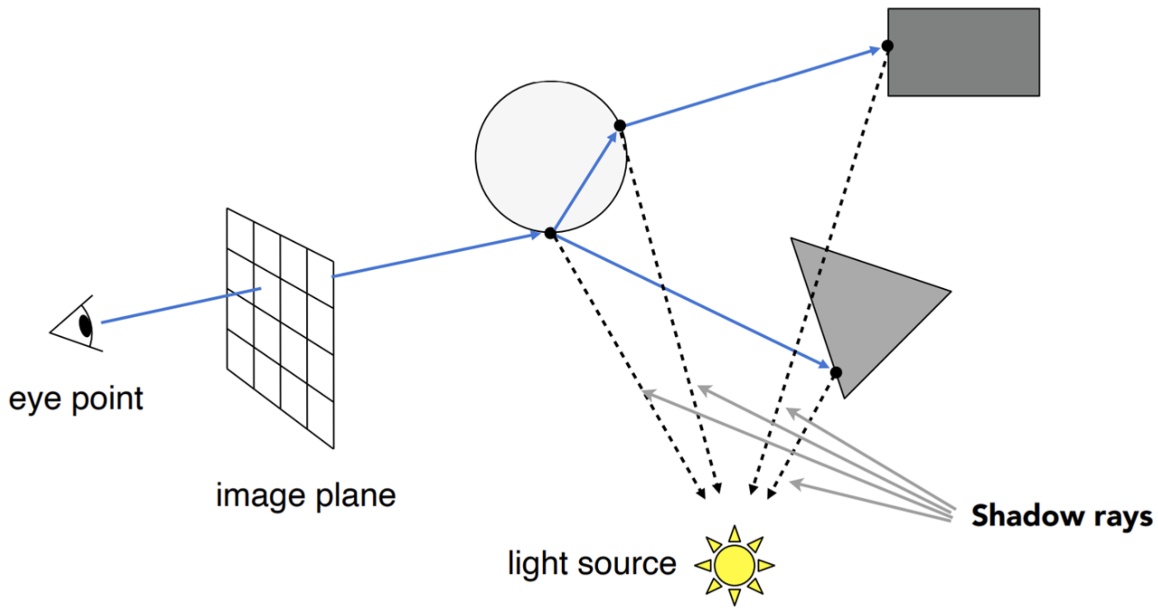
# Ray Types

- We'll distinguish four ray types:
  - Eye rays: originating at the eye
  - Shadow rays: from surface point toward light source
  - Reflection rays: from surface point in mirror direction
  - Transmission rays: from surface point in refracted direction



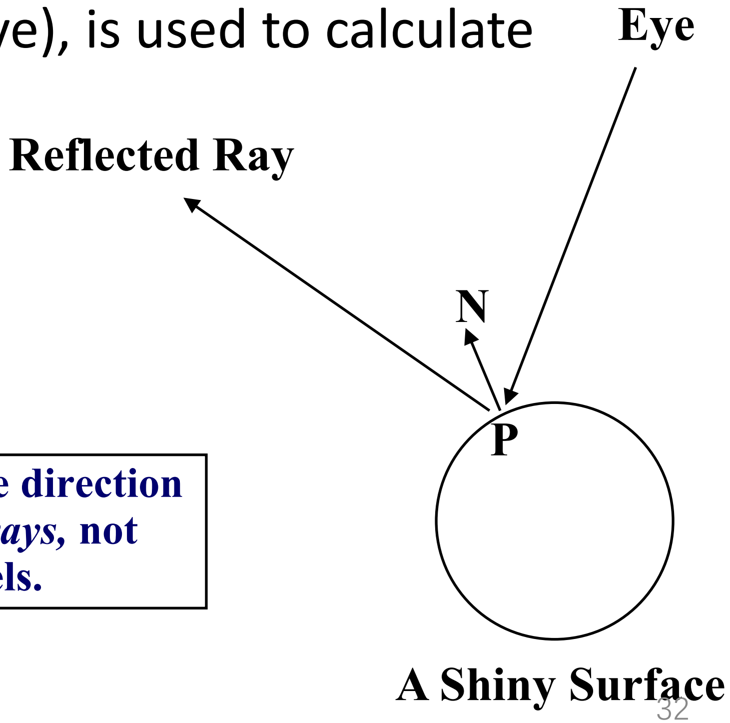
# Ray Tracing Algorithm

1. send ray from eye through each pixel
2. compute point of closest intersection with a scene surface
3. shade that point by computing shadow rays
4. **spawn reflected and refracted rays, repeat 2-4 steps**



# Specular Reflection Rays

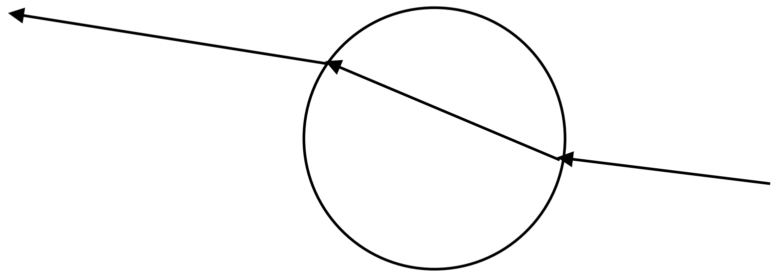
- An eye ray hits a shiny surface
  - We know the direction from which a specular reflection would come, based on the surface normal
  - Fire a ray in this reflected direction
  - The reflected ray is treated just like an eye ray: it hits surfaces and spawns new rays
  - Light flows in the direction opposite to the rays (towards the eye), is used to calculate shading
  - It's easy to calculate the reflected ray direction




**Note: arrowheads show the direction in which we're *tracing the rays*, not the direction the light travels.**

# Specular Transmission Rays

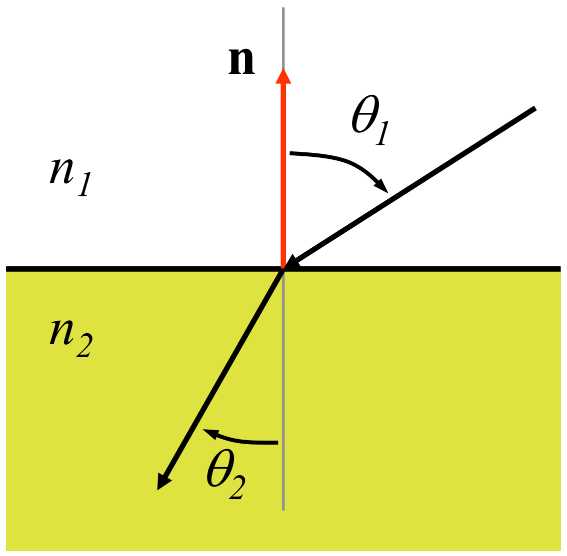
- To add transparency:
    - Add a term for light that's coming from within the object
    - These rays are refracted (bent) when passing through a boundary between two media with different refractive indices
    - When a ray hits a transparent surface fire a *transmission ray* into the object at the proper refracted angle
    - If the ray passes through the other side of the object then it bends again (the other way)



# Refraction

- Refraction:
  - The bending of light due to its different velocities through different materials
  - rays bend toward the normal when going from sparser to denser materials (e.g. air to water), away from normal in opposite case
- Refractive index:
  - Light travels at speed $c/n$ in a material of refractive index $n$
  - $c$ is the speed of light in a vacuum
  - $c$ varies with wavelength, hence rainbows and prisms
  - Use Snell's law $n_1 \sin\theta_1 = n_2 \sin\theta_2$ to derive refracted ray direction
    - note: ray dir. can be computed without trig functions (only sqrts)

| MATERIAL | INDEX OF REFRACTION |
|---|---|
| air/vacuum | 1 |
| water | 1.33 |
| glass | about 1.5 |
| diamond | 2.4 |



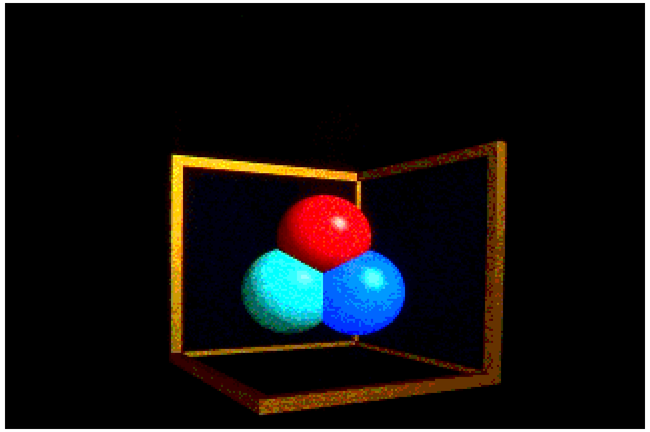
# From a Ray Caster to a Ray Tracer

```
Trace(ray)              // fire a ray, return RGB radiance
                           // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Shade(point, ray)           /* return radiance along ray */
    radiance = black;       /* initialize color vector */
    for each light source
        shadow_ray = calc_shadow_ray(point,light)
        if !in_shadow(shadow_ray,light)
            radiance += phong illumination(point,ray,light)
    if material is specularly reflective
        radiance += spec_reflectance *
            Trace(reflected_ray(point,ray)))
    if material is specularly transmissive
        radiance += spec_transmittance *
            Trace(refracted_ray(point,ray)))
    return radiance
```

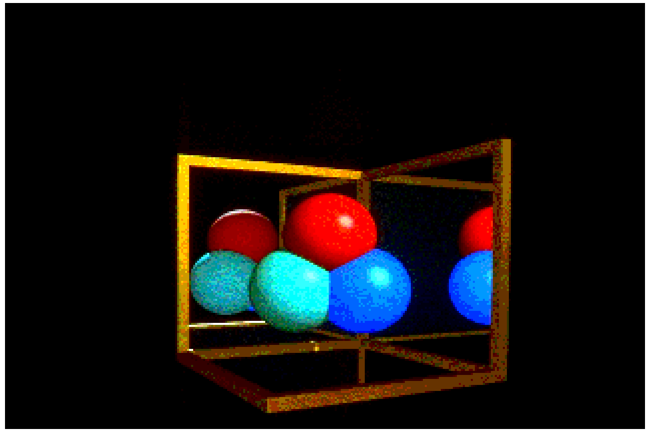
```
ray eye_ray;
eye_ray.level = 0;

reflected_ray(ray in):
        ray out;
        out.level = in.level++
        return out
```

# Ray Casting vs. Ray Tracing

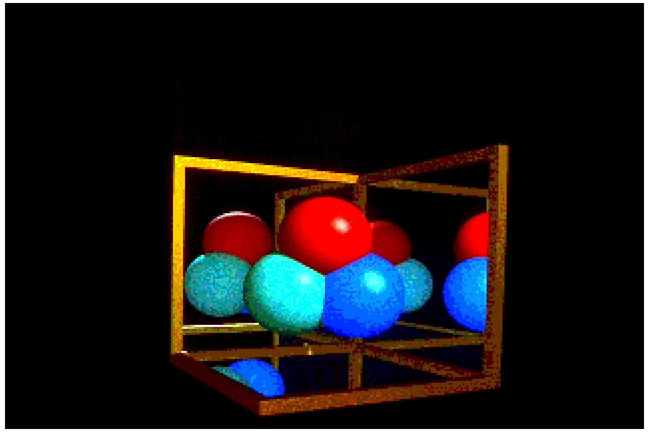
```
Trace(ray) // fire a ray, return RGB radiance of light traveling backward along it
    if ray.level > n return Background_Color;
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color
```



**Ray Casting -- 1 bounce**

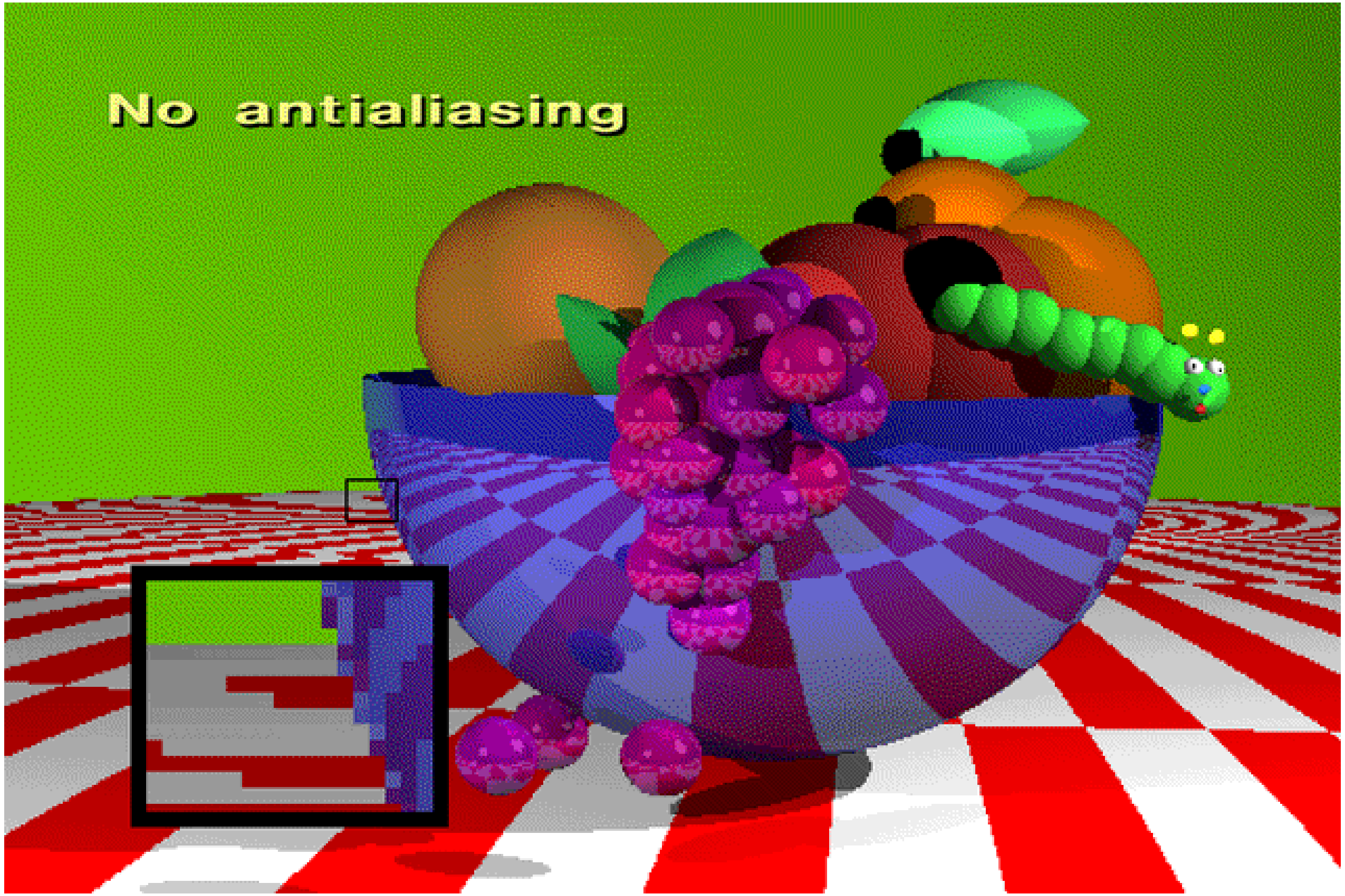


**Ray Tracing -- 2 bounce**



**Ray Tracing -- 3 bounce**

# Problem with Simple Ray Tracing

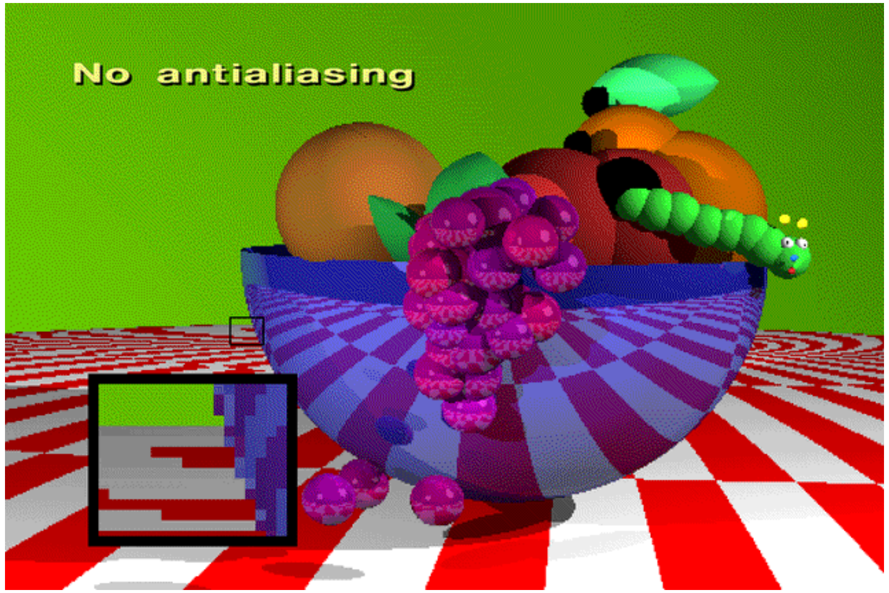


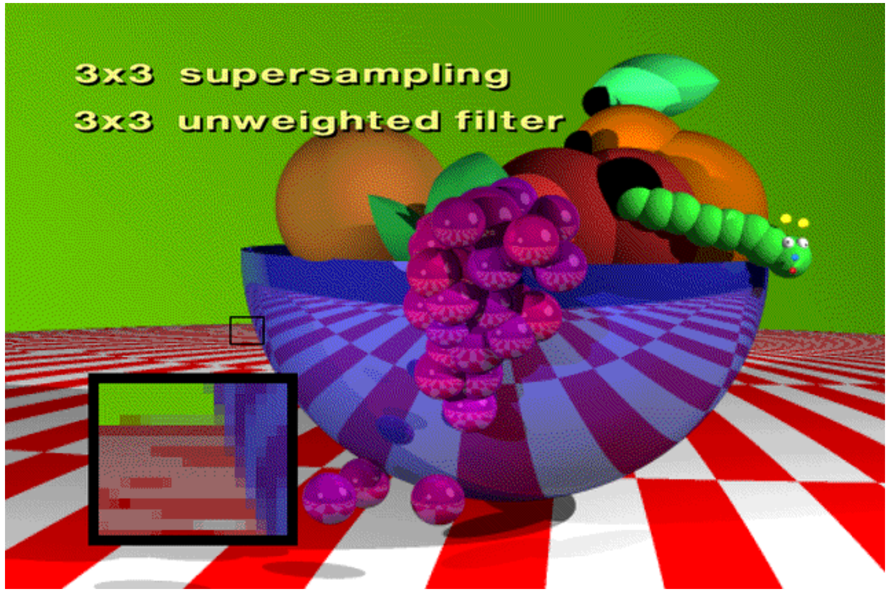
# Aliasing

- Ray tracing shoots one ray per pixel
- But a pixel represents an area; one ray samples only one point with the area; an area consists *infinite* number of points
  - These points may not all have the same color
  - This leads to *aliasing*
    - jaggies
    - moire patterns
- How do we fix this problem?
  - Recall antialiasing we talked earlier

# Antialiasing: Supersampling

- We talked about two antialiasing methods
  - Supersampling
  - Pre-filtering (MIP-mapping)
- Here we use supersampling
  - Fire more than one ray for each pixel (e.g., a 3x3 grid of rays)
  - Average the results using a filter (or some kind of filter)



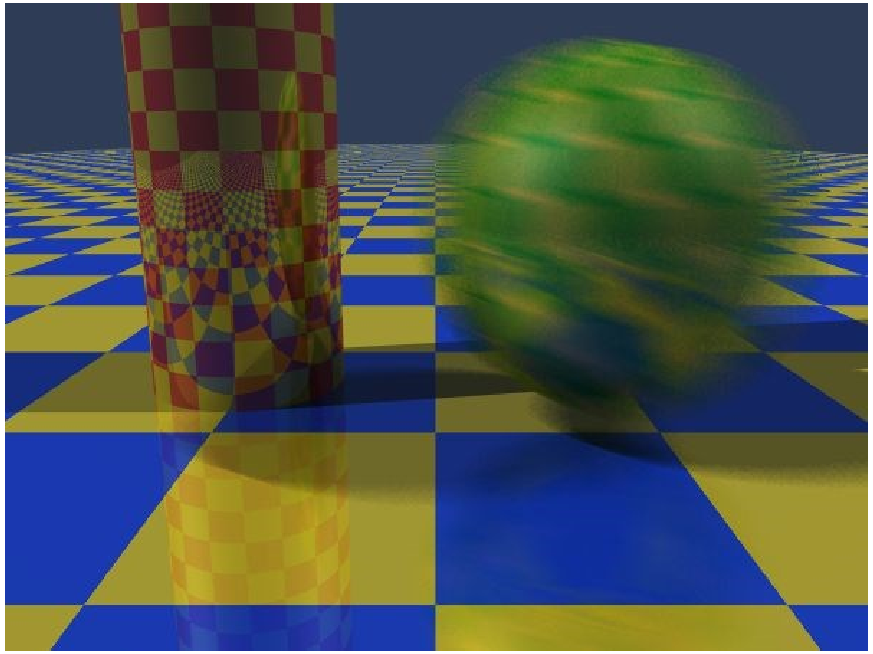
# Antialiasing: Adaptive Supersampling

- Supersampling can be done **adaptively**
  - divide pixel into 2x2 grid, trace 5 rays (4 at corners, 1 at center)
  - if the colors are similar then just use their average
  - otherwise recursively subdivide each cell of grid
  - keep going until each 2x2 grid is close to uniform or limit is reached
  - filter the result

- Behavior of adaptive supersampling
  - Areas with fairly constant appearance are sparsely sampled
  - Areas with lots of variability are heavily sampled

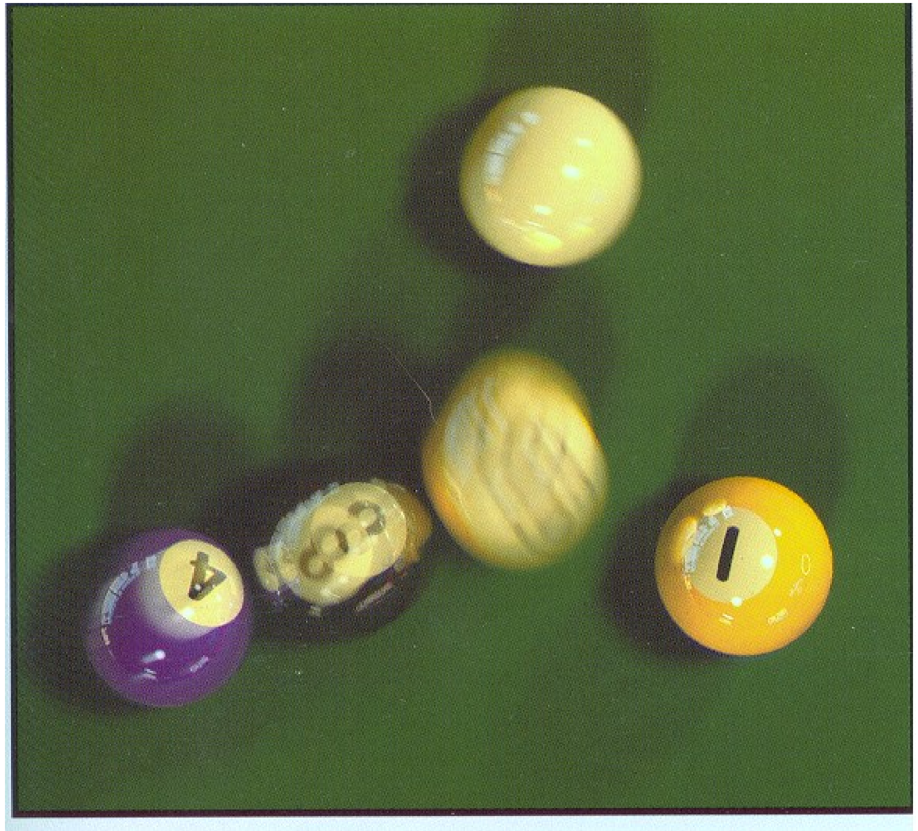
# Motion Blur

- Apply stochastic sampling to time as well as space
- Assign a time as well as an image position to each ray
- The result is still-frame motion blur and smooth animation
- This is an example of **distribution ray tracing**



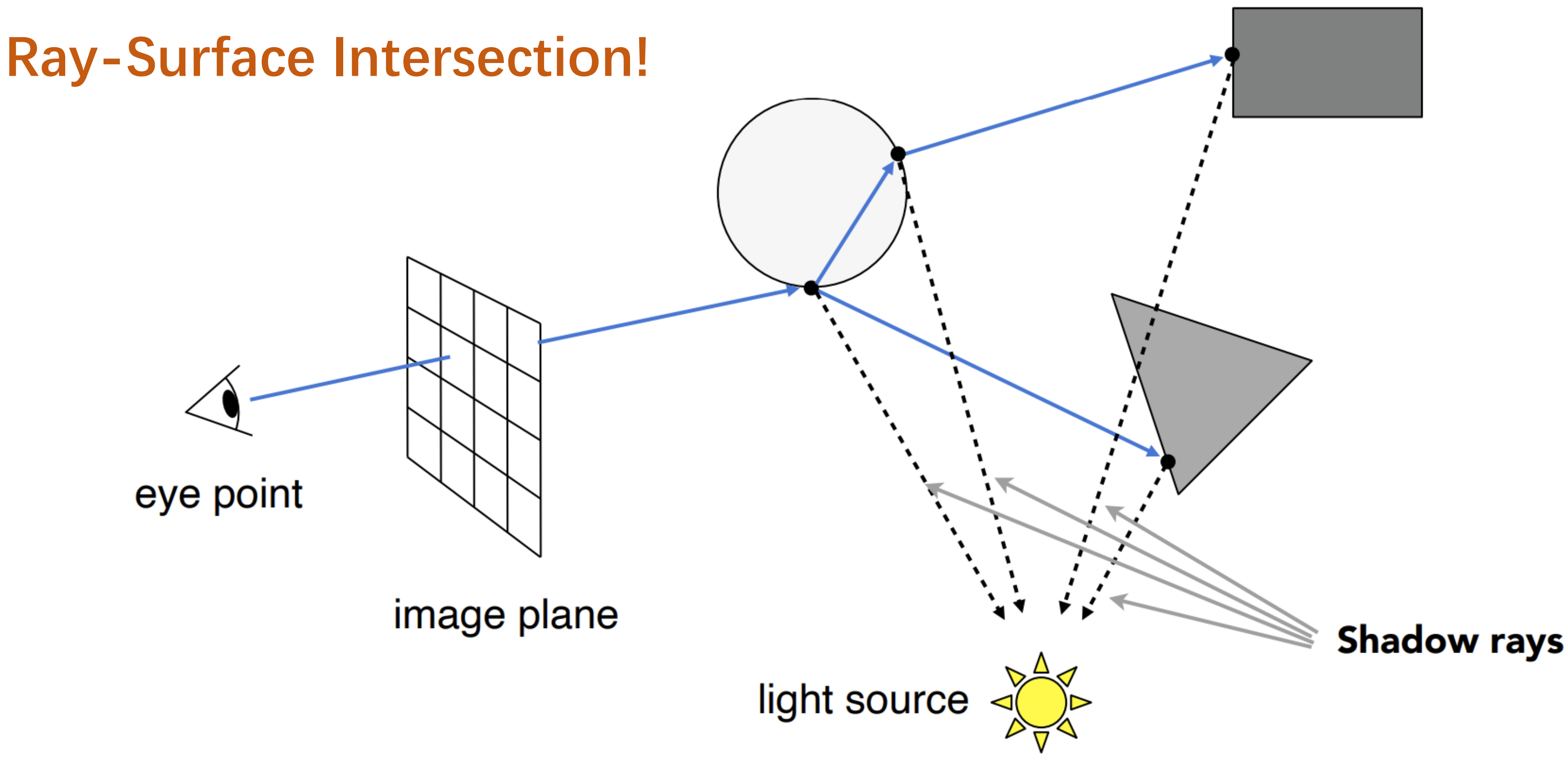
# Motion Blur: a classic example

- From Foley et. al. Plate III.16
- Rendered using distribution ray tracing at 4096x3550 pixels, 16 samples per pixel.
- Note motion-blurred reflections and shadows with penumbrae cast by extended light sources.

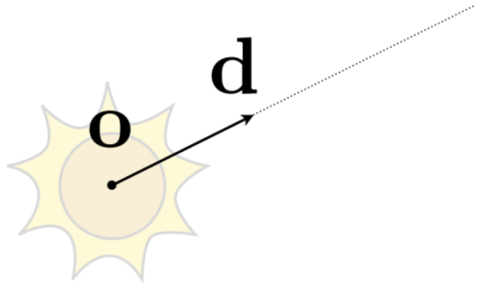


# Ray Tracing Acceleration

# Whitted-Style Ray Tracing

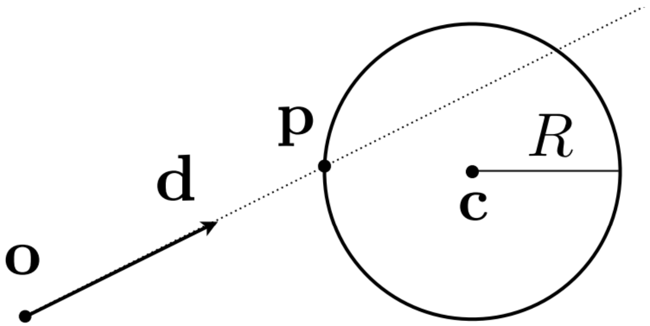


# Ray-Surface Intersection




- $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$

Sphere




Solve $(\mathbf{r}(t) - c)^2 = R^2$

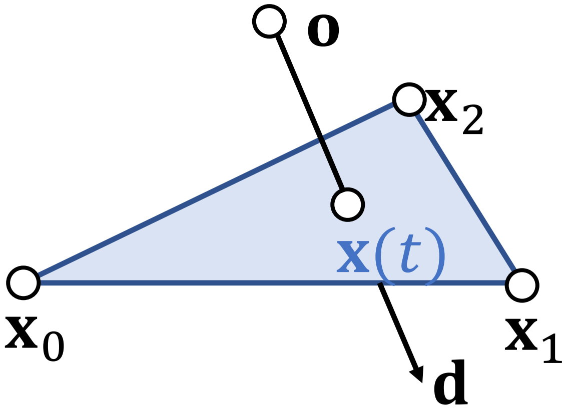
$$a = \mathbf{d} \cdot \mathbf{d}$$

$$b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}$$

$$c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Triangle




Solve $(\mathbf{r}(t) - \mathbf{x}_0) \cdot (\mathbf{x}_{10} \times \mathbf{x}_{20}) = 0$

$$t = \frac{\mathbf{x}_{0\boldsymbol{o}} \cdot \mathbf{x}_{10} \times \mathbf{x}_{20}}{\boldsymbol{d} \cdot \mathbf{x}_{10} \times \mathbf{x}_{20}}$$

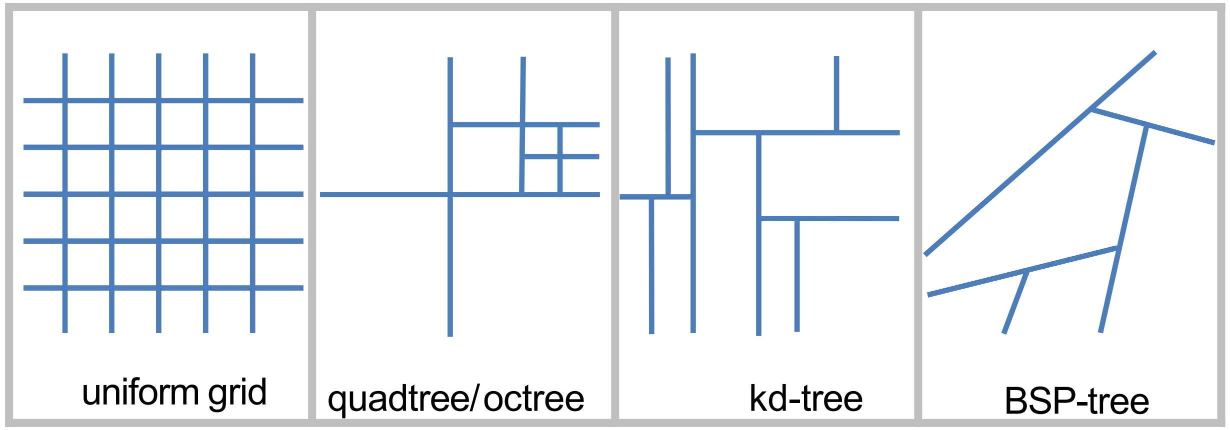
If $t > 0$ and $\mathbf{x}(t)$ *inside*:
return Intersection point, $\mathbf{x}(t)$

# Ray Tracing – Performance Challenges

# Ray Tracing – Performance Challenges

- Checking intersections with everything!

  **Spatial partitioning**

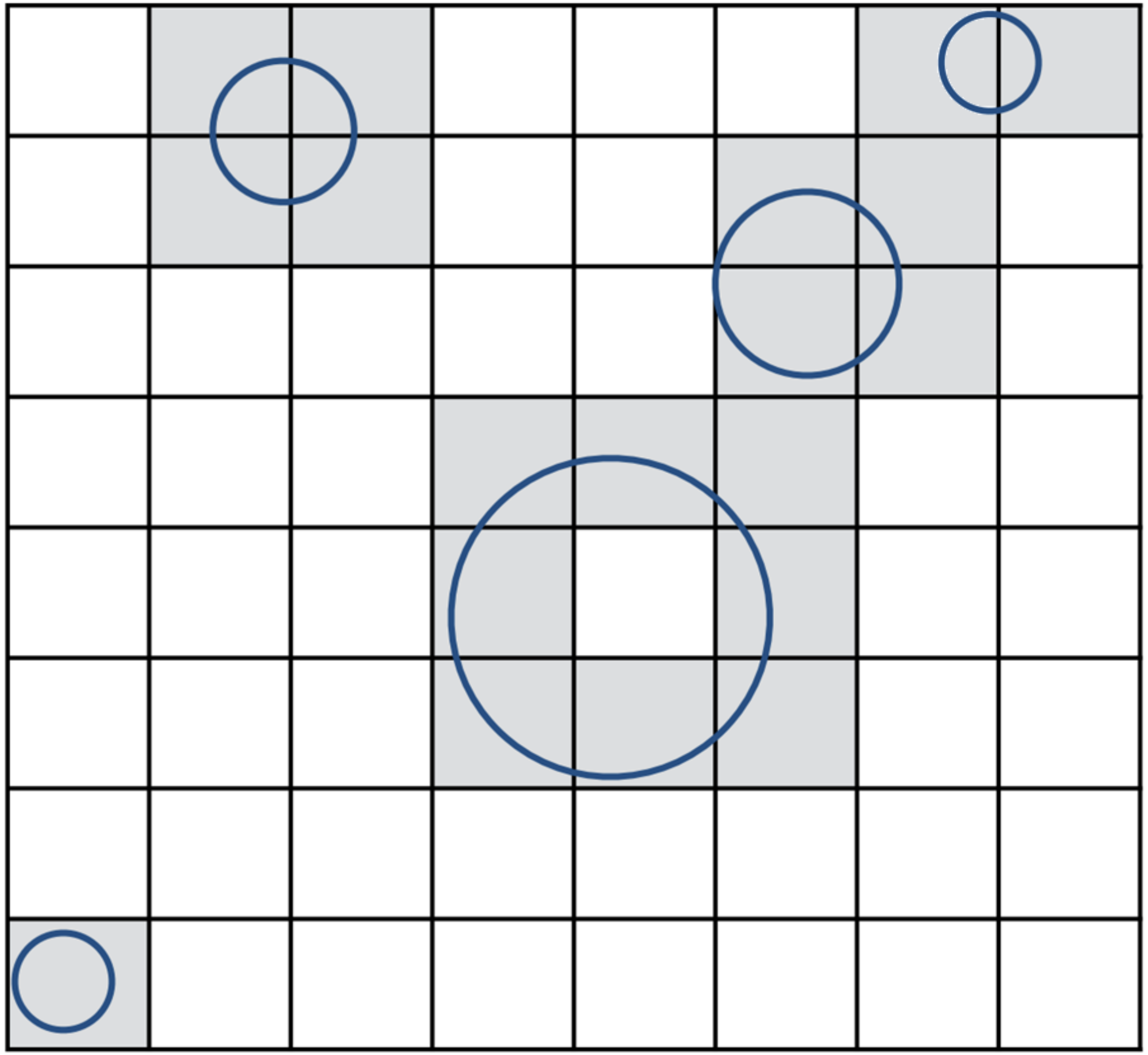



- Checking intersections with complex geometry!

  **Bounding Volumes**

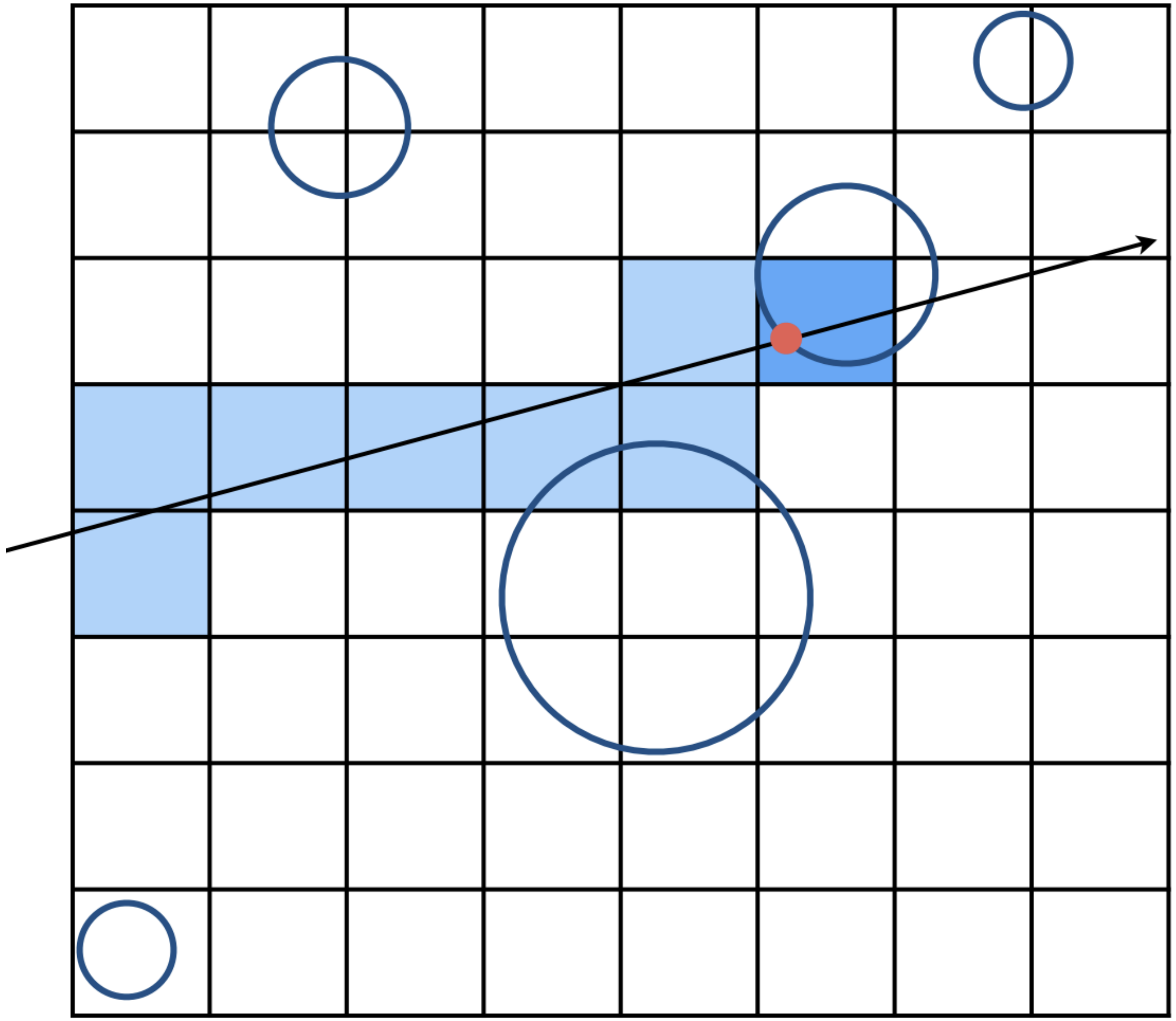
# Grid Acceleration

- 1. Find bounding box
- 2. Create grid
- 3. Store each object in overlapping cells



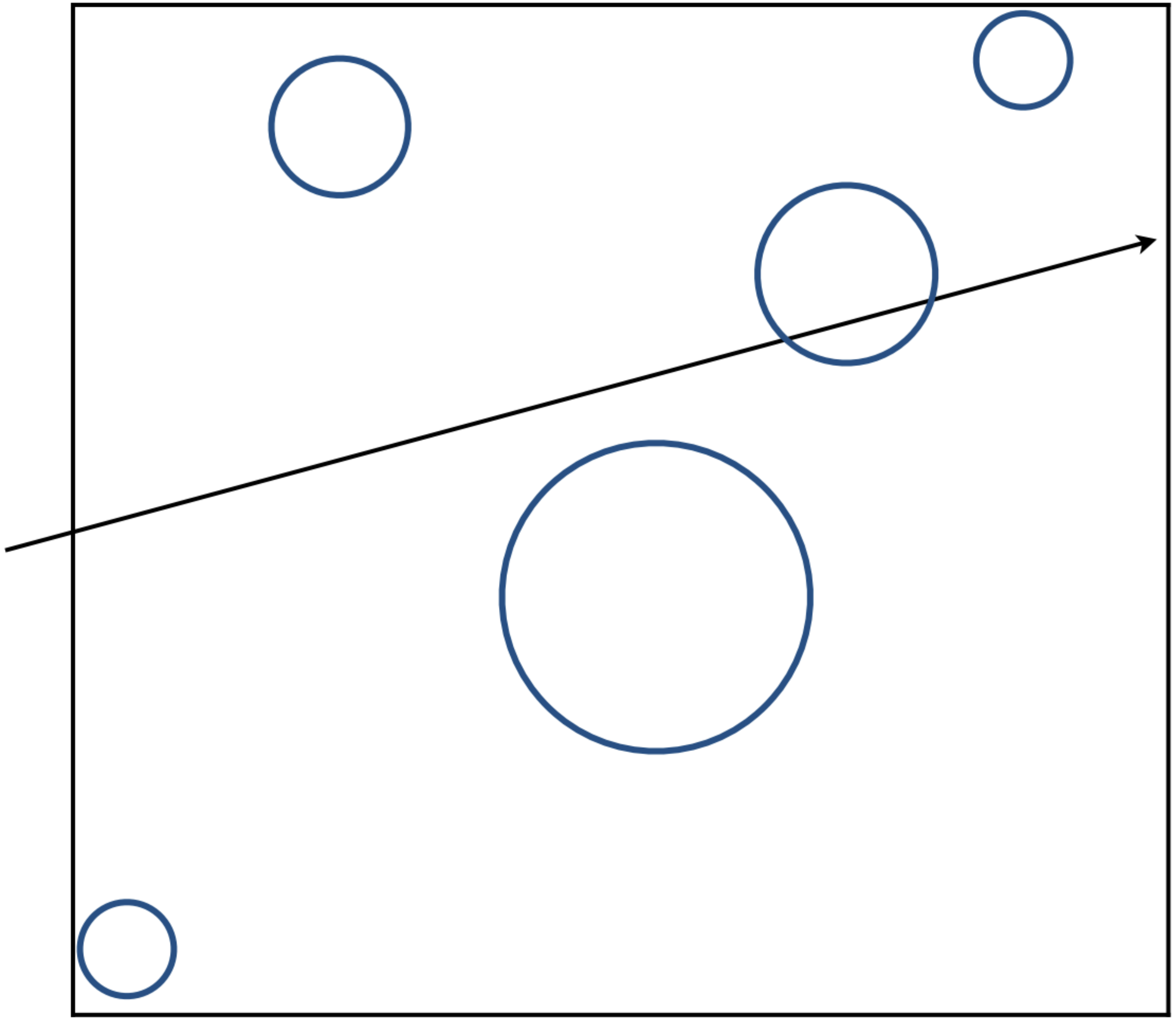
# Grid Acceleration

- Step through grid in ray traversal order
- For each grid cell :
  - Test intersection with all objects stored at that cell



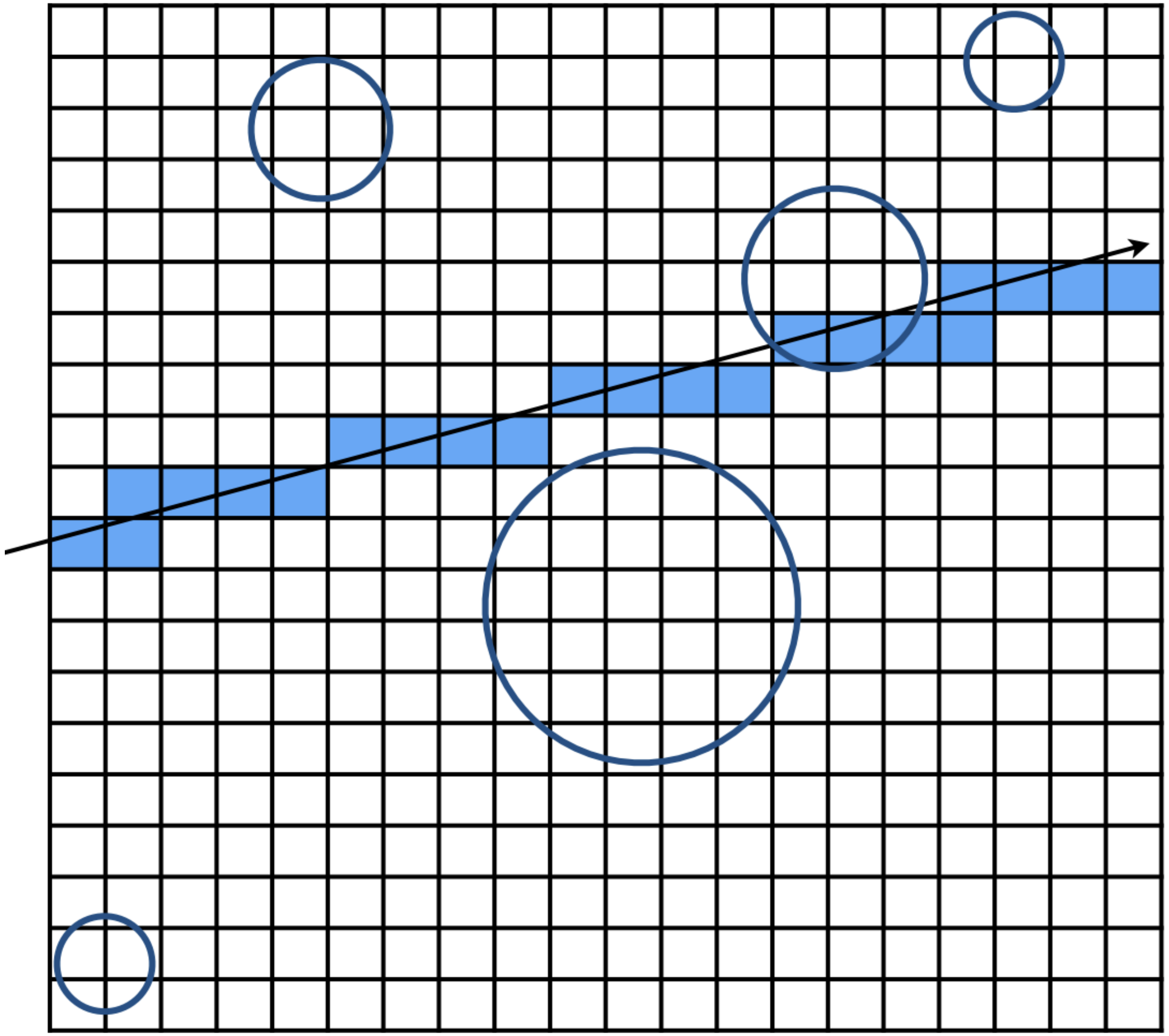
# Grid Resolution?

- One cell
  - No speedup



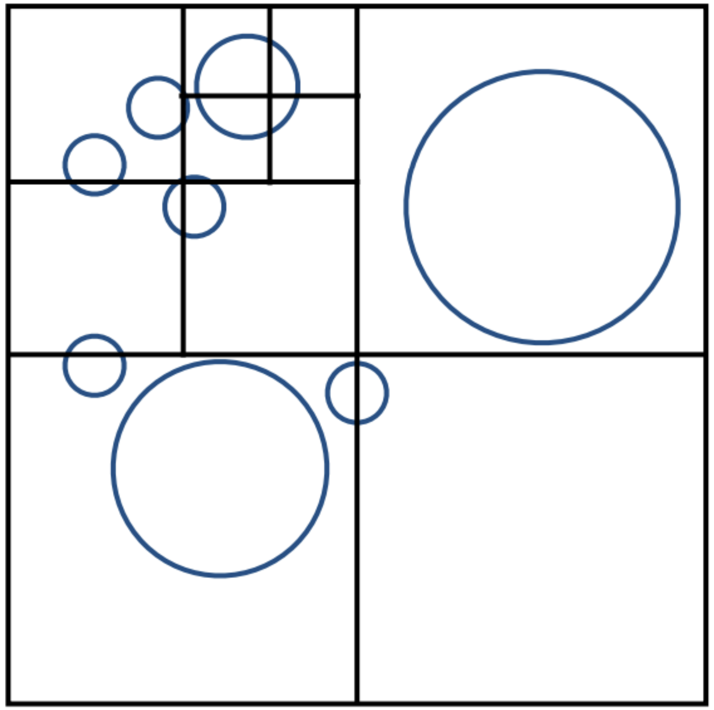
# Grid Resolution?

- Too many cells
  - Inefficiency due to extraneous grid traversal

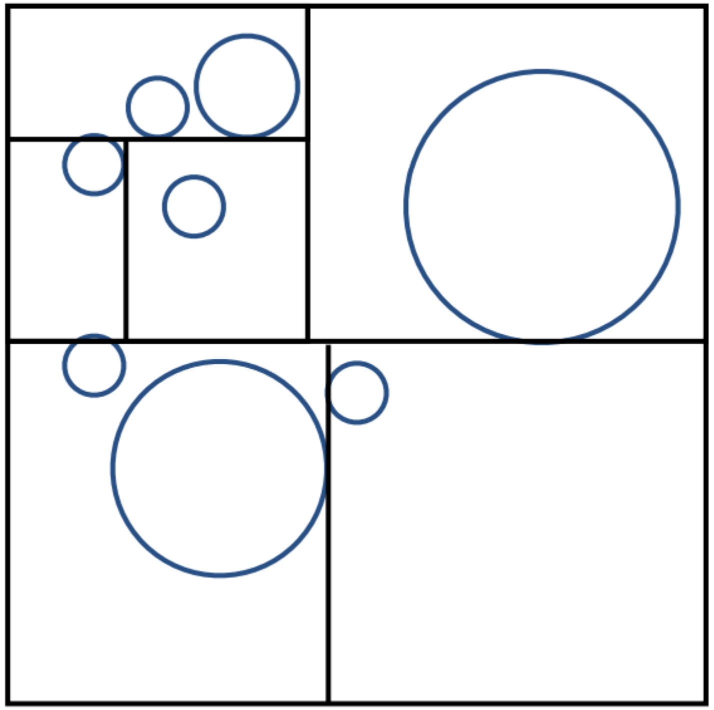


# Ray Tracing – Grid Resolution?

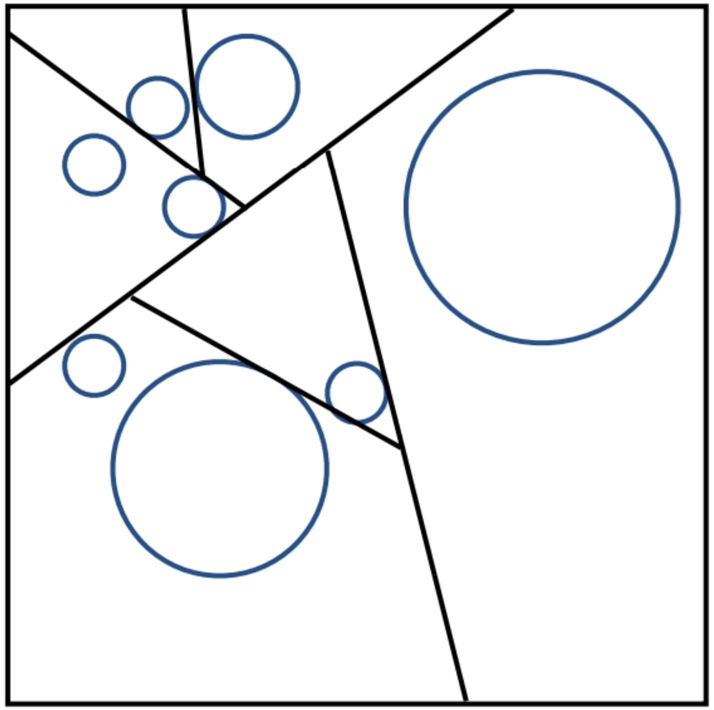
# Spatial Partitioning



Oct-Tree

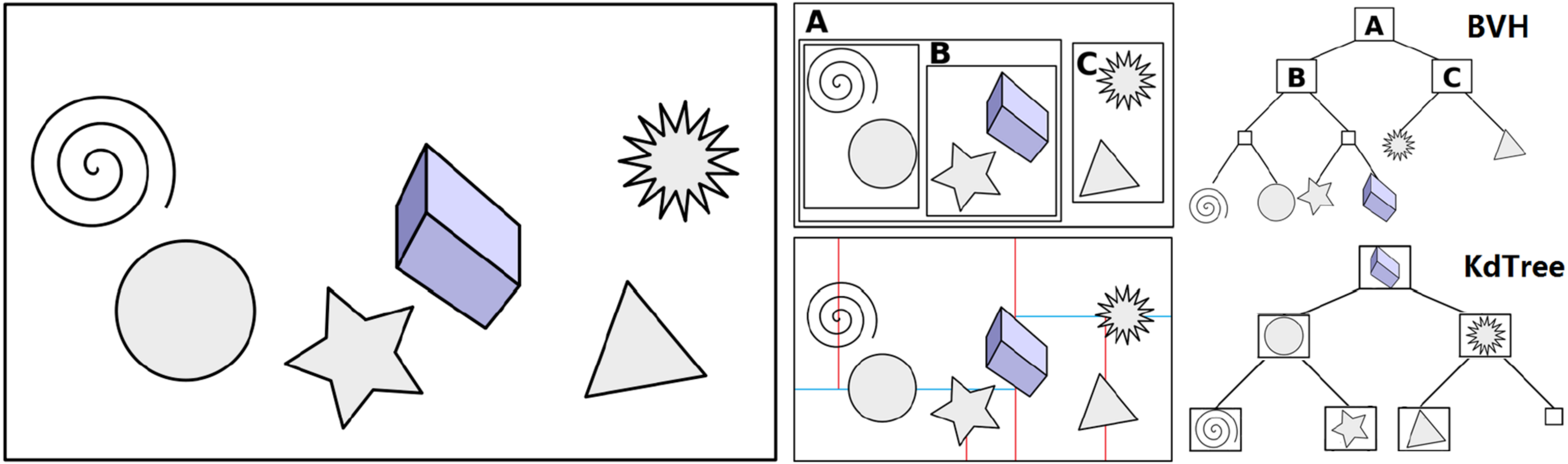


**KD-Tree**



BSP-Tree

# Spatial Partitioning

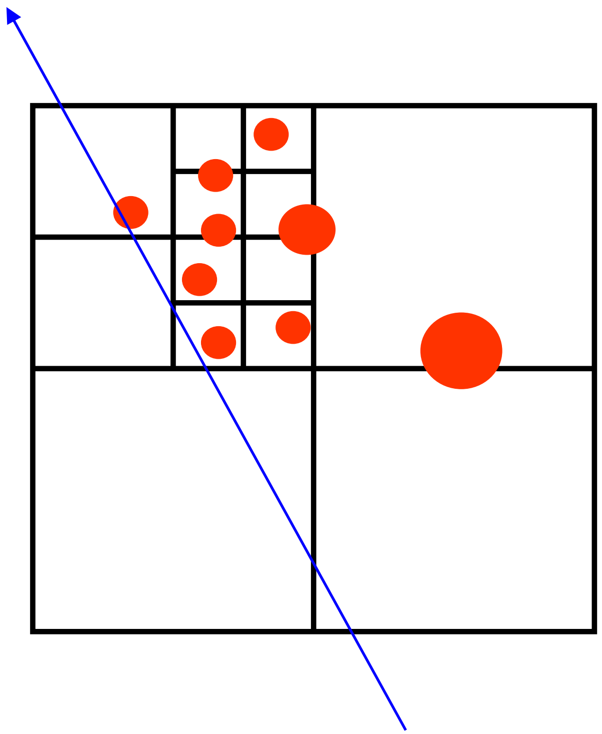



General task:

- 1. Build the tree
- 2. For a given point, travel the root–to-leaf path and test intersections

# Octrees

- Quadtree is the 2-D generalization of binary tree
  - node (cell) is a square
  - recursively split into four equal sub-squares
  - stop when leaves get “simple enough”



# Octrees

- Octree is the 3-D generalization of quadtree
  - node (cell) is a cube, recursively split into eight equal sub-cubes
  - for ray tracing:
    - stop splitting when the number of objects intersecting the cell gets “small enough” or the tree depth exceeds a limit
    - internal nodes store pointers to children, leaves store list of surfaces
  - more expensive to traverse than a grid
  - but an octree adapts to nonhomogeneous, clumpy scenes better

```
trace(cell, ray) {                    // returns object hit or NONE
    if cell is leaf, return closest (objects_in_cell(cell))
    for child cells pierced by ray, in order          // 1 to 4 of these
        obj = trace(child, ray)
        if obj!=NONE return obj
    return NONE
}
```

# Which Data Structure is Best for Ray Tracing?

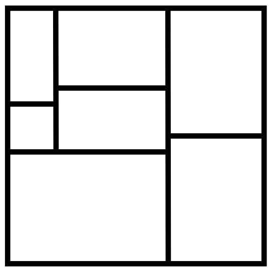
- Grids are easy to implement, but they're memory hogs (and slow) for nonhomogeneous scenes, i.e. most scenes
- Octrees are pretty good, but not as fast as grids for some scenes
- Nested grids seem to be the fastest on static scenes

- If scene is dynamic, the cost of regenerating or updating the data structure may become an issue
- In such cases, hierarchical bounding volumes may be best
- Hierarchical bounding volumes easy to implement if your model is naturally hierarchical (e.g. human), otherwise not

- For other visibility algorithms:
  - BSP trees useful for Painter's algorithm...

# k-d Trees

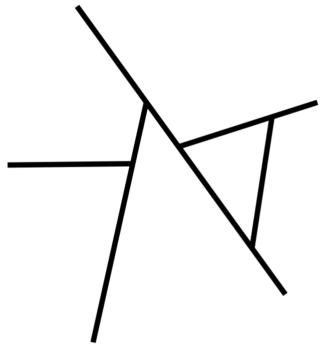
- Relax the rules for quadtrees and octrees:



- first variant: *k-dimensional (k-d) tree*
  - don’t always split at midpoint
  - split only one dimension at a time (i.e. *x* or *y* or *z*)
  - useful for clustering and choosing colormaps for color image quantization

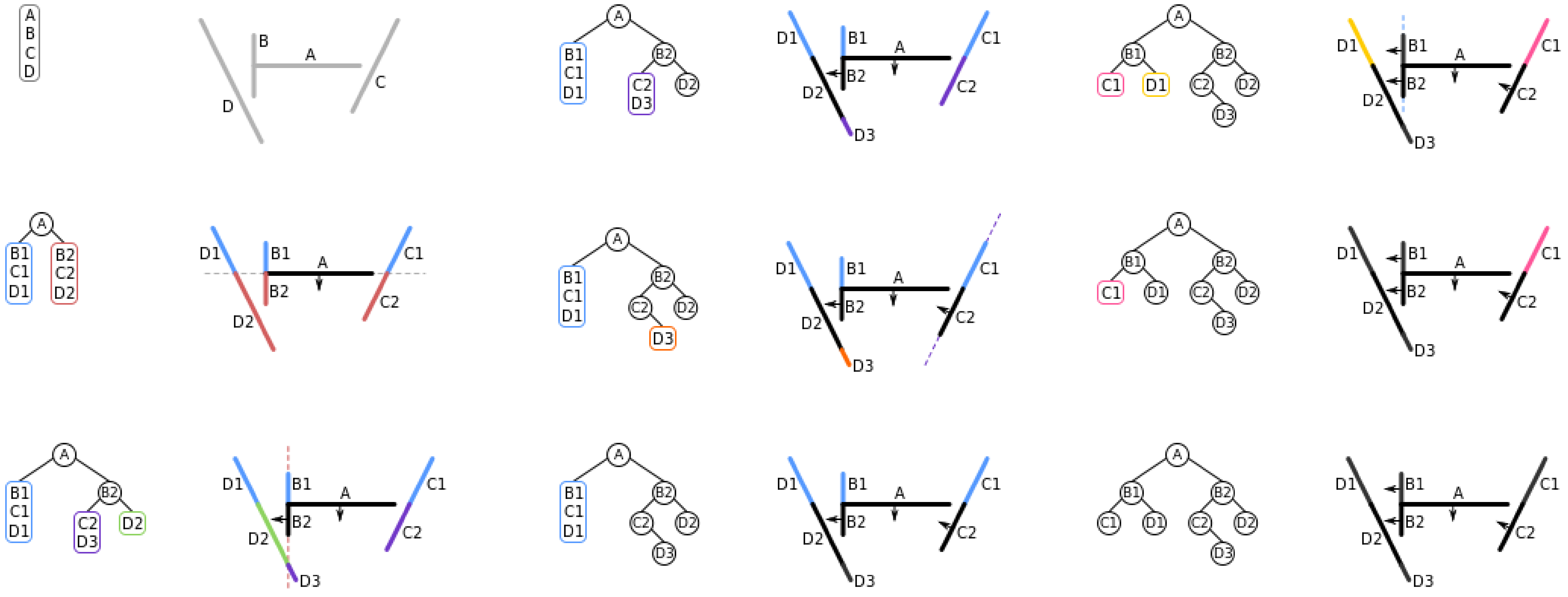
# BSP Trees

- Relax the rules for quadtrees and octrees:



- second variant: *binary space partitioning (BSP) tree*
  - permit splits with any line
  - in general, split *k* dimensional space with *k*-1 dimensional hyperplane
    - 2-D space split with lines (most of our examples)
    - 3-D space split with planes
    - each node corresponds to a (potentially unbounded) convex polyhedron
  - useful for Painter’s algorithm

# Building a BSP Tree



https://en.wikipedia.org/wiki/Binary_space_partitioning

# Building a Good Tree - the tricky part

- A naïve partitioning of *n* polygons will yield $O(n^3)$ polygons!

- Algorithms exist to find partitionings that produce $O(n^2)$.
  - For example, try all remaining polygons and add the one which causes the fewest splits (greedy algorithm!)
  - Fewer splits -> larger polygons -> better polygon fill efficiency

- Also, we want a balanced tree.
  - More important for ray casting than scan conversion.

- These goals conflict.

- *note: in the examples we've shown, the geometric objects being stored are planar, and we split using the planes of these objects, but that needn't be so – could theoretically split with any plane*

# Uses for Binary Space Partitioning (BSP) Trees

- Painter’s algorithm rendering
  - good for
    - static 3-D scenes with moving viewpoint (flight simulators)
    - architectural scenes with a small number of polygons (DOOM)
    - if you don’t have z-buffer hardware
  - Add a few monsters and such after the environment is drawn
- Ray tracing
- Solid modeling with polyhedra

- History:
  - BSP trees first used by Naylor, Fuchs, et al. for Painter’s algorithm ~1980
  - theoreticians scoffed at their worst-case performance
  - considered unpromising
  - revived by John Carmack, author of Quake, and the PC game community
    - out of necessity: no z-buffer hardware for PC’s at the time