

Introduction to Computer Graphics

Toshiya Hachisuka

Last Time

- Shading models
 - BRDF
 - Lambertian
 - Specular
- Simple lighting calculation
- Tone mapping

Today

- Acceleration data structure
 - How to handle lots of objects
- Light transport simulation
 - Rendering equation

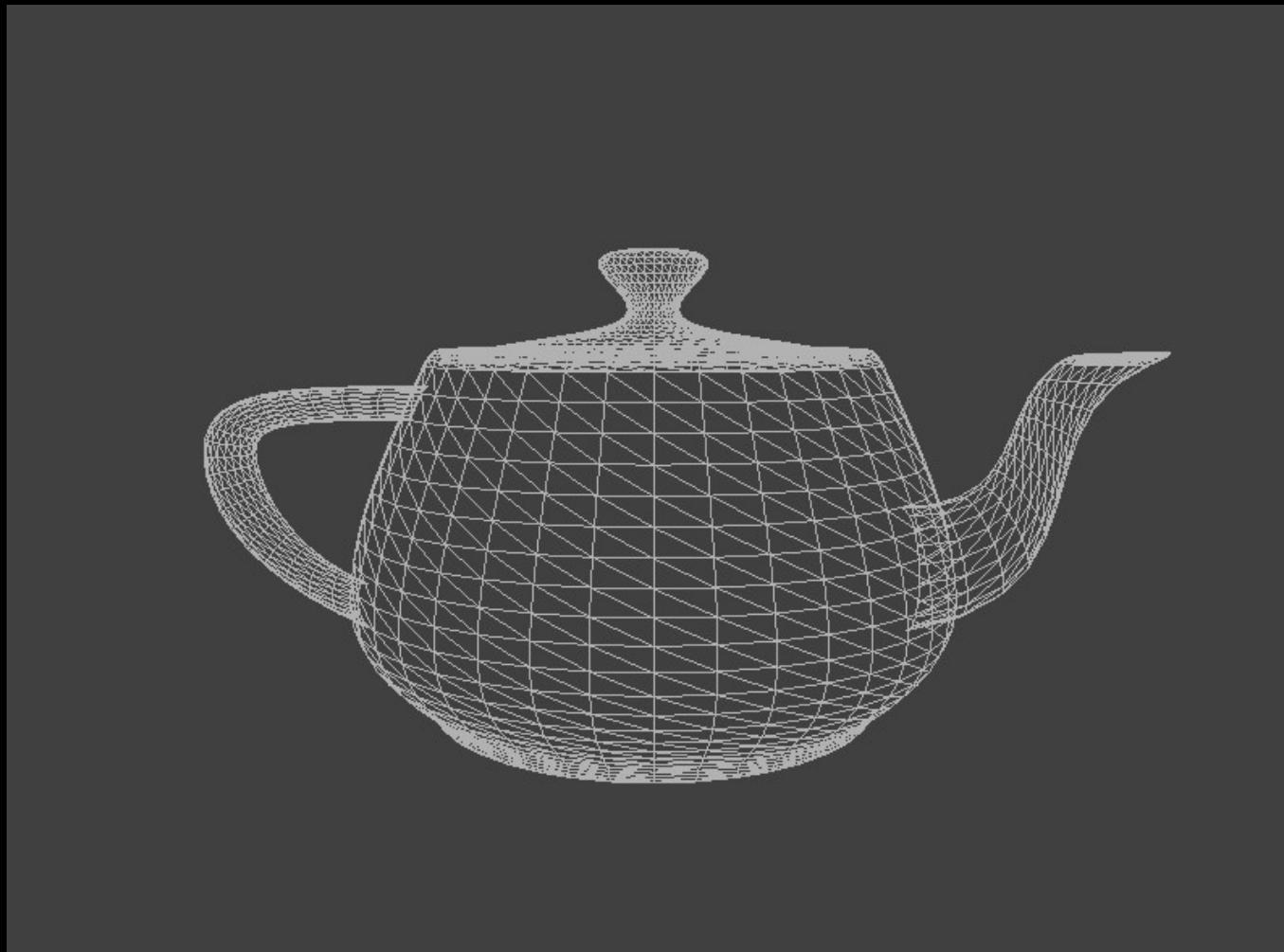
Cost

```
for all pixels {  
    ray = generate_camera_ray( pixel )  
    for all objects {  
        hit = intersect( ray, object )  
        if "hit" is closer than "first_hit" {first_hit = hit}  
    }  
    pixel = shade( first_hit )  
}
```

Cost

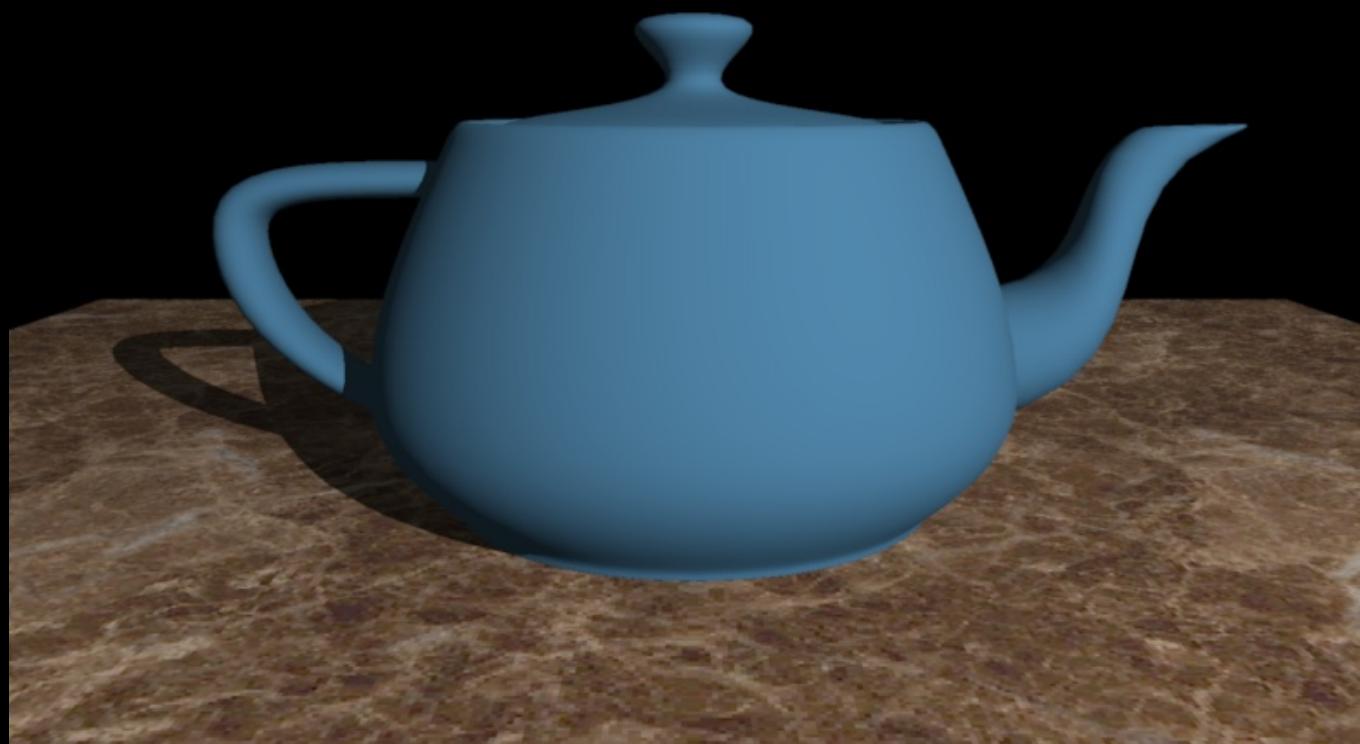
- Number of objects × Number of rays
- Example:
 - 1000 × 1000 image resolution
 - 1000 objects

Many Objects (Triangles)



The teapot has 6320 triangles

Many Objects (Triangles)

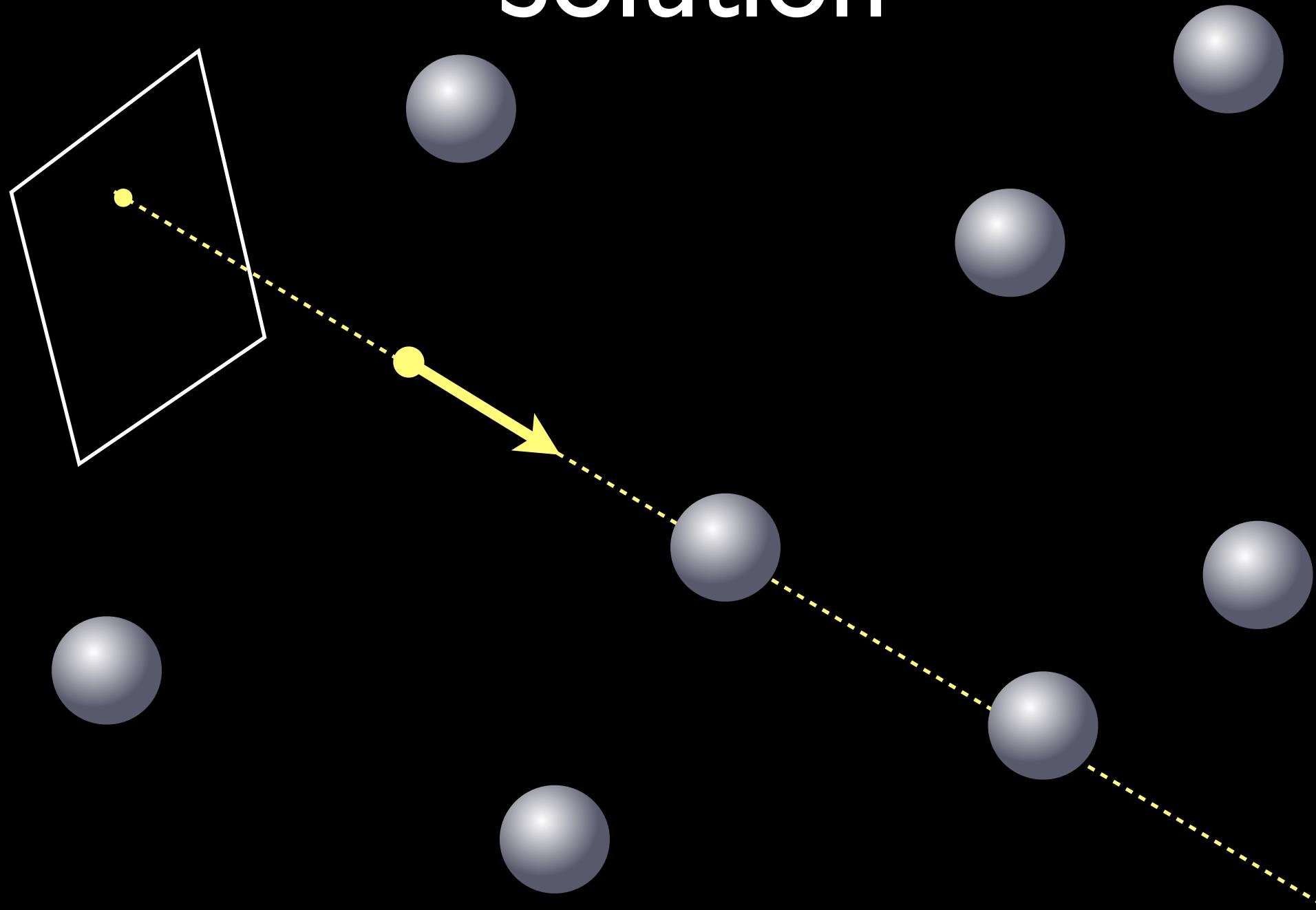


The teapot has 6320 triangles

Solution

- Acceleration data structure
 - Reorganize the list of objects
 - Don't touch every single object per ray

Solution



Solution

```
for all pixels {  
    ray = generate_camera_ray( pixel )  
    for all objects {  
        hit = intersect( ray, object )  
        if "hit" is closer than "first_hit" {first_hit = hit}  
    }  
    pixel = shade( first_hit )  
}
```

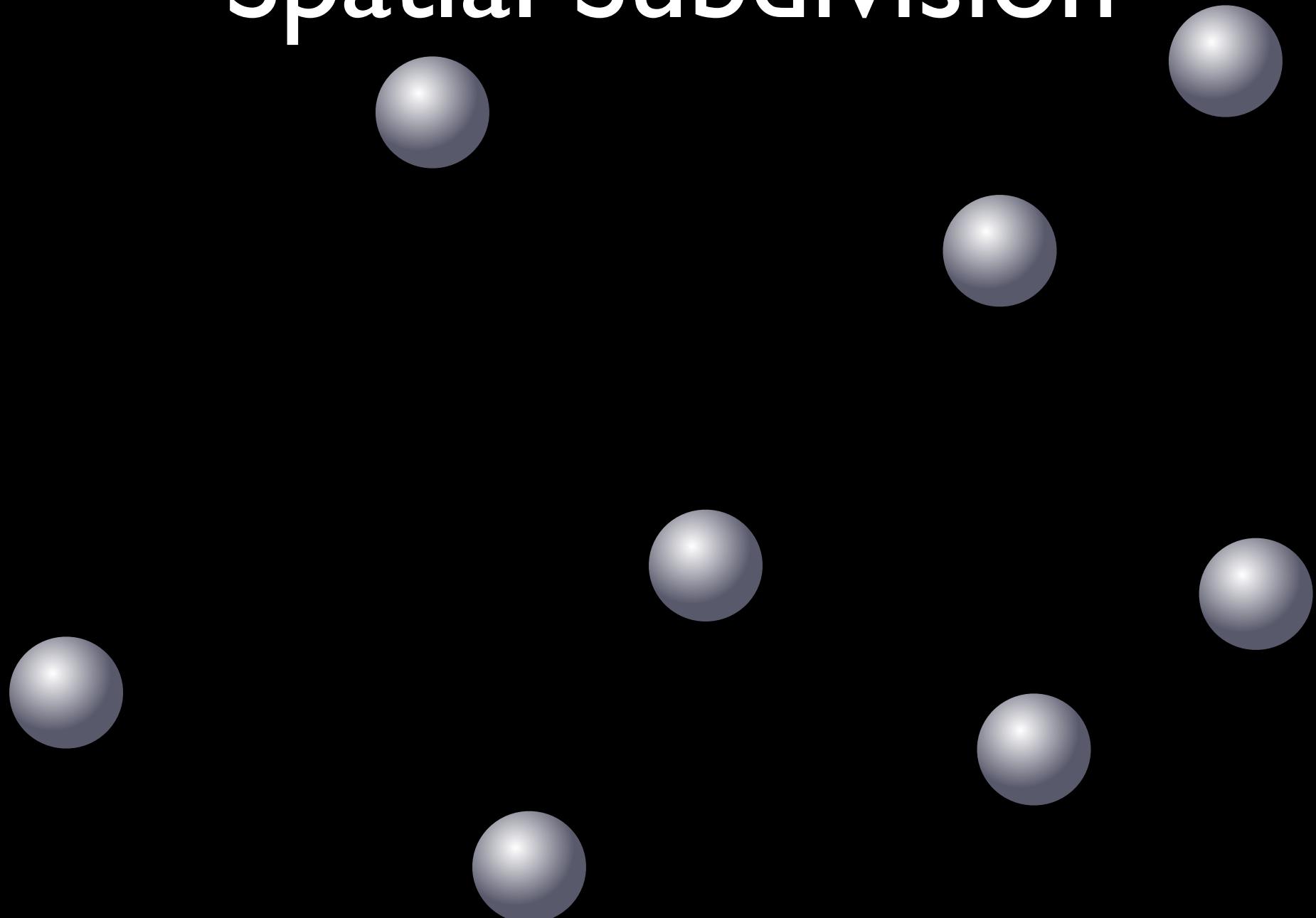
Solution

```
for all pixels {  
    ray = generate_camera_ray( pixel )  
    first_hit = traverse( ray, accel_data_struct )  
    pixel = shade( first_hit )  
}
```

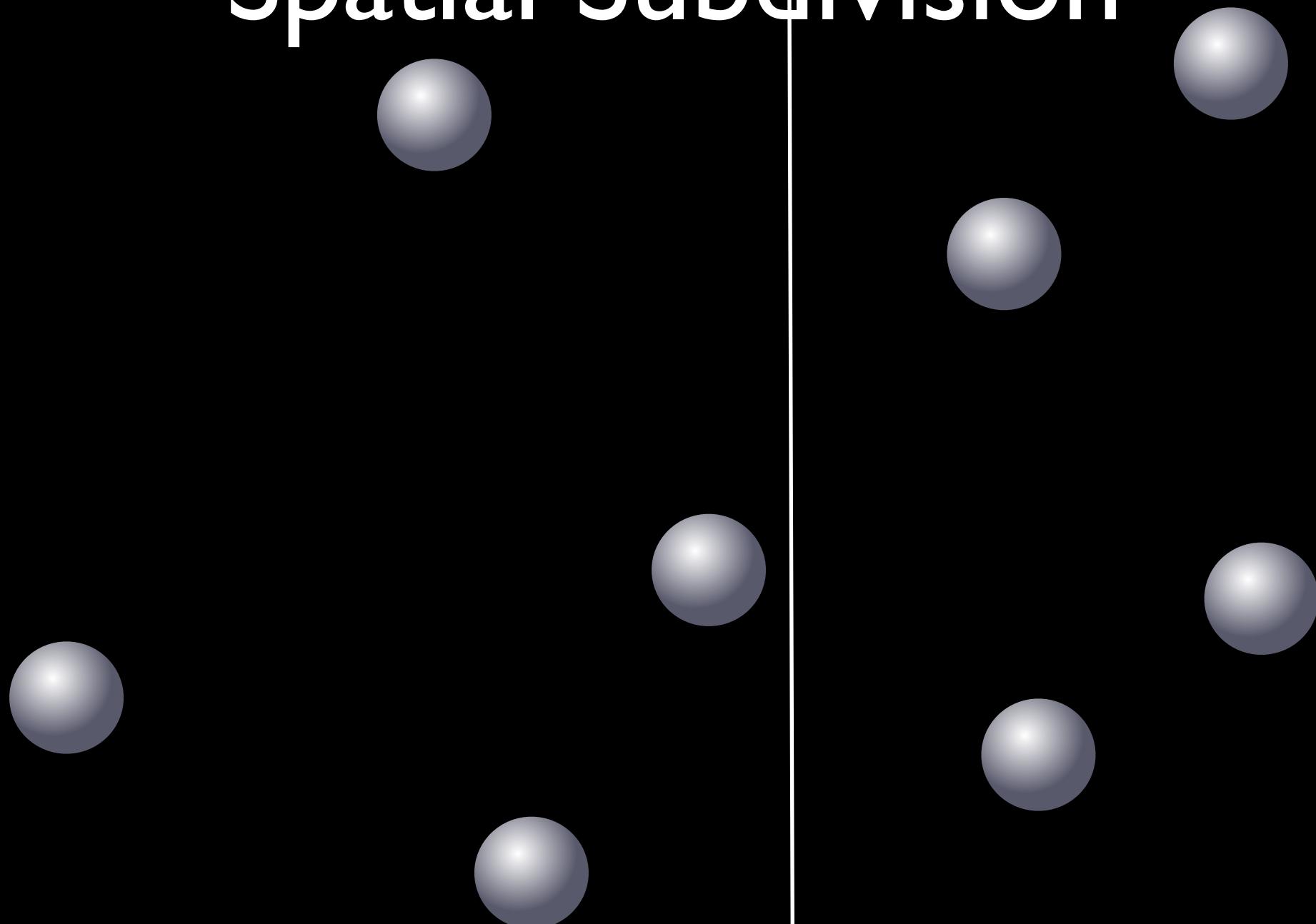
Two Basic Approaches

- Reorganize the space - spatial subdivision
- Reorganize the list - object subdivision

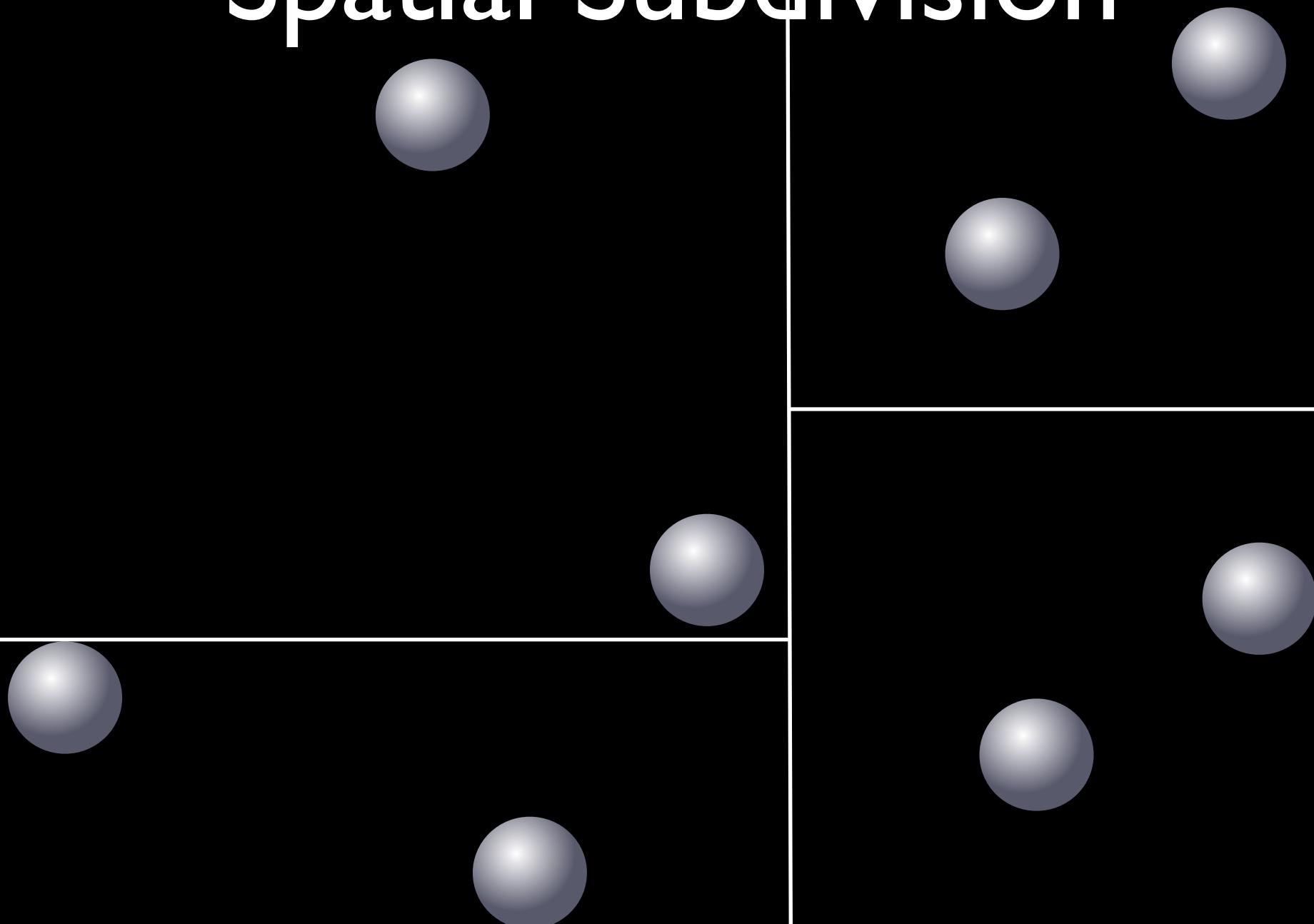
Spatial Subdivision



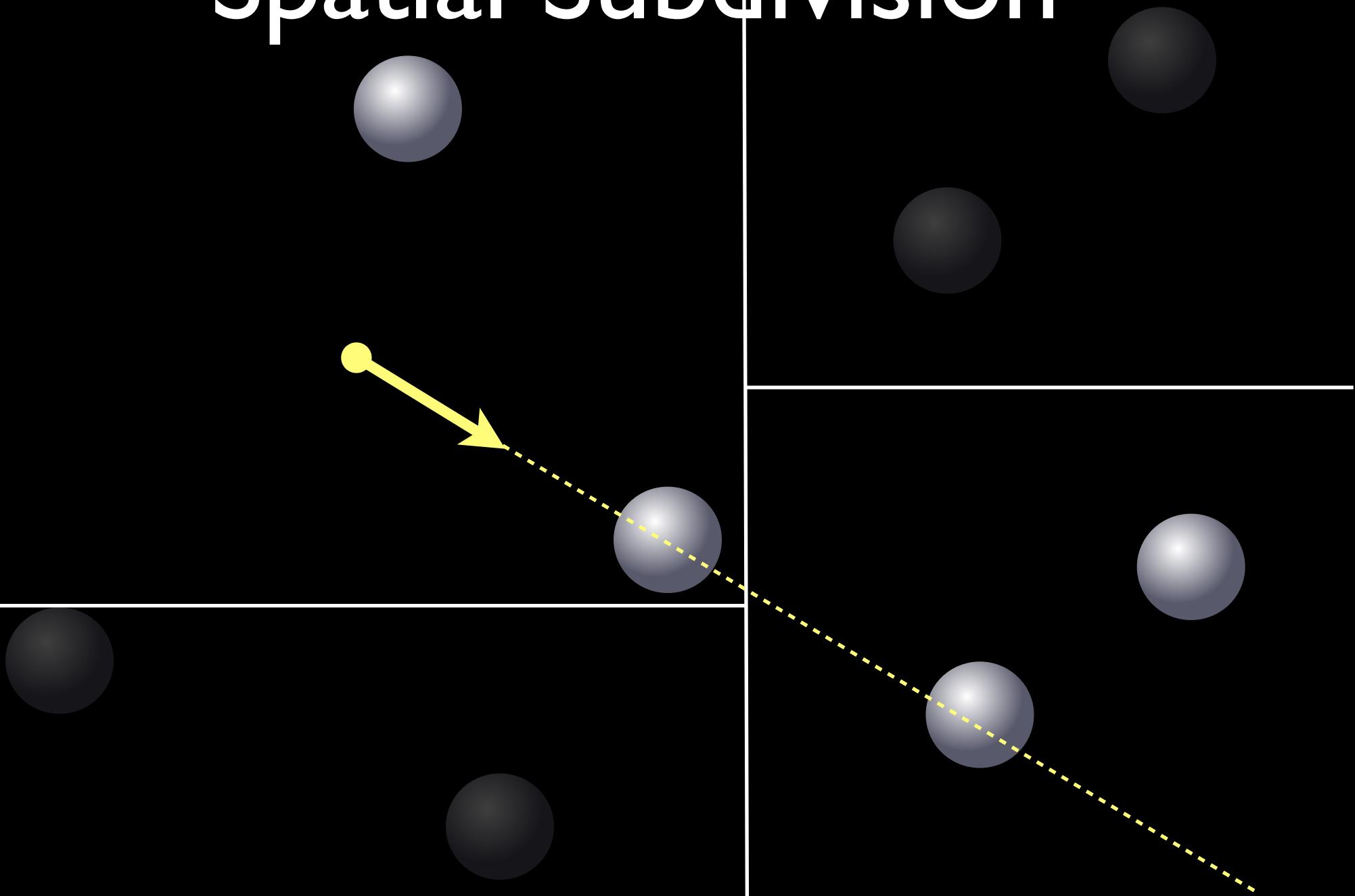
Spatial Subdivision



Spatial Subdivision



Spatial Subdivision



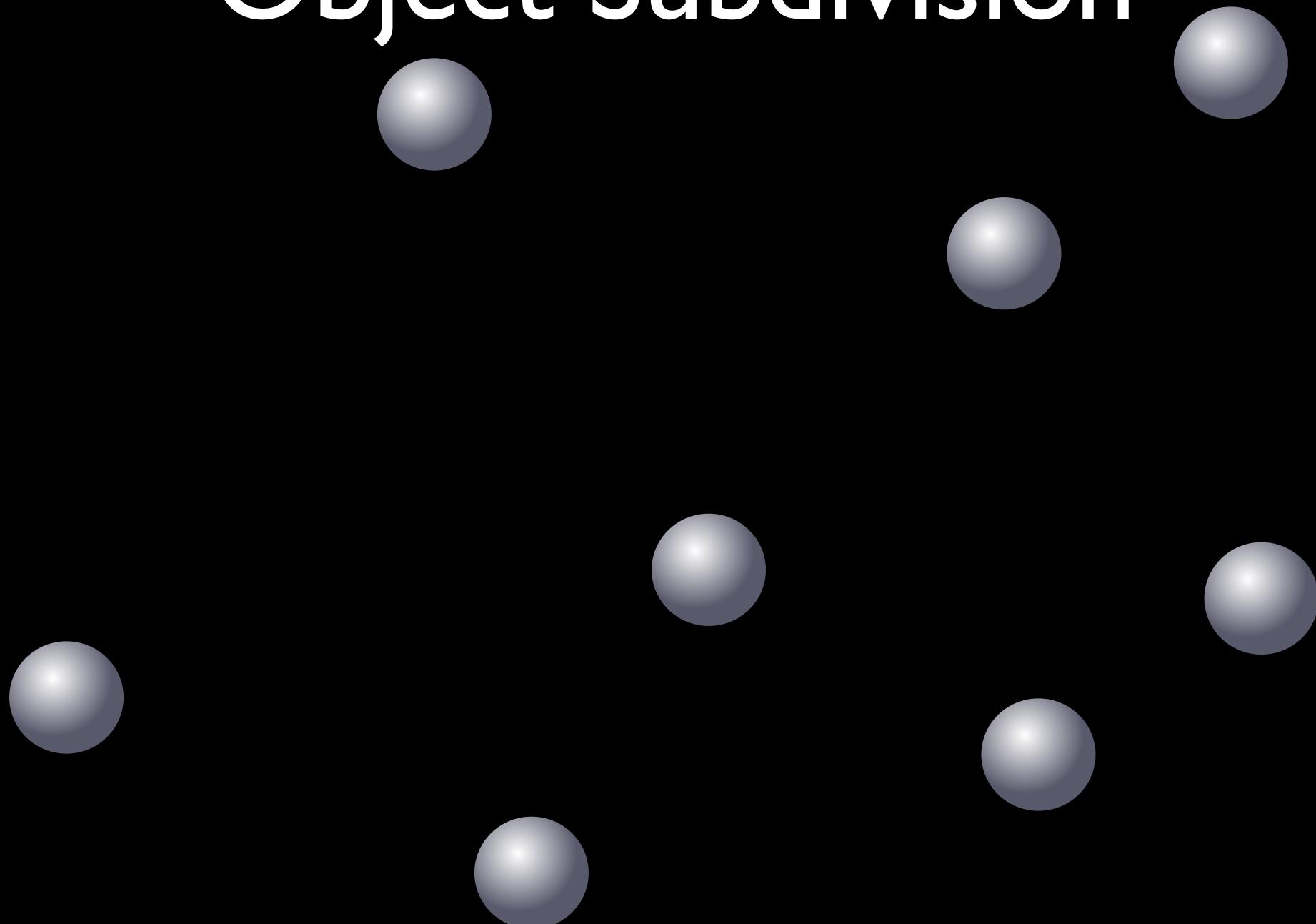
Spatial Subdivision

- Hierarchically subdivide **the space**
 - kD-tree (axis-aligned split)
 - BSP-tree (non-axis-aligned split)
- Each node stores pointers to **the subspaces**
- Leaf node stores the list of objects that overlap with the subspace

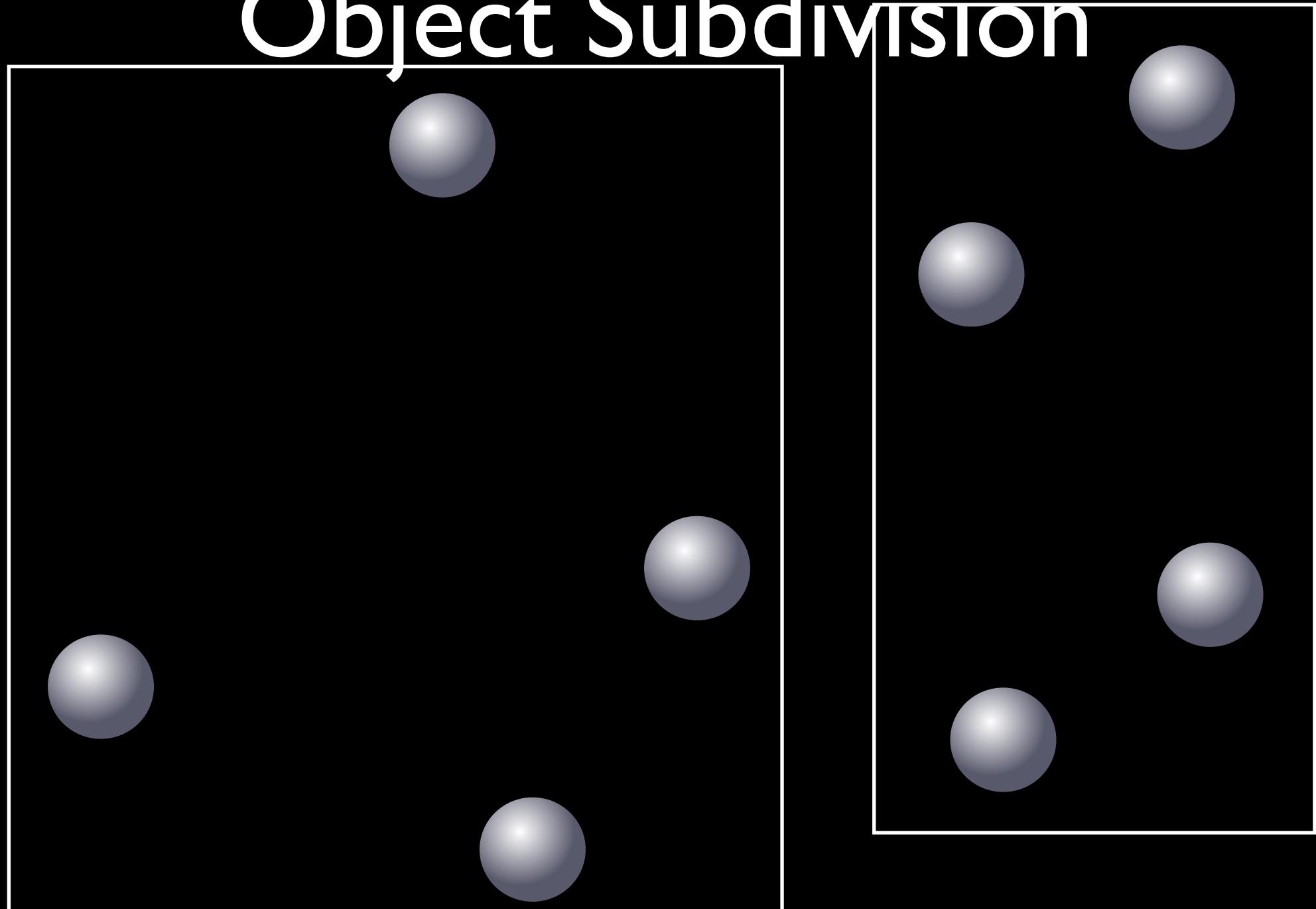
Spatial Subdivision

```
node subdivision( objects, space ) {  
    if ( space is small enough ) {  
        return make_leaf( objects )  
    }  
    space.split ( &subspace1, &subspace2 )  
    for all objects {  
        if (overlap(subspace1, object)) subspace1.add(object)  
        if (overlap(subspace2, object)) subspace2.add(object)  
    }  
    tree.add_node( subdivision( objects1, subspace1 ) )  
    tree.add_node( subdivision( objects2, subspace2 ) )  
}
```

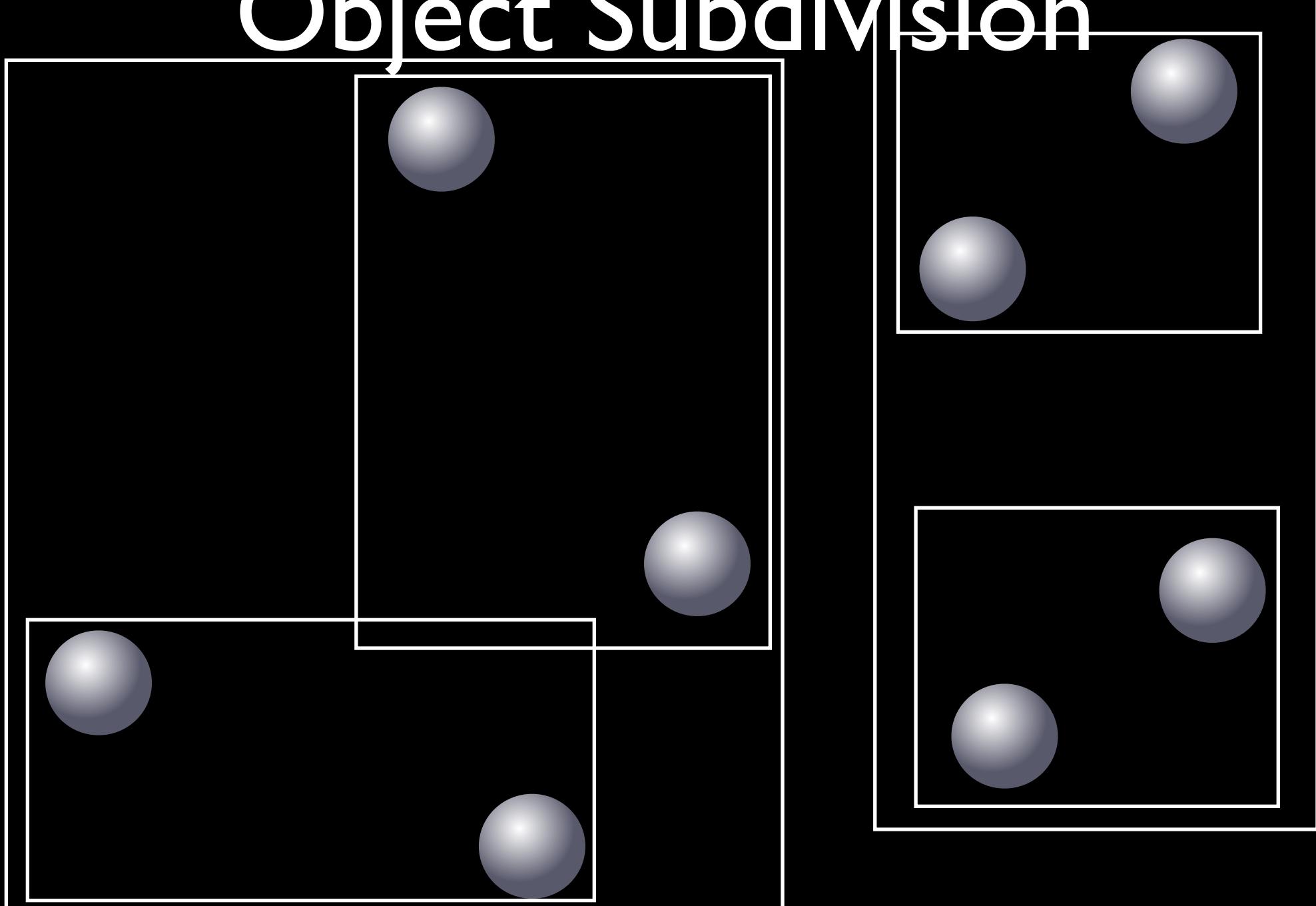
Object Subdivision



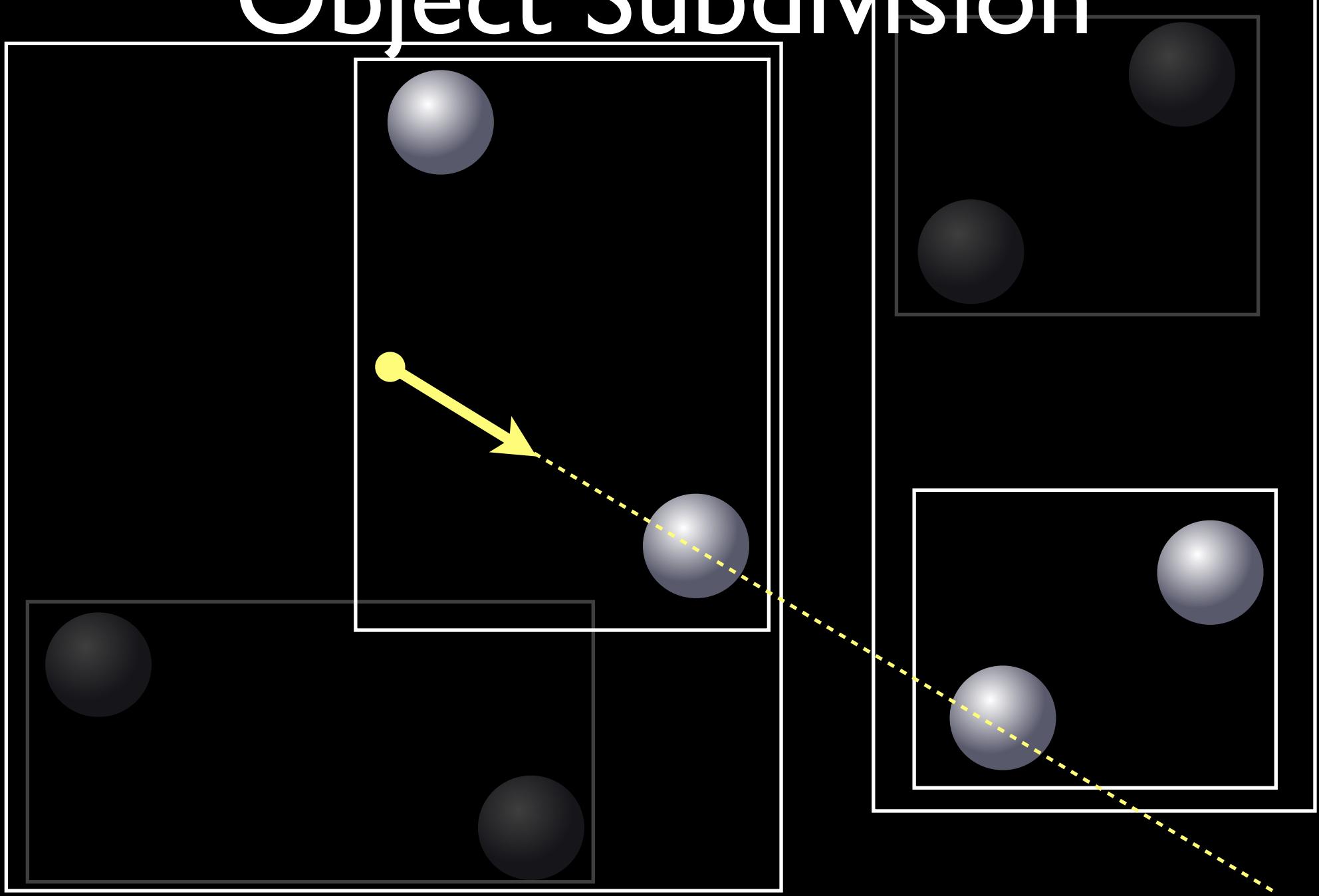
Object Subdivision



Object Subdivision



Object Subdivision



Object Subdivision

- Hierarchically subdivide **the list of objects**
 - Bounding volume hierarchy (BVH)
 - Choice of volume : sphere, box etc.
- Each node stores pointers to **the sublists**
- Leaf node stores the list of objects

Object Subdivision

```
node subdivision( objects, space ) {  
    if ( number of objects is small enough ) {  
        return make_leaf( objects )  
    }  
    objects.split ( &objects1, &object2 )  
    subspace1 = bounding_volume( objects1 )  
    subspace2 = bounding_volume( objects2 )  
  
    tree.add_node( subdivision( objects1, subspace1 ) )  
    tree.add_node( subdivision( objects2, subspace2 ) )  
}
```

Object vs Spatial

- Two approaches
 - Spatial subdivision (kD-tree)
 - Object subdivision (BVH)
- Still debatable
- Hybrid is possible and explored
- Similar to database queries

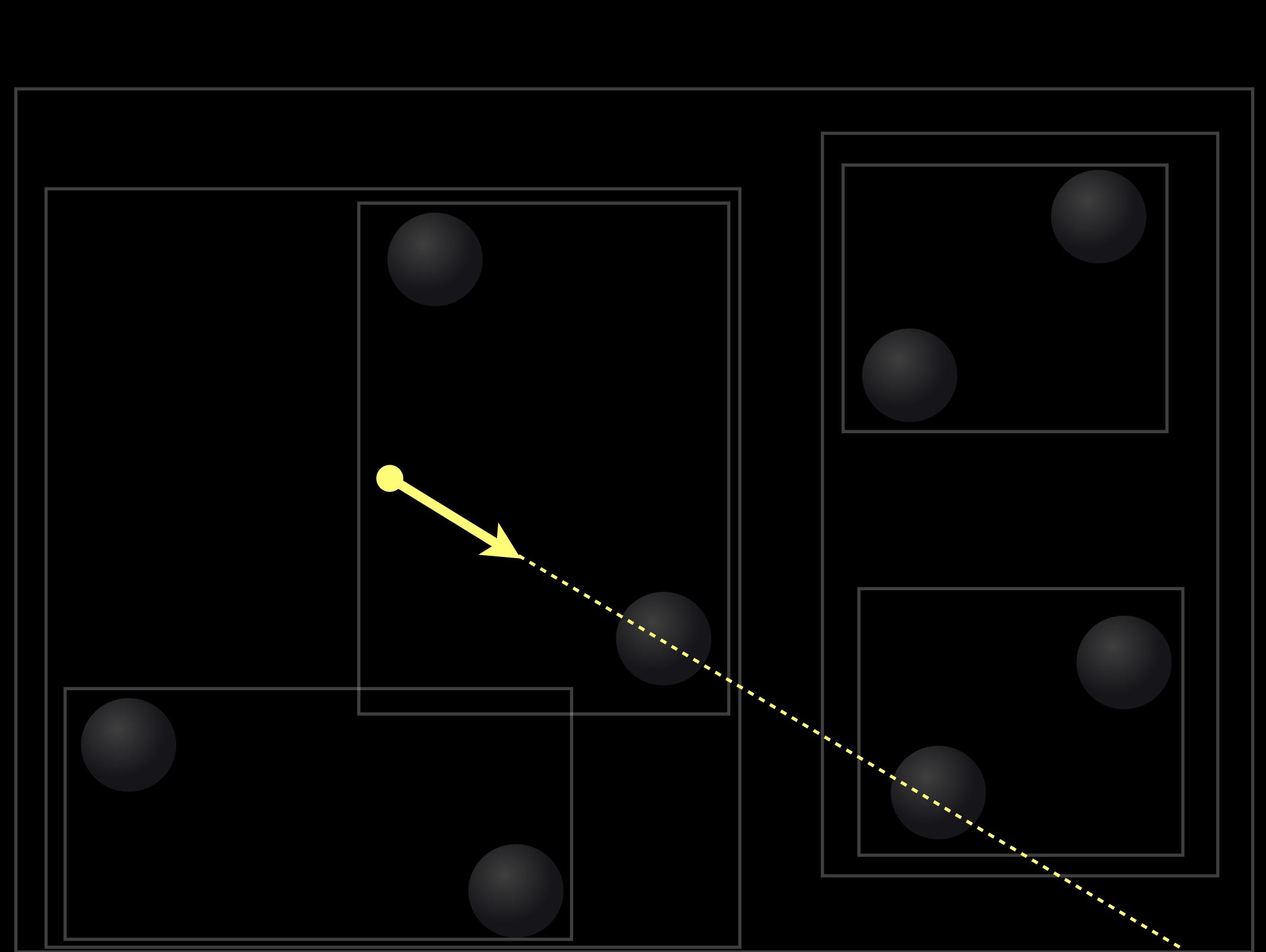
Traversal

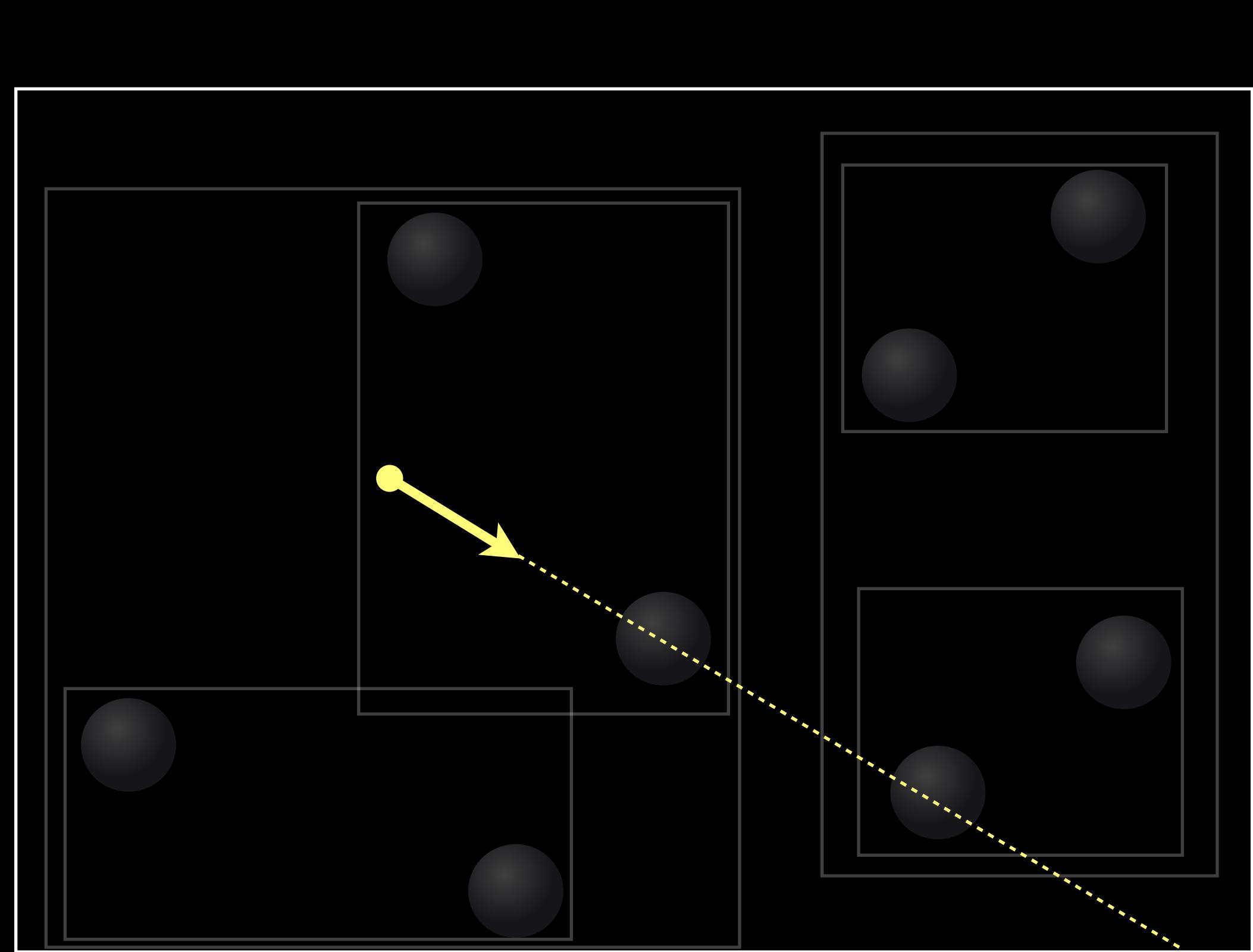
- Goal : Don't touch every single object
- Given an acceleration data structure
 - Start from the root (entire space)
 - Intersect the ray with child nodes
 - Perform ray-triangle intersections at leaf

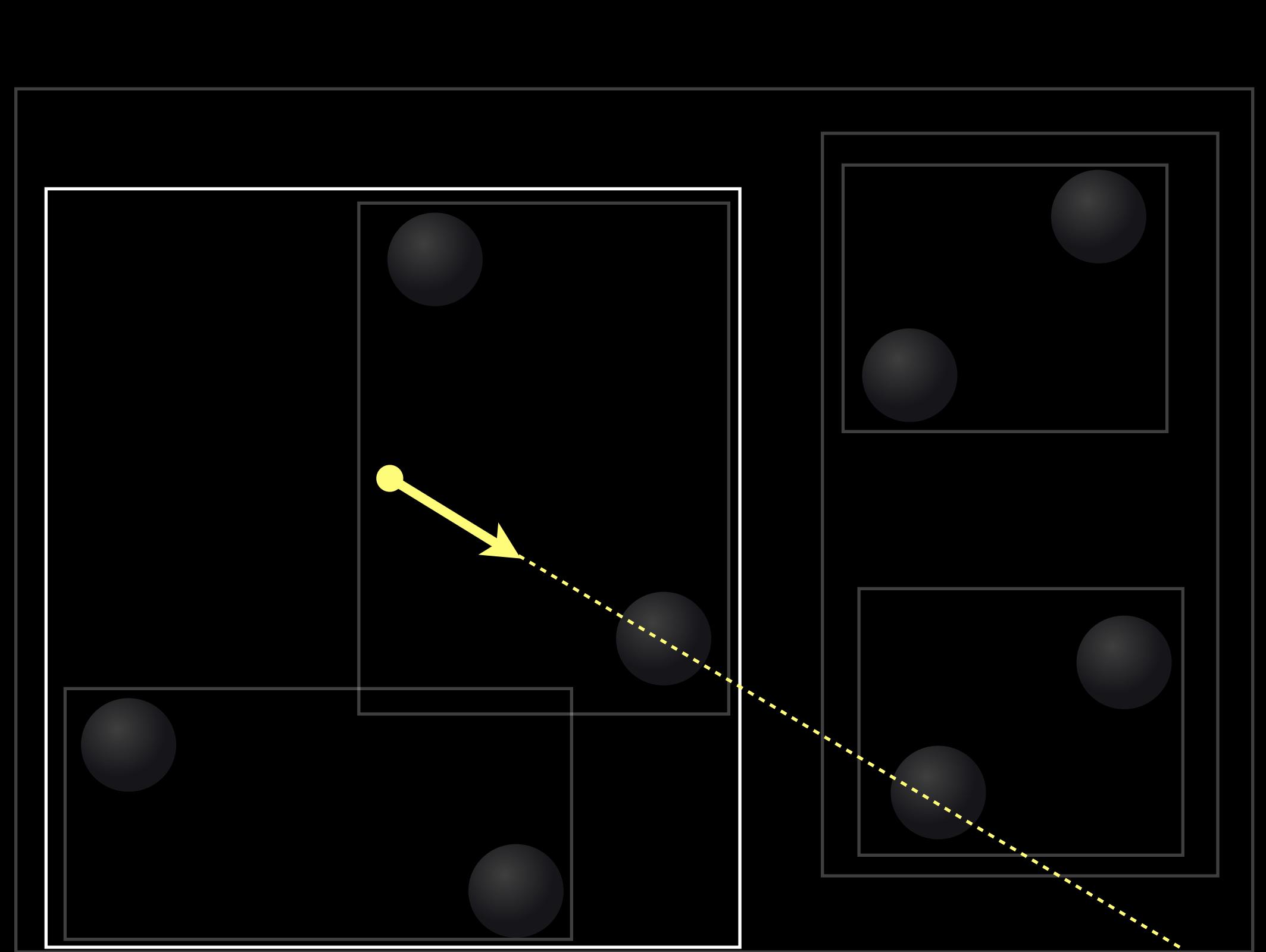
Traversal

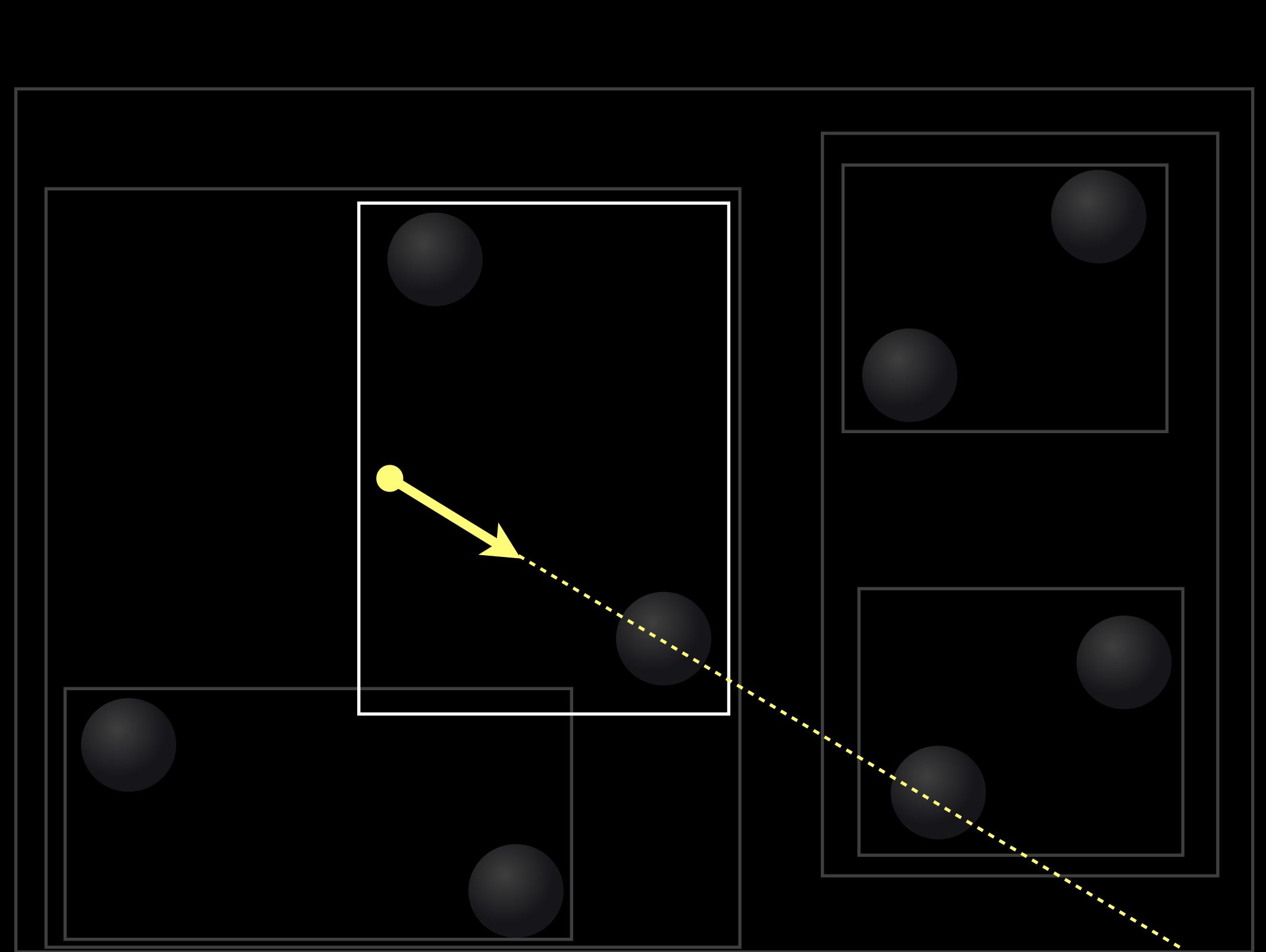
```
hit traverse( ray, node ) {
    if ( IsLeaf( node ) ) {
        for all objects in the node {
            hit = closer( hit, intersect( ray, object ) )
        }
    }
    if (overlap(ray, node.child1)) traverse( ray, node.child1 )
    if (overlap(ray, node.child2)) traverse( ray, node.child2 )
}

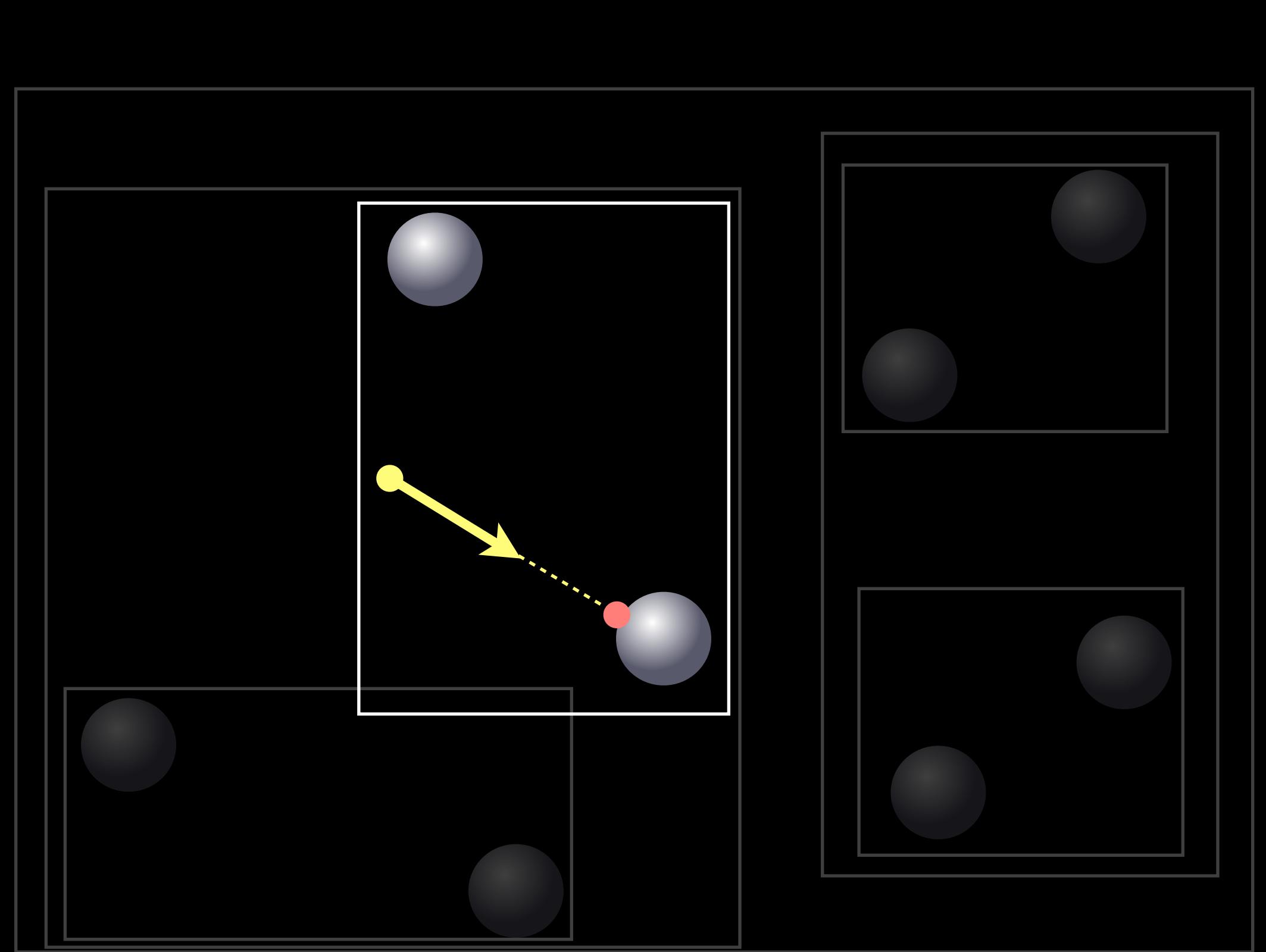
hit = traverse( ray, root )
```

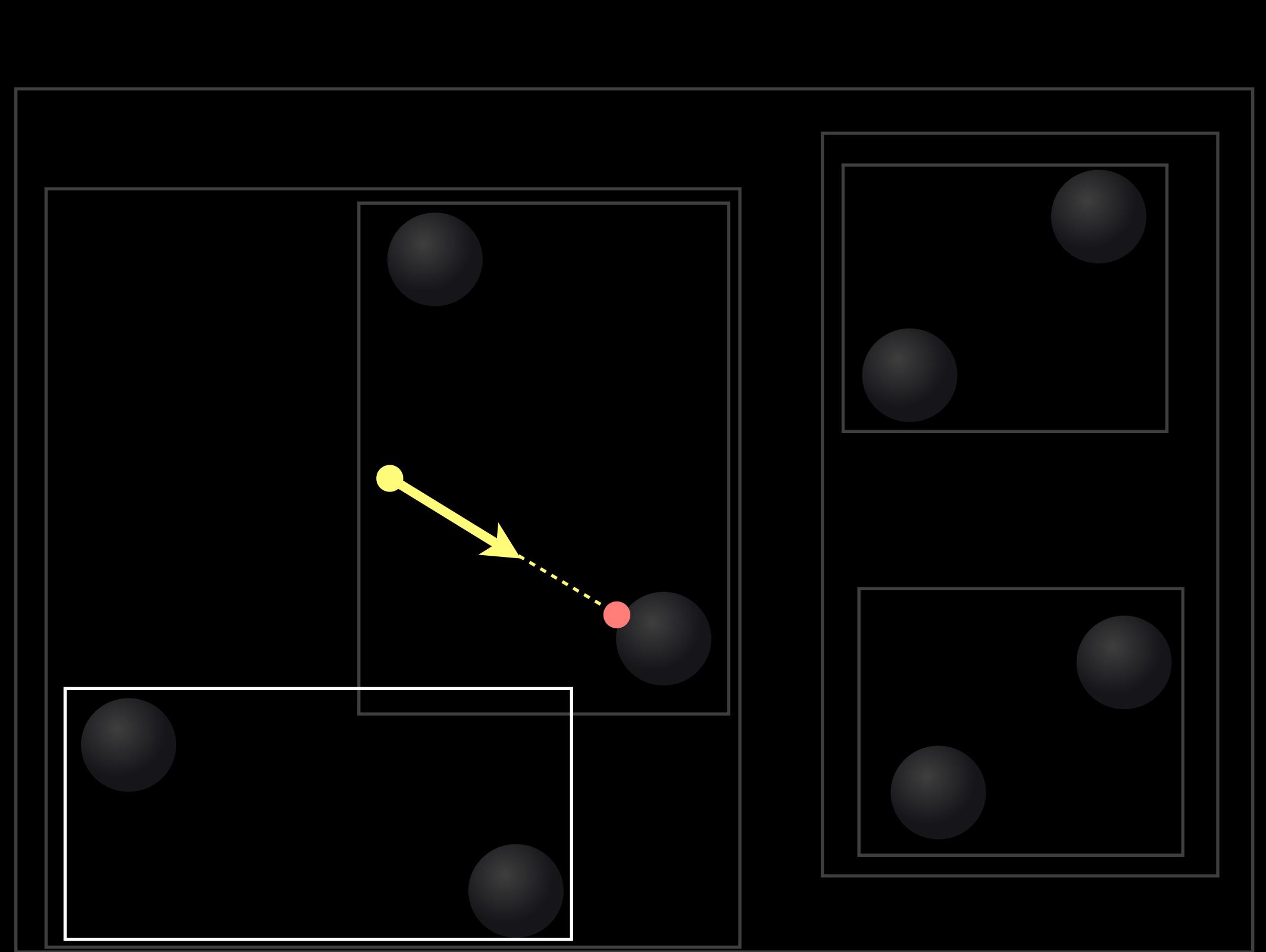


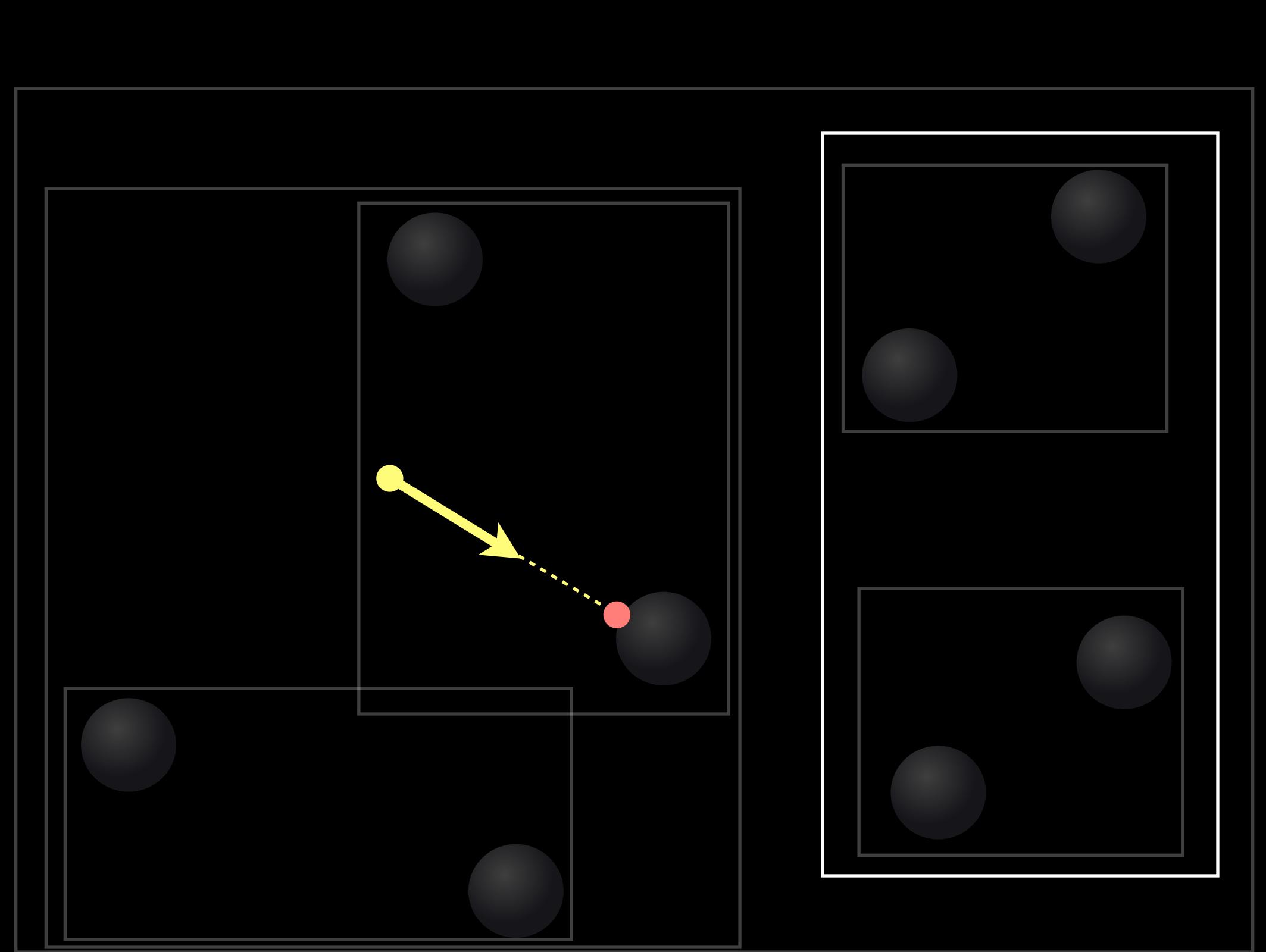




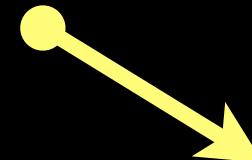








Only 2 intersection tests out of 8



Cost

```
for all pixels {  
    ray = generate_camera_ray( pixel )  
    first_hit = traverse( ray, accel_data_struct )  
    pixel = shade( first_hit )  
}
```

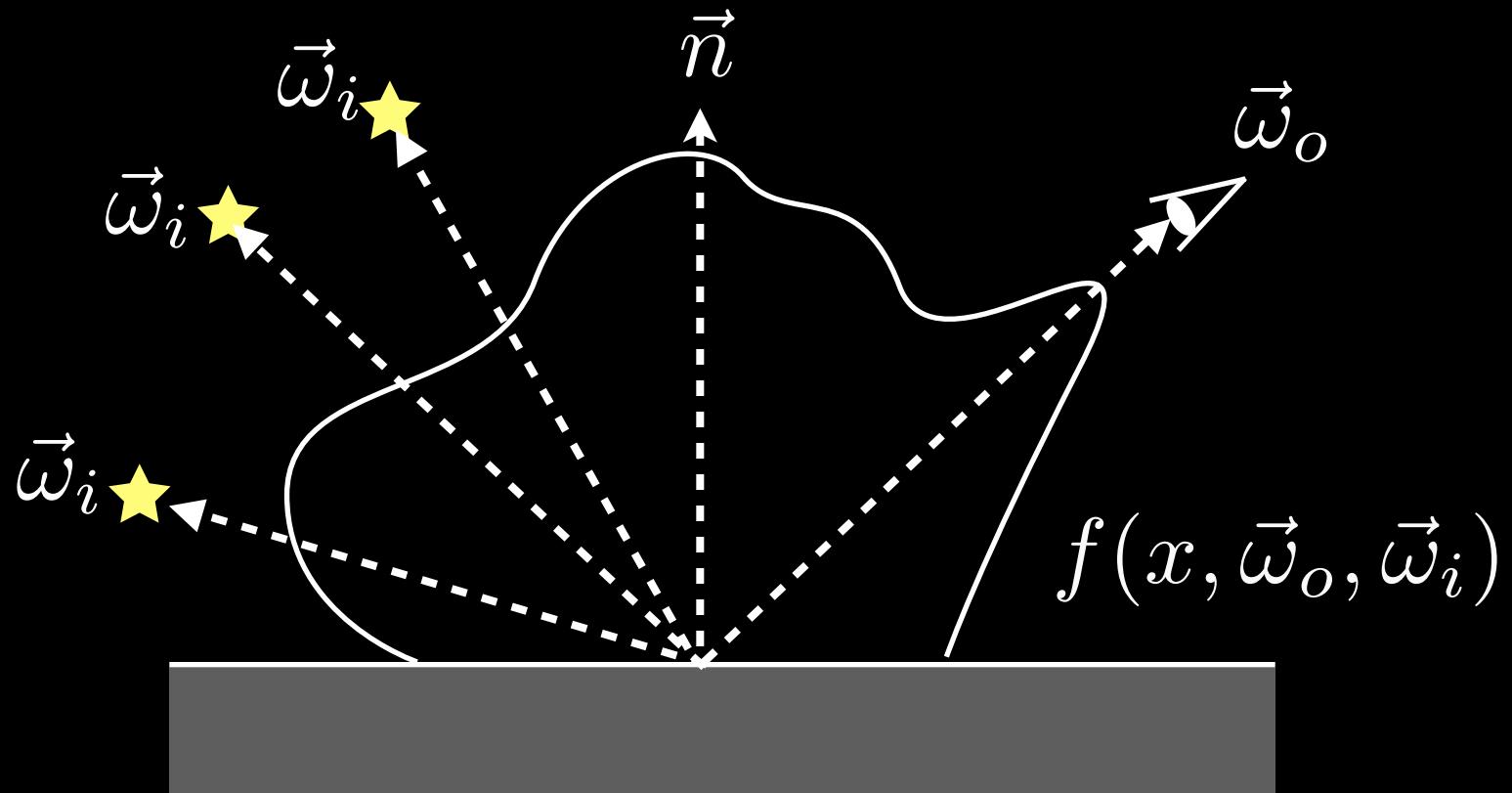
Cost

- No longer as simple as
“Number of objects × Number of rays”
 - Order analysis is not helpful
- Cost of traversal vs intersection tests
- When should we stop subdivision?

⇒ Surface Area Heuristic (SAH)

Reflected Radiance

$$L_o(x, \vec{\omega}_o) = \int_{\Omega} f(x, \vec{\omega}_o, \vec{\omega}_i) L_i(x, \vec{\omega}_i) \cos \theta_i d\omega_i$$



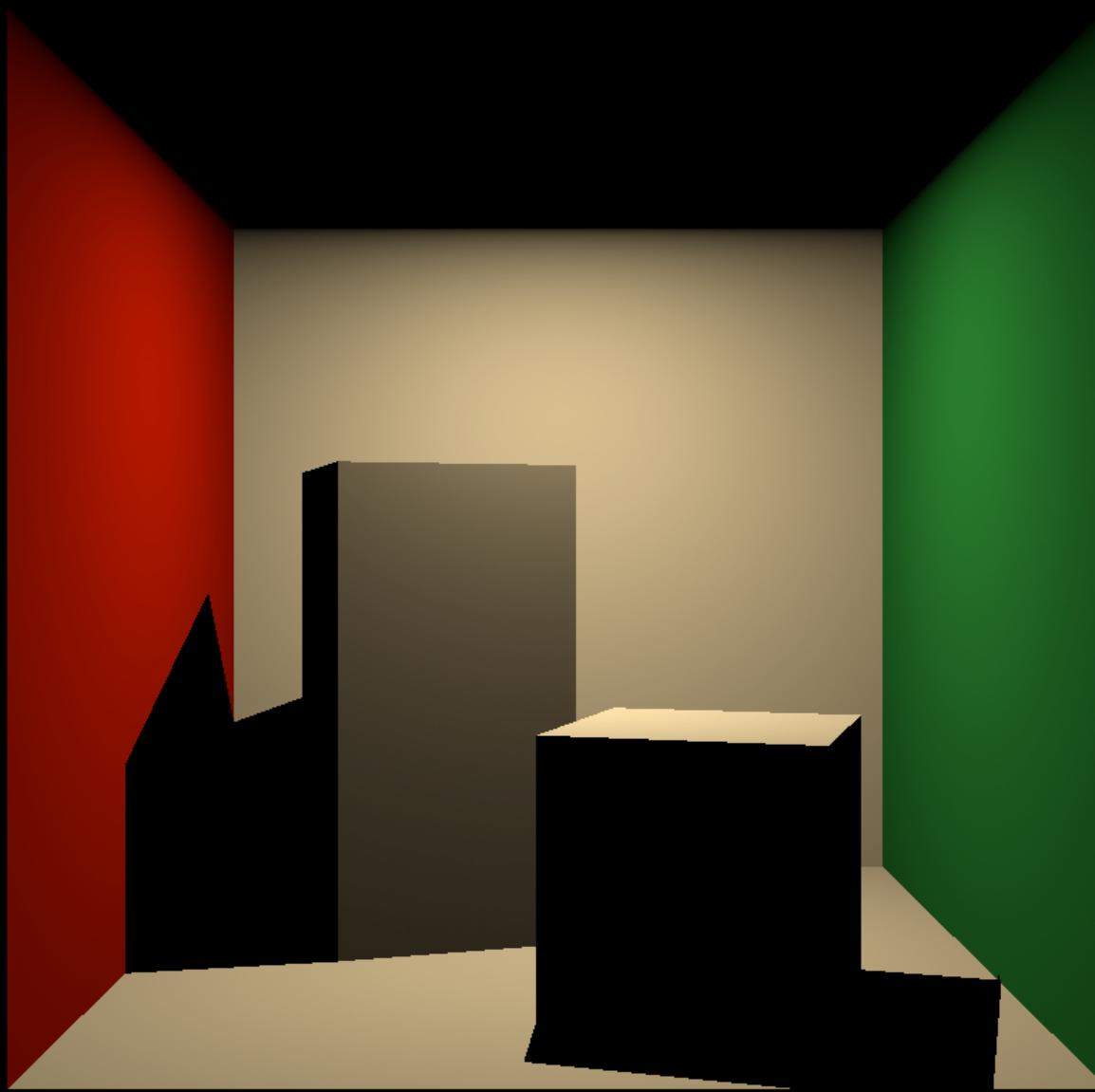
Incident Illumination

$$L_o(x, \vec{\omega}_o) = \int_{\Omega} f(x, \vec{\omega}_o, \vec{\omega}_i) L_i(x, \vec{\omega}_i) \cos \theta_i d\omega_i$$

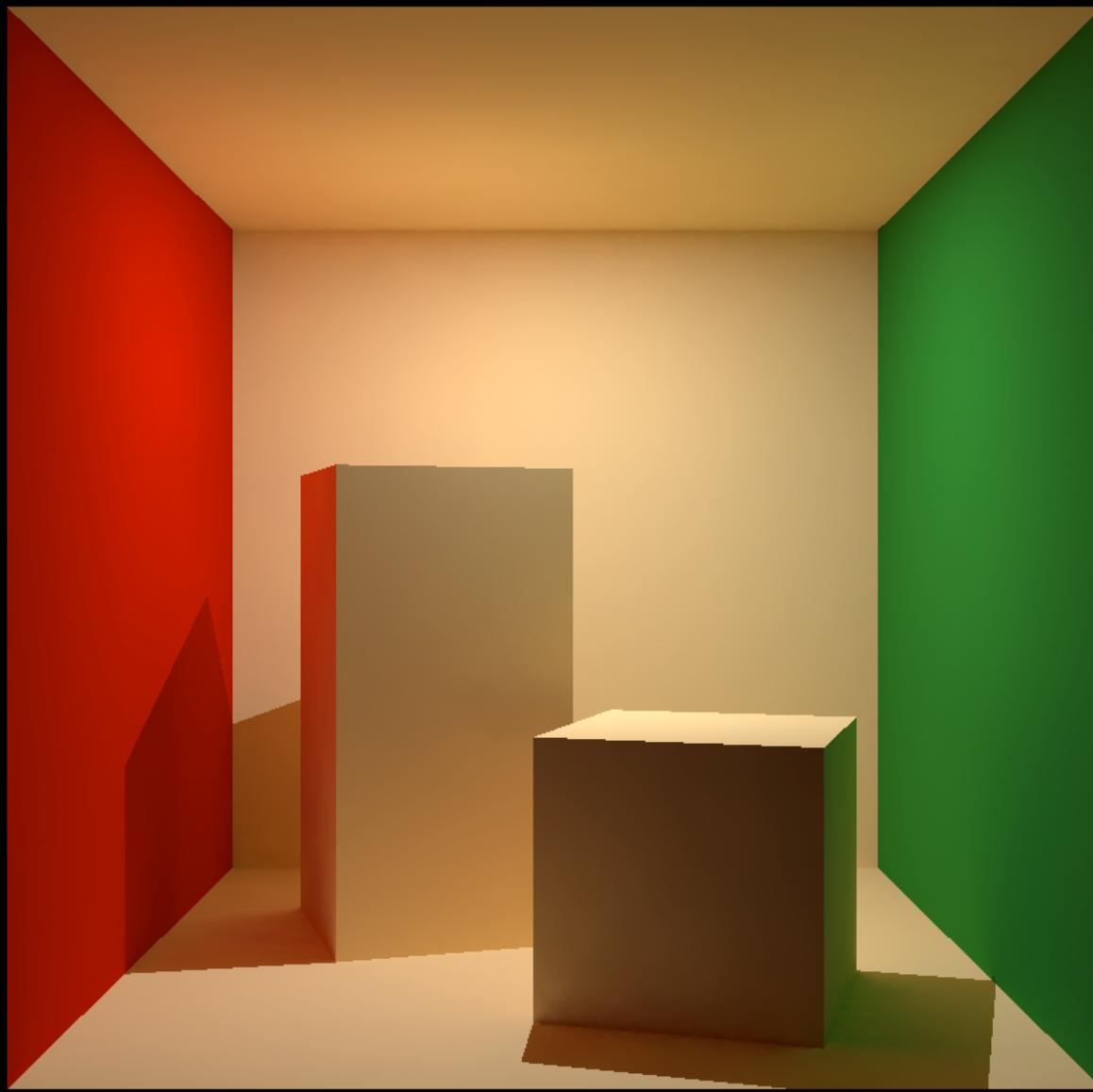
- Given by light sources or images



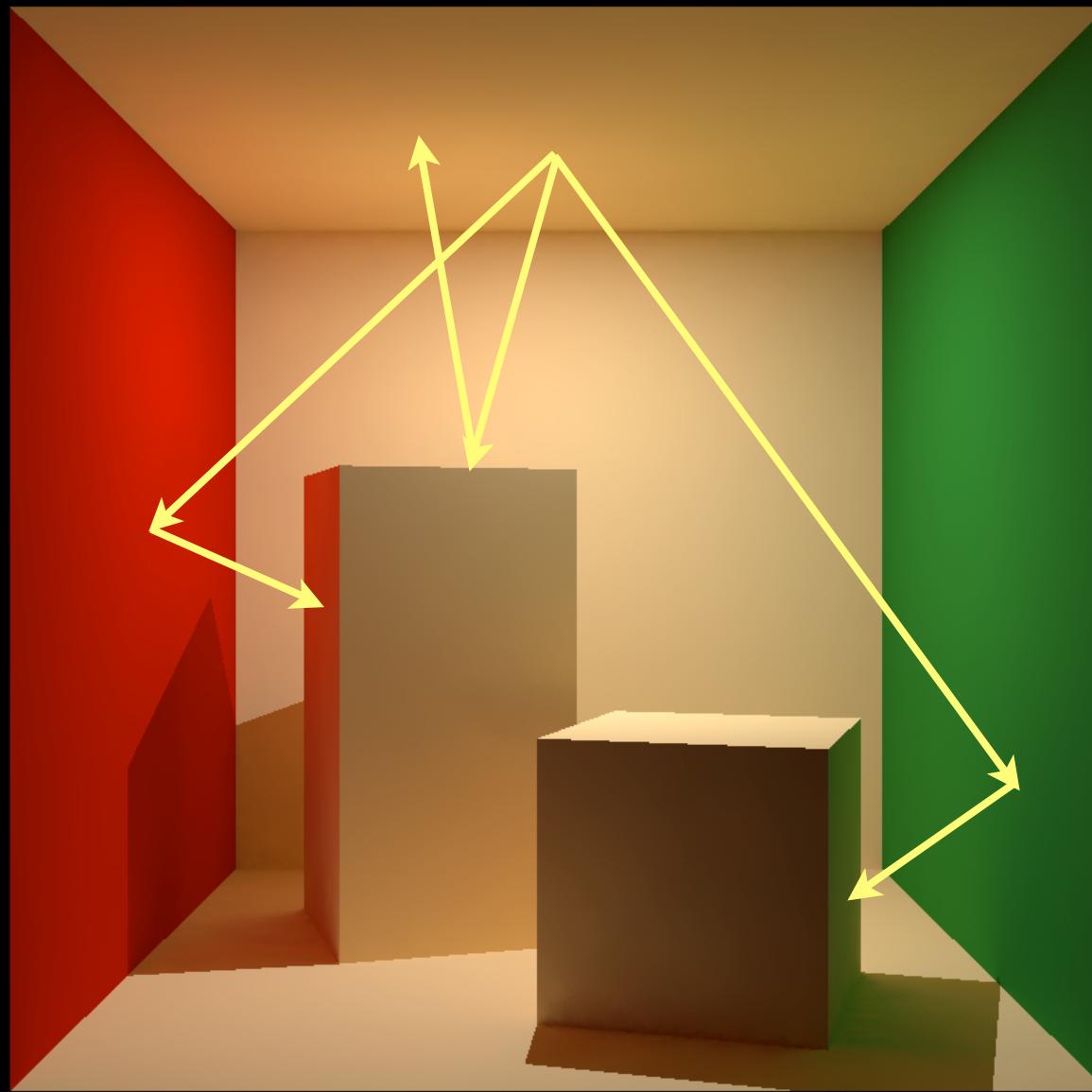
Missing Piece



Missing Piece



Missing Piece



Missing Piece

- Light can bounce off from other surfaces
- Multiple bounces
- Global illumination
 - Other objects affect illumination
 - Shadowing is one example
 - In contrast to local illumination

Transport Operator

- We use the following operator
 - Input: illumination
 - Output: reflected radiance
- Simplify the notation

$$L_o(x \rightarrow e) = T[L_i]$$



$$L_o(x \rightarrow e) = \int_{\Omega} f(l \rightarrow x \rightarrow e) L_i(l \rightarrow x) \cos \theta d\omega$$

Using Transport Operator

- One bounce (i.e., direct)

$$I^0 L_o(x \rightarrow e) = T[L_i]$$

- Two bounces

$$I^1 L_o(x \rightarrow e) = T[I^0 L_o]$$

Using Transport Operator

- One bounce (i.e., direct)

$$I^0 L_o(x \rightarrow e) = T[L_i]$$

- Two bounces

$$I^1 L_o(x \rightarrow e) = T[I^0 L_o]$$

Using Transport Operator

- One bounce (i.e., direct)

$$I^0 L_o(x \rightarrow e) = T[L_i]$$

- Two bounces

$$\begin{aligned} I^1 L_o(x \rightarrow e) &= T[I^0 L_o] \\ &= T[T[L_i]] = T^2[L_i] \end{aligned}$$

Using Transport Operator

- One bounce (i.e., direct)

$$I^0 L_o(x \rightarrow e) = T[L_i]$$

- Two bounces

$$\begin{aligned} I^1 L_o(x \rightarrow e) &= T[I^0 L_o] \\ &= T[T[L_i]] = T^2[L_i] \end{aligned}$$

$$L_o(x \rightarrow e) = T[L_i] + T^2[L_i]$$

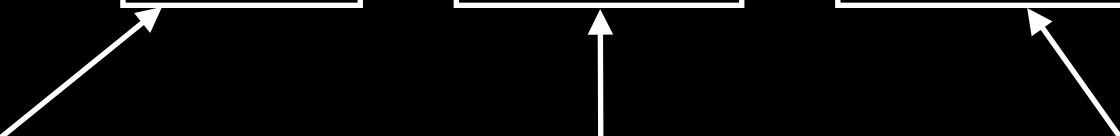
Including All Bounces

$$L_o(x \rightarrow e) = T[L_i] + T^2[L_i] + T^3[L_i] + \dots$$

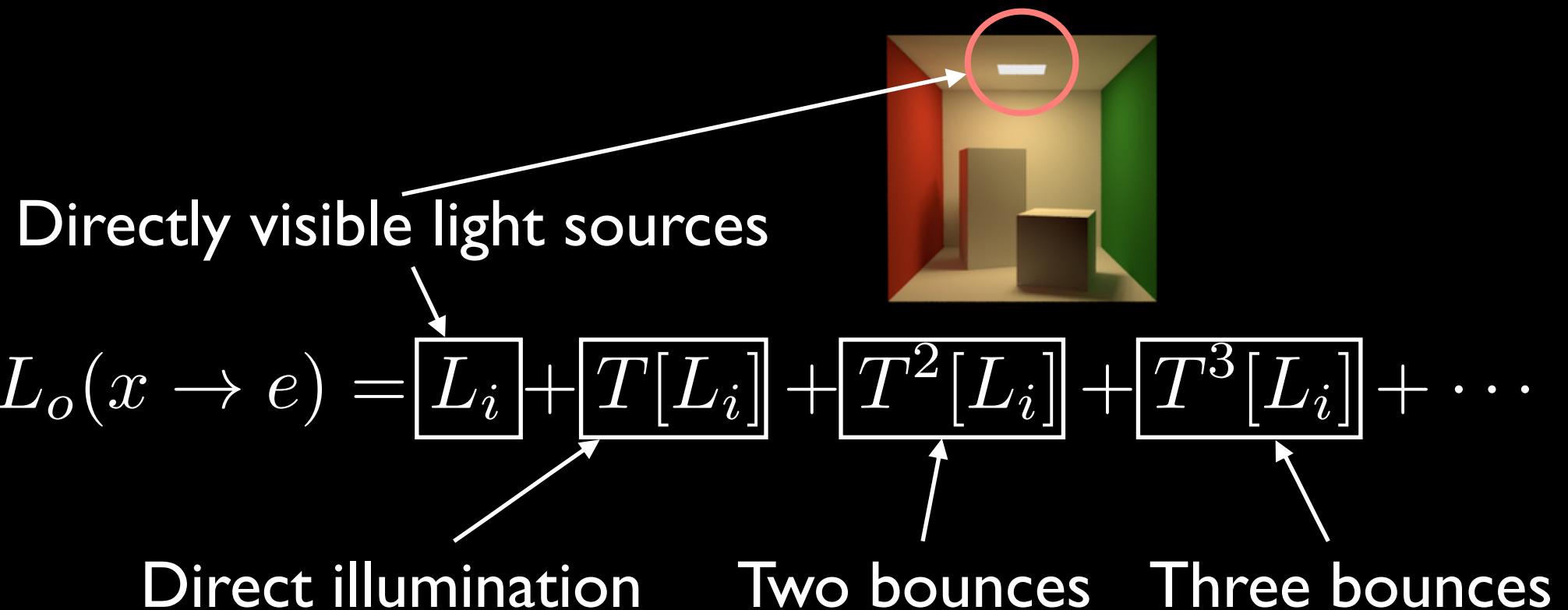
Including All Bounces

$$L_o(x \rightarrow e) = \boxed{T[L_i]} + \boxed{T^2[L_i]} + \boxed{T^3[L_i]} + \dots$$

Direct illumination Two bounces Three bounces



Including All Bounces



To the Rendering Equation

- Remember the Neumann series

$$\frac{I}{I - K} = I + K + K^2 + K^3 + \dots$$

To the Rendering Equation

- Remember the Neumann series

$$\frac{I}{I - K} = I + K + K^2 + K^3 + \dots$$

$$L_o(x \rightarrow e) = L_i + T[L_i] + T^2[L_i] + T^3[L_i] + \dots$$

$$= \frac{I}{I - T}[L_i]$$

To the Rendering Equation

- Remember the Neumann series

$$\frac{I}{I - K} = I + K + K^2 + K^3 + \dots$$

$$L_o(x \rightarrow e) = L_i + T[L_i] + T^2[L_i] + T^3[L_i] + \dots$$

$$= \frac{I}{I - T}[L_i]$$

$$(I - T)[L_o(x \rightarrow e)] = L_i$$

To the Rendering Equation

$$(I - T)[L_o(x \rightarrow e)] = L_i$$

$$L_o(x \rightarrow e) - T[L_o(x \rightarrow e)] = L_i$$

$$L_o(x \rightarrow e) = L_i + T[L_o(x \rightarrow e)]$$

$$L(x \rightarrow e) = L_i(x \rightarrow e) + \int_{\Omega} f(\omega, x \rightarrow e) L(\omega) \cos \theta d\omega$$

To the Rendering Equation

$$(I - T)[L_o(x \rightarrow e)] = L_i$$

$$L_o(x \rightarrow e) - T[L_o(x \rightarrow e)] = L_i$$

$$L_o(x \rightarrow e) = L_i + T[L_o(x \rightarrow e)]$$

$$L(x \rightarrow e) = L_i(x \rightarrow e) + \int_{\Omega} f(\omega, x \rightarrow e) L(\omega) \cos \theta d\omega$$

Rendering Equation

Rendering Equation

- Describe the equilibrium of radiance
- Rendering algorithms = solvers of R.E.
 - James Kajiya in 1986

$$L(x \rightarrow e) = \boxed{L_i(x \rightarrow e)} + \int_{\Omega} \boxed{f(\omega, x \rightarrow e)L(\omega) \cos \theta d\omega}$$

↑ ↑
Self-emission BRDF
(zero if x is not light source)



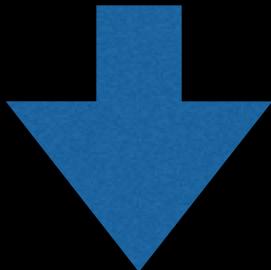
Figure 6. A sample image. All objects are neutral grey. Color on the objects is due to caustics from the green glass balls and color bleeding from the base polygon.

“The Rendering Equation” [Kajiya 1986]

Path Tracing

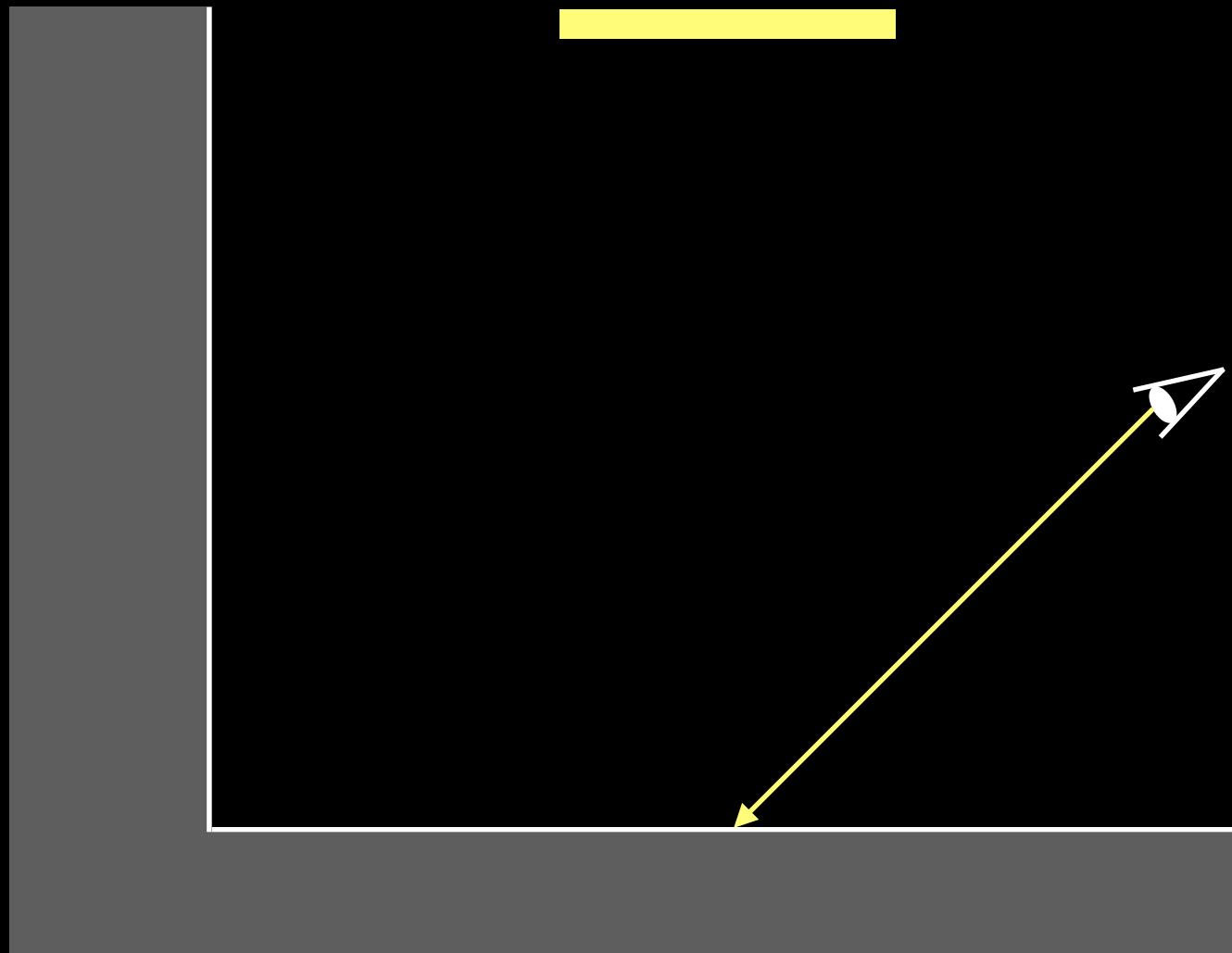
- Recursive expansion by random sampling

$$L(x \rightarrow e) = L_i(x \rightarrow e) + \int_{\Omega} f(\omega, x \rightarrow e) L(\omega) \cos \theta d\omega$$

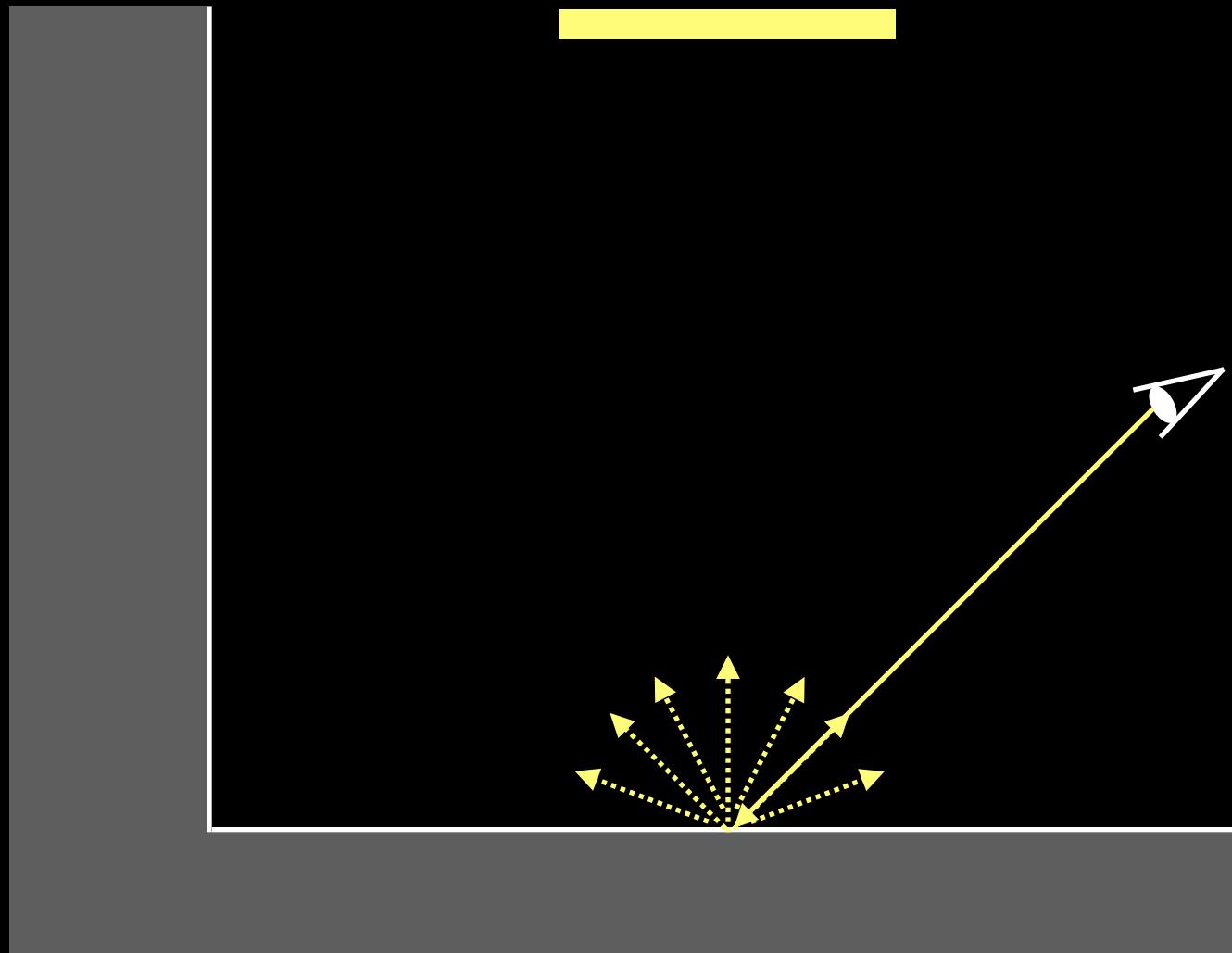


$$L(x \rightarrow e) \approx L_i(x \rightarrow e) + \frac{f(\omega_0, x \rightarrow e) L(\omega_0) \cos \theta_0}{p(\omega_0)}$$

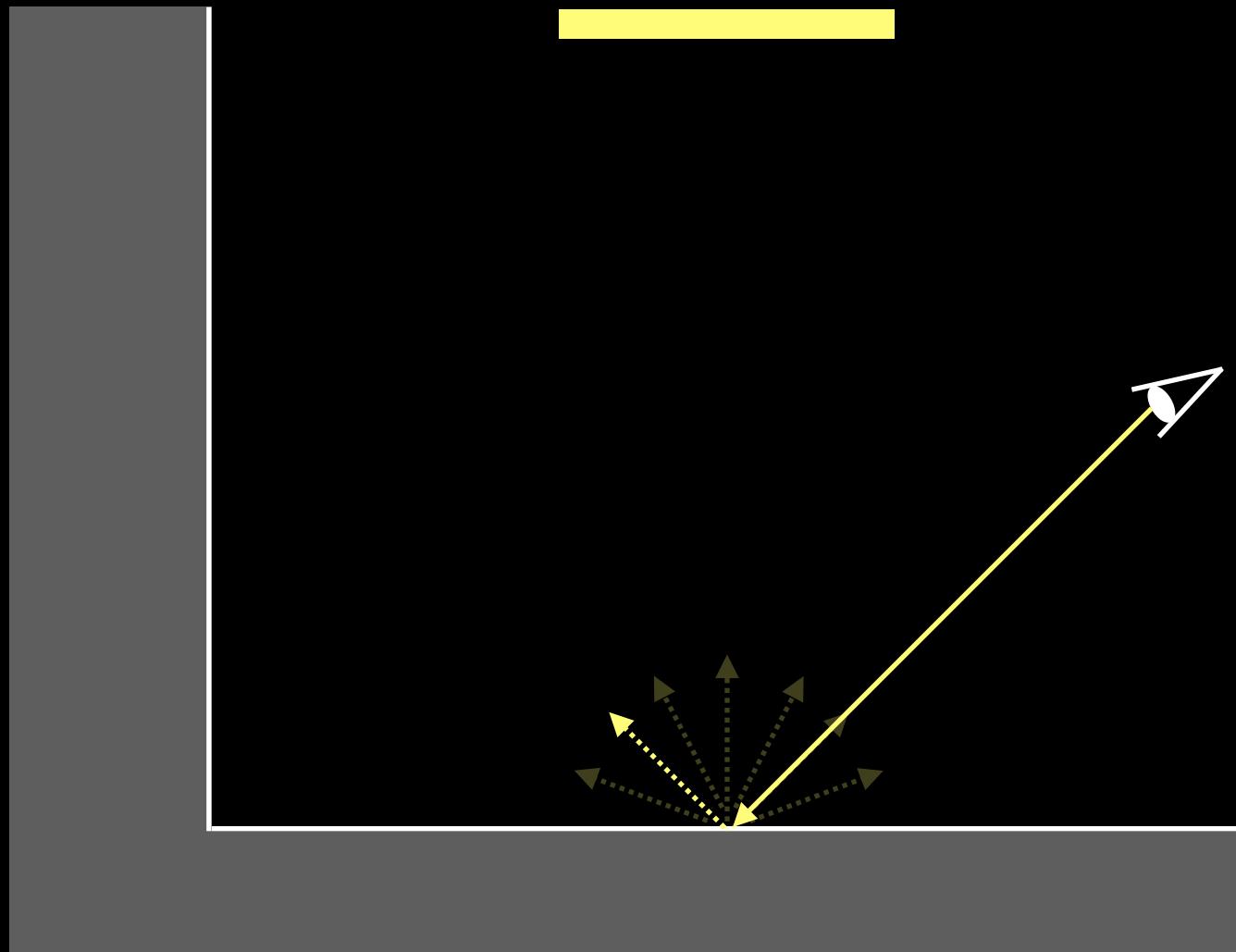
Path Tracing



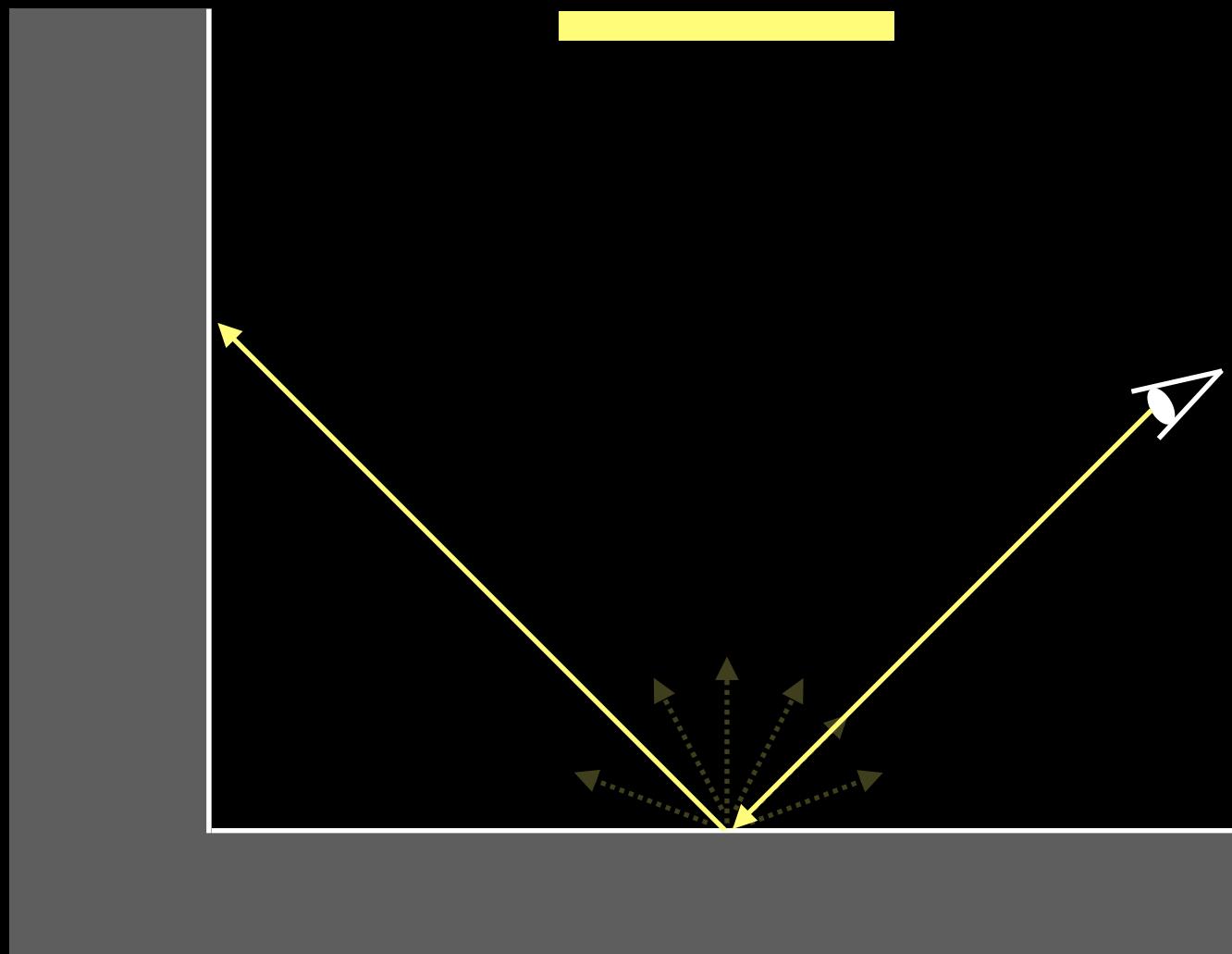
Path Tracing



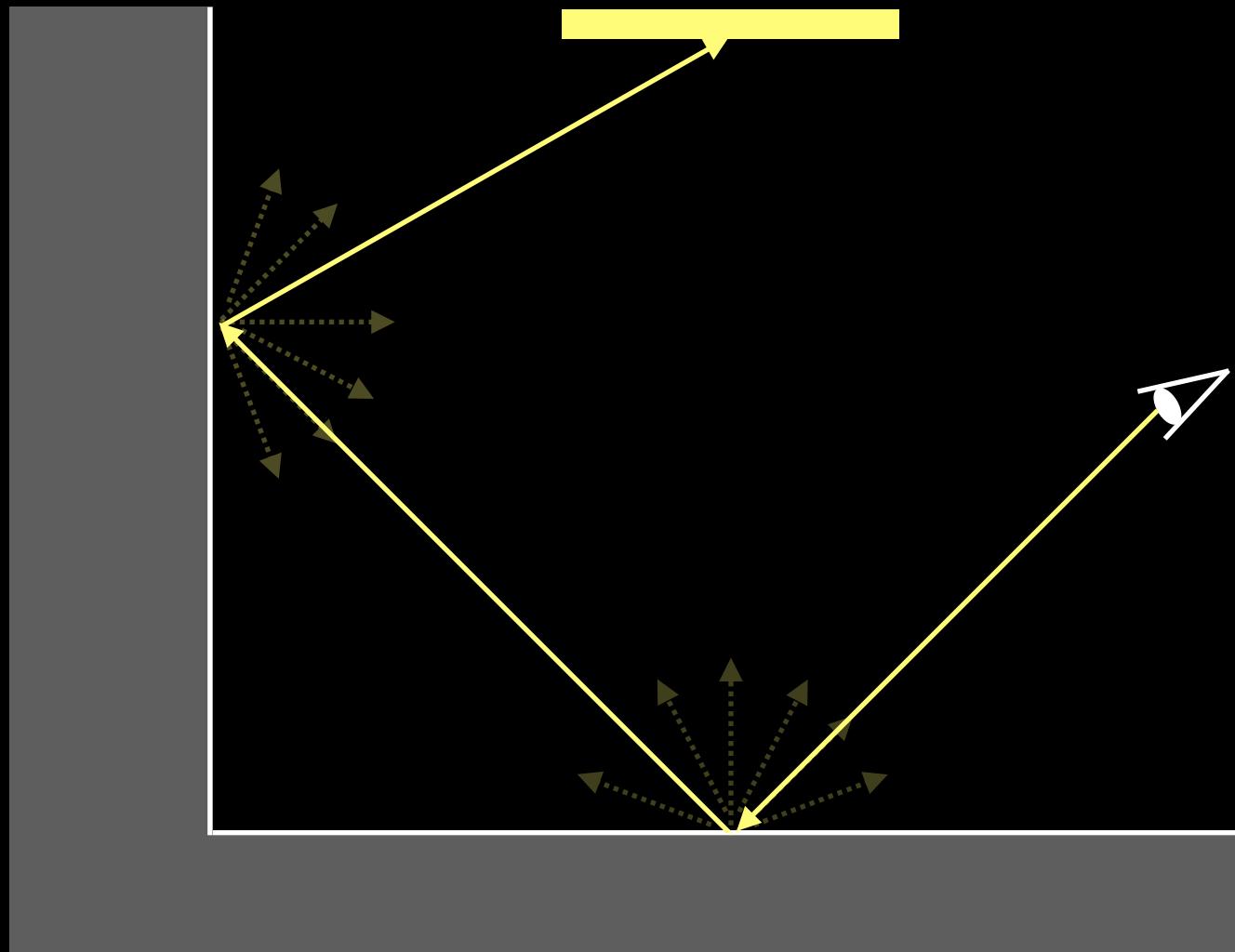
Path Tracing



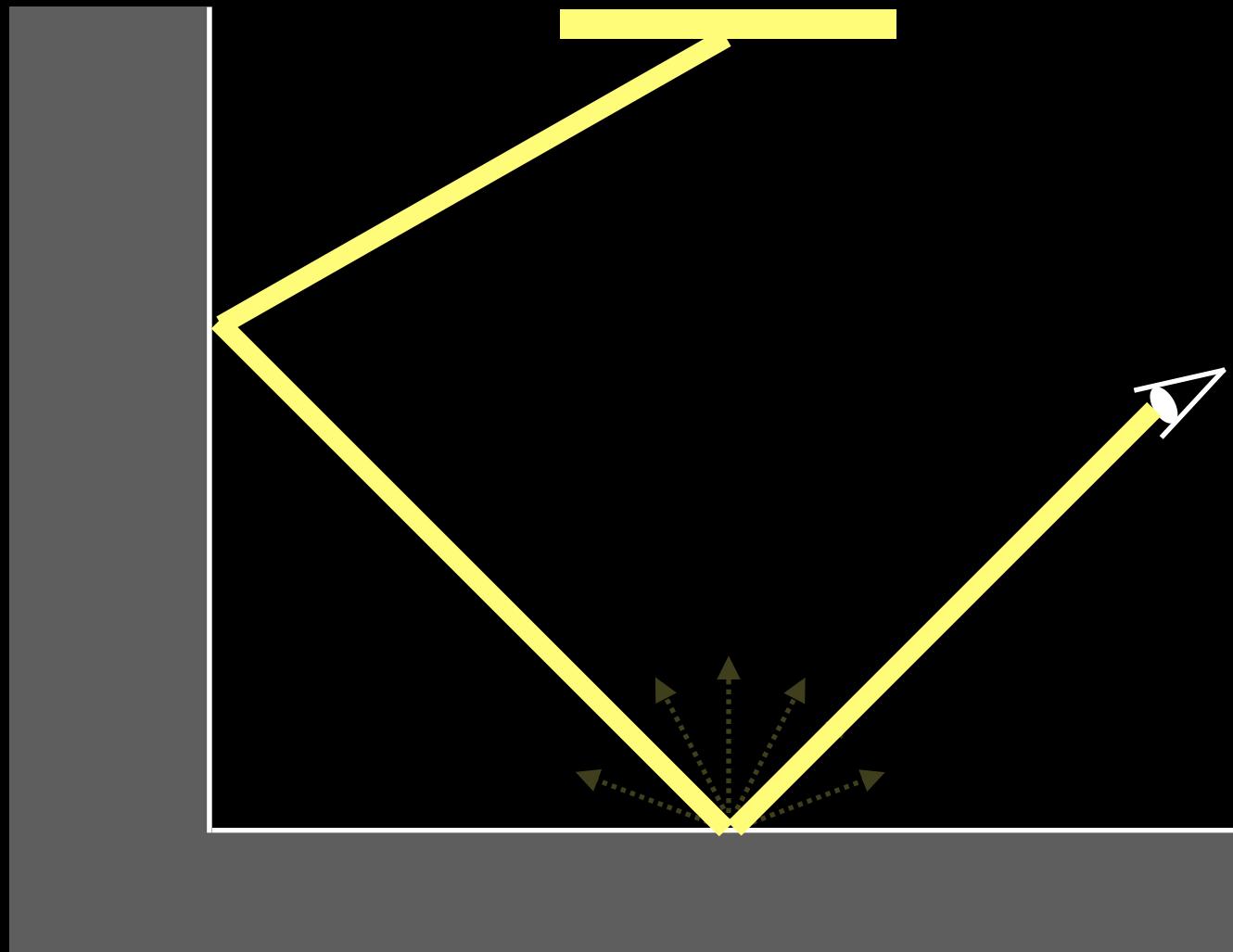
Path Tracing



Path Tracing



Path Tracing



Path Tracing

- Recursive call even for Lambertian
 - Use random directions around the normal
 - Add the emission term for light sources

```
color shade (hit) {  
    return (Kd / PI) * get_irradiance(hit)  
}
```

Path Tracing

- Recursive call even for Lambertian
 - Use random directions around the normal
 - Add the emission term for light sources

```
color shade (hit) {  
    w = random_dir(hit.normal)  
    c = max(dot(w, hit.normal), 0)  
    d = ray(hit.position, w)  
    return Le + (Kd / PI) * shade(trace(d)) * c / p(w)  
}
```

Generating Random Directions

- Use spherical coordinates (then get xyz)

$$p(\omega_i) = \frac{1}{2\pi}$$

$$\theta = \arccos(u_1)$$

$$\phi = 2\pi u_2$$

$$u_1, u_2 \in [0, 1]$$

Generating Random Directions

- θ and ϕ are around the normal, not y axis
- Use an orthonormal basis (similar to eye rays)

$$\vec{n}_y = \vec{n}$$

$$\vec{n}_x = \frac{\vec{c} \times \vec{n}}{|\vec{c} \times \vec{n}|} \quad \vec{\omega}_i = x\vec{n}_x + y\vec{n}_y + z\vec{n}_z$$

$$\vec{n}_z = \vec{n}_x \times \vec{n}$$

\vec{c} : a vector that is not parallel to normal

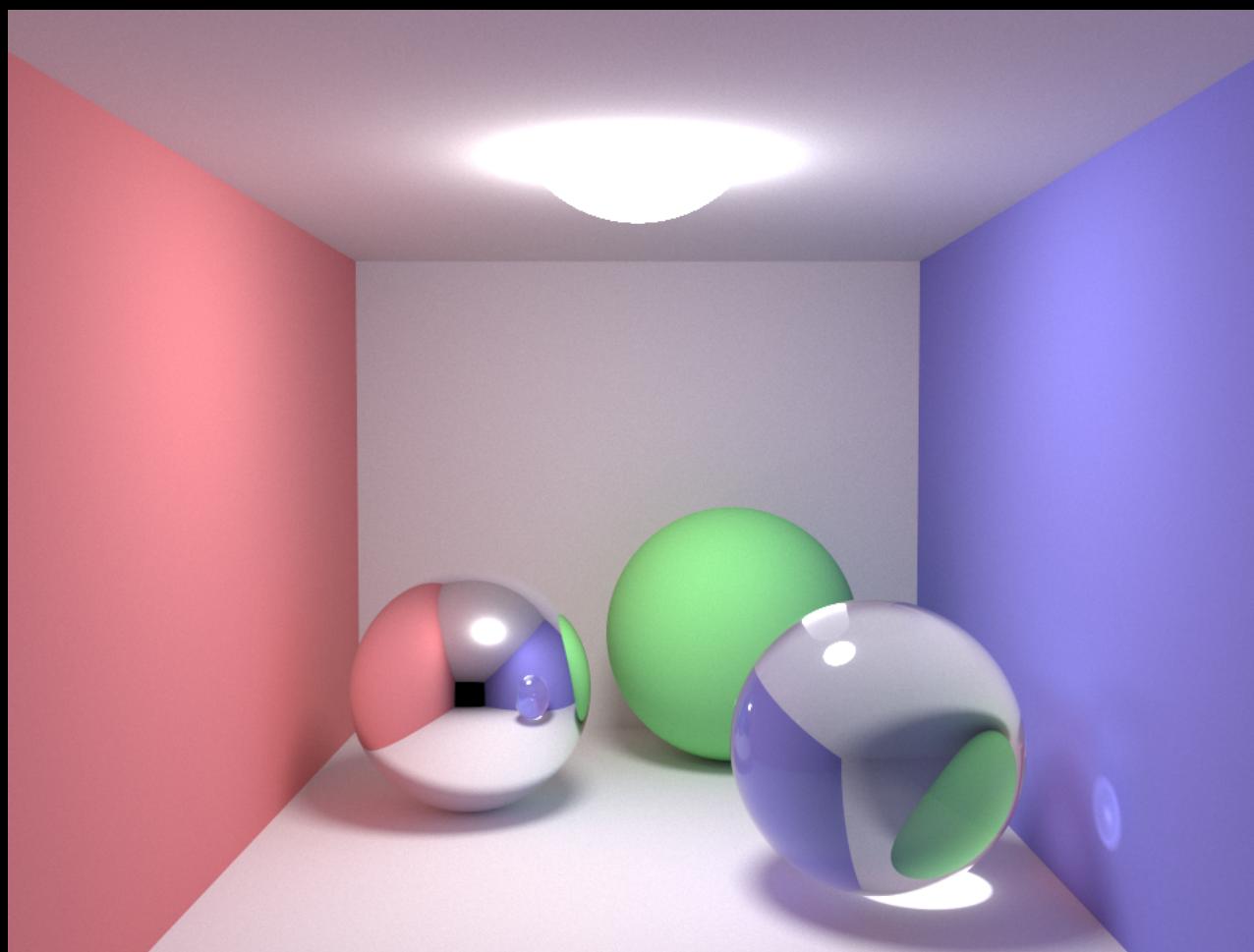
Path Tracing

- Take the average of random samples

```
for all pixels {  
    ray = generate_camera_ray( pixel )  
    first_hit = traverse( ray, accel_data_struct )  
    pixel = 0  
    for i = 1 to N {  
        pixel = pixel + shade( first_hit )  
    }  
    pixel = pixel / N  
}
```

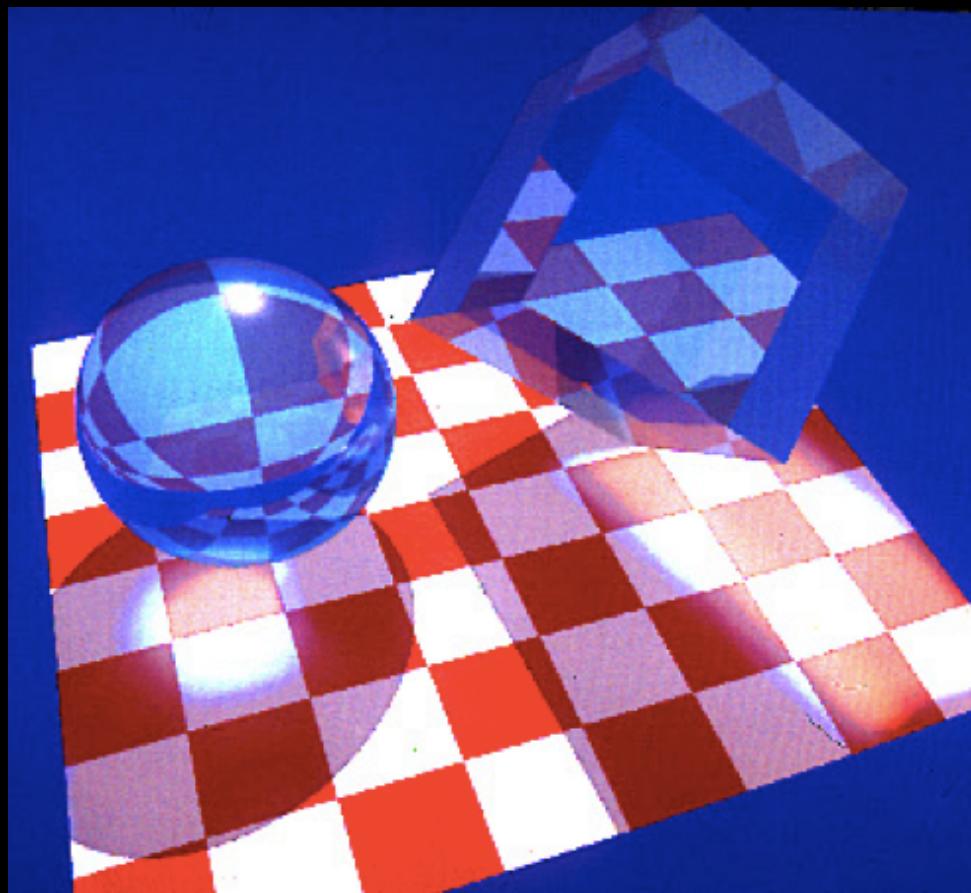
Example

- edupt (<http://kagamin.net/hole/edupt/>)



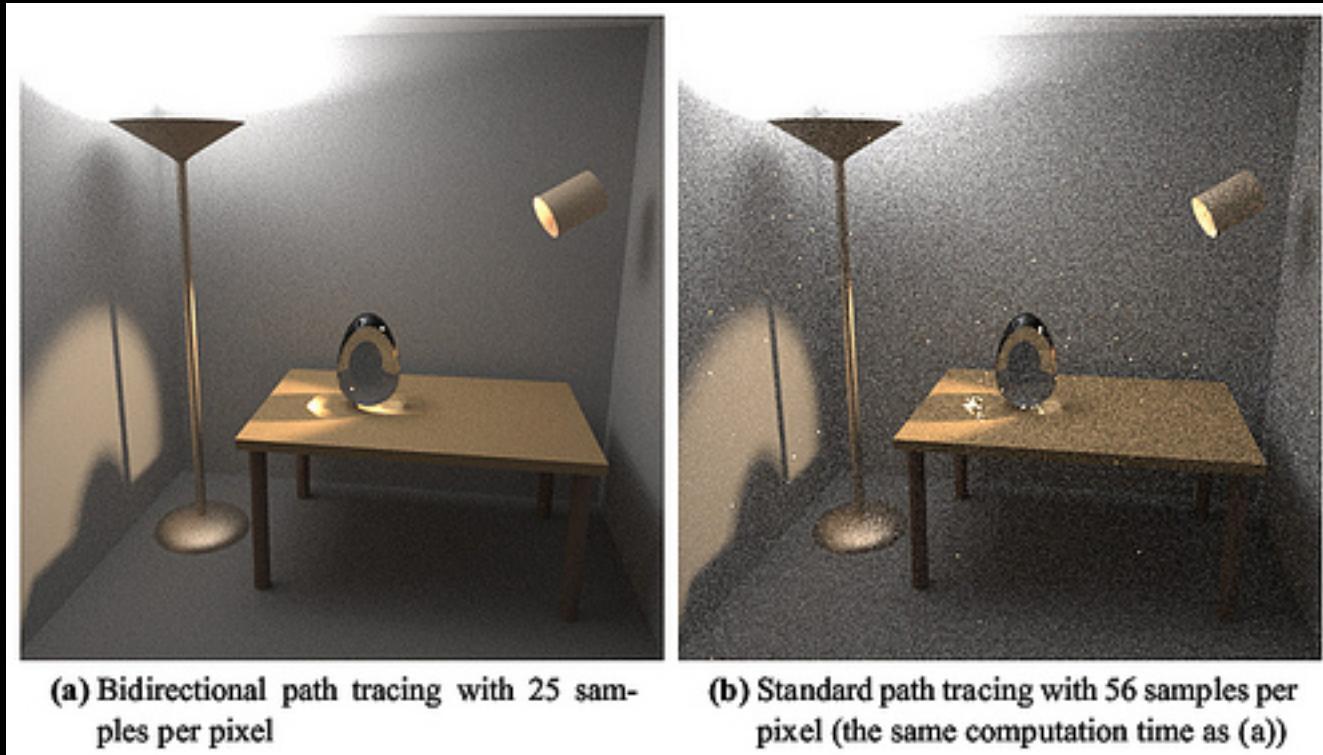
Light Tracing

- Trace paths from light sources



Bidirectional Path Tracing

- Trace paths from both light sources and eye



Markov Chain Monte Carlo

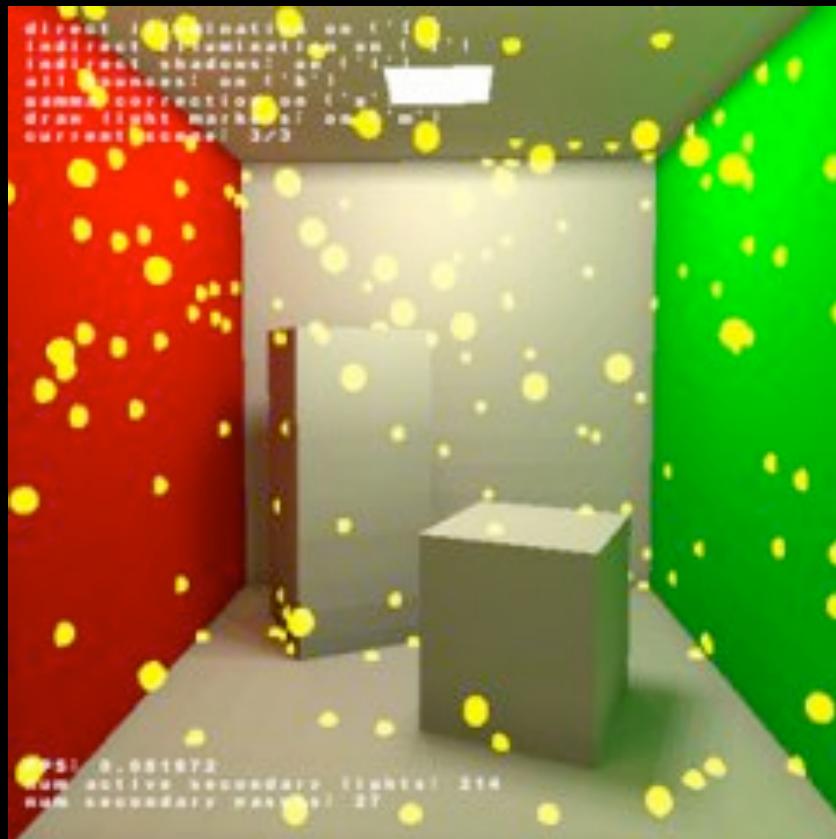
- Generate a new sample by mutating the old



<https://graphics.stanford.edu/papers/metro/metro.pdf>

Many Lights Methods

- Lots of point lights for simulating diffuse bounces

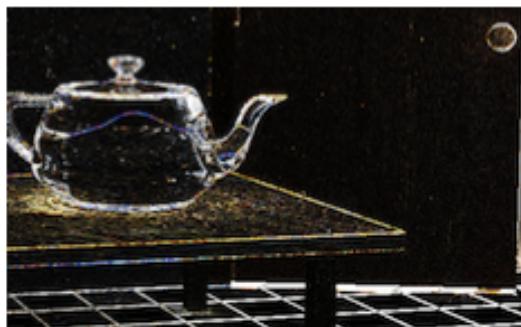


Gradient-Domain Rendering

- Render derivatives and reconstruct the image



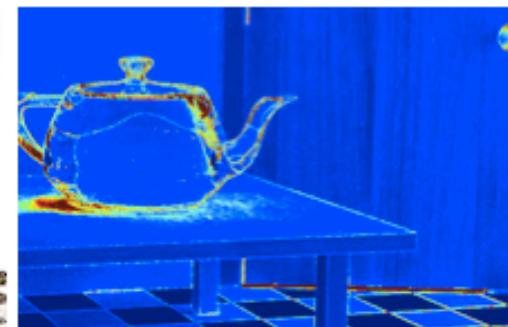
Horiz. differences I^{dx}



Vertical differences I^{dy}



Coarse image I^g



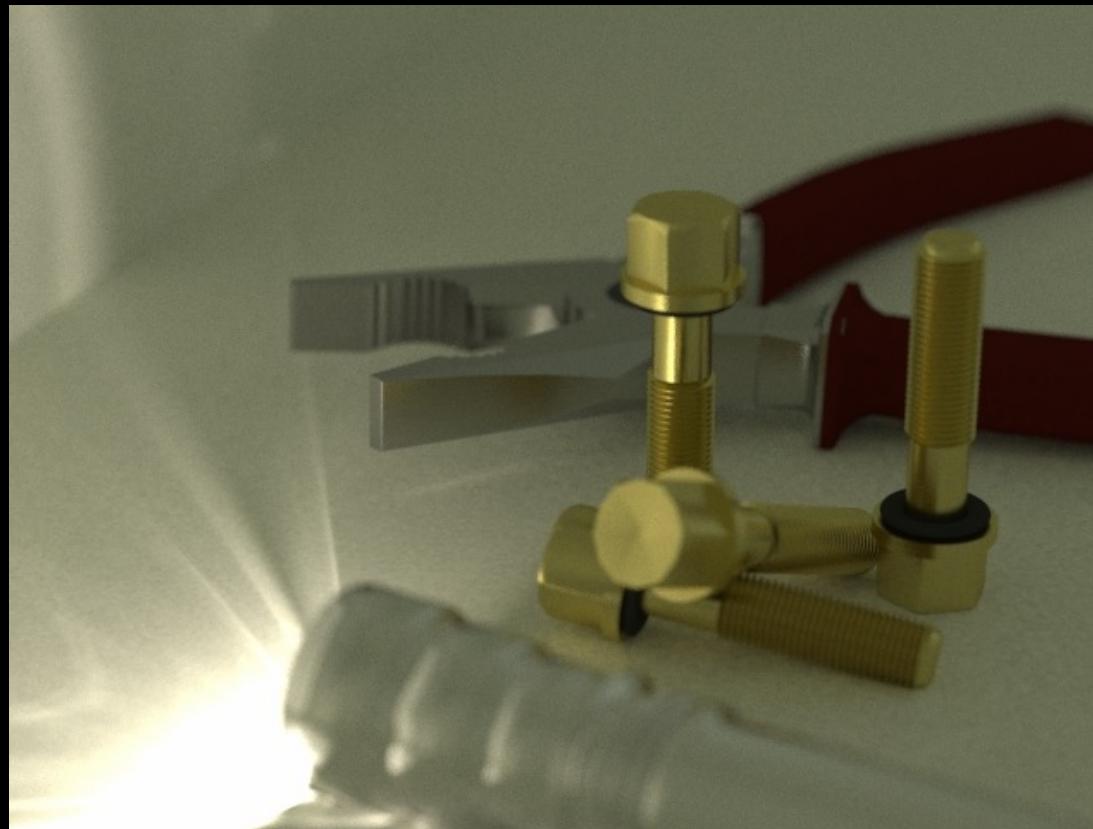
Sample density

<https://mediatech.aalto.fi/publications/graphics/GPT/>

<https://mediatech.aalto.fi/publications/graphics/GMLT/>

Photon Density Estimation

- Estimate densities of path vertices



<http://www.ci.i.u-tokyo.ac.jp/~hachisuka/starpm2013a/>

Bidirectional Path Tracing + Photon Density Estimation

- Automatically combine two algorithms



Monte Carlo Path Integration (RMSE: 0.07836)

Photon Density Estimation (RMSE: 0.04638)

Unified Framework (RMSE: 0.01246)

<http://www.ci.i.u-tokyo.ac.jp/~hachisuka/ups.pdf>