# CMP205: Computer Graphics


## Lecture 12: Ray Tracing III

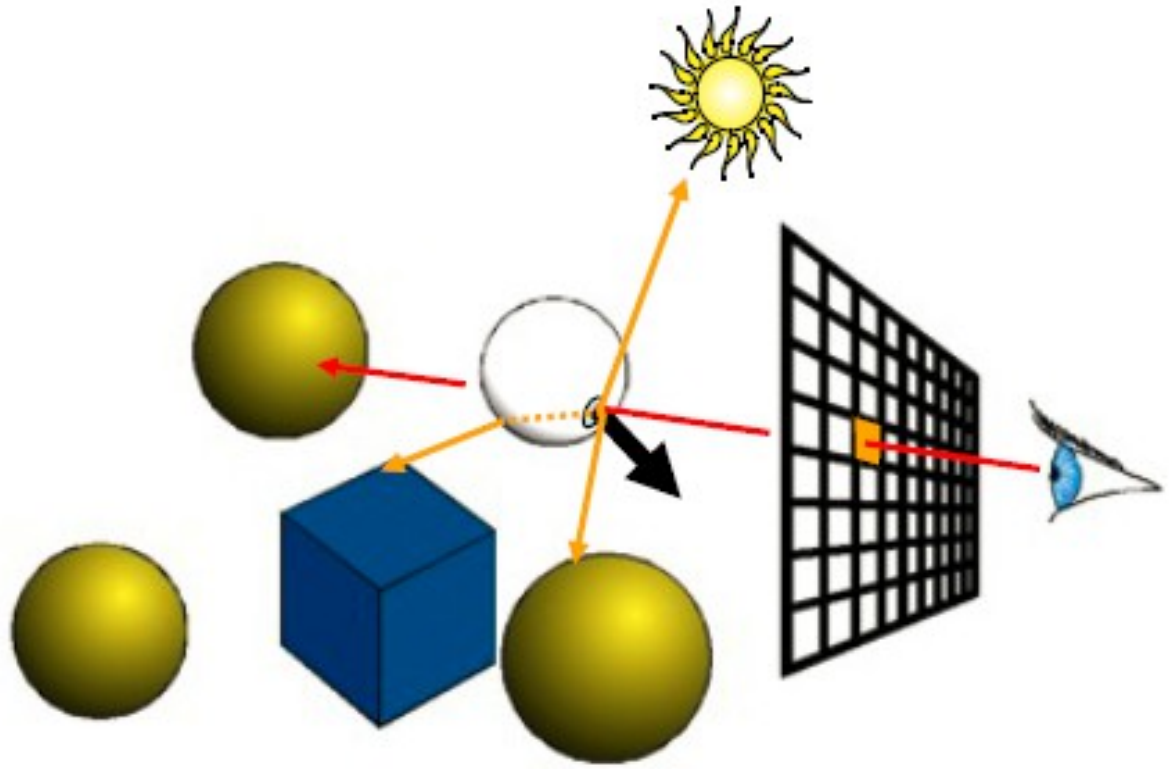
Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

# Agenda

- Ray Tree
- Accelerating Ray Tracing
  - Bounding Volume Hierarchies (BVHs)
  - Binary Space Partitioning Trees (BSP Trees)
  - Uniform Spatial Subdivision
- Constructive Solid Geometry

**Acknowledgment**: Some slides adapted from Steve Marschner, Maneesh Agrawala, and Fredo Durand

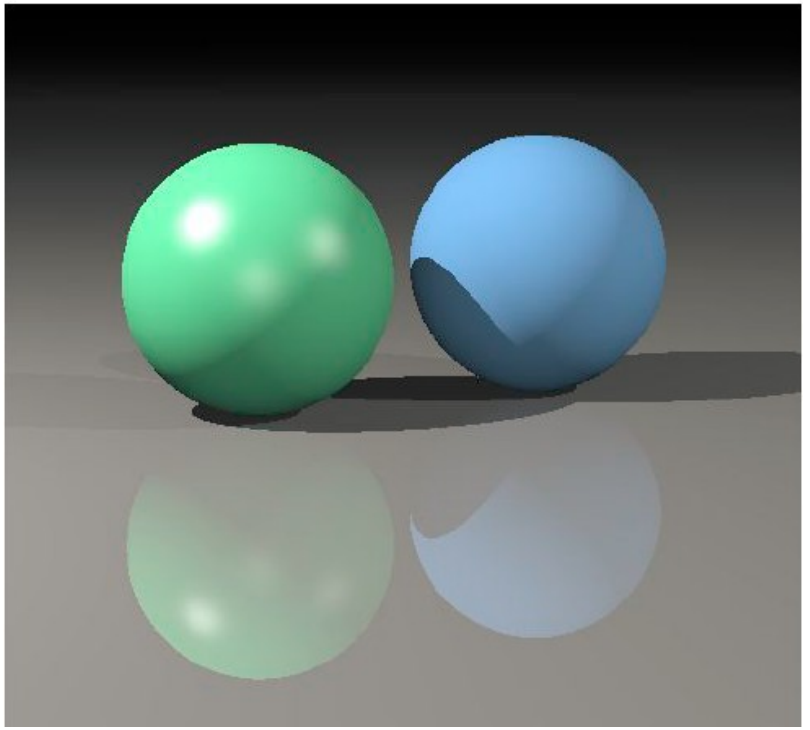
# Recursive Ray Tracing



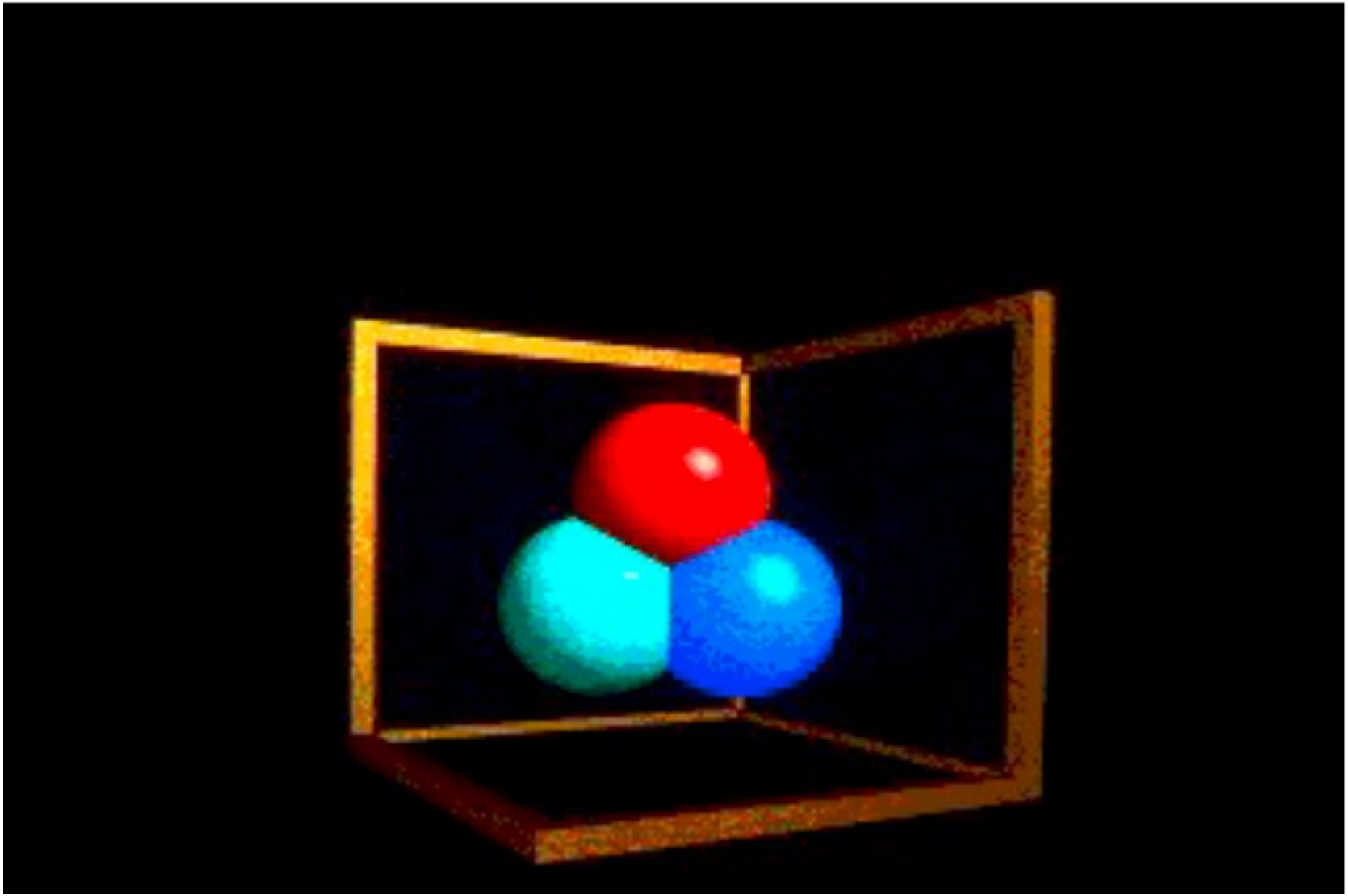
Many many rays !

# Recursive Ray Tracing

```
function ComputeShading(ray, t0, t1)
  Get intersection of ray with scene
  If intersection != null
    Color = ambient
    Get n, h, l, r
    If !blocked(shadowray, ε, ∞)
      Color += kd * max(0, <n,l>) + ks * <h, n>^p
          + ks * ComputeShading(reflected ray, ε, ∞)
  Else
    Color = background
```

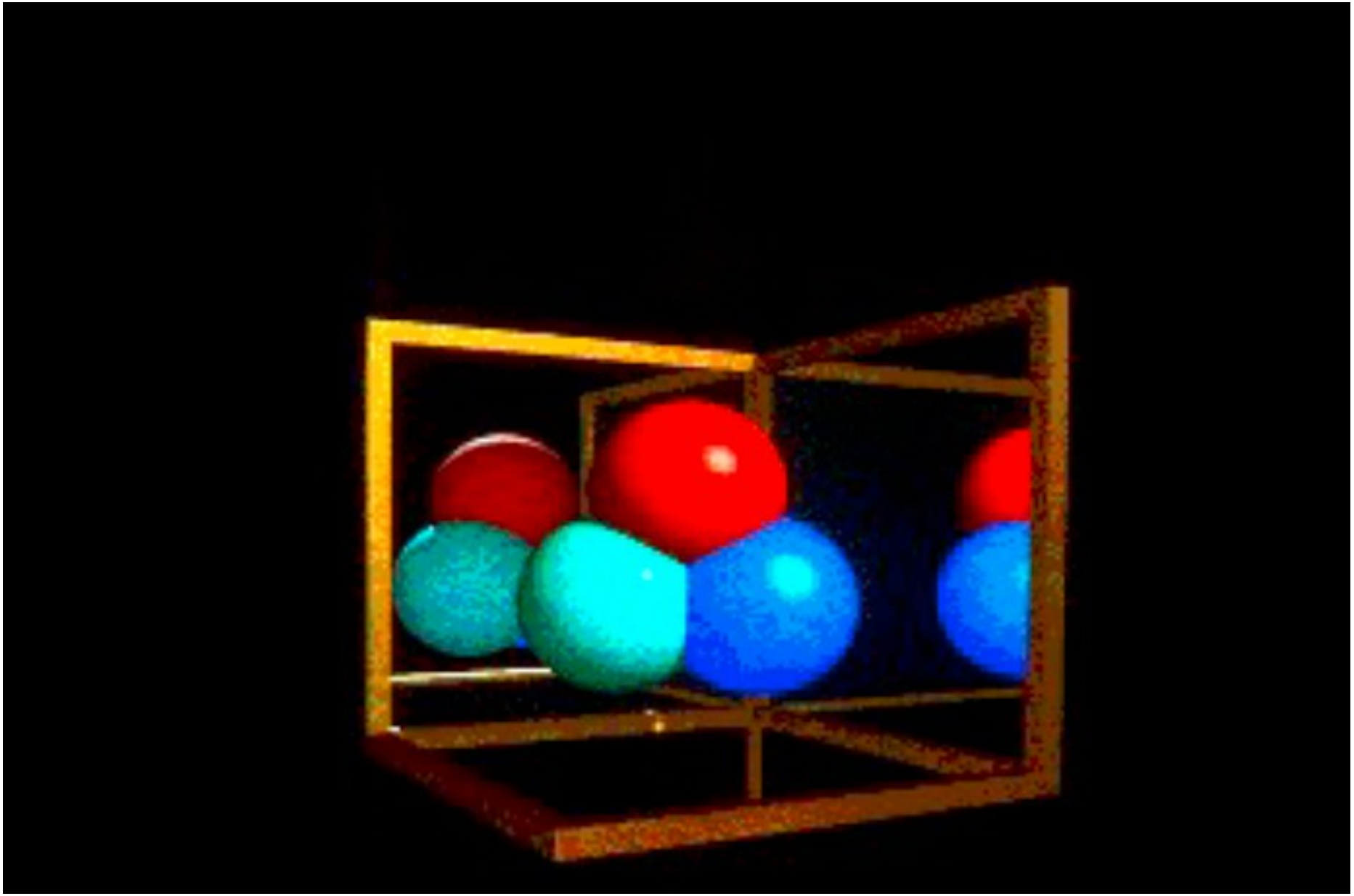


# Recursive Ray Tracing



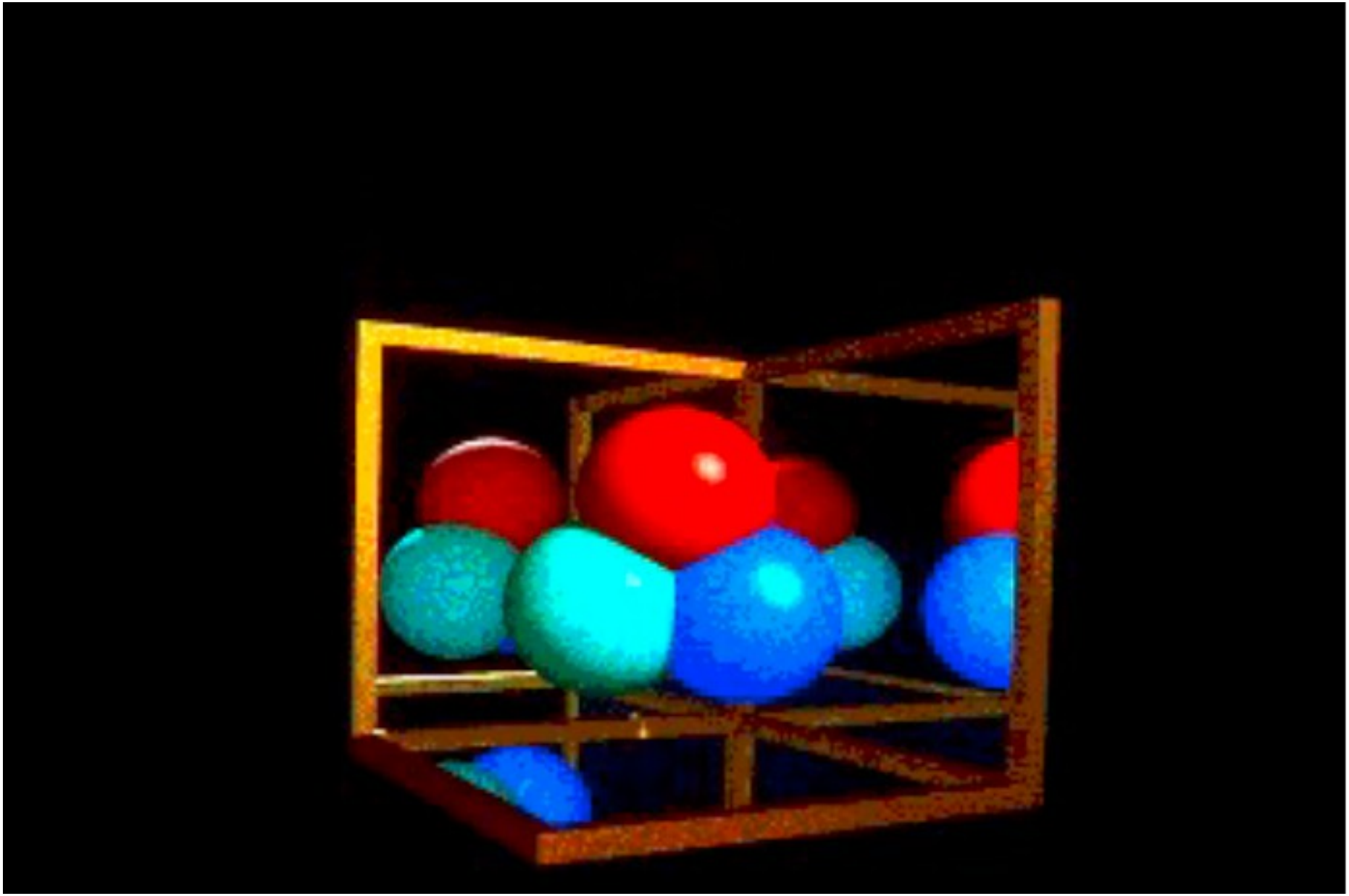
Recursion Depth: 0

# Recursive Ray Tracing



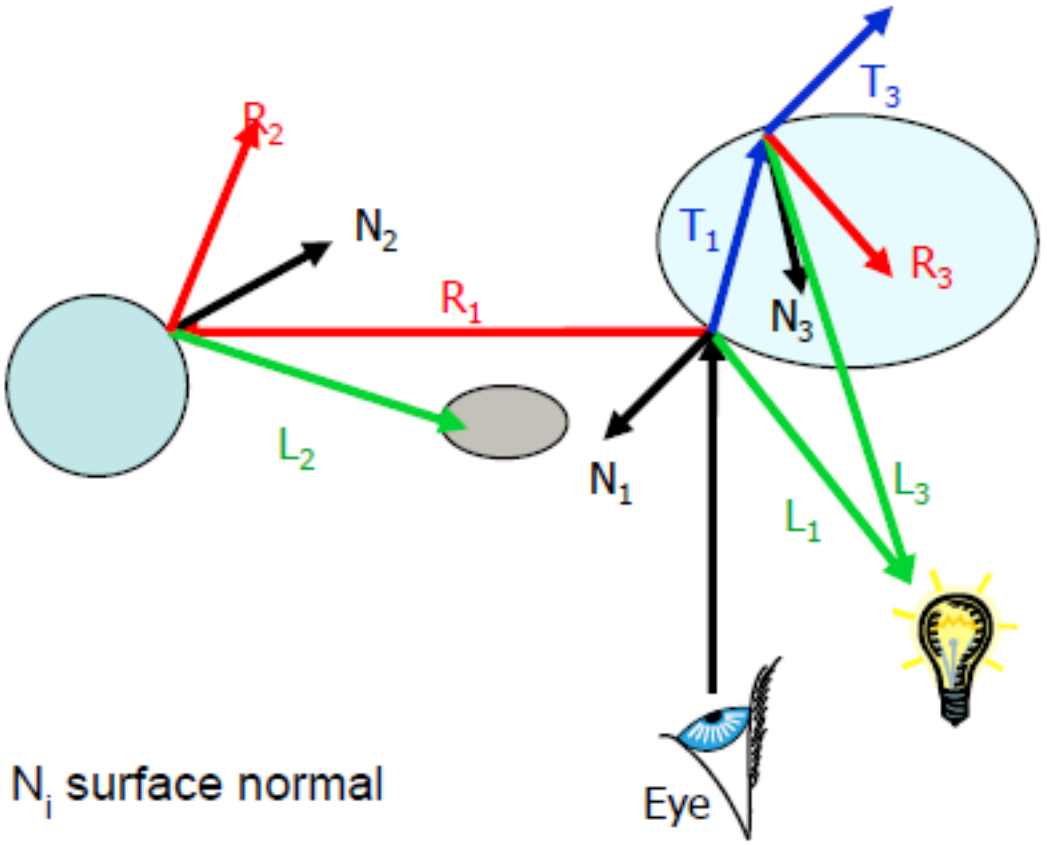
Recursion Depth: 1

# Recursive Ray Tracing



Recursion Depth: 2

# Ray Tree

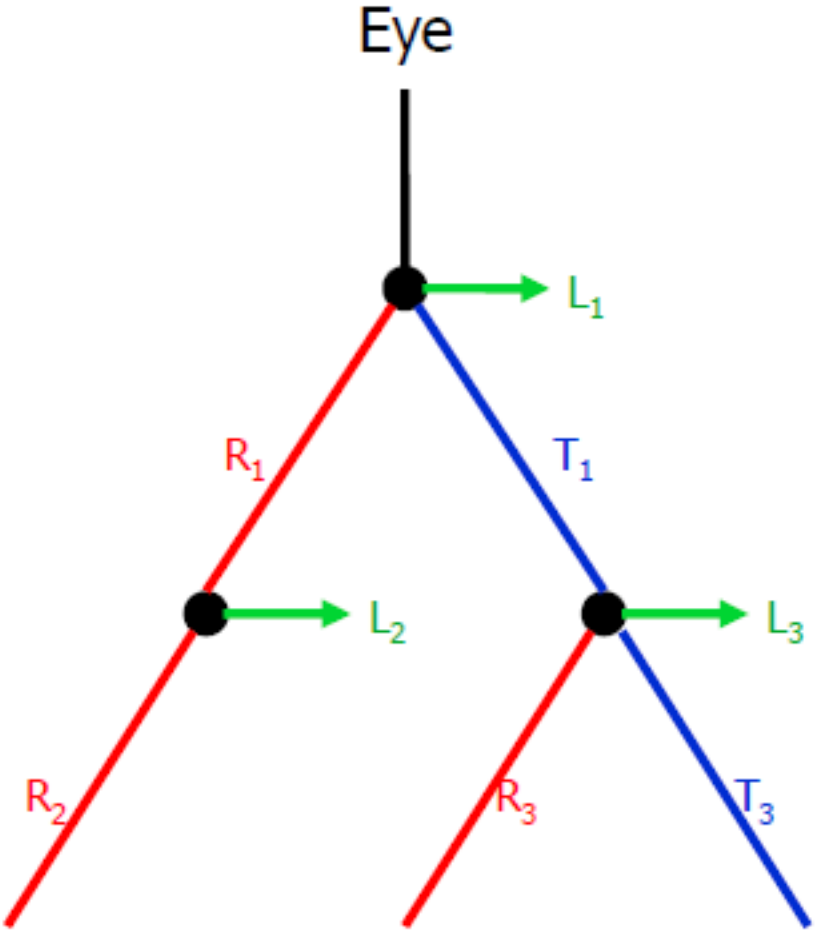



$N_i$ surface normal

$R_i$ reflected ray

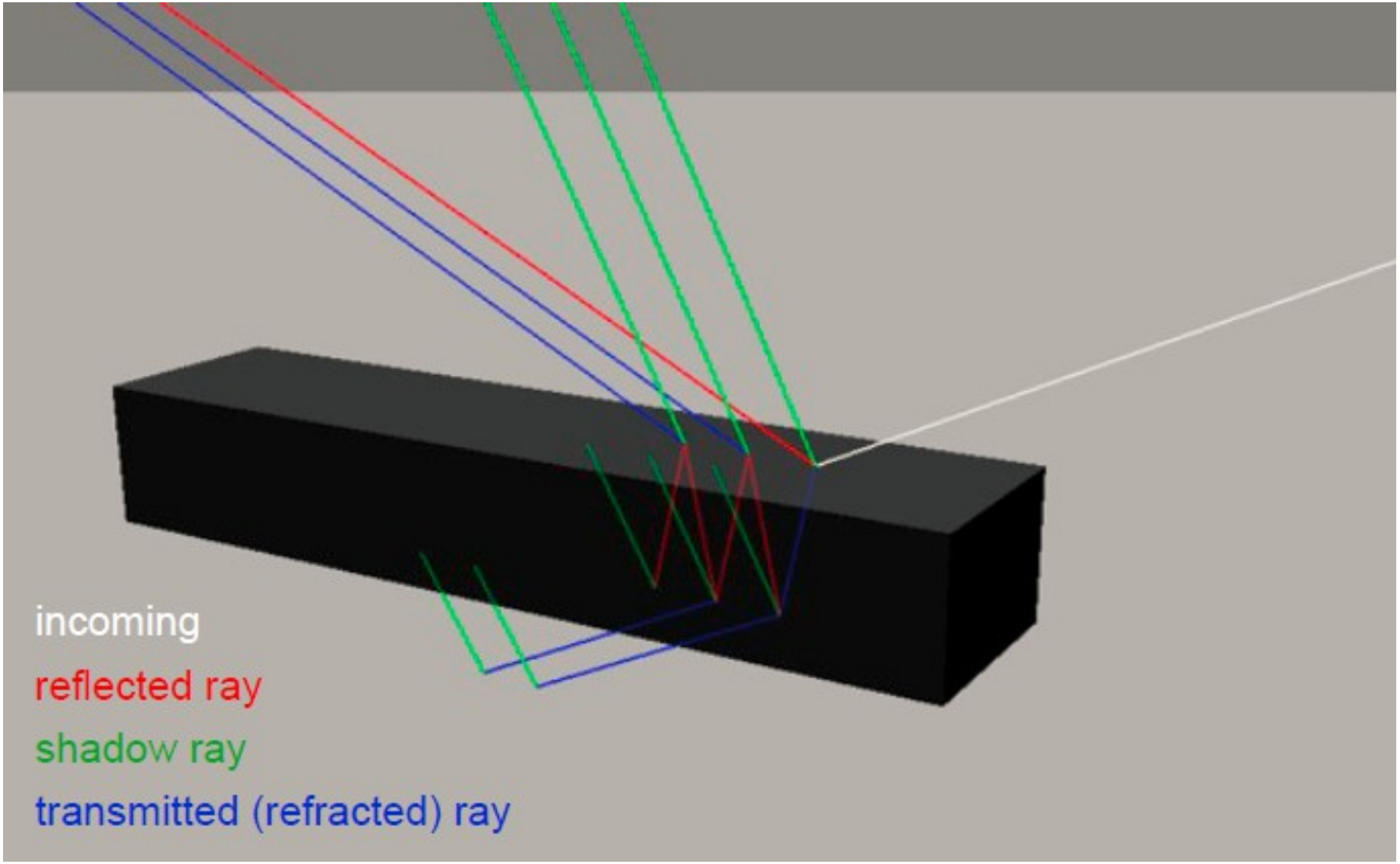
$L_i$ shadow ray

$T_i$ transmitted (refracted) ray




Complexity?

# Ray Tree



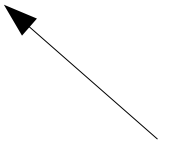

It gets complicated very quickly !

# Ray Tracing Complexity

- Number of pixels
- Intersection cost * Number of objects (primitives)
- Number of shadow rays
- Size of recursive ray tree
- Distribution ray tracing
  - Glossy reflection
  - Glossy refraction
  - Soft shadows
  - …

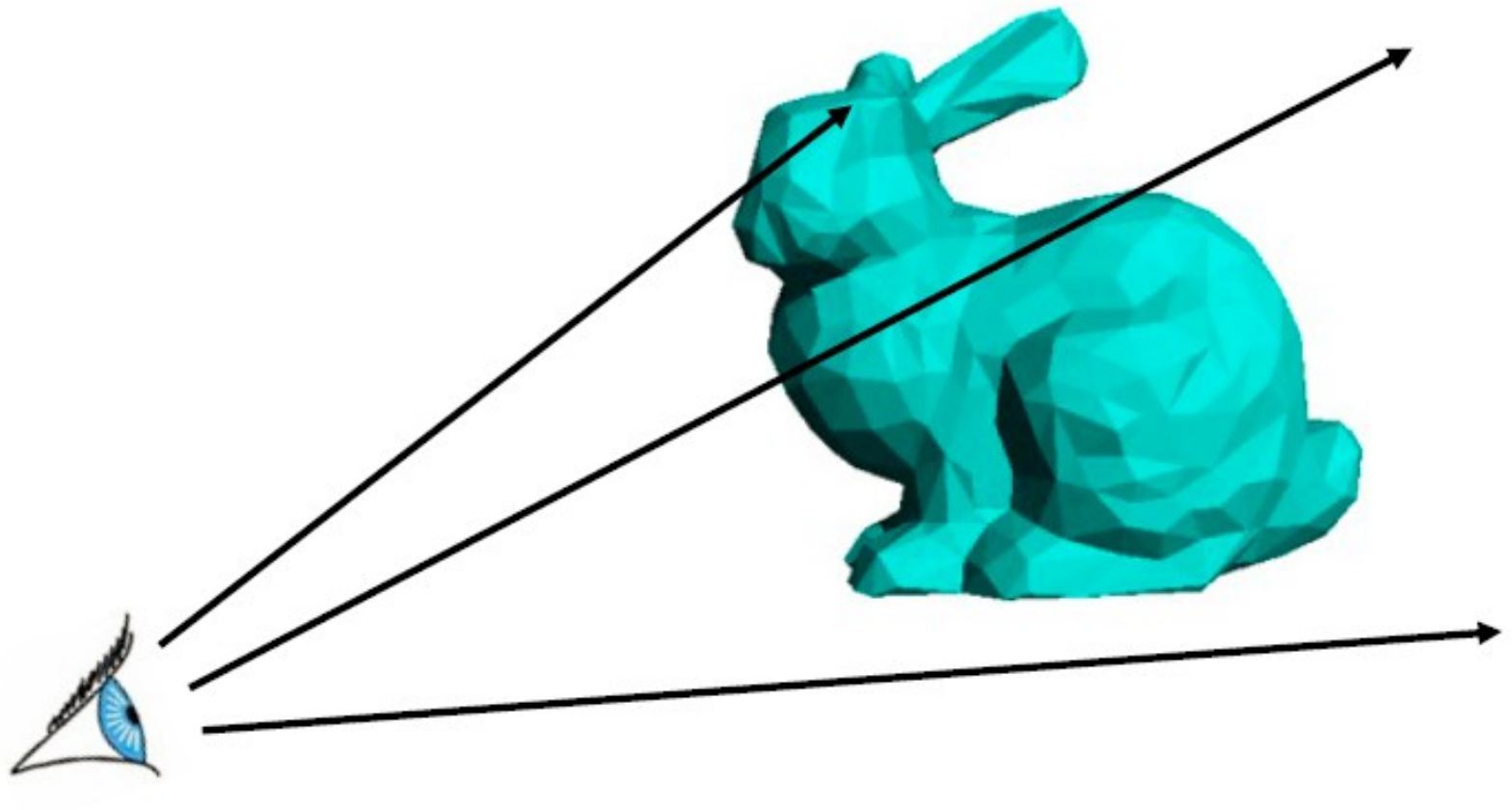
# Ray Tracing Complexity

- Number of pixels
- Intersection cost * Number of objects (primitives)
- Number of shadow rays
- Size of recursive ray tree
- Distribution ray tracing
  - Glossy reflection
  - Glossy refraction
  - Soft shadows
  - …



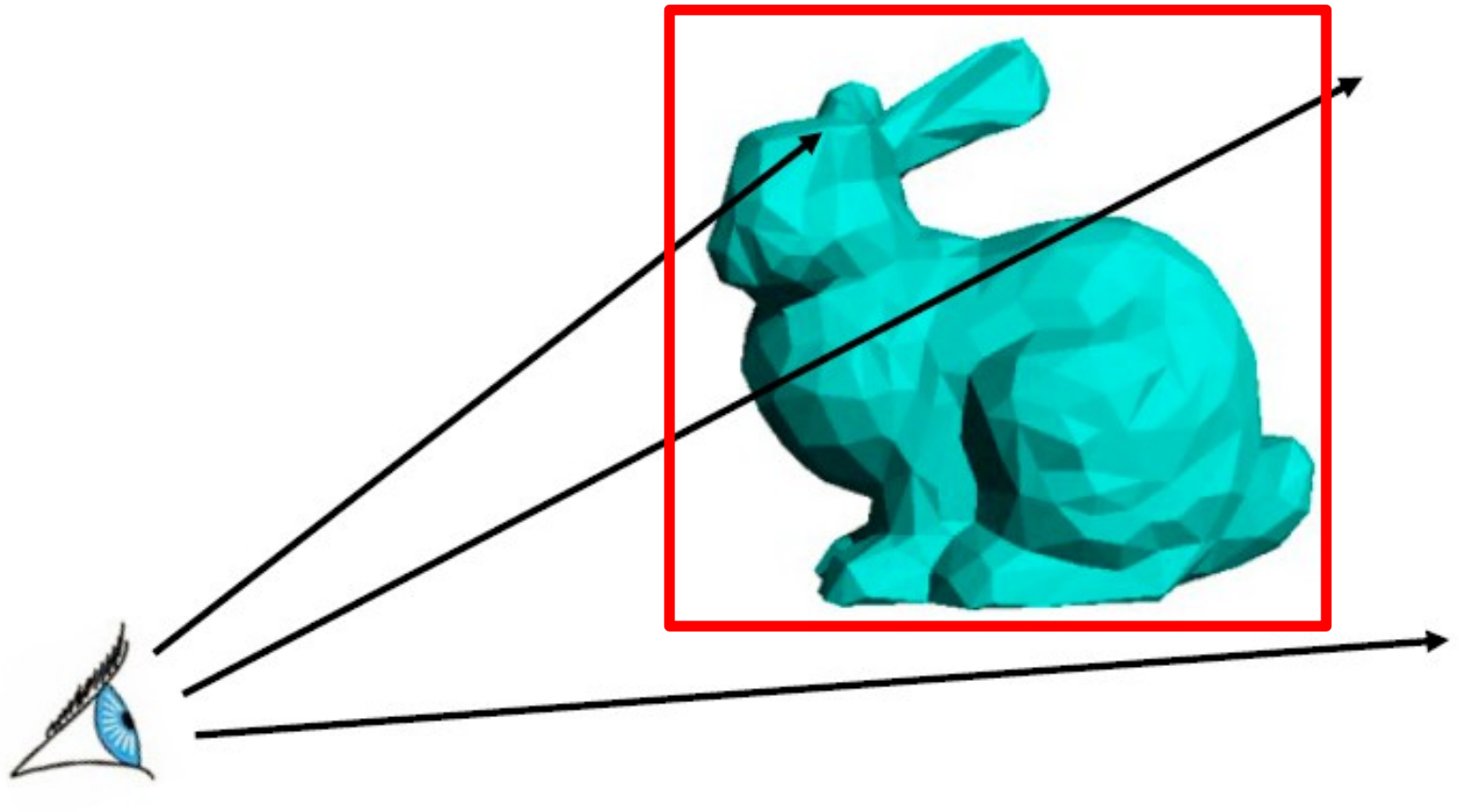
Accelerate by reducing number of intersections required

# Bounding Volumes



Need to test every ray intersection with every triangle

# Bounding Volumes

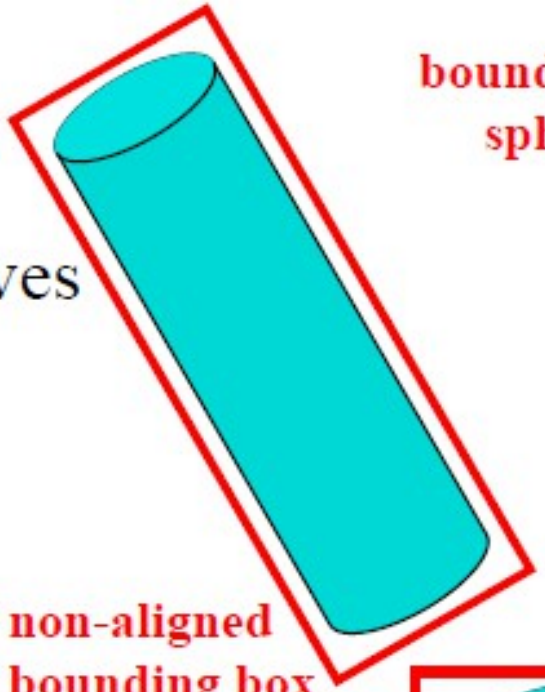


First check intersection with a bounding volume

# Bounding Volumes

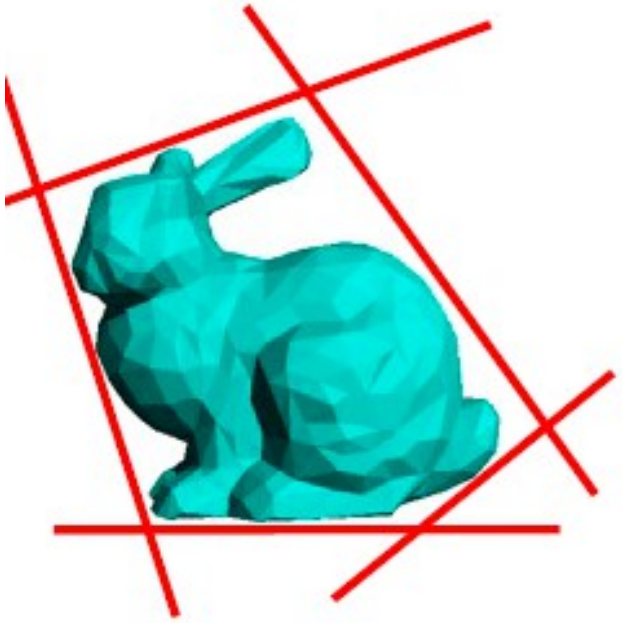
## Desiderata

- Tight →
  avoid false positives
- Fast to intersect

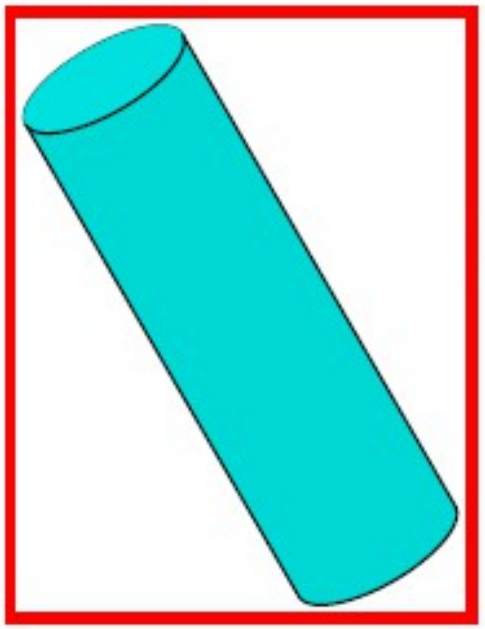


non-aligned
bounding box

bounding
sphere



arbitrary convex region
(bounding half-spaces)



axis-aligned
bounding box

15

# Recall: Ray Box Intersection

Ray equation: $\boldsymbol{p}(t) = \boldsymbol{e} + t\,\boldsymbol{d}$

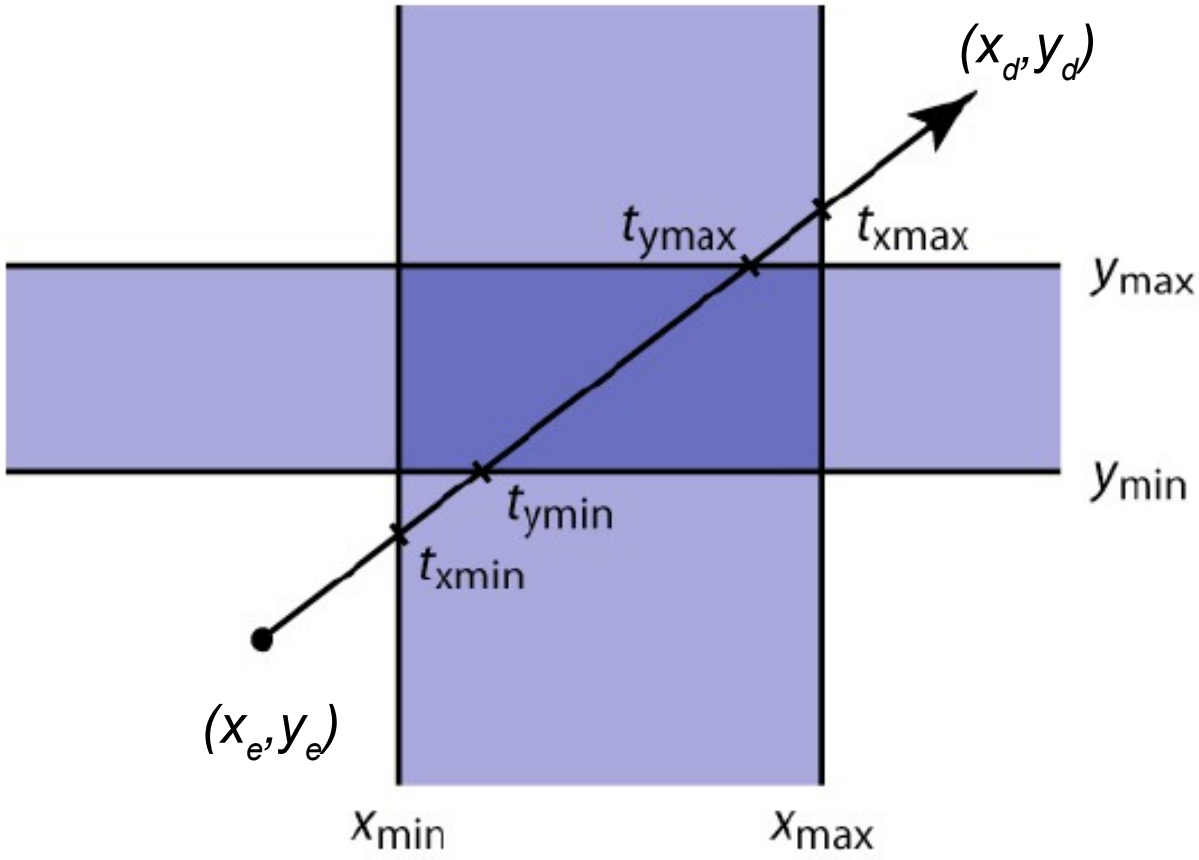
2D Box: $x_p = x_{min}, x_p = x_{max}, y_p = y_{min}, y_p = y_{max}$

$$x_e + t_{xmin}\,x_d = x_{min}$$
$$\rightarrow t_{xmin} = (x_{min} - x_e)/x_d$$

$$x_e + t_{xmax}\,x_d = x_{max}$$
$$\rightarrow t_{xmax} = (x_{max} - x_e)/x_d$$

$$y_e + t_{ymin}\,y_d = y_{min}$$
$$\rightarrow t_{ymin} = (y_{min} - y_e)/y_d$$

$$y_e + t_{ymax}\,y_d = y_{max}$$
$$\rightarrow t_{ymax} = (y_{max} - y_e)/y_d$$



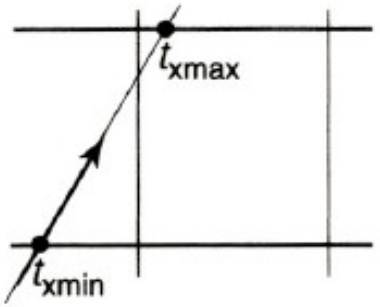
# Recall: Ray Box Intersection

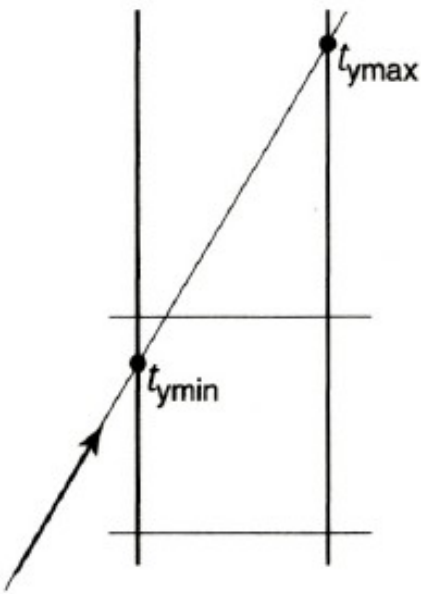
Each intersection gives an interval

We want the last entry point and first exit point !

$$t_{min} = max(t_{xmin}, t_{ymin})$$

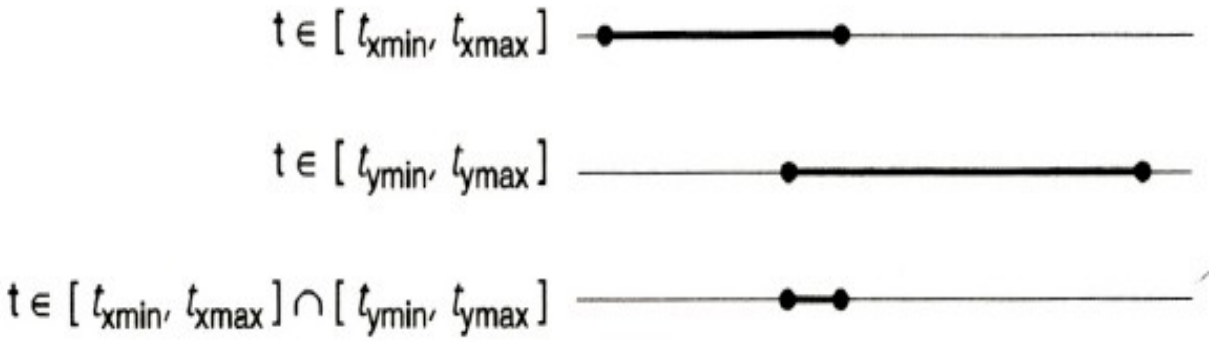
$$t_{max} = min(t_{xmax}, t_{ymax})$$



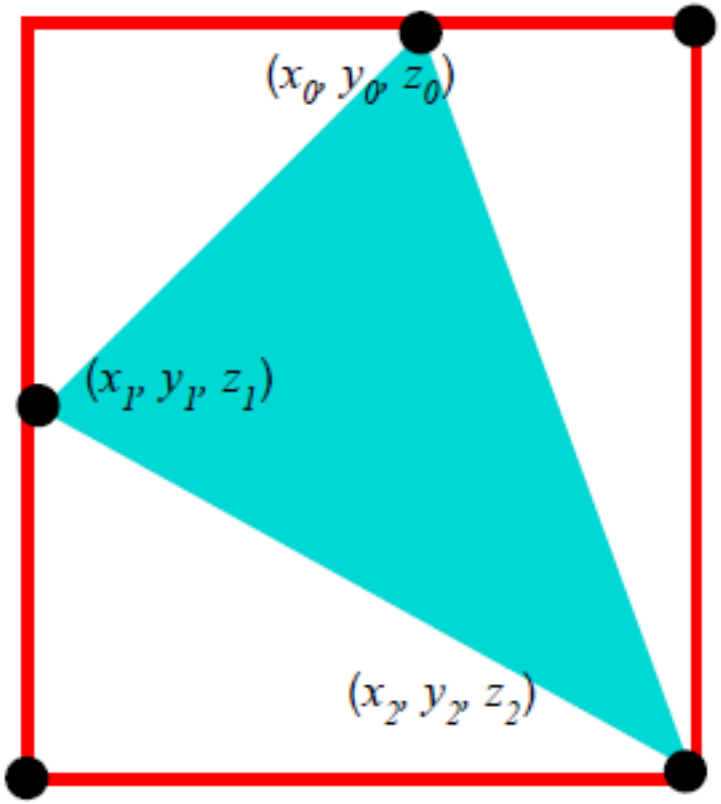




Intersection?
→ $t_{min} < t_{max}$

Intersection point?
→ $p(t_{min})$



# Bounding Box: Triangle



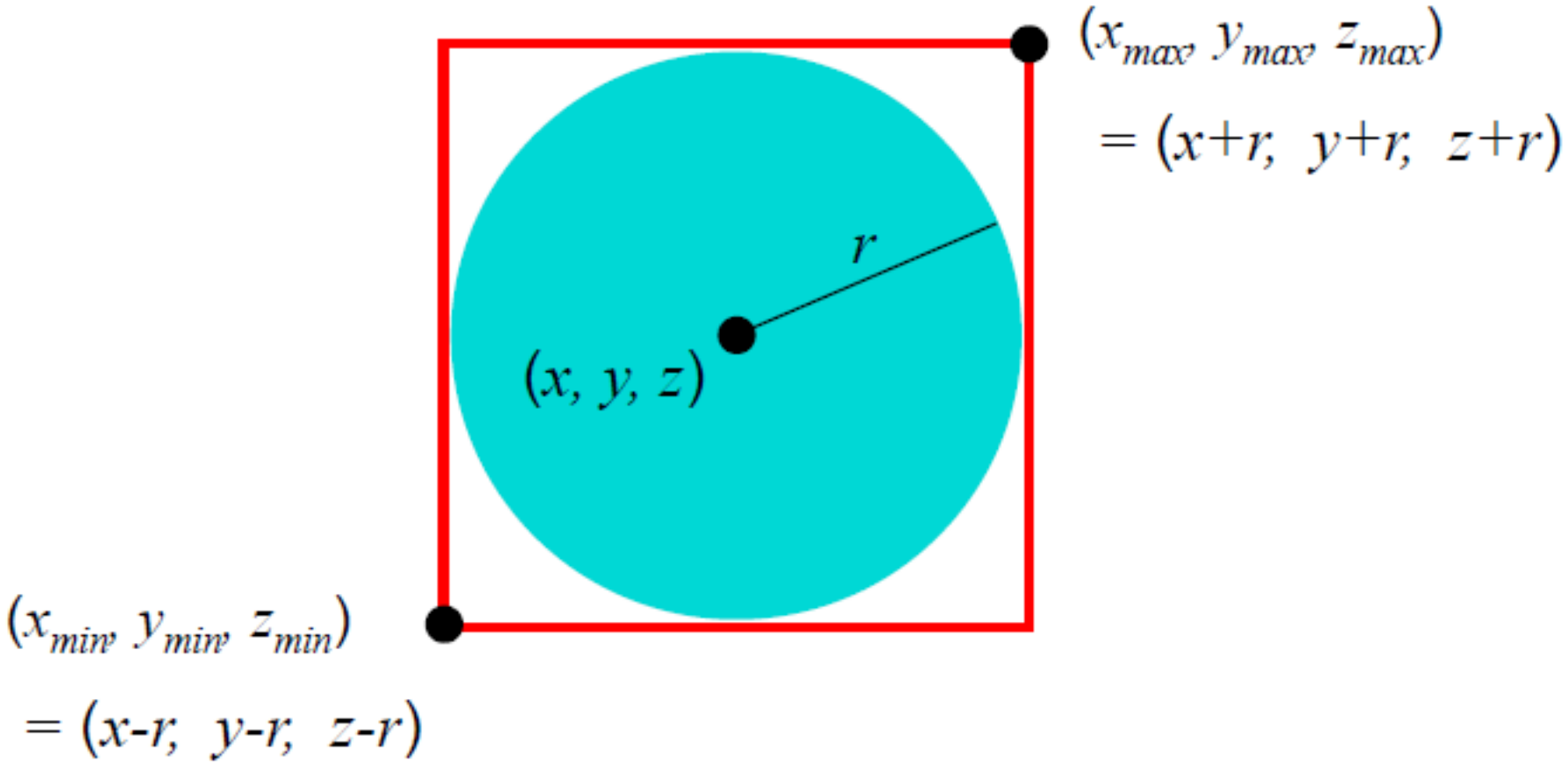

$(x_{max}, y_{max}, z_{max})$

$= (\max(x_0,x_1,x_2),$
$\max(y_0,y_1,y_2),$
$\max(z_0,z_1,z_2))$

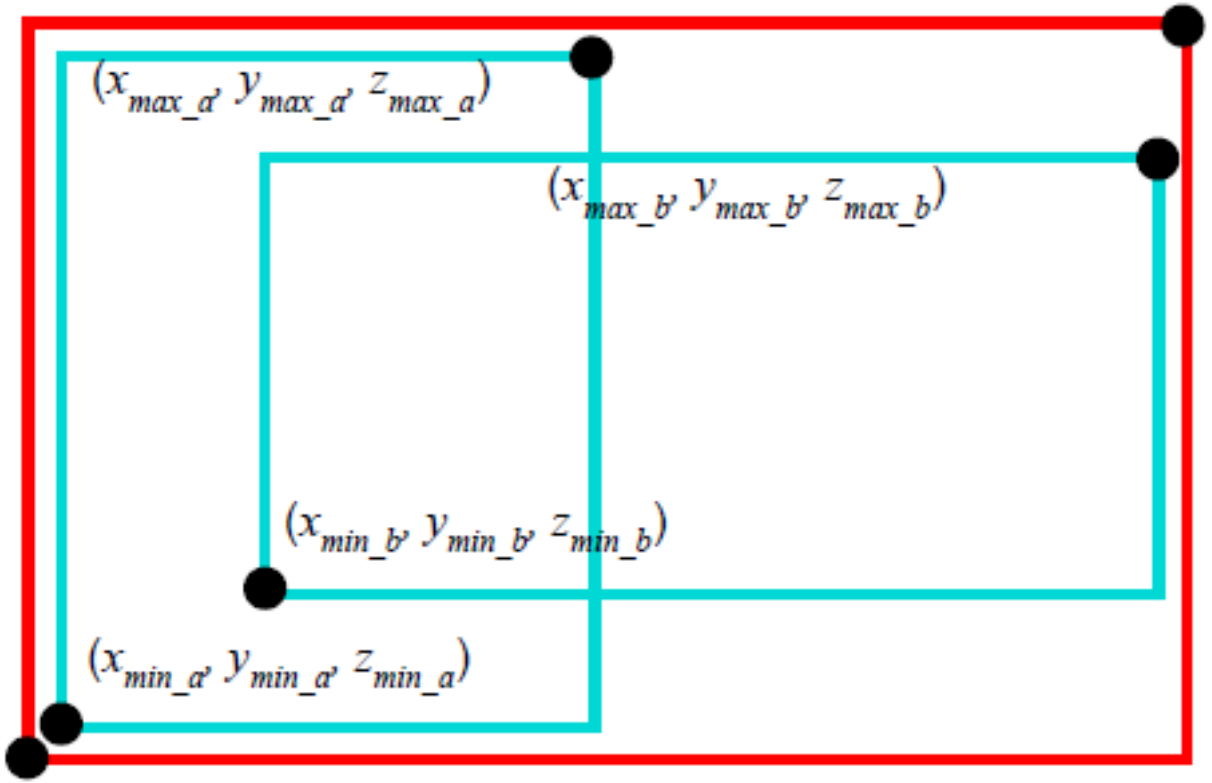
$(x_{min}, y_{min}, z_{min})$

$= (\min(x_0,x_1,x_2),$
$\min(y_0,y_1,y_2),$
$\min(z_0,z_1,z_2))$

# Bounding Box: Sphere



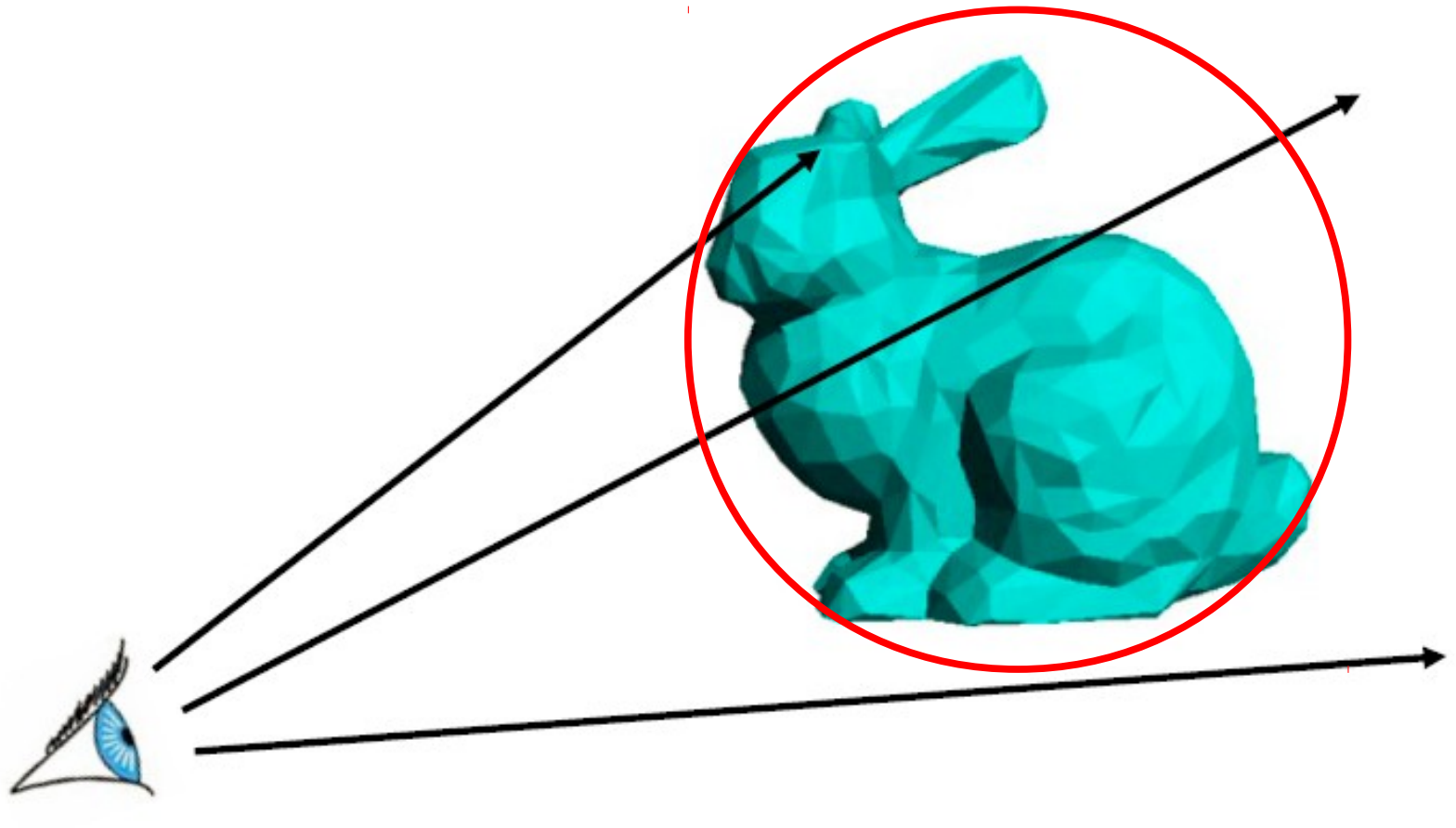
# Bounding Box: Group of Boxes




$$(x_{max}, y_{max}, z_{max}) = (max(x_{max\_a}, x_{max\_b}), max(y_{max\_a}, y_{max\_b}), max(z_{max\_a}, z_{max\_b}))$$

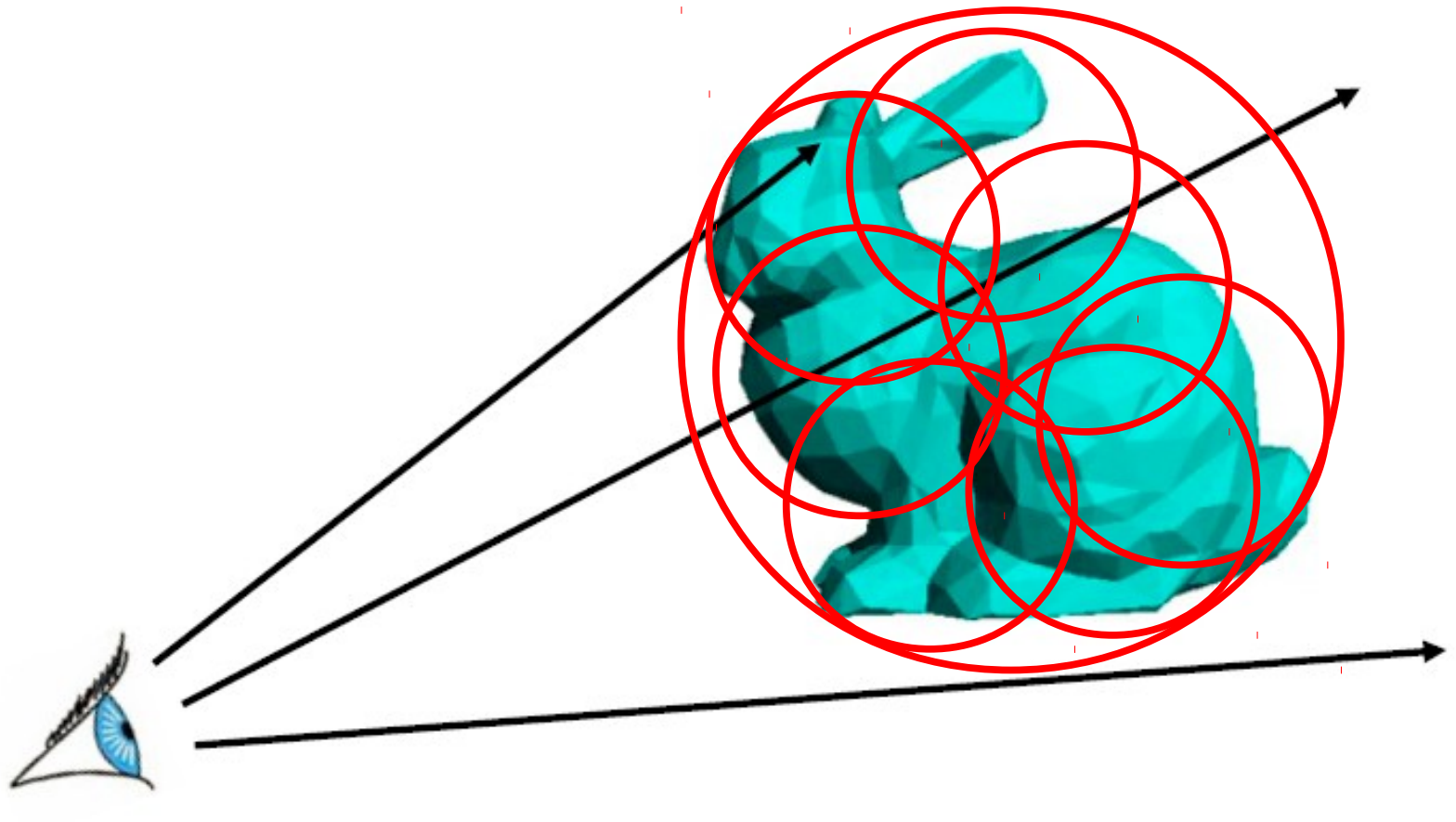
$$(x_{min}, y_{min}, z_{min}) = (min(x_{min\_a}, x_{min\_b}), min(y_{min\_a}, y_{min\_b}), min(z_{min\_a}, z_{min\_b}))$$

# Bounding Volume Hierarchies



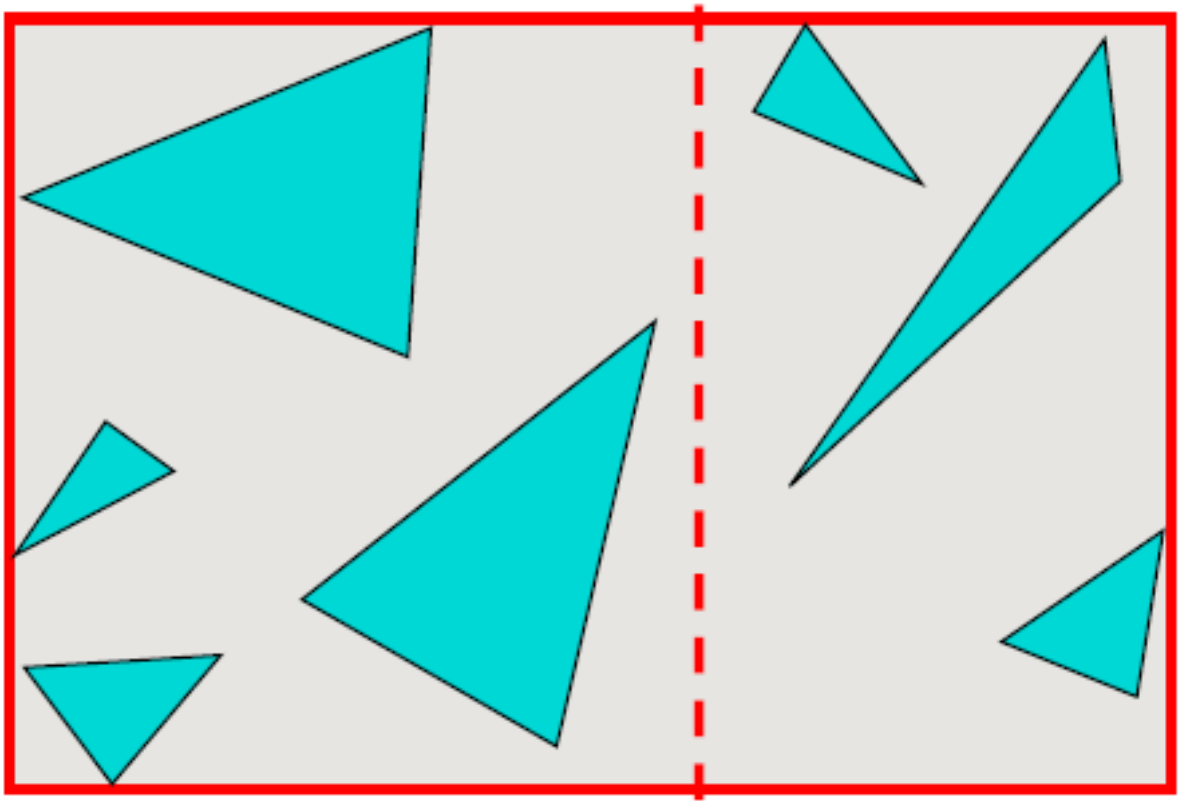
If ray hits bounding volume, still need to intersect all triangles

# Bounding Volume Hierarchies



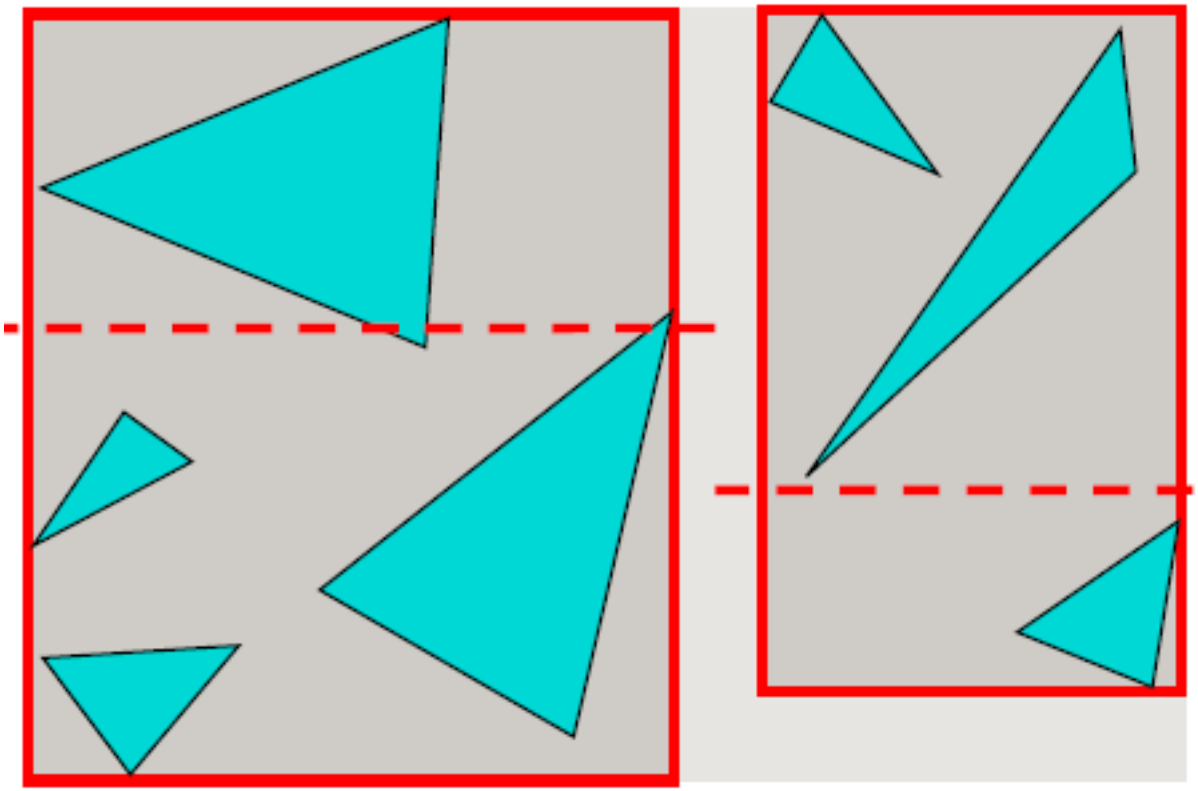
Divide the triangles into groups and build recursive bounding volumes

# Bounding Volume Hierarchies



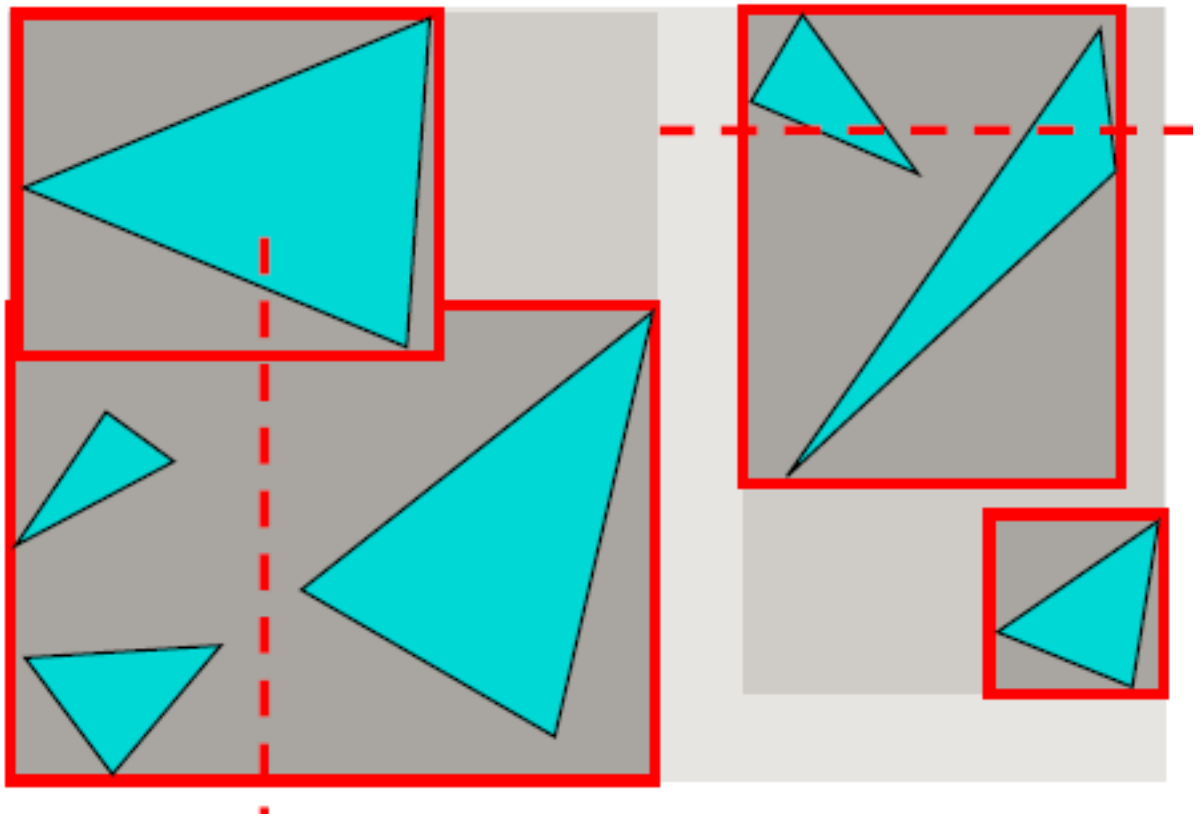
1. Find bounding volume of primitives

# Bounding Volume Hierarchies



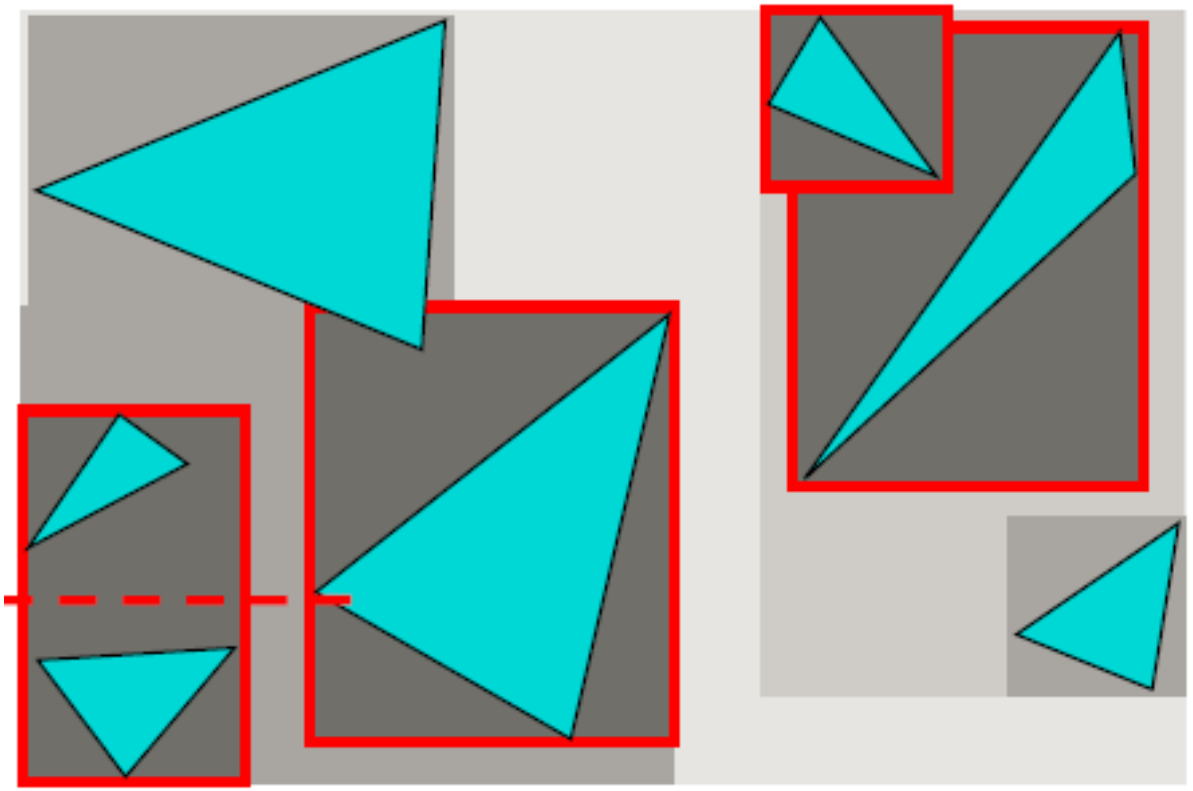
2. Split primitives in two groups, find their BV

# Bounding Volume Hierarchies



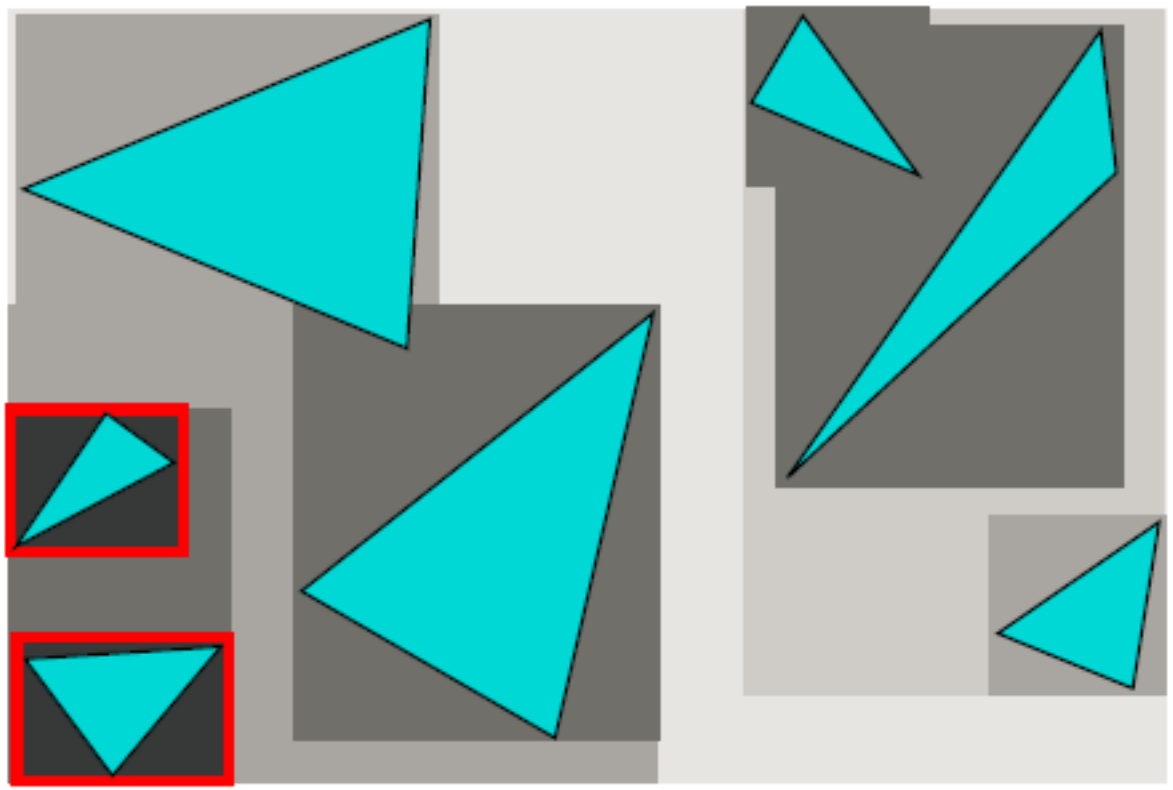
3. Repeat 1 & 2

# Bounding Volume Hierarchies



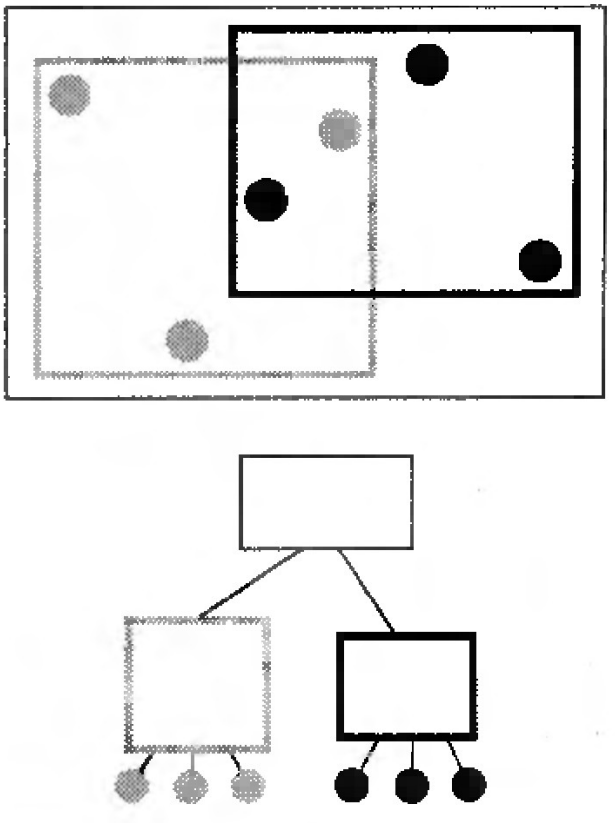
3. Repeat 1 & 2

# Bounding Volume Hierarchies



3. Repeat 1 & 2

# Bounding Volume Hierarchies

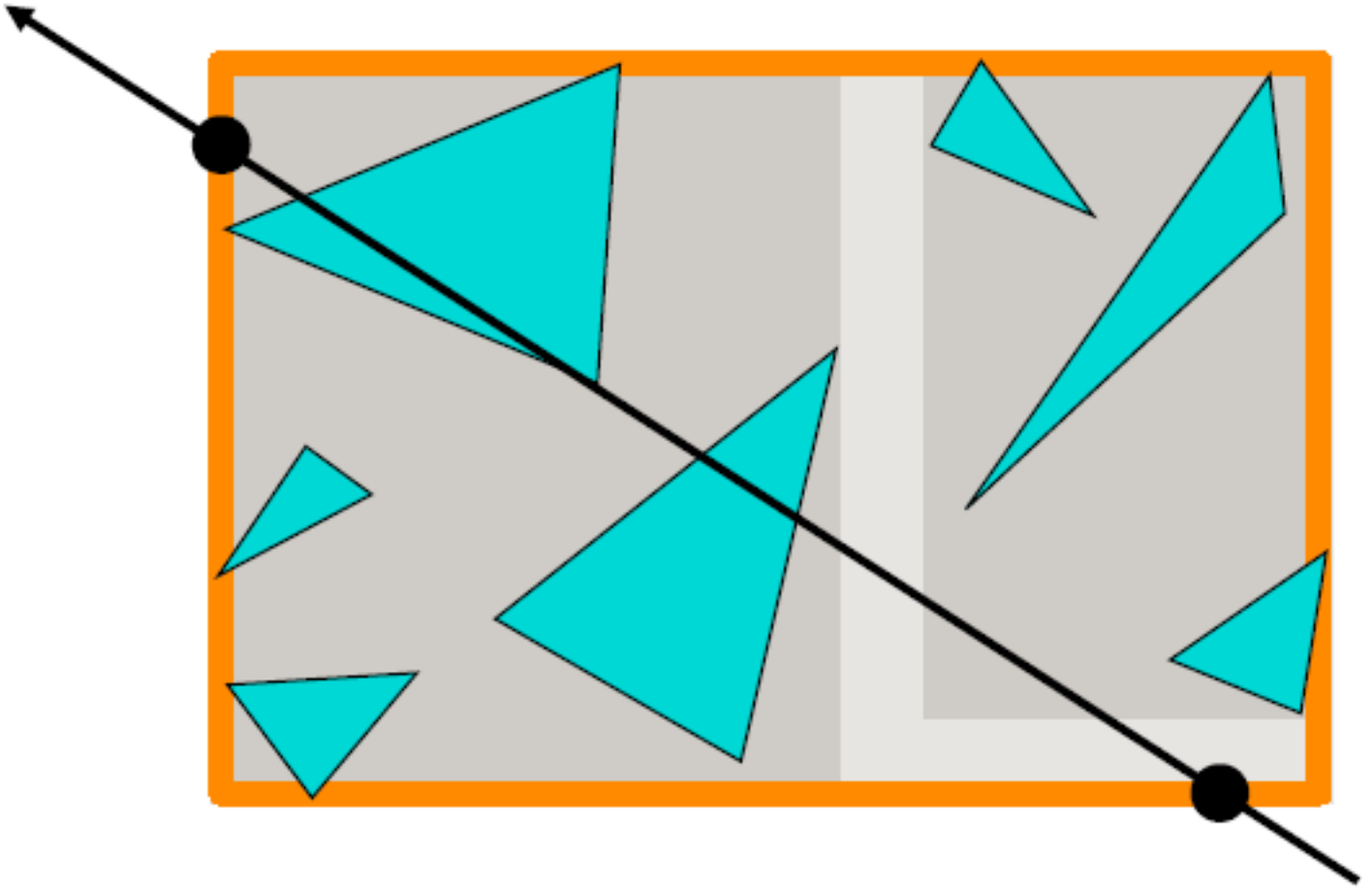


Bounding volumes can be overlapping !

# Bounding Volume Hierarchies

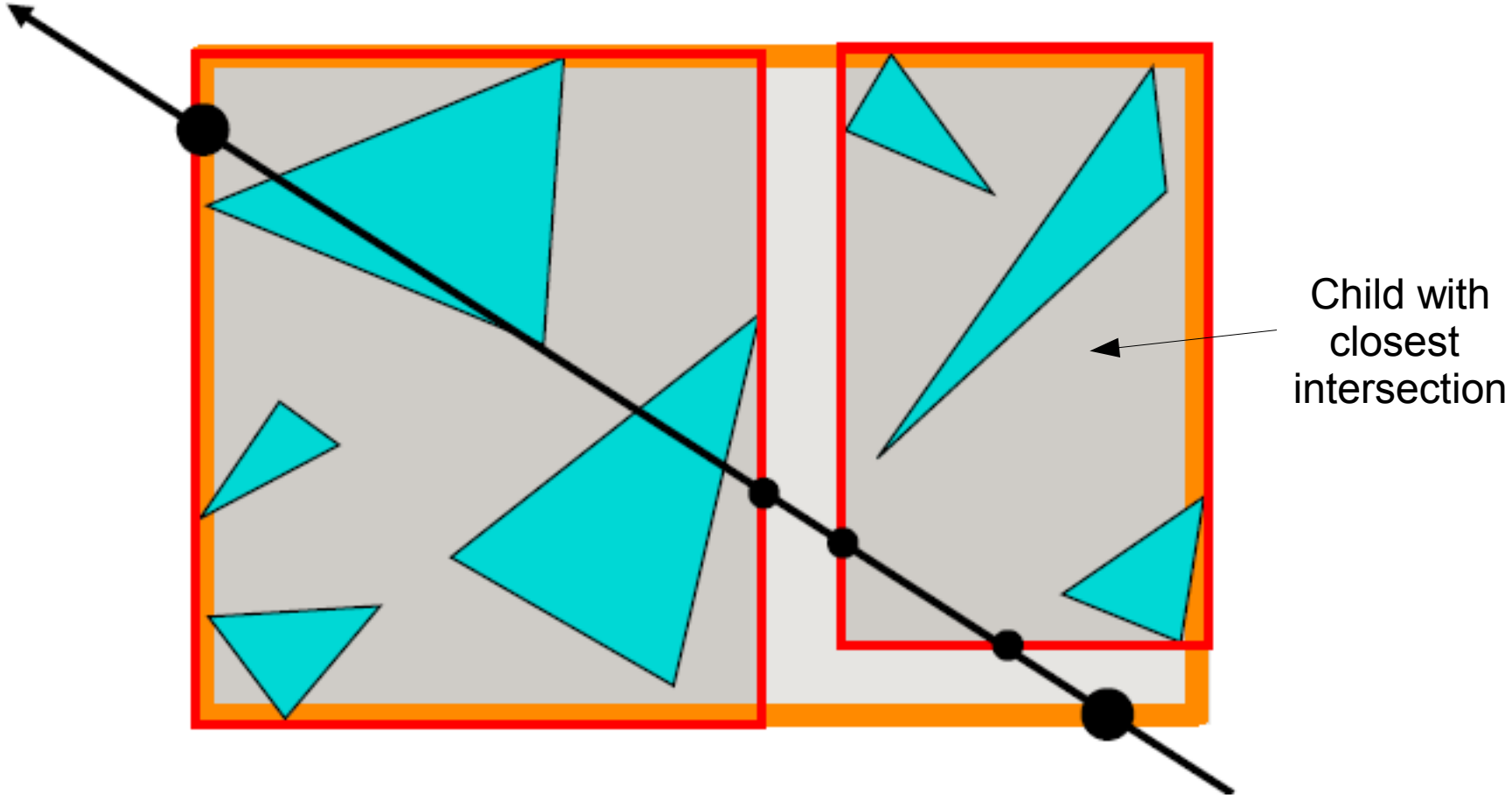
```
struct BVHNode {
  // Bounding Volume
  float xmin, ymin;
  float xmax, ymax;
  float zmin, zmax;

  // Left and right children
  BVHNode *left, *right;

  // Leaf?
  bool isLeaf;

  // List of primitives
  Primitive* list;
}
```

# Ray Intersection: BVH



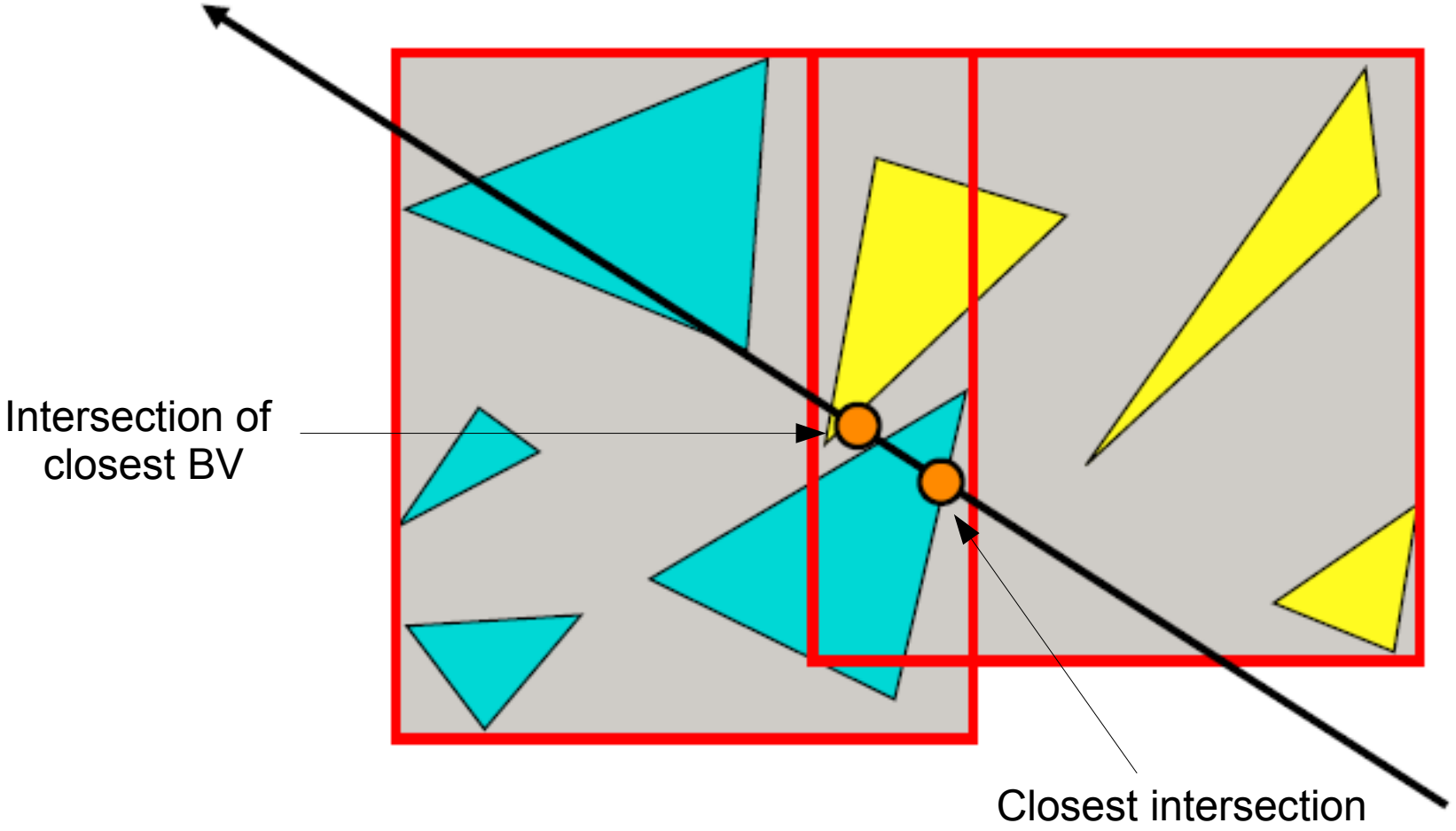
Find $t_{min}$ and $t_{max}$ for the parent node

# Ray Intersection: BVH




Find $t_{min}$ and $t_{max}$ for child nodes and recursive check child with closer intersection first

# Ray Intersection: BVH




Make sure to check the other child and return closest intersection.
BVs overlap!

# Ray Intersection: BVH

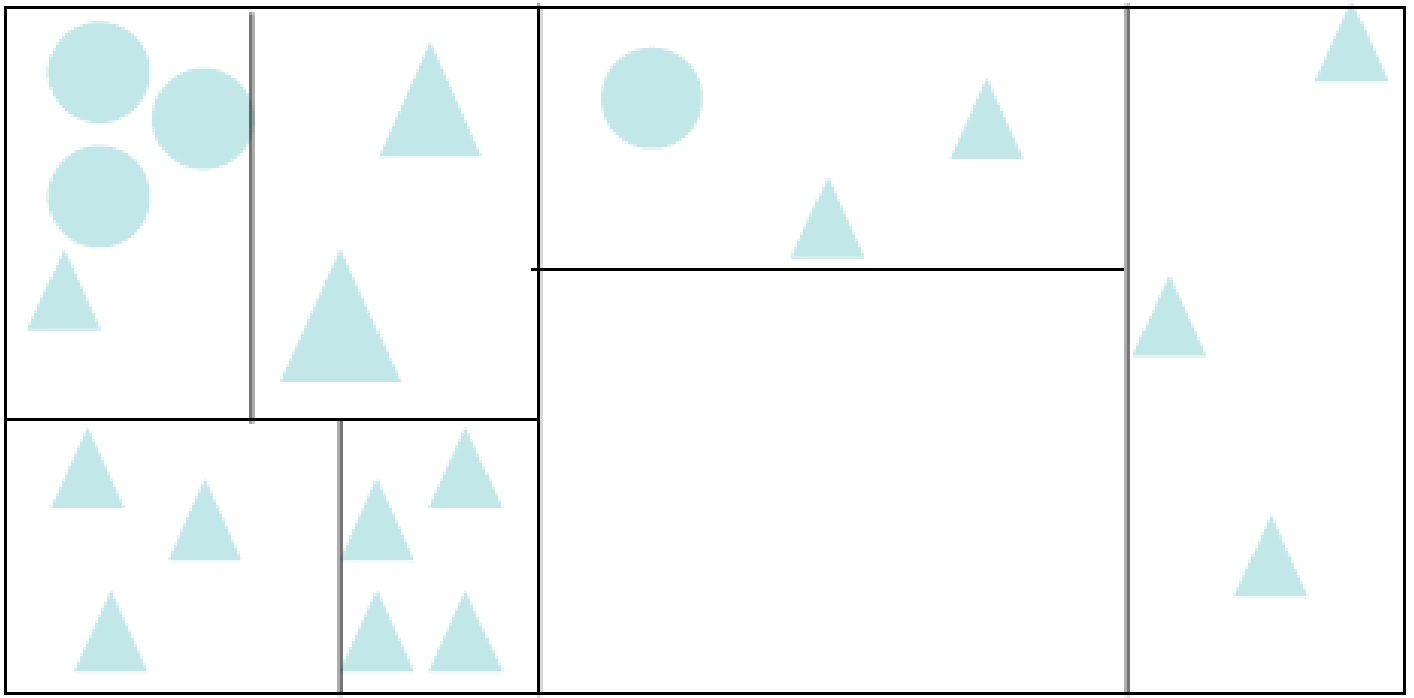
```
bool intersection(ray, node, t0, t1, record) {
  if (node.isLeaf)
    return intersection(ray, list, t0, t1, record);

 if (!bbox.intersect(ray))
    return false;

  left_hit = intersection(ray, node.left, t0, t1, lrecord);
  right_hit = intersection(ray, node.right, t0, t1, rrecord);

  if (left_hit && right_hit)
    if (lrecord.t < rrecord.t)
      record = lrecord;
    else
      record = rrecord;
    return true;
  else if (left_hit)
    record = lrecord;
    return true;
  else if (right_hit)
    record = rrecord;
    return true;
  else
    return false;
}
```

# Bounding Volume Hierarchies

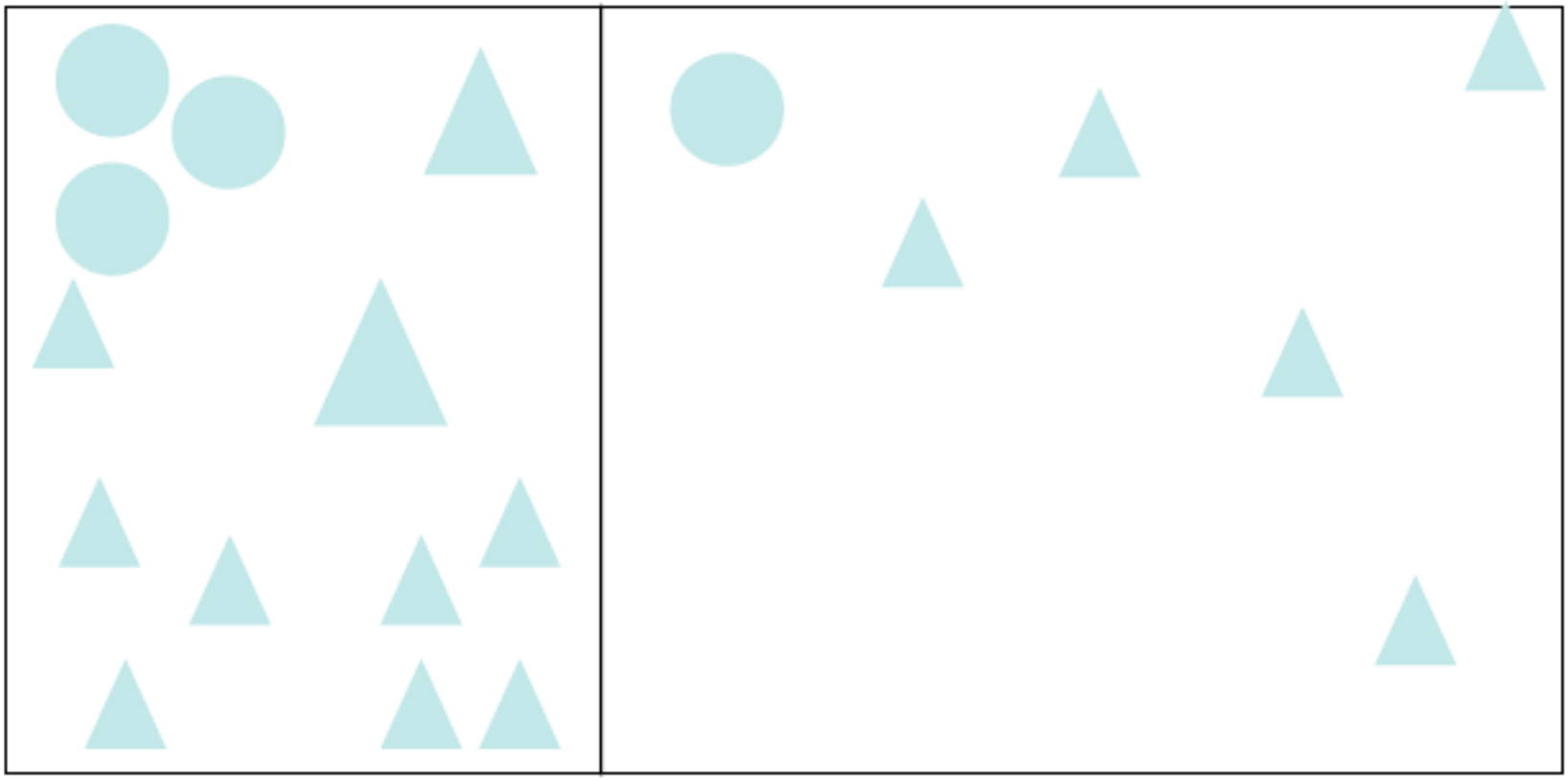
- Pros
  - Easy to build
  - Easy to intersect
- Cons
  - Not so easy to find a split
  - Bad splits result in bad performance

# Binary Space Partitioning (BSP) Trees

- Binary Tree
- Axis aligned splits
- Non-overlapping nodes (unlike BVHs)
- A.K.A. Kd-Trees

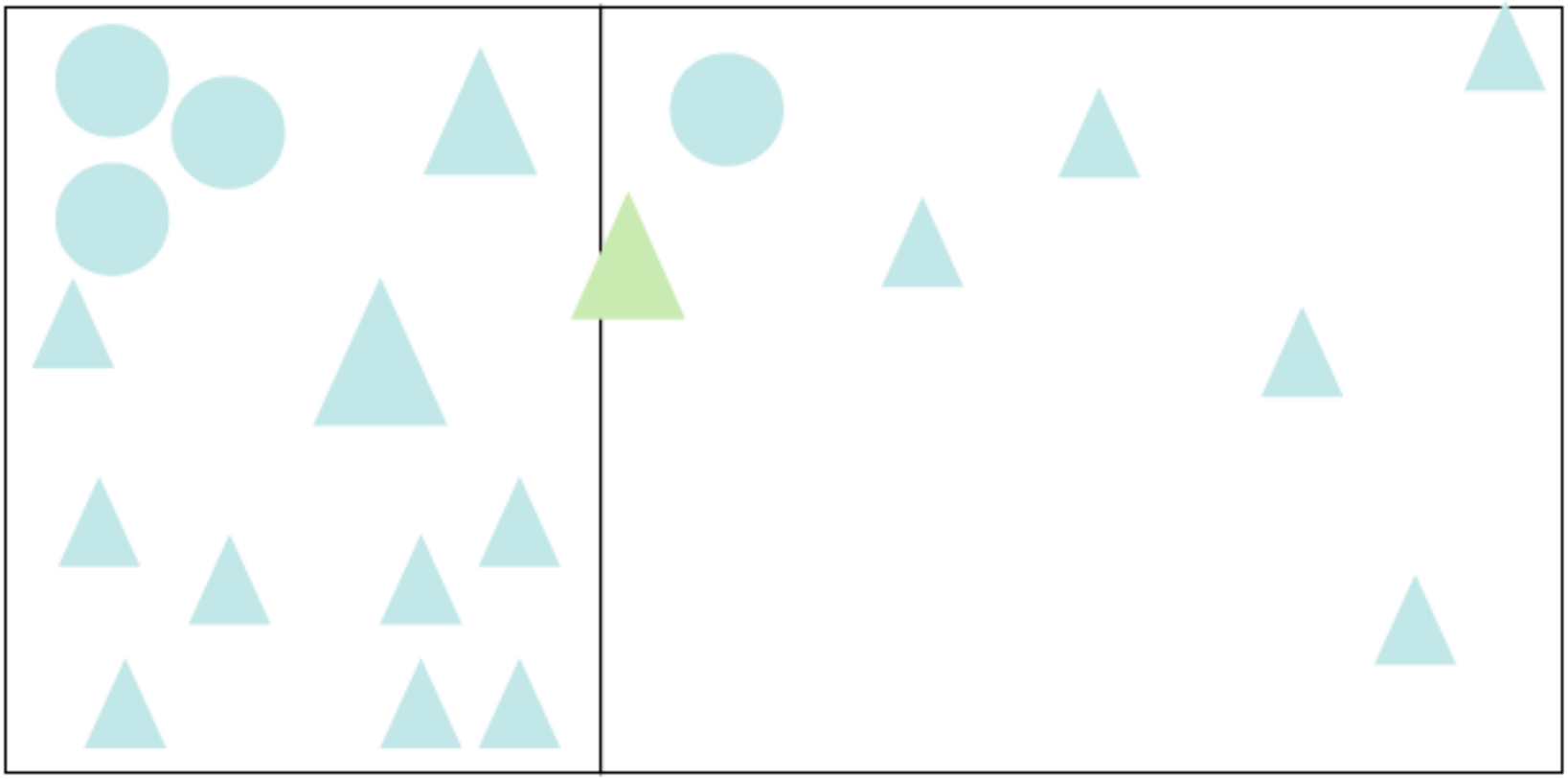


# BSP Tree Construction



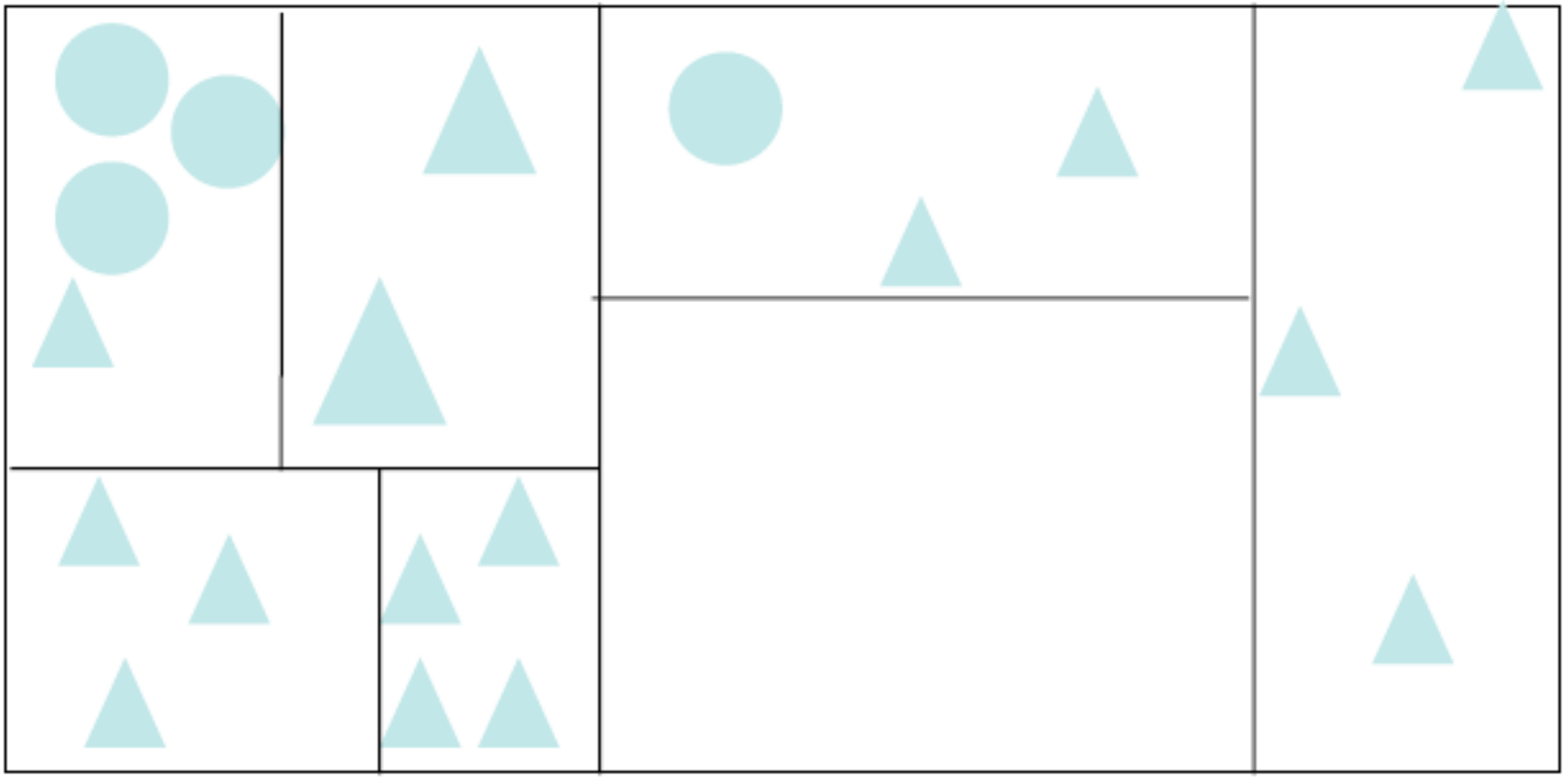
1. Start with bounding volume
2. Choose a dimension to split on
3. Choose a value to split on
4. Distribute primitives to each side

# BSP Tree Construction



Primitive cuts through plane → goes to both sides

# BSP Tree Construction

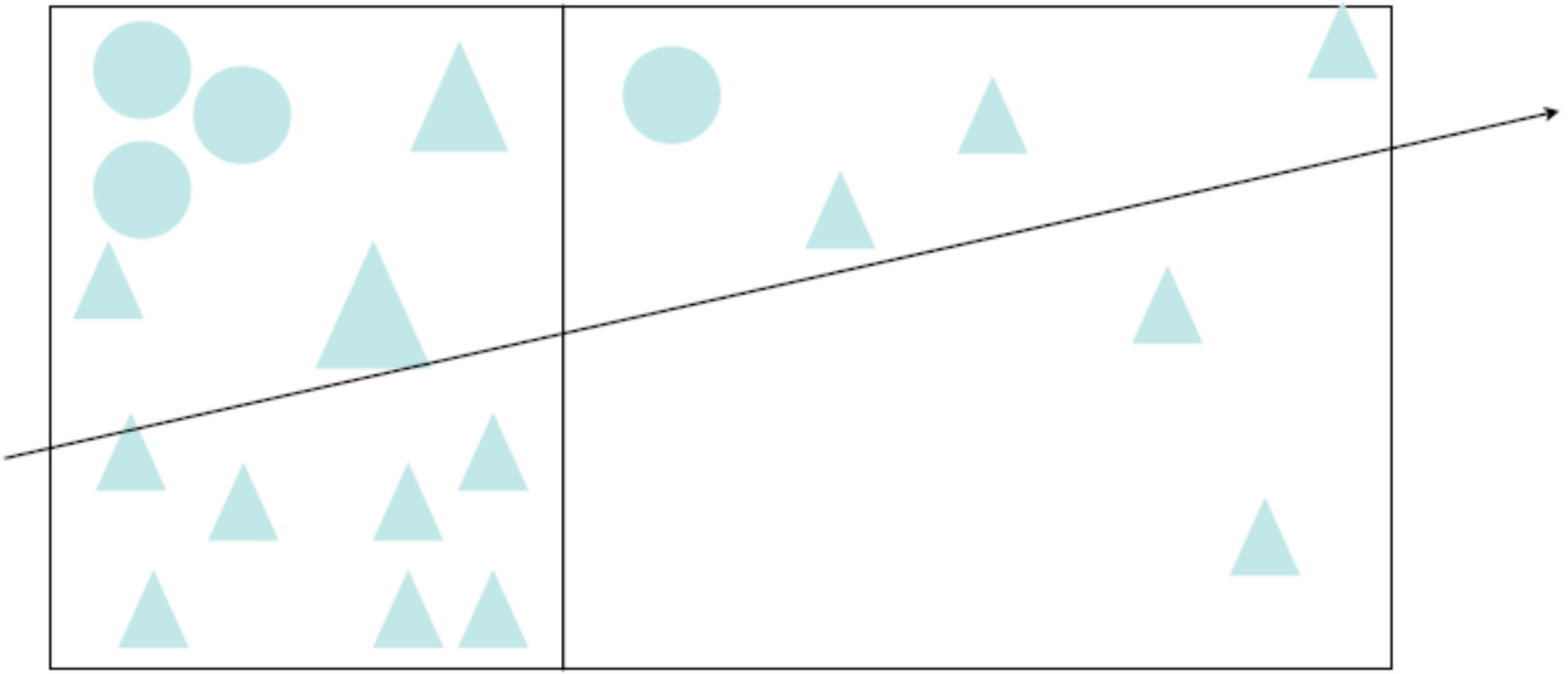


5. Repeat recursively

# BSP Tree Construction

```
struct BSPNode {
  // Split
  int dimension;
  float value;

  // Left and right children
  BSPNode* left, right;

  bool isLeaf;

  // List of primitives
  Primitive* list;
}
```

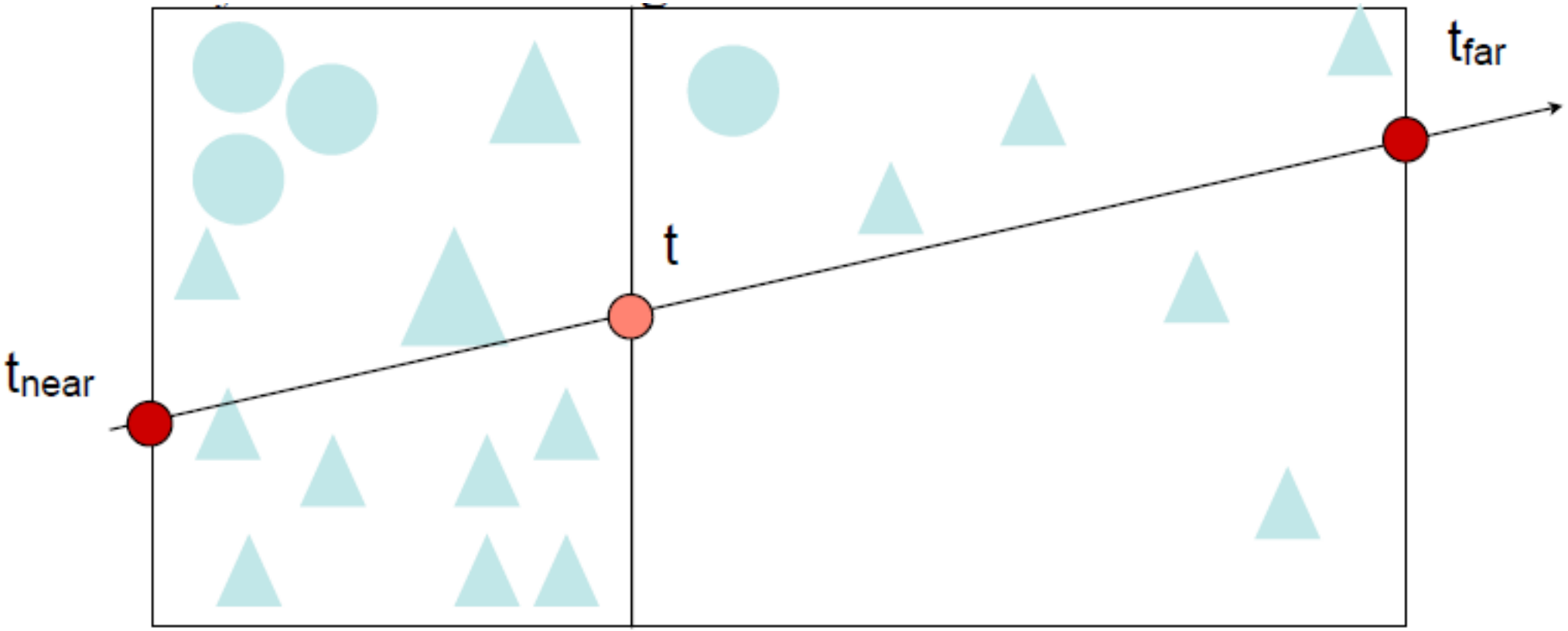
# Ray Intersection: BSP Tree



If leaf, intersect with primitives
If intersect left child, recurse
If intersect right child, recurse

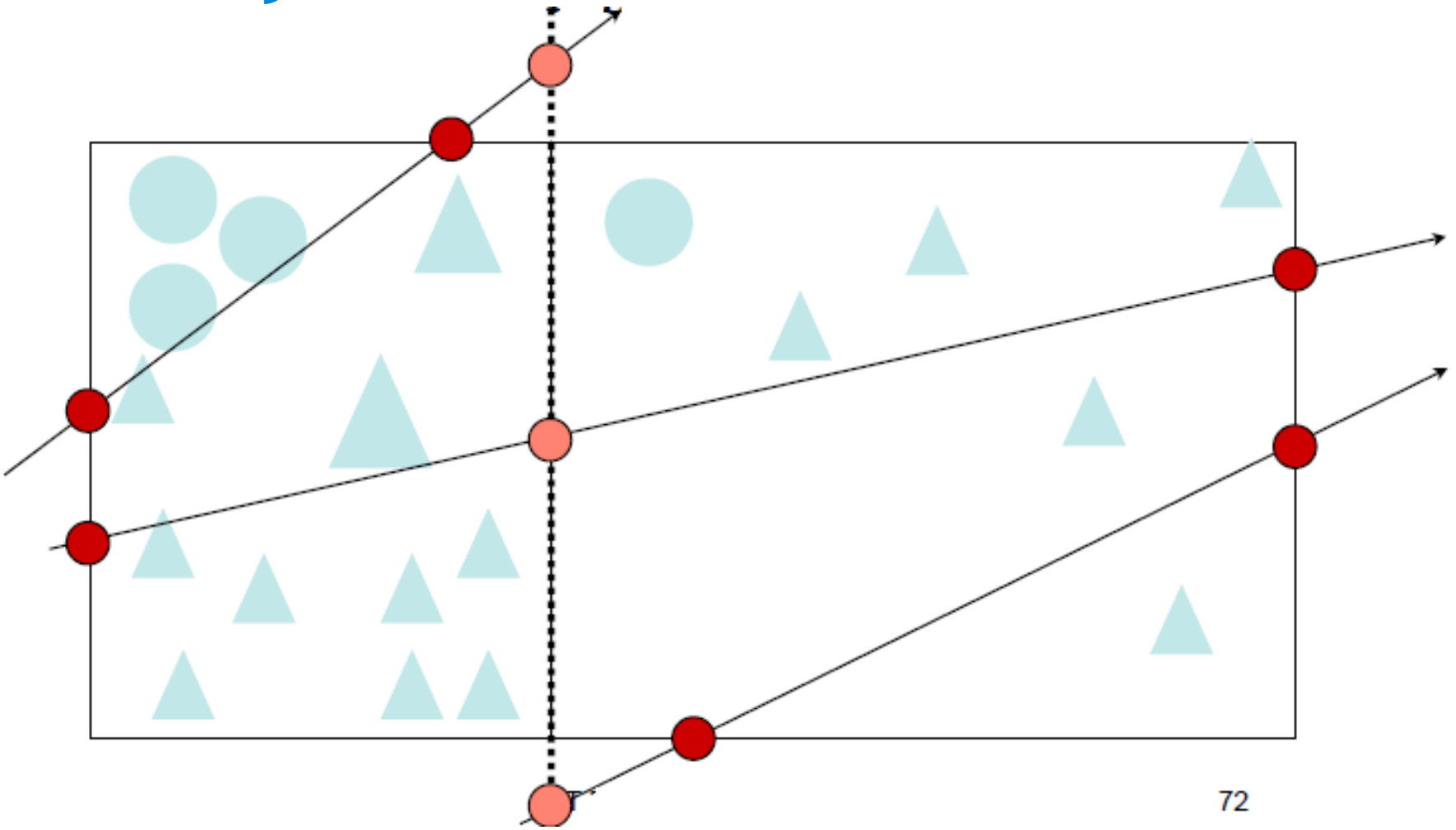
How to find out if intersect with children?

# Ray Intersection: BSP Tree




Find intersection with node's BV and splitting plane

# Ray Intersection: BSP Tree




Three cases:

1. $t > t_{far} \rightarrow$ intersect left child
2. $t < t_{near} \rightarrow$ intersect right child
3. $t_{near} < t < t_{far} \rightarrow$ intersect both children

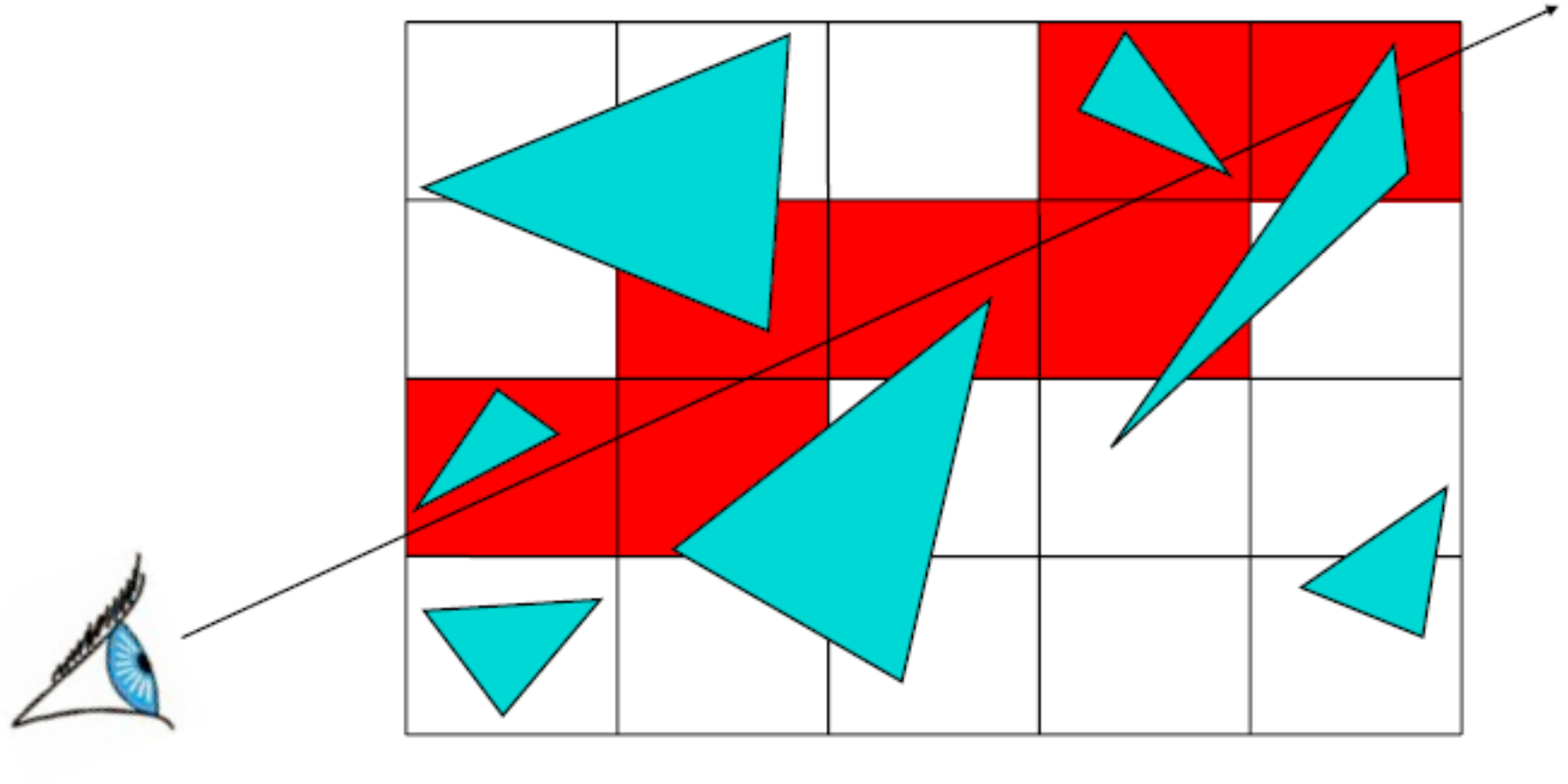
# Ray Intersection: BSP Tree

```
float intersection(node, ray, tnear, tfar){
  // intersect list of primitives
  if (node.isLeaf)
    return intersection(list, ray, t0, t1);

  compute t

  // Left only.
  if (t > tfar)
    return intersection(node.left, ray, tnear, tfar);

  // Right only.
  if (t < tnear)
    return intersection(node.right, ray, tnear, tfar);

  // Intersect left first, then right (if necessary)
  t_left = intersection(node.left, ray, tnear, t);
  // Early termination
  if (t_left <= t)
    return t_left;
  return intersection(node.right, ray, t, tfar);
}
```

# BSP Tree

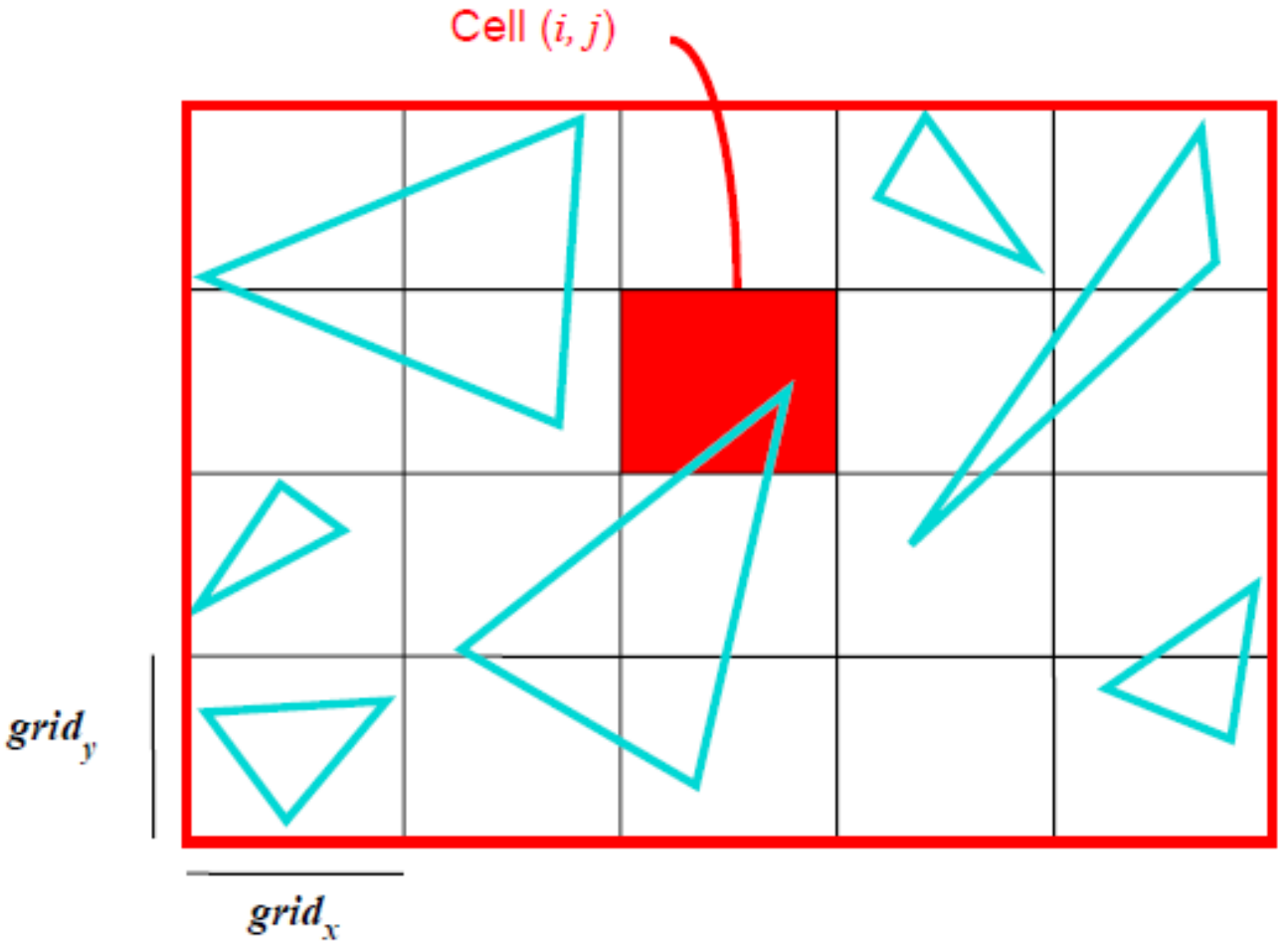
- Pros
  - Simple code
  - Easy traversal
- Cons
  - Expensive construction, specially for moving objects

# Uniform Spatial Subdivision



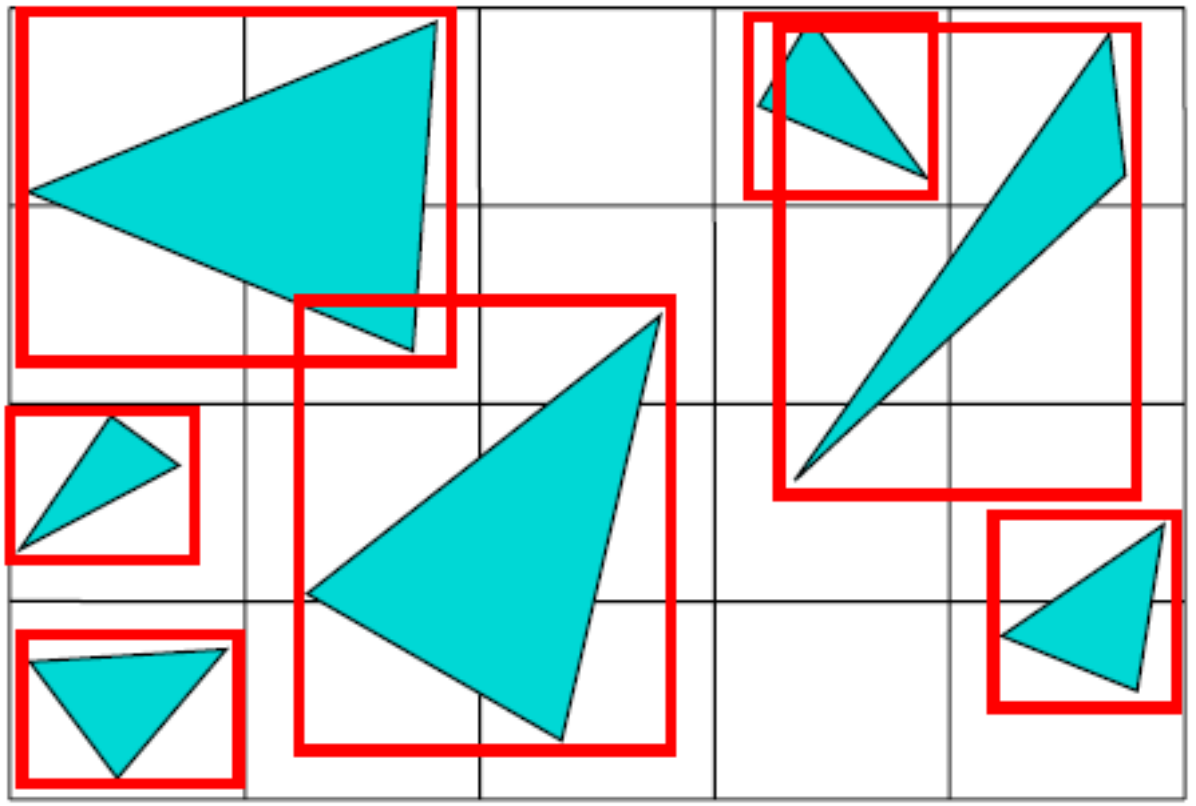
Divide space into grid and intersect with cells along the way

# Uniform Spatial Subdivision




Find scene BV and choose grid resolution

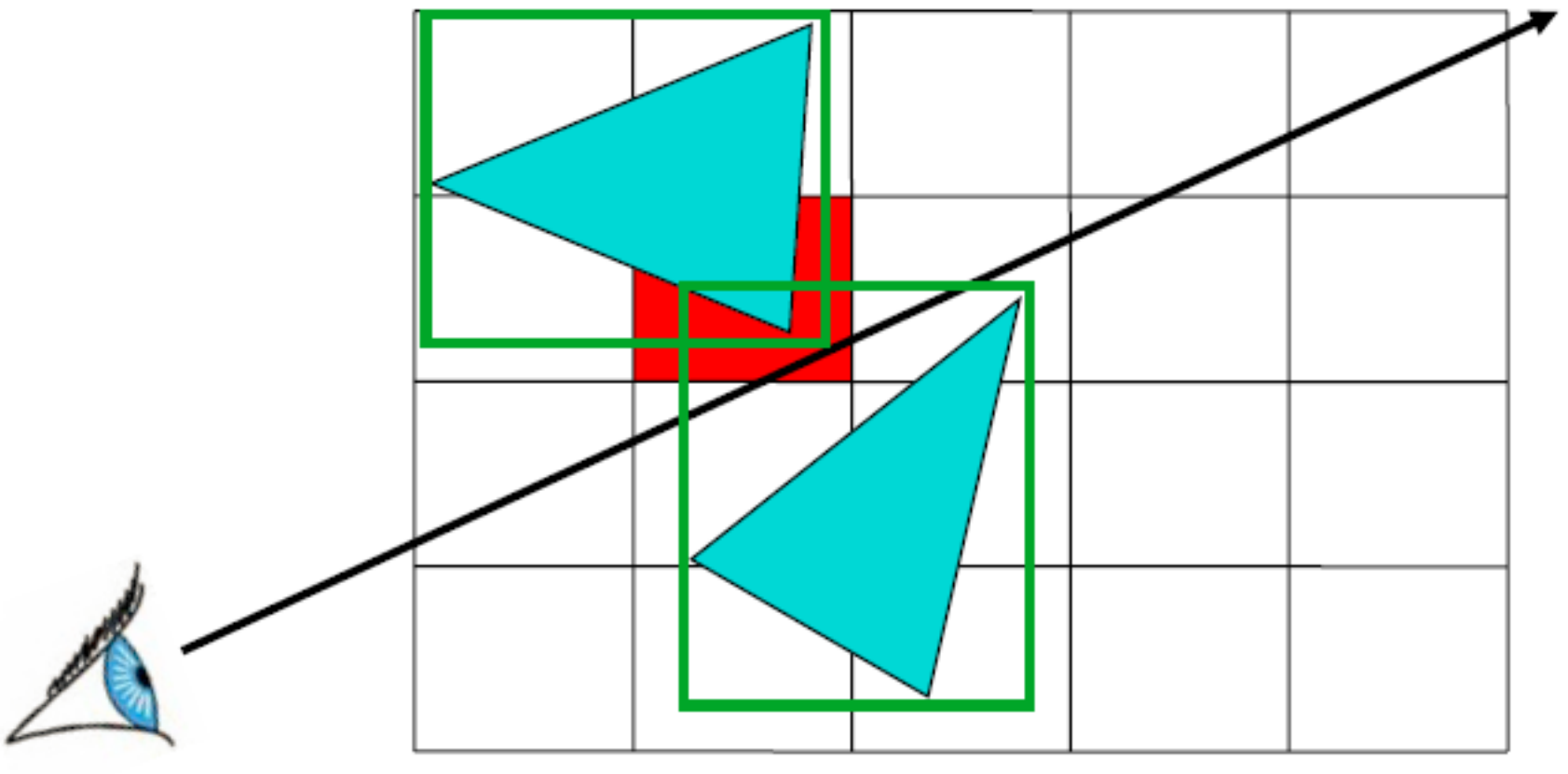
# Uniform Spatial Subdivision



Insert primitives into cells.

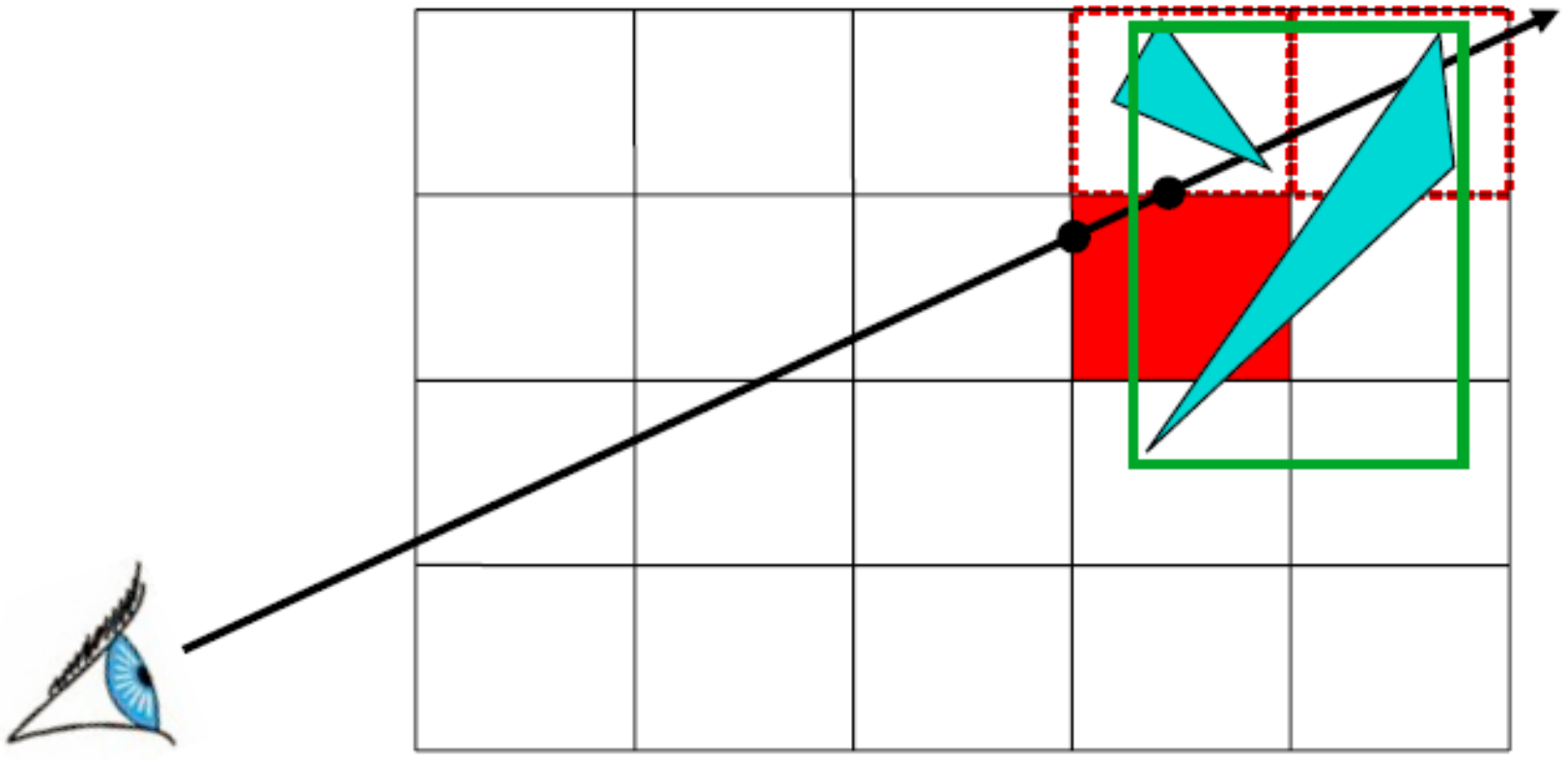
Primitives that overlap multiple cells goes to all of them

# Ray Intersection: Uniform Spatial Subdivision



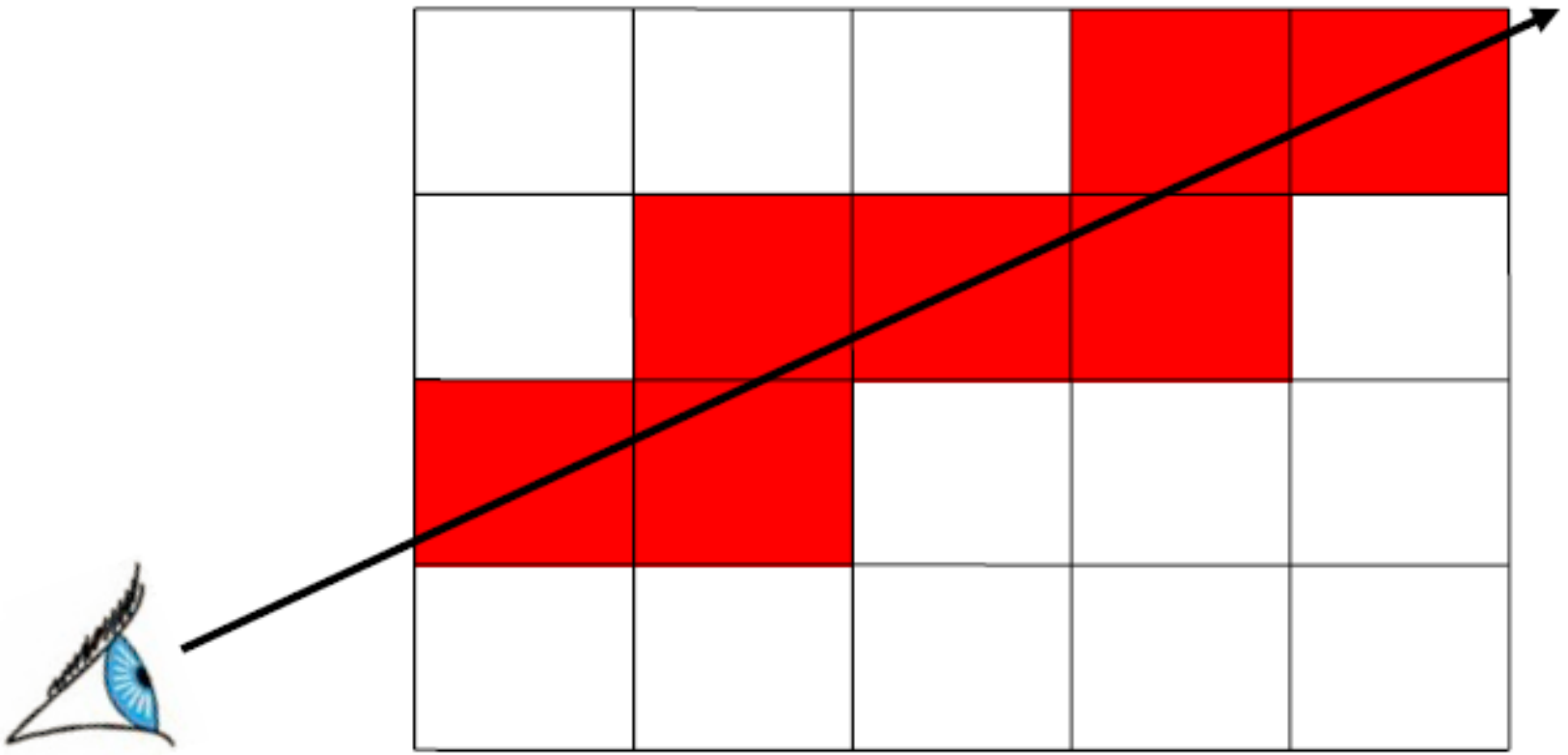
If cell has an intersection, return closest.

# Ray Intersection: Uniform Spatial Subdivision



Beware to return an intersection *within* the cell!

# Ray Intersection: Uniform Spatial Subdivision

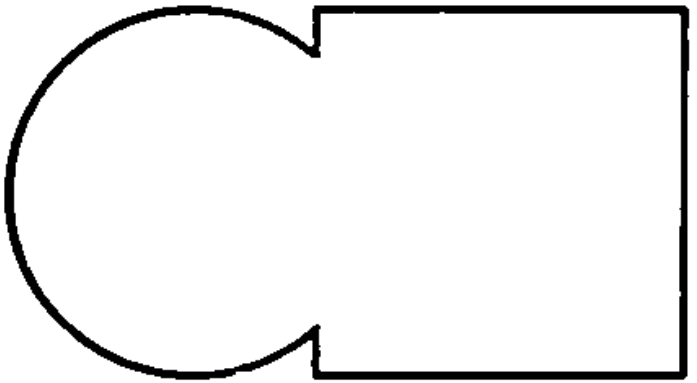


Intersect only with cells along the ray direction
Similar to line rasterization!

# Uniform Spatial Subdivision

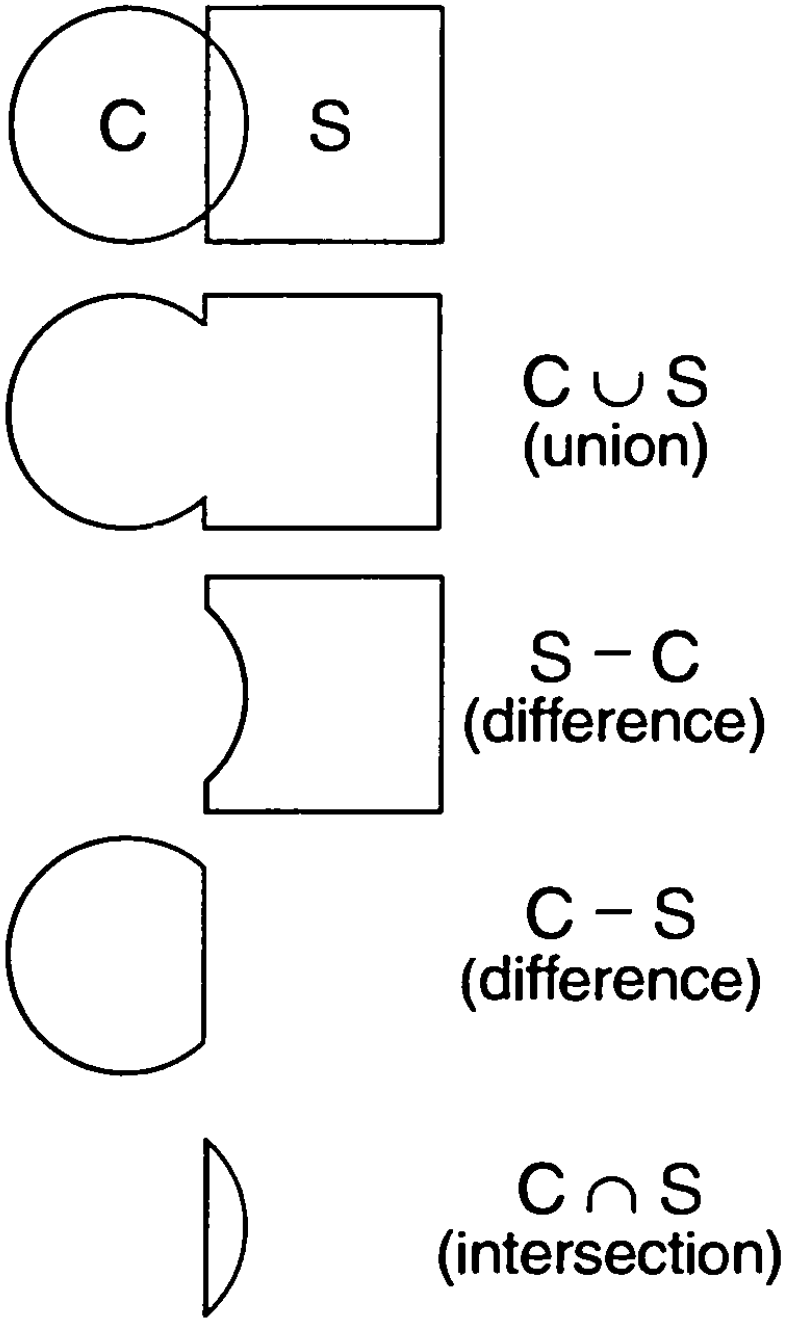
- Pros
  - Very easy to build
  - Very easy to traverse
- Cons
  - Cells can have many primitives

# Constructive Solid Geometry (CSG)



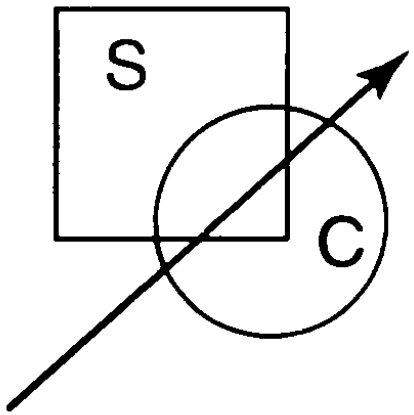
We know how to intersect a ray with sphere and box, but how about this?

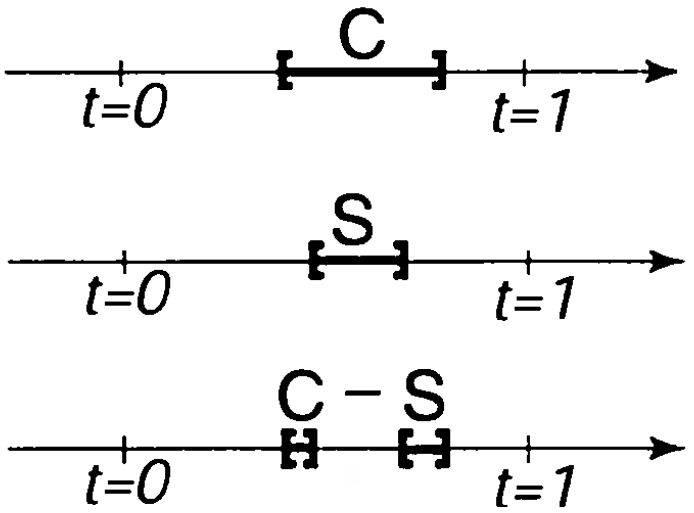
# Constructive Solid Geometry

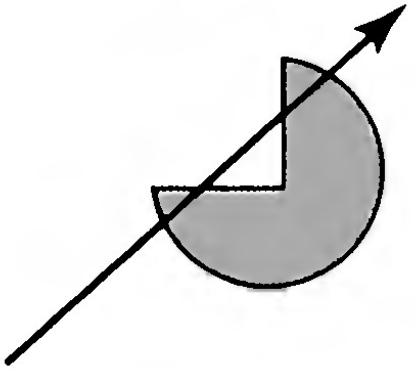



Build objects using set operations on simpler objects

# Constructive Solid Geometry



Find ray intersection using set operations
on the individual intersection intervals

# Recap

- Ray Tree
- Accelerating Ray Tracing
  - Bounding Volume Hierarchies (BVHs)
  - Binary Space Partitioning Trees (BSP Trees)
  - Uniform Spatial Subdivision
- Constructive Solid Geometry