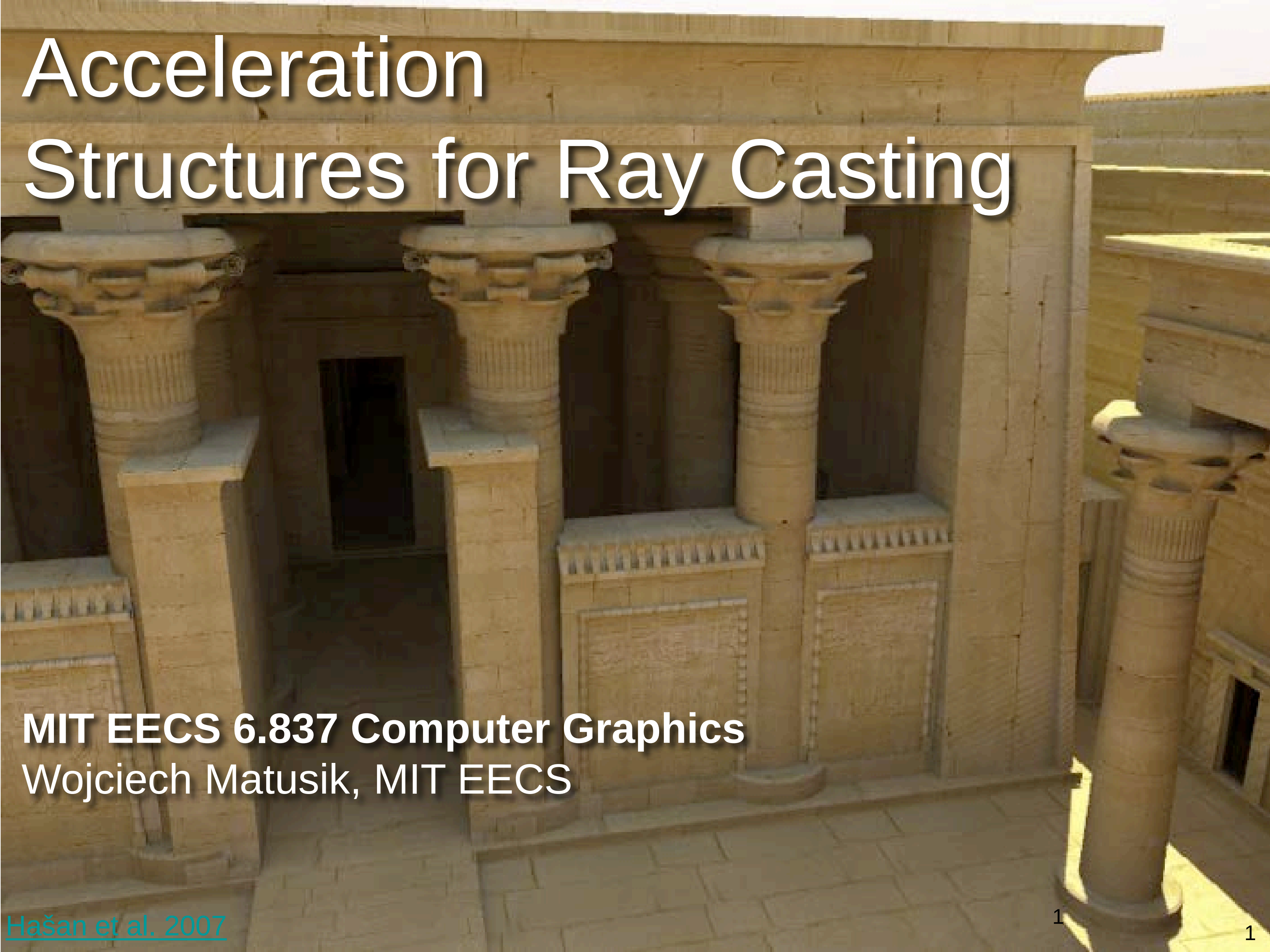
Acceleration Structures for Ray Casting



MIT EECS 6.837 Computer Graphics

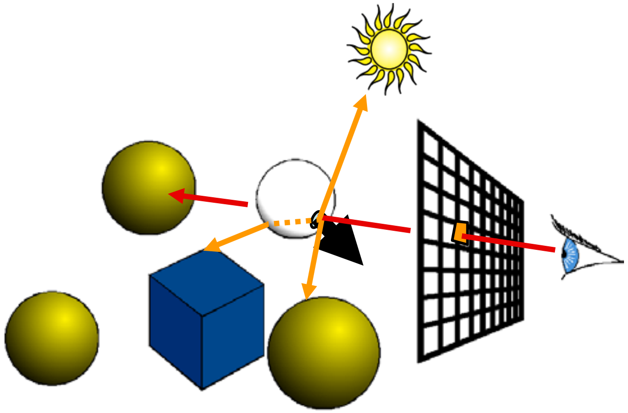
Wojciech Matusik, MIT EECS

Hašan et al. 2007

Recap: Ray Tracing

```
trace ray
   Intersect all objects
   color = ambient term
   For every light
      cast shadow ray
      color += local shading term
   If mirror
      color += $color_{refl}$ *
               trace reflected ray
   If transparent
      color += $color_{trans}$ *
               trace transmitted ray
```

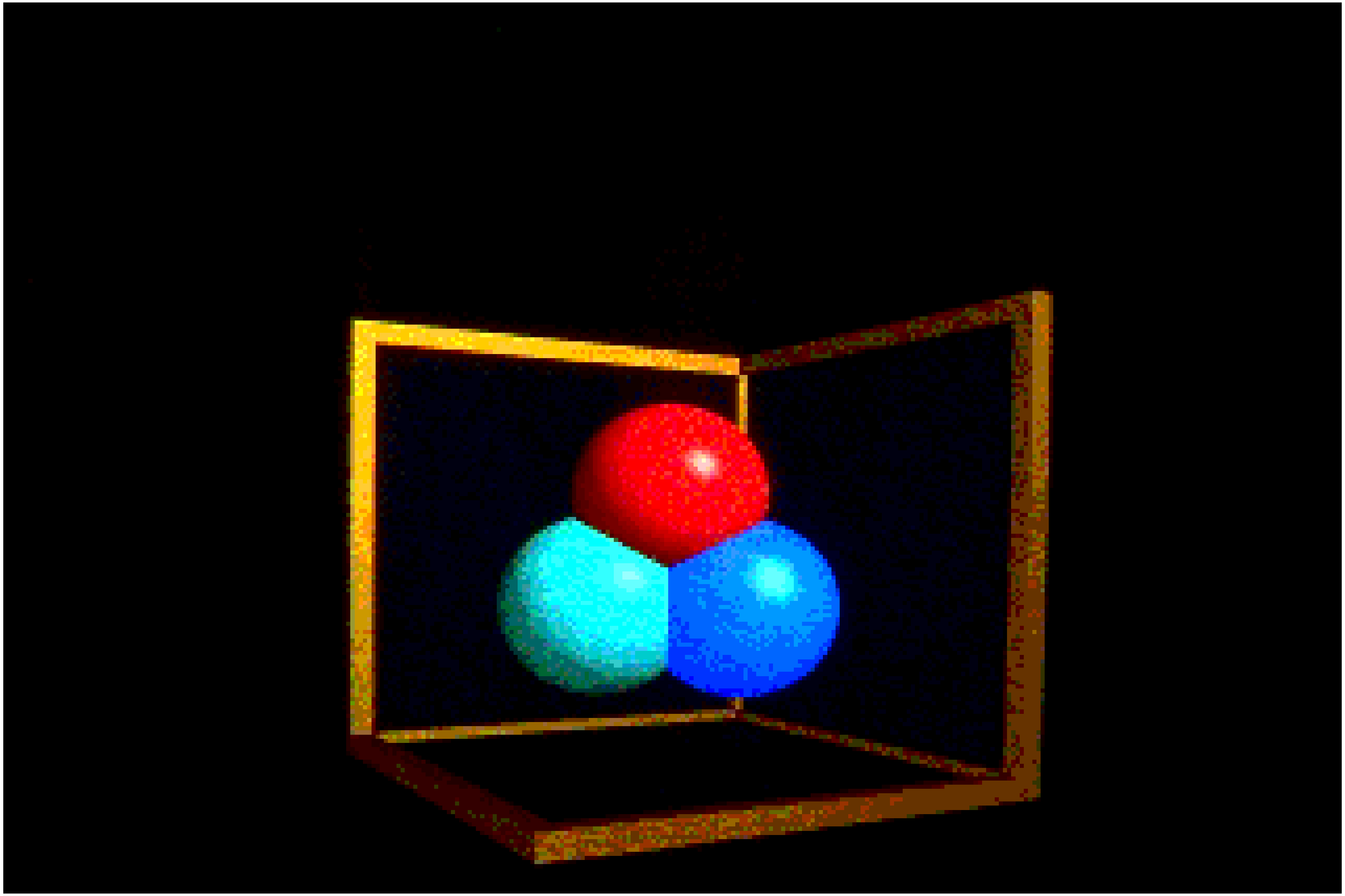
- ***Does it ever end?***



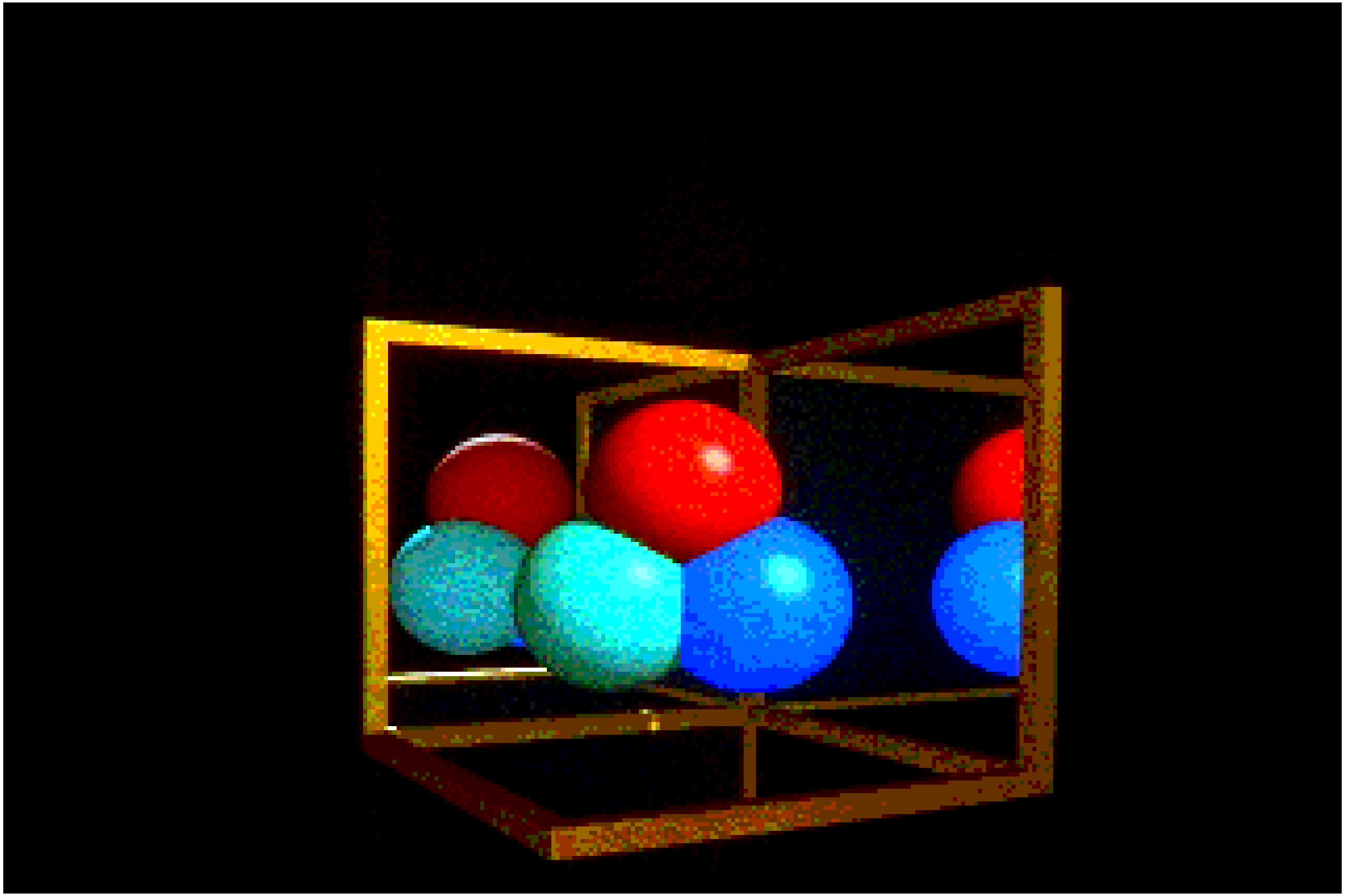
Stopping criteria:

- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small

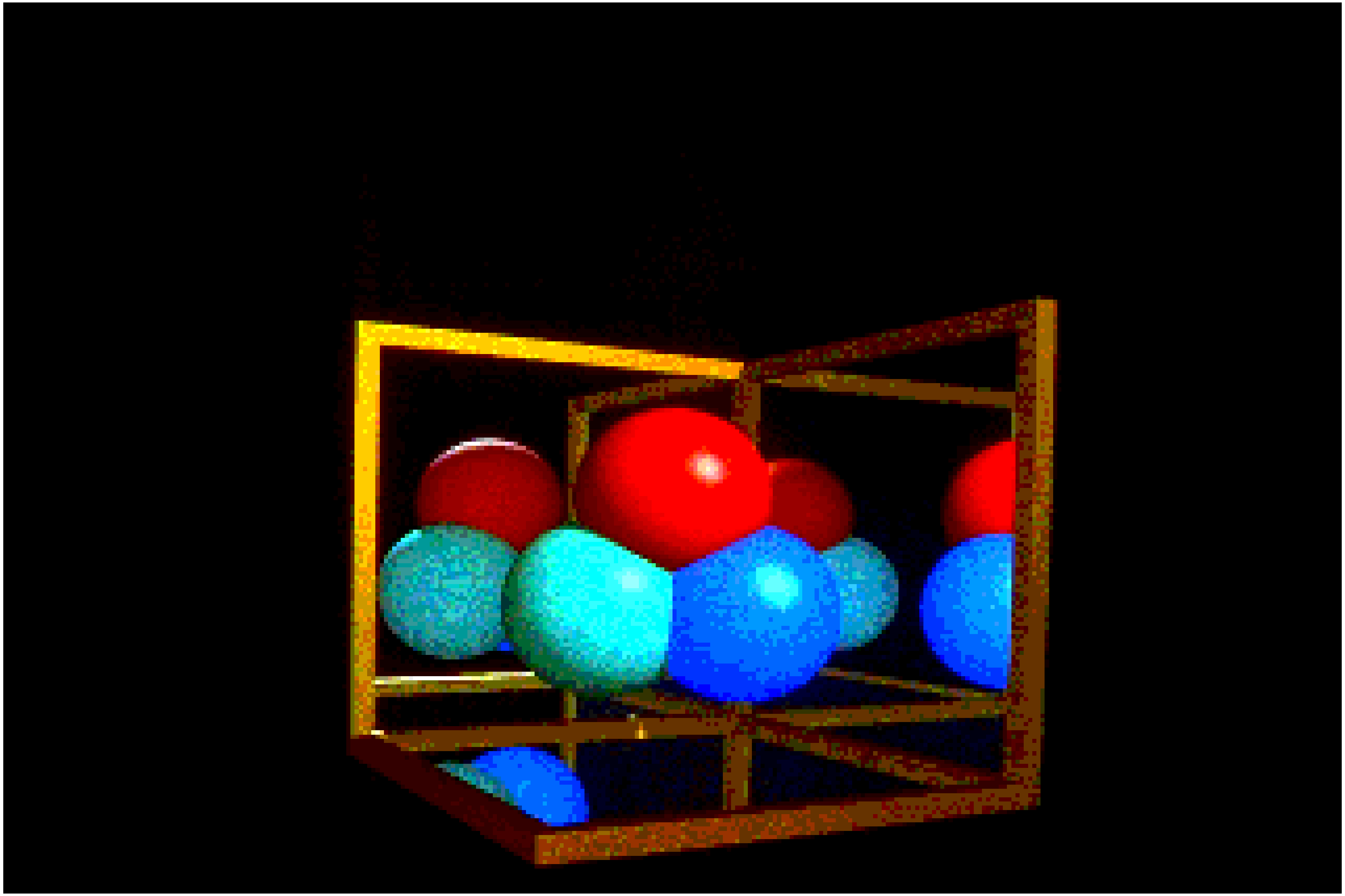
Recursion For Reflection: None



Recursion For Reflection: 1

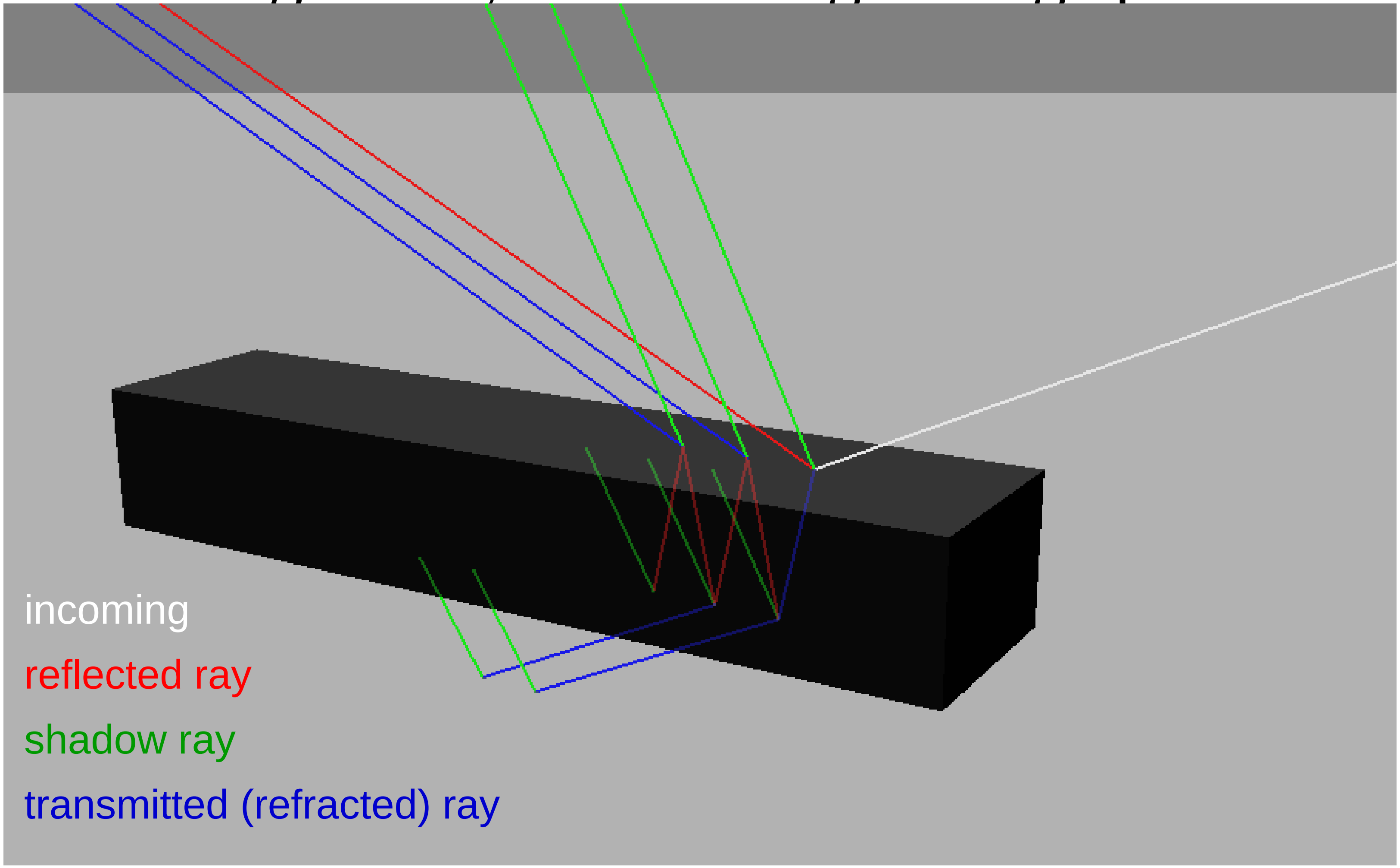


Recursion For Reflection: 2



Ray tree

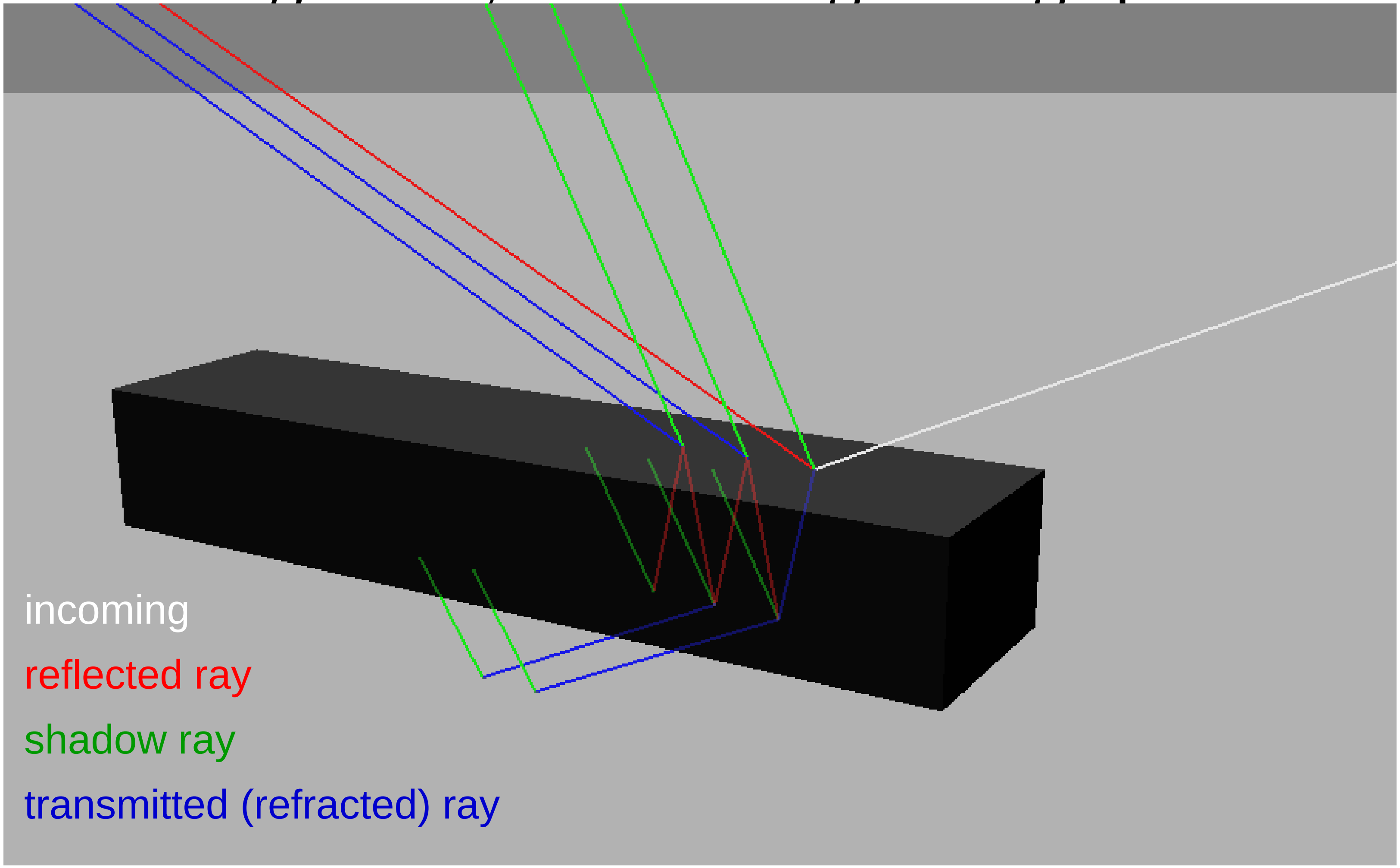
- Visualizing the ray tree for single image pixel



Ray tree

This gets pretty complicated pretty fast!

- Visualizing the ray tree for single image pixel



Questions?

Ray Tracing Algorithm Analysis

- Lots of primitives
- Recursive
- Distributed Ray Tracing
  - – Means using many rays for non-ideal/non-pointlike phenomena
    - Soft shadows
    - Anti-aliasing
    - Glossy reflection
    - Motion blur
    - Depth of field

cost ≈ height * width *
num primitives *
intersection cost *
size of recursive ray tree *
num shadow rays *
num supersamples *
num glossy rays *
num temporal samples *
num aperture samples *
. . .

Can we reduce this?

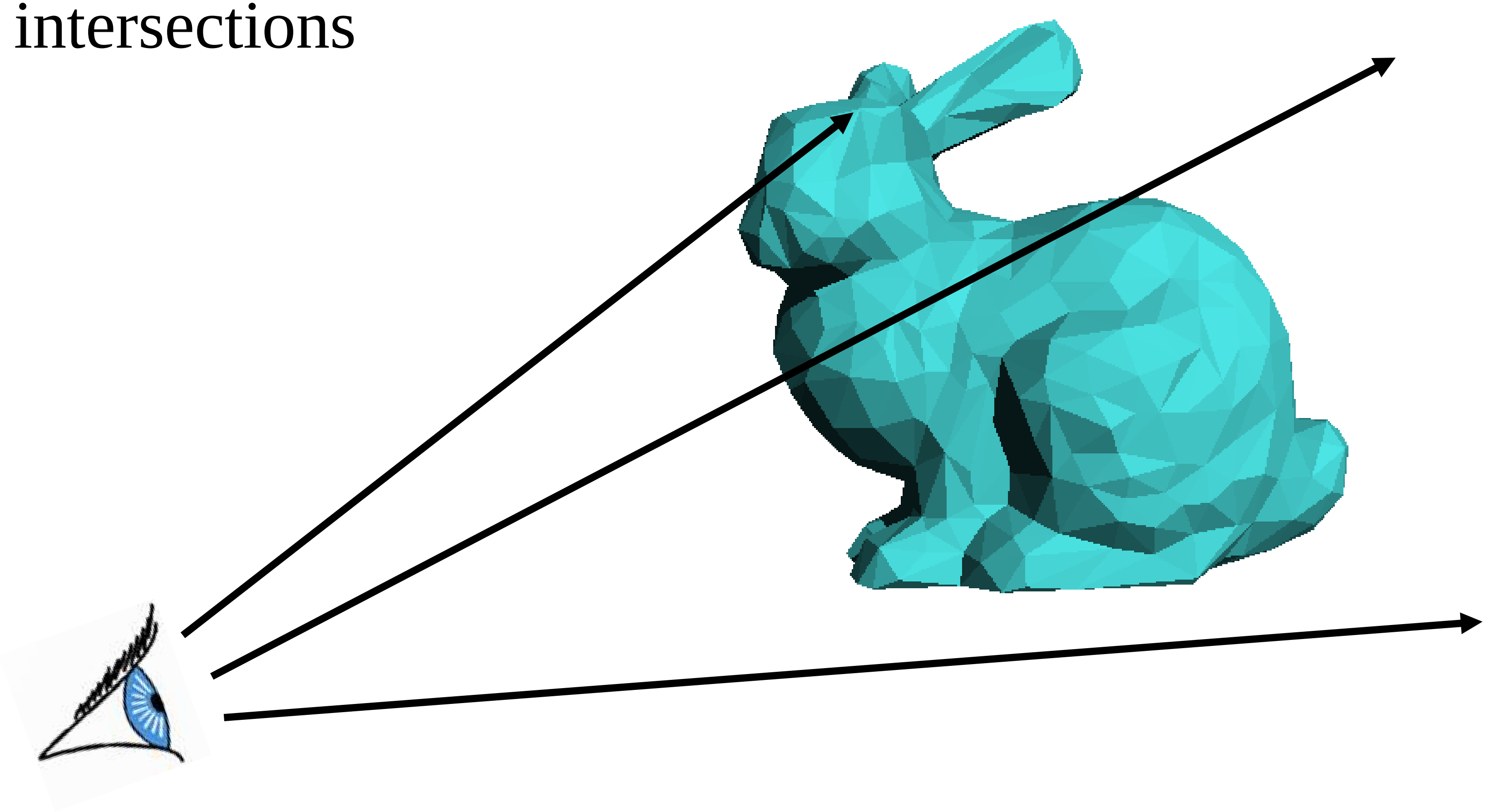
Today

- Motivation
  - You need LOTS of rays to generate nice pictures
  - Intersecting every ray with every primitive becomes the bottleneck
- Bounding volumes
- Bounding Volume Hierarchies, Kd-trees

```
For every pixel
   Construct a ray from the eye
   For every object in the scene
        Find intersection with the ray
        Keep if closest
         Shade
```

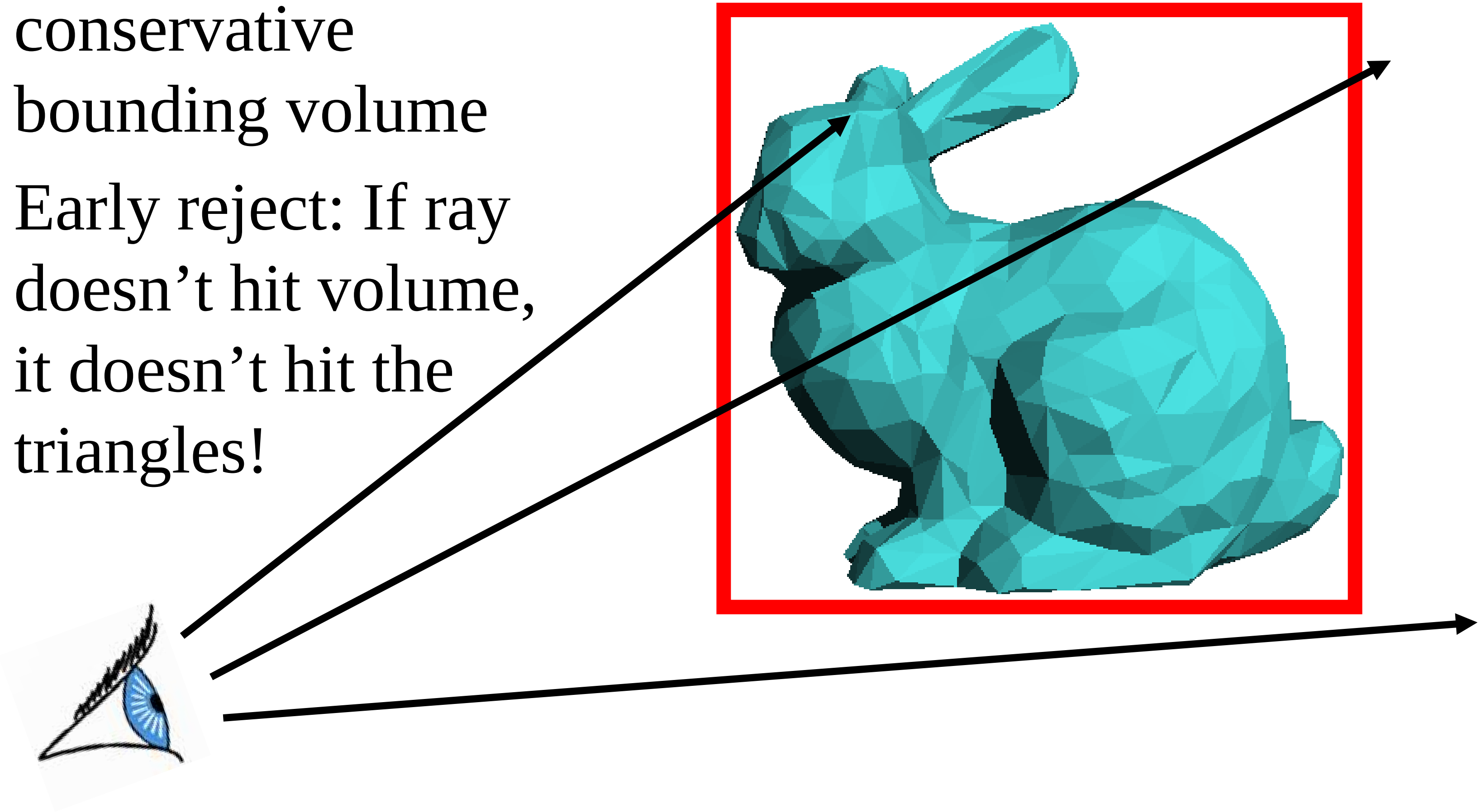
Accelerating Ray Casting

- Goal: Reduce the number of ray/primitive intersections



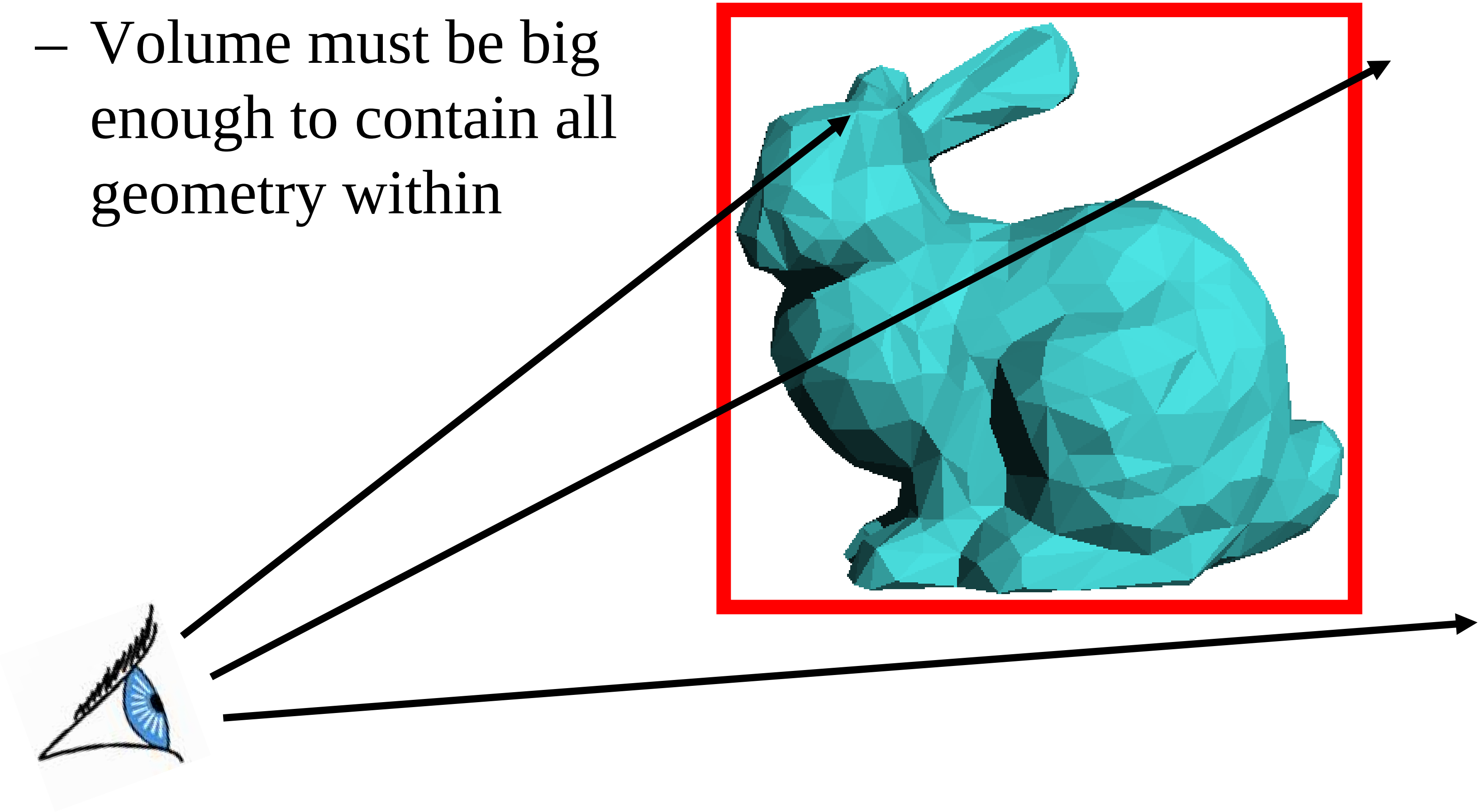
Conservative Bounding Volume

- First check for an intersection with a conservative bounding volume
- Early reject: If ray doesn’t hit volume, it doesn’t hit the triangles!



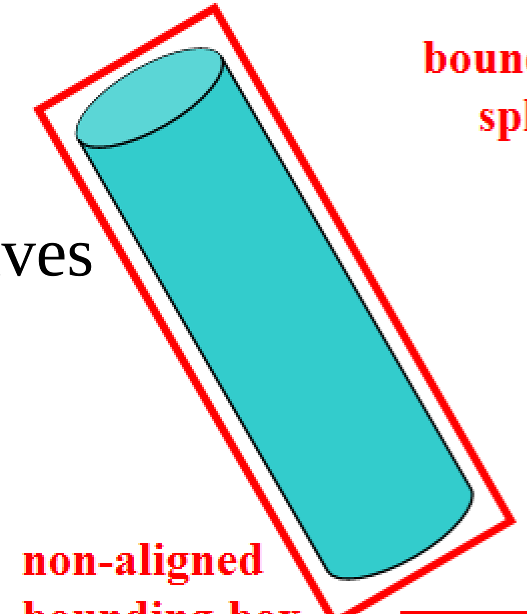
Conservative Bounding Volume

- What does “conservative” mean?
  - Volume must be big enough to contain all geometry within

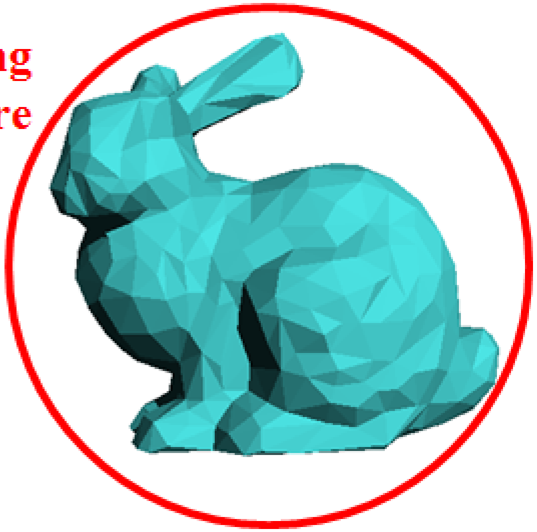


Conservative Bounding Regions

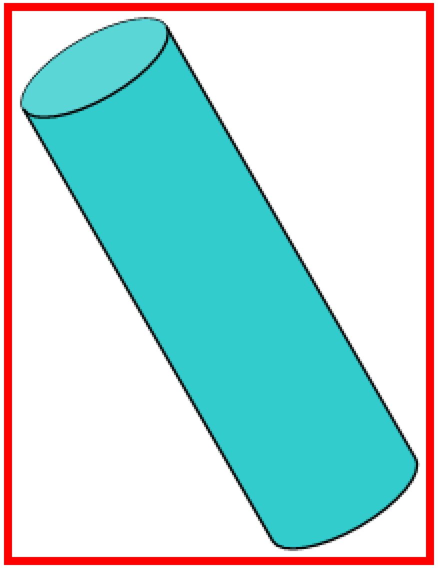
- Desiderata
  - Tight → avoid false positives
  - Fast to intersect



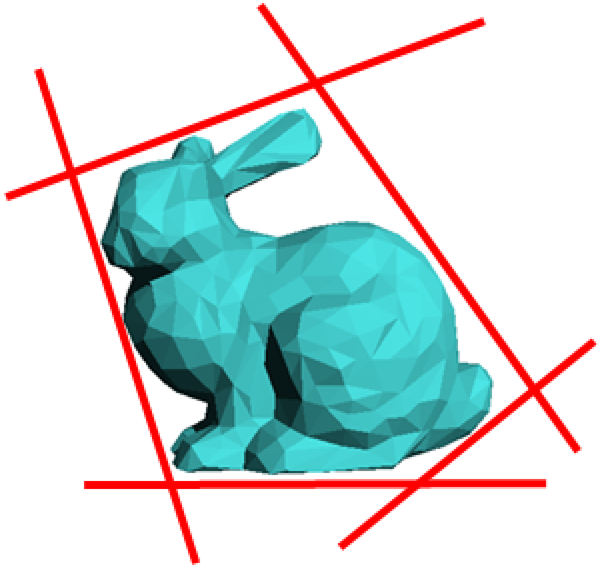
non-aligned bounding box



bounding sphere



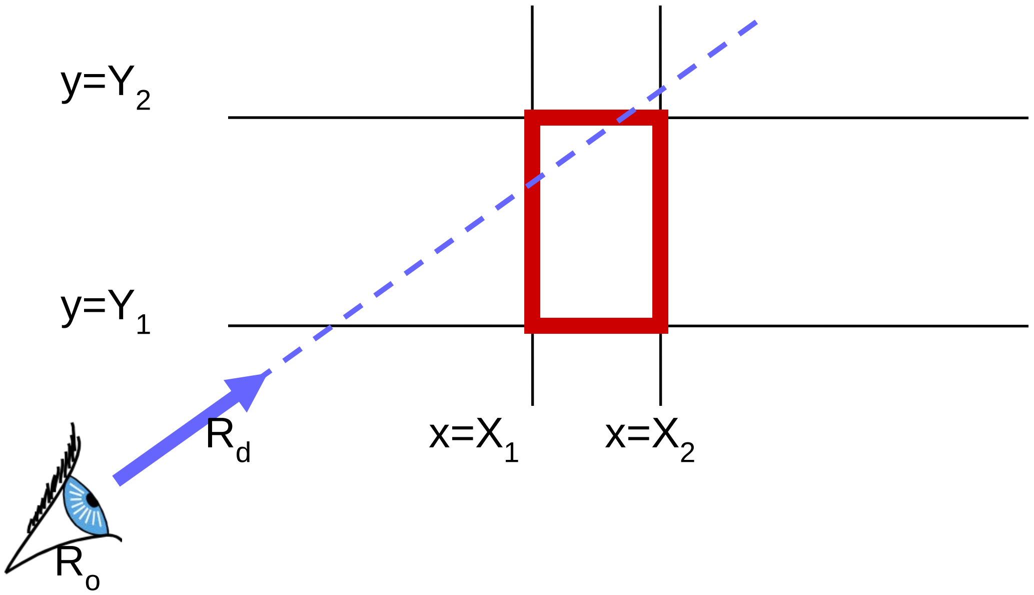
axis-aligned bounding box



arbitrary convex region (bounding half-spaces)

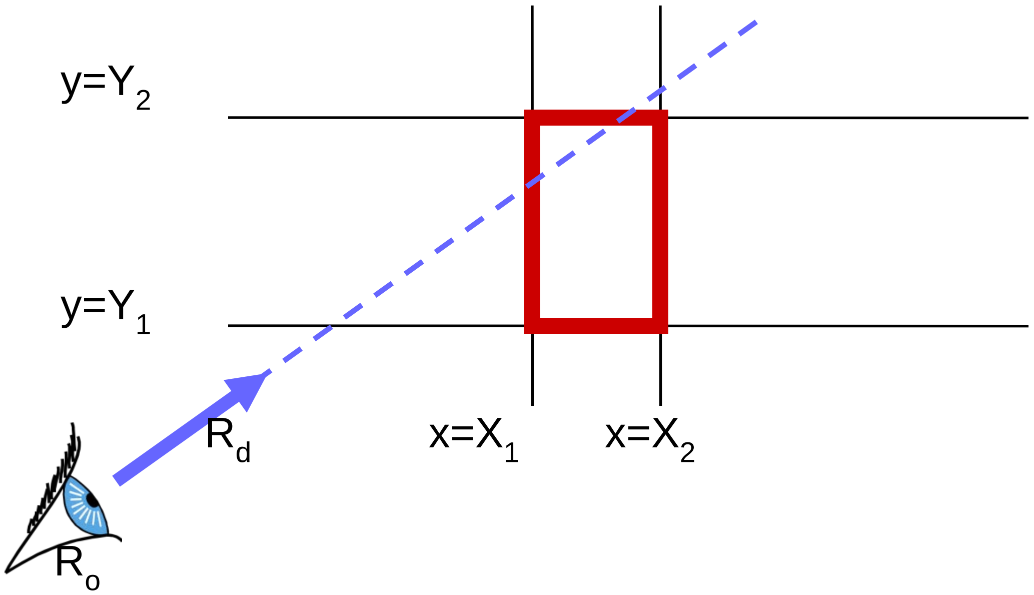
Ray-Box Intersection

- Axis-aligned box
- Box: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- Ray: $P(t) = R_o + tR_d$



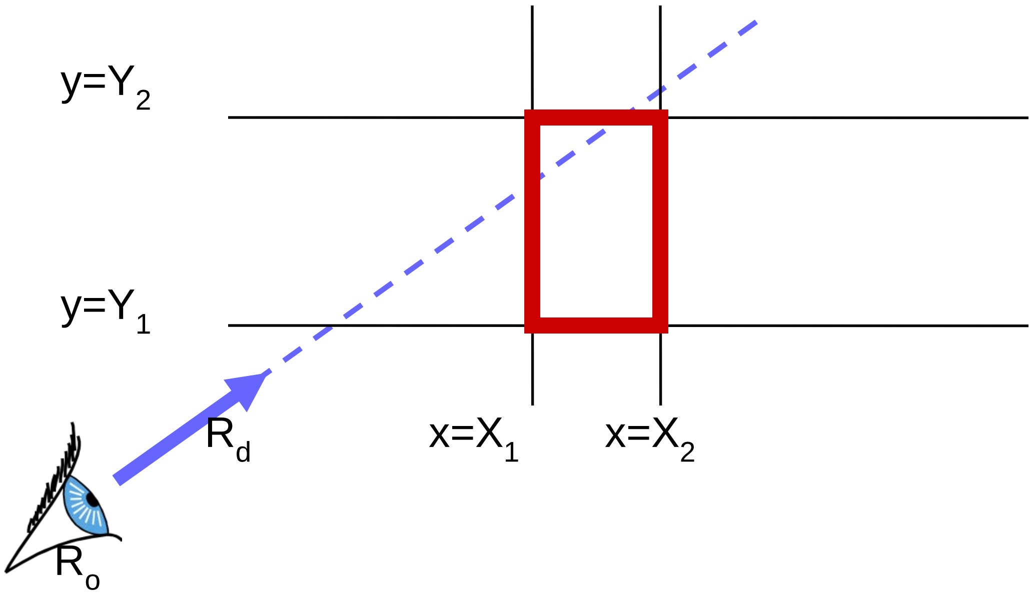
Naïve Ray-Box Intersection

- 6 plane equations: Compute all intersections
- Return closest intersection *inside the box*
  - Verify intersections are on the correct side of each plane: $Ax+By+Cz+D < 0$



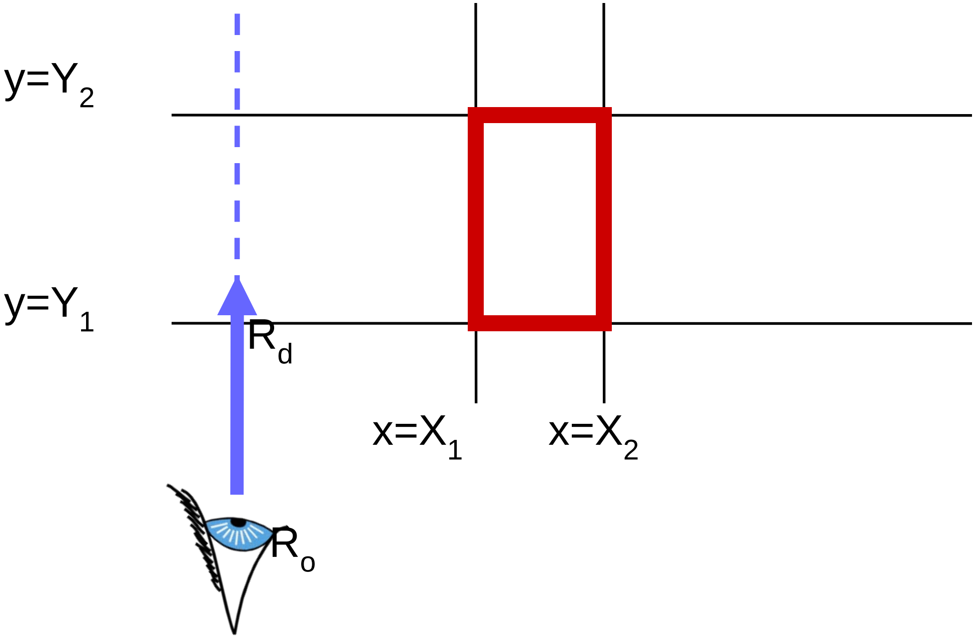
Reducing Total Computation

- Pairs of planes have the same normal
- Normals have only one non-zero component
- Do computations one dimension at a time



Test if Parallel

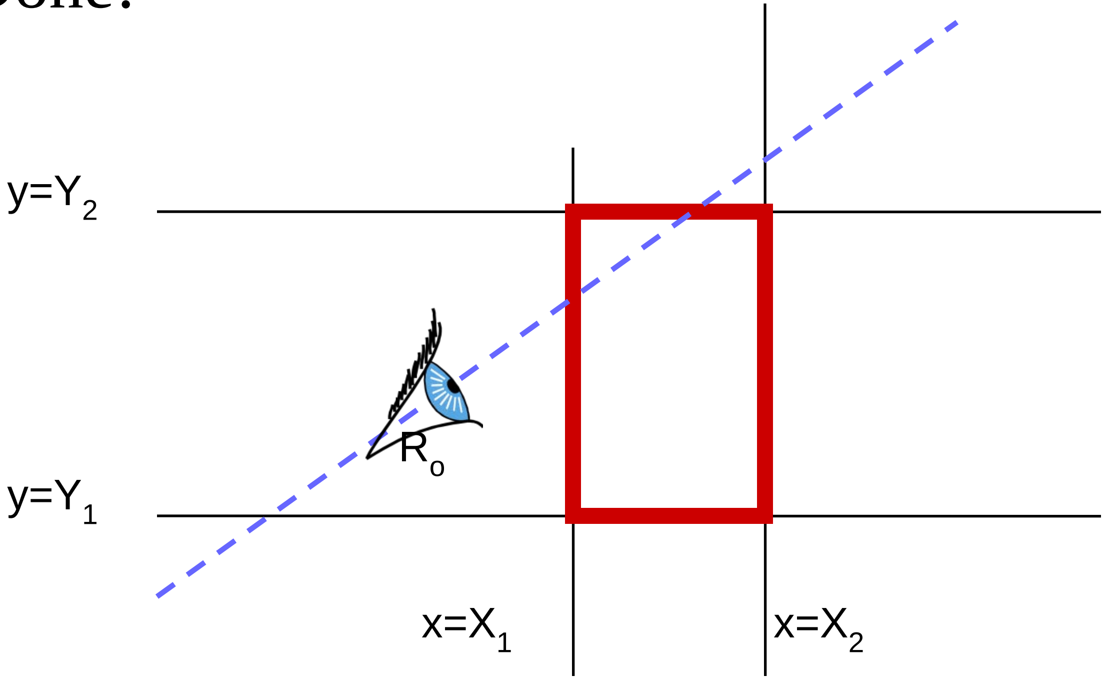
- If $R_{dx} = 0$ (ray is parallel) AND $R_{ox} < X_1$ or $R_{ox} > X_2$ → **no intersection**



(The same for Y and Z, of course)

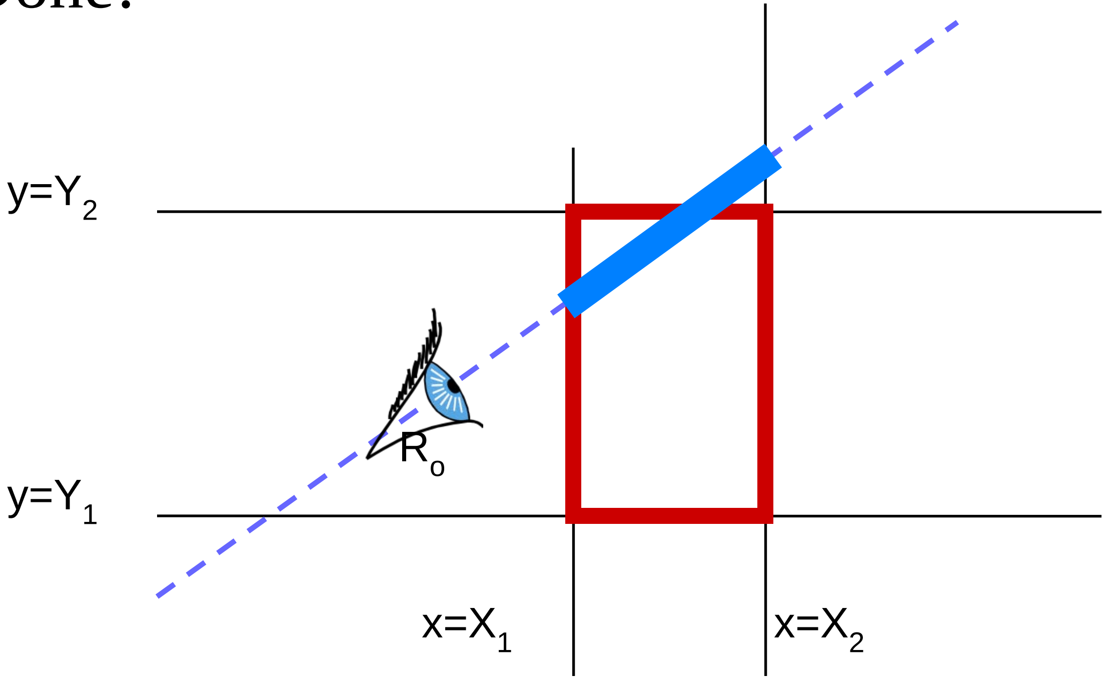
Find Intersections Per Dimension

- Basic idea
  - – Determine an interval along the ray for each dimension
  - – The intersect these 1D intervals (remember CSG!)
  - – Done!



Find Intersections Per Dimension

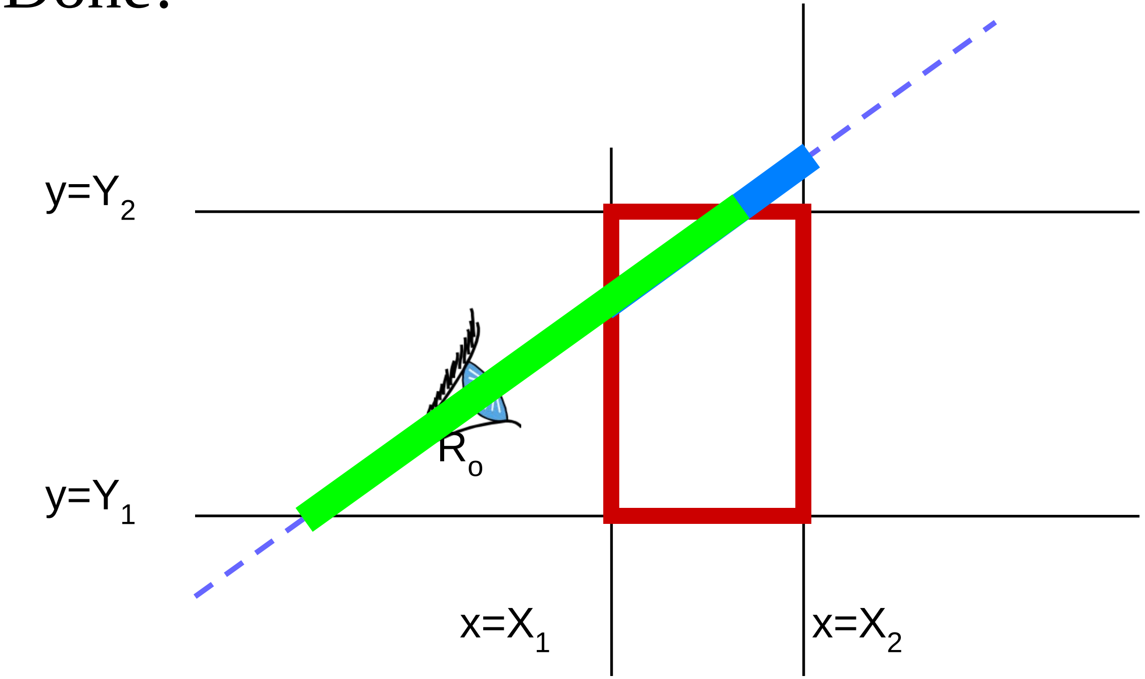
- Basic idea
  - Determine an interval along the ray for each dimension
  - The intersect these 1D intervals (remember CSG!)
  - Done!



Interval between X_1 and X_2

Find Intersections Per Dimension

- Basic idea
  - Determine an interval along the ray for each dimension
  - The intersect these 1D intervals (remember CSG!)
  - Done!

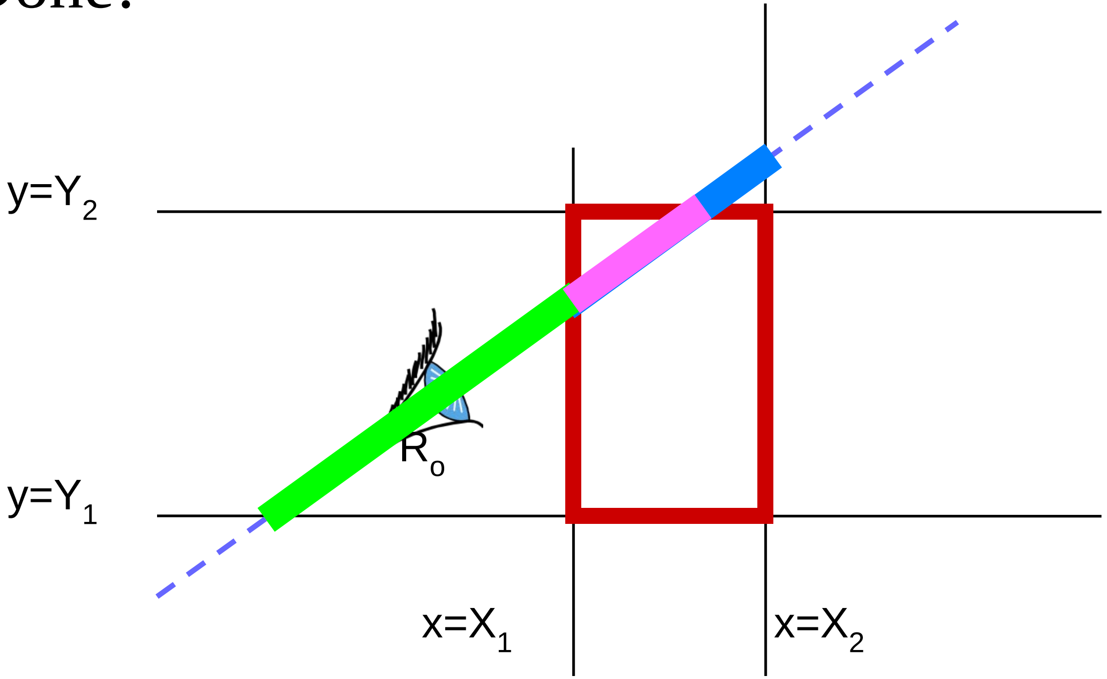


Interval between X_1 and X_2

Interval between Y_1 and Y_2

Find Intersections Per Dimension

- Basic idea
  - Determine an interval along the ray for each dimension
  - The intersect these 1D intervals (remember CSG!)
  - Done!

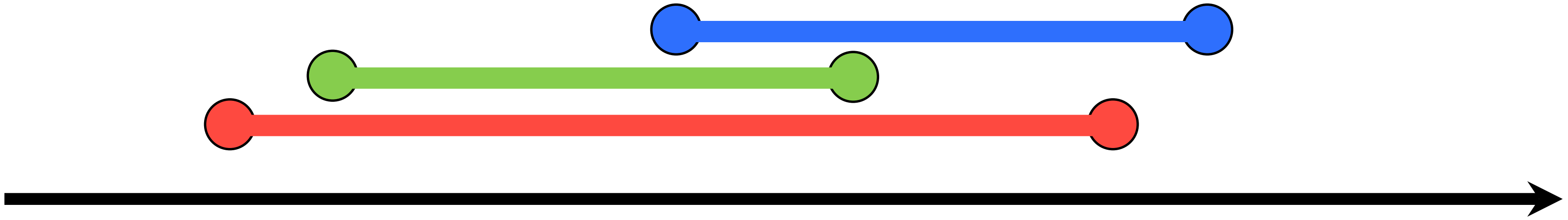


Interval between X_1 and X_2

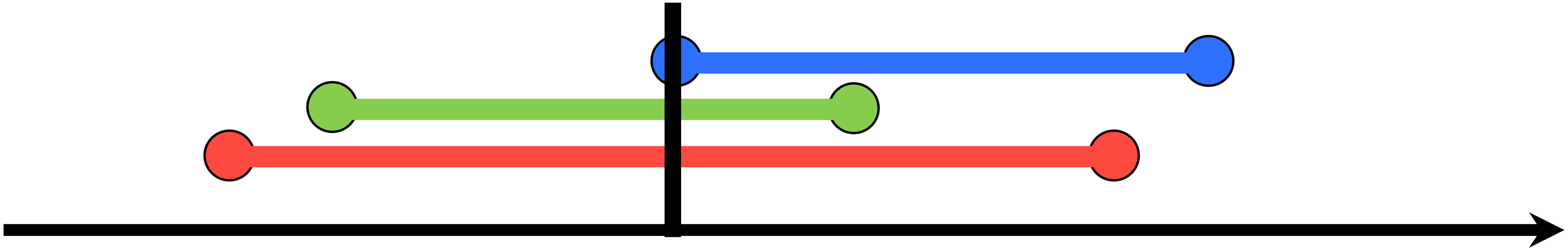
Interval between Y_1 and Y_2

Intersection

Intersecting 1D Intervals

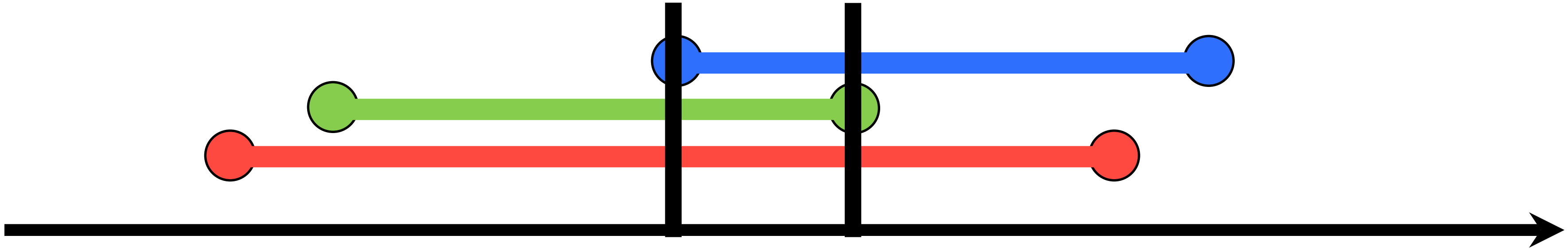


Intersecting 1D Intervals



Start=
max of mins

Intersecting 1D Intervals

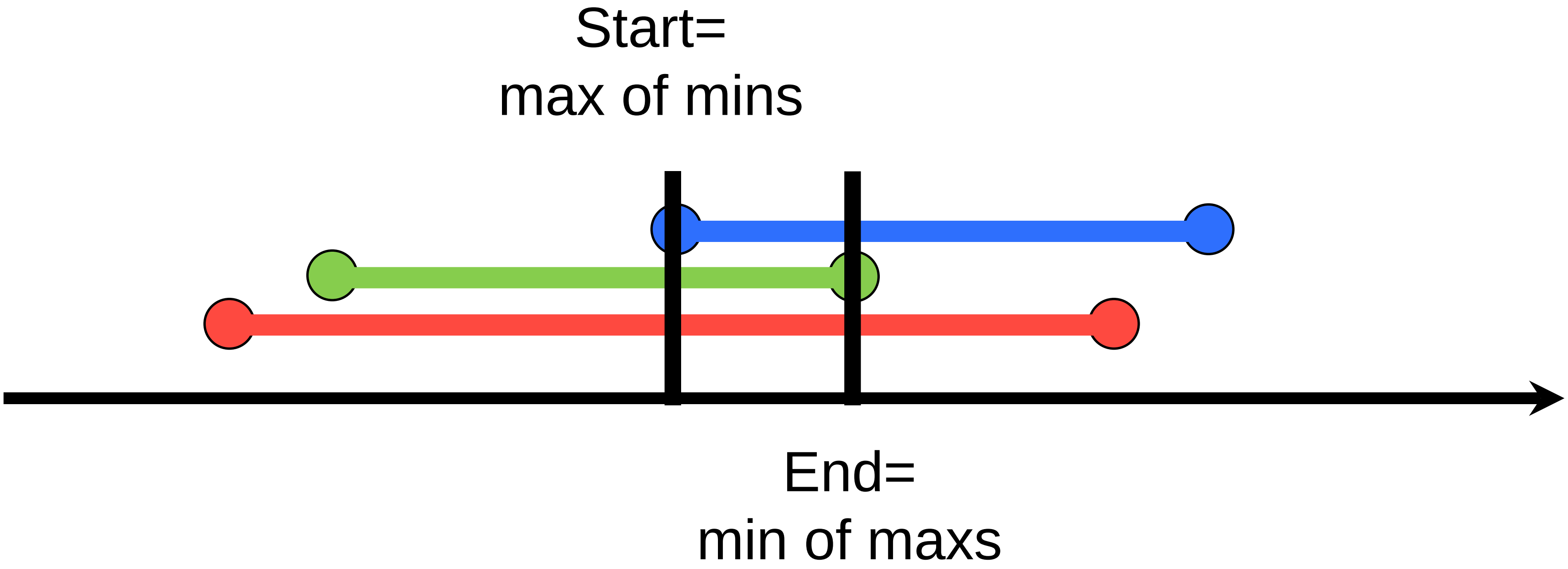


Start=
max of mins

End=
min of maxs

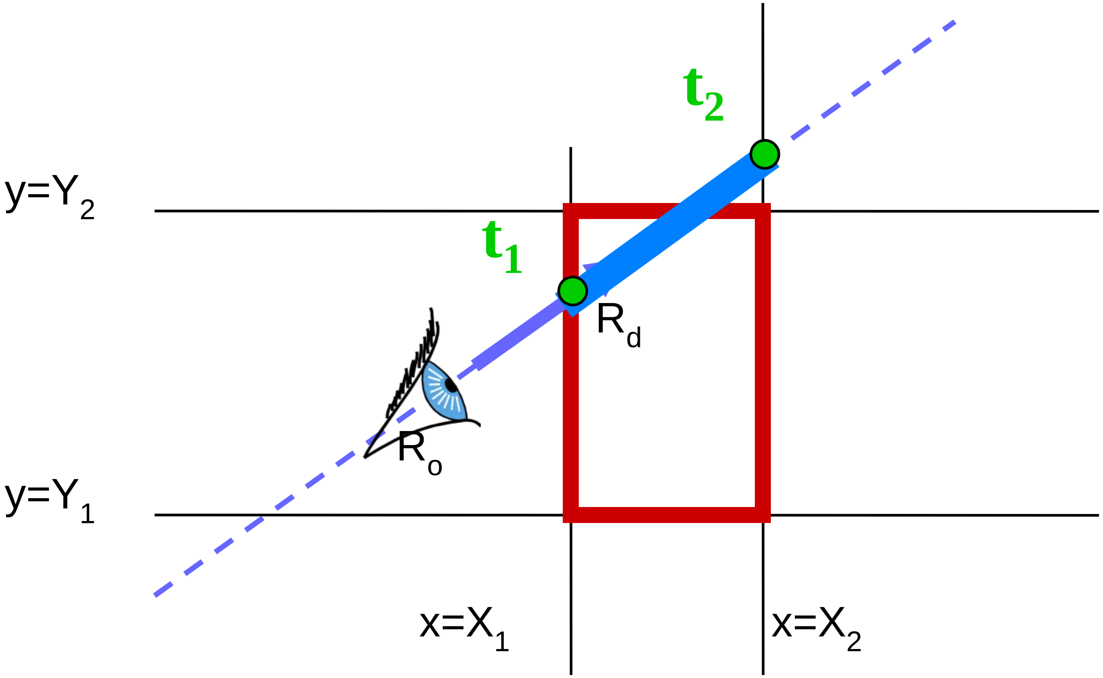
Intersecting 1D Intervals

If Start > End, the intersection is empty!



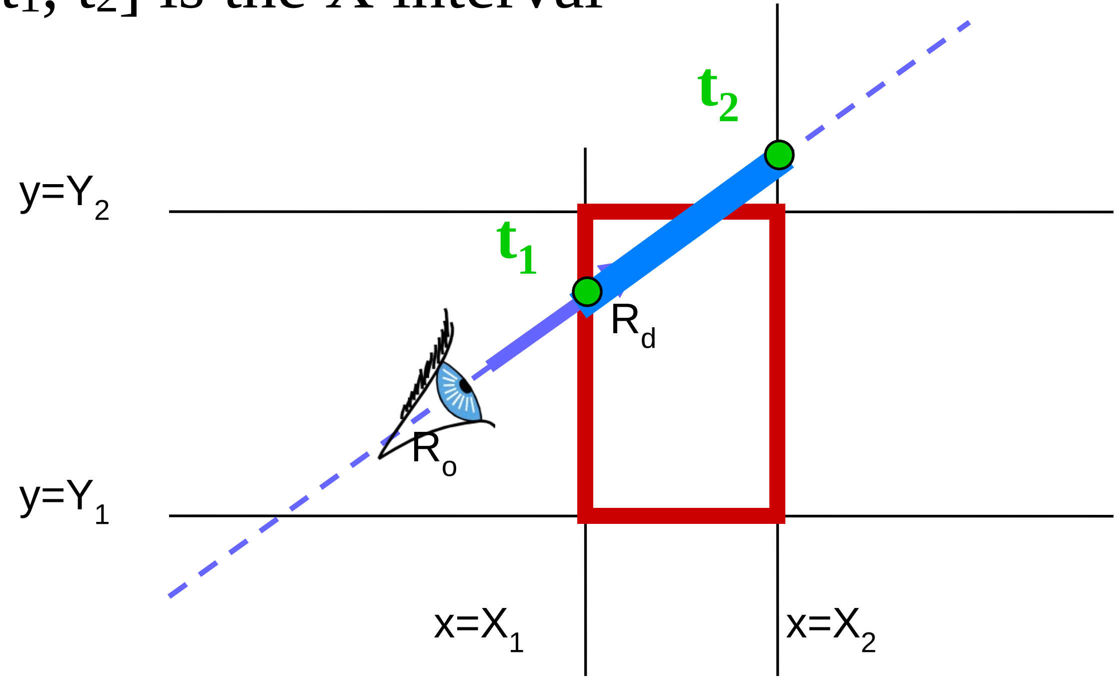
Find Intersections Per Dimension

- Calculate intersection distance t_1 and t_2



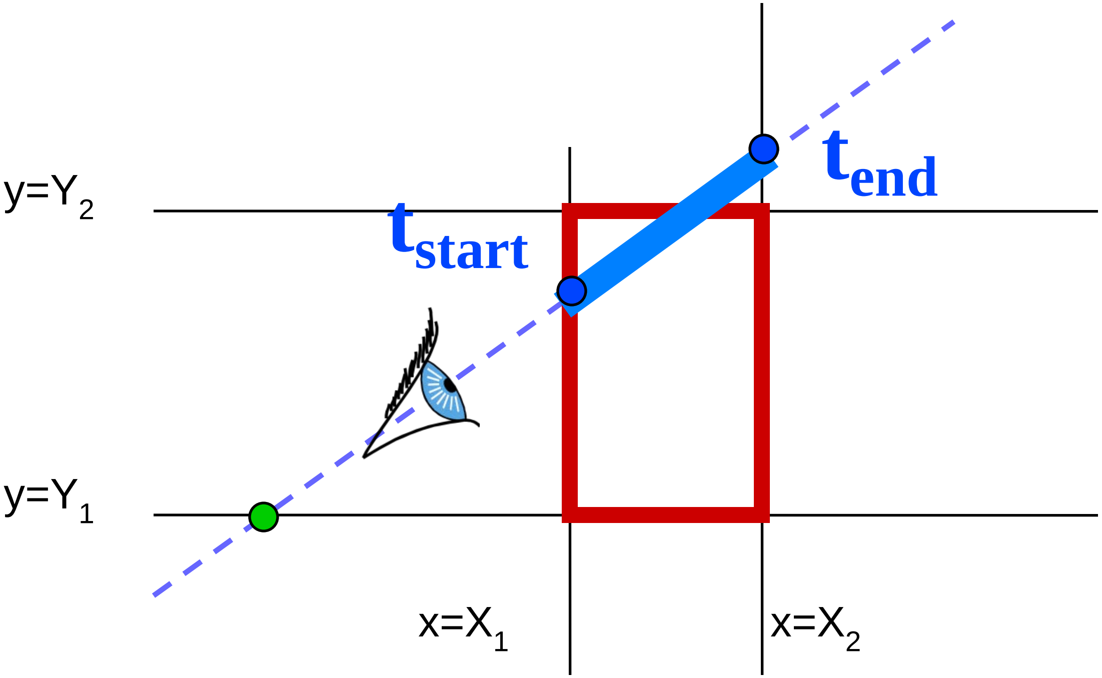
Find Intersections Per Dimension

- Calculate intersection distance t_1 and t_2
  - $t_1 = (X_1 - R_{ox}) / R_{dx}$
  - $t_2 = (X_2 - R_{ox}) / R_{dx}$
  - $[t_1, t_2]$ is the X interval



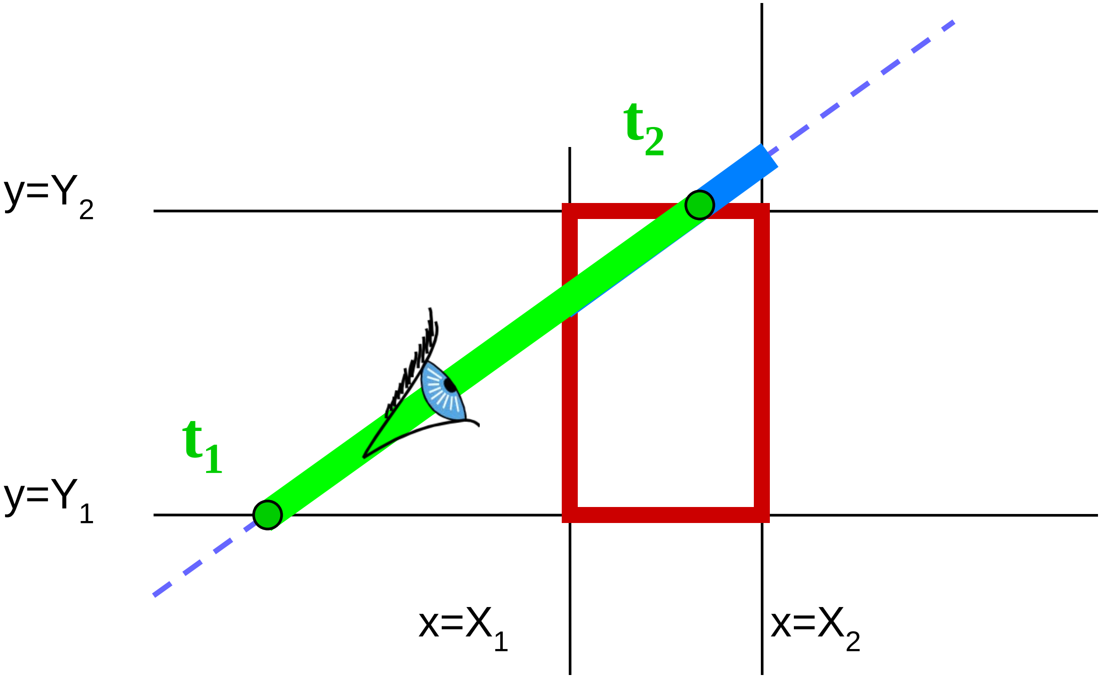
Then Intersect Intervals

- Init t_{start} & t_{end} with X interval
- Update t_{start} & t_{end} for each subsequent dimension



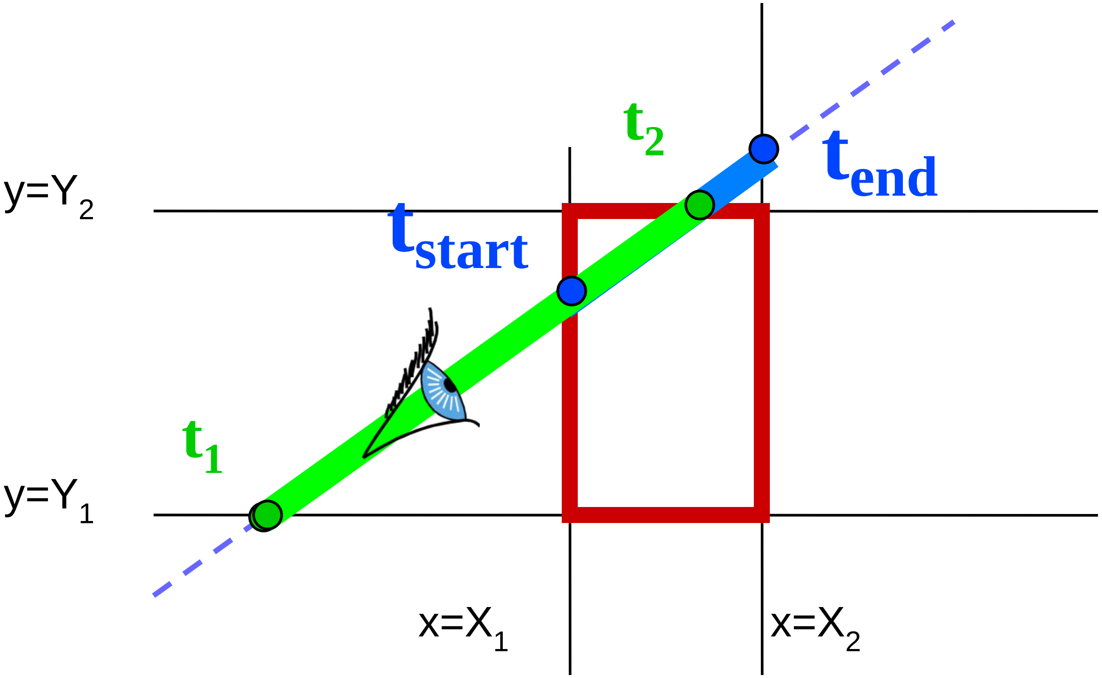
Then Intersect Intervals

- Compute t_1 and t_2 for Y…



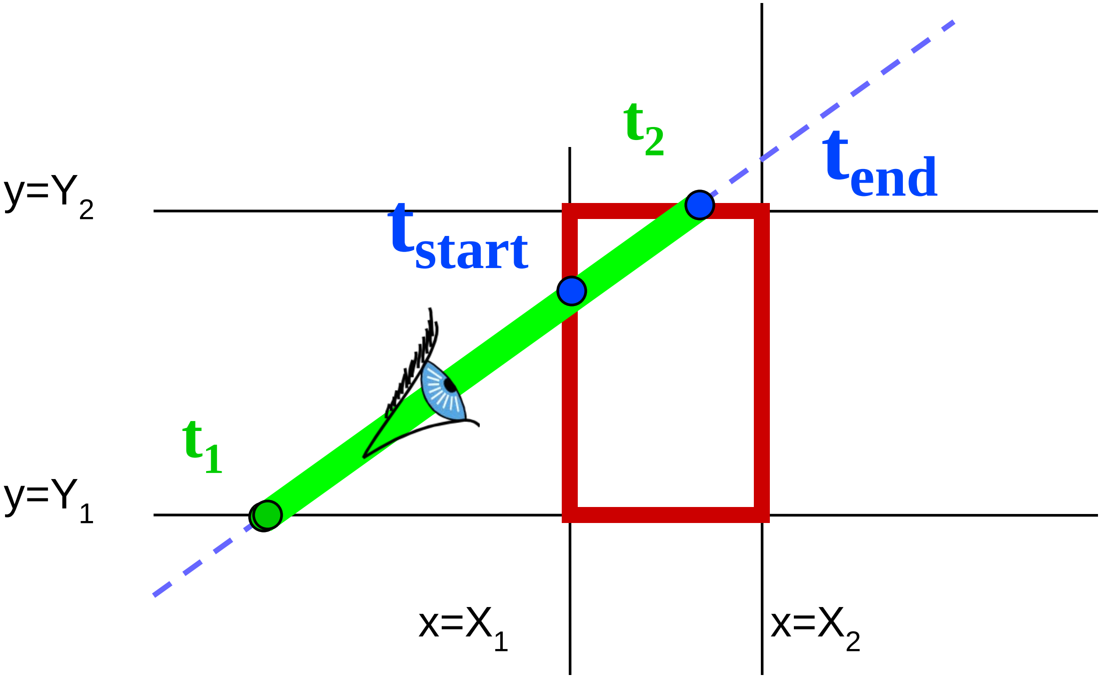
Then Intersect Intervals

- Update t_{start} & t_{end} for each subsequent dimension
  - If $t_1 > t_{start}$, $t_{start} = t_1$
  - If $t_2 < t_{end}$, $t_{end} = t_2$



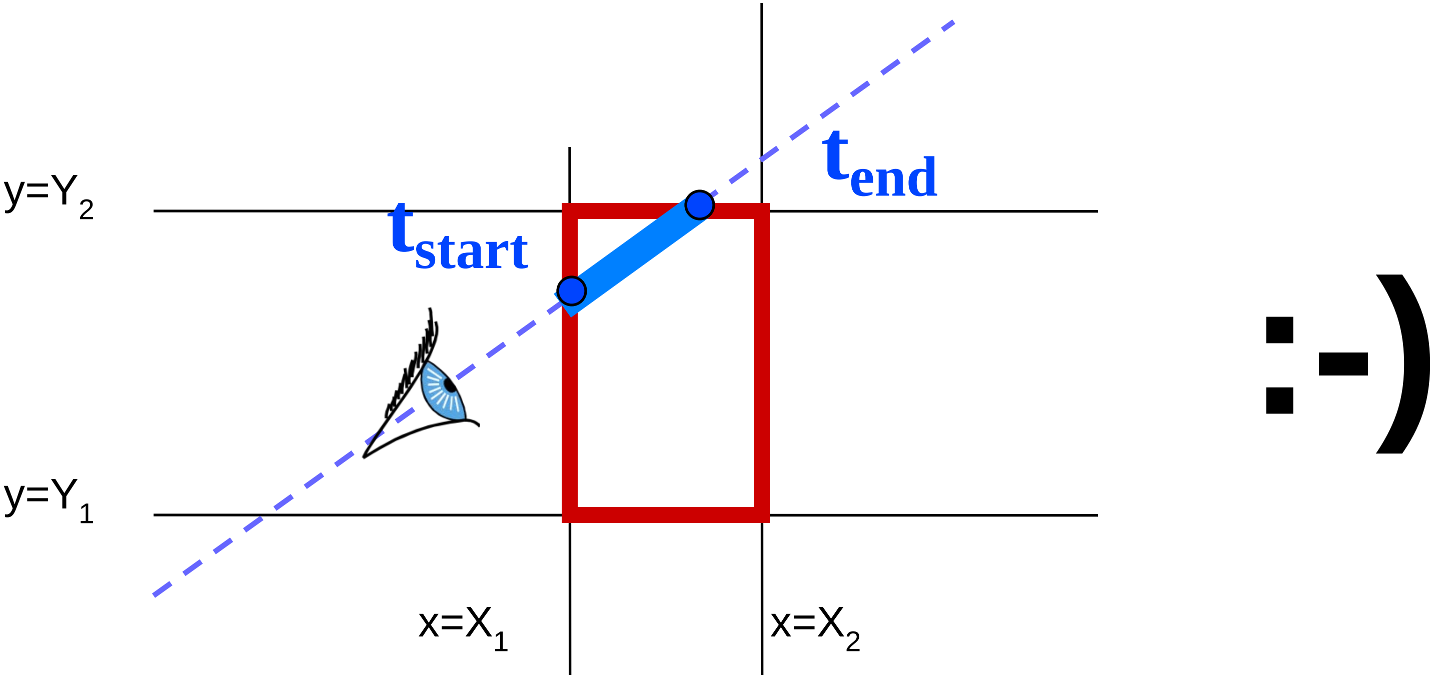
Then Intersect Intervals

- Update t_{start} & t_{end} for each subsequent dimension
  - If $t_1 > t_{start}$, $t_{start} = t_1$
  - If $t_2 < t_{end}$, $t_{end} = t_2$



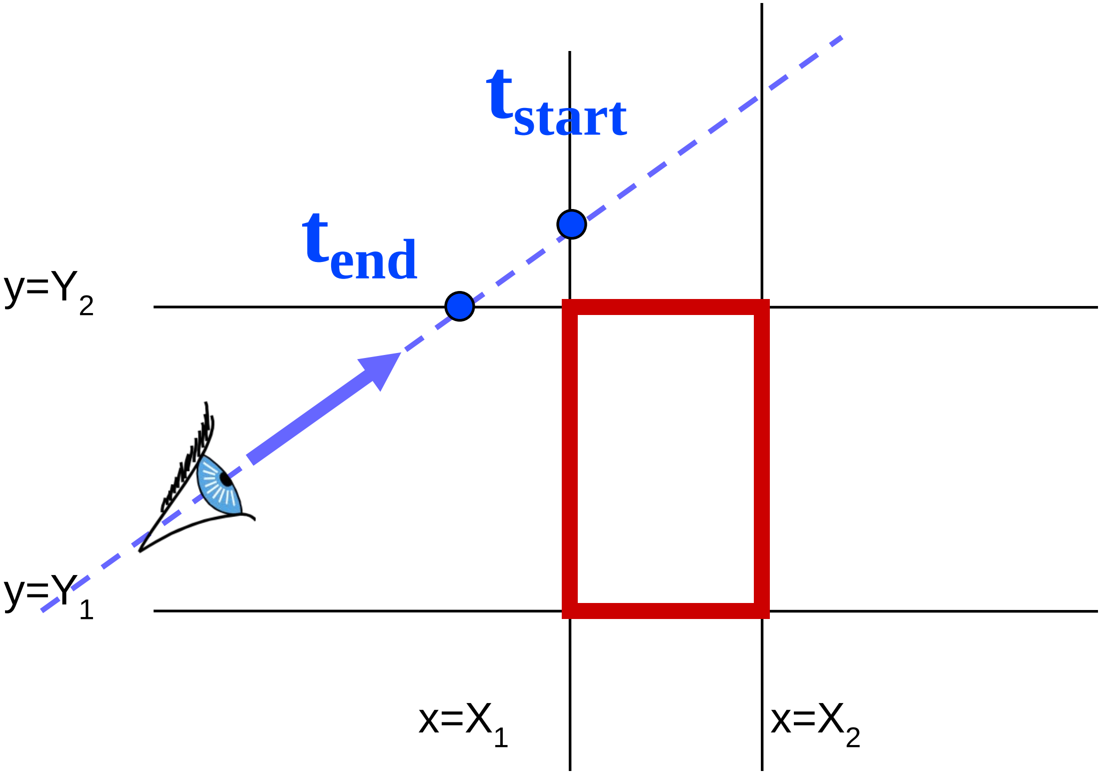
Then Intersect Intervals

- Update t_{start} & t_{end} for each subsequent dimension
  - If $t_1 > t_{start}$, $t_{start} = t_1$
  - If $t_2 < t_{end}$, $t_{end} = t_2$



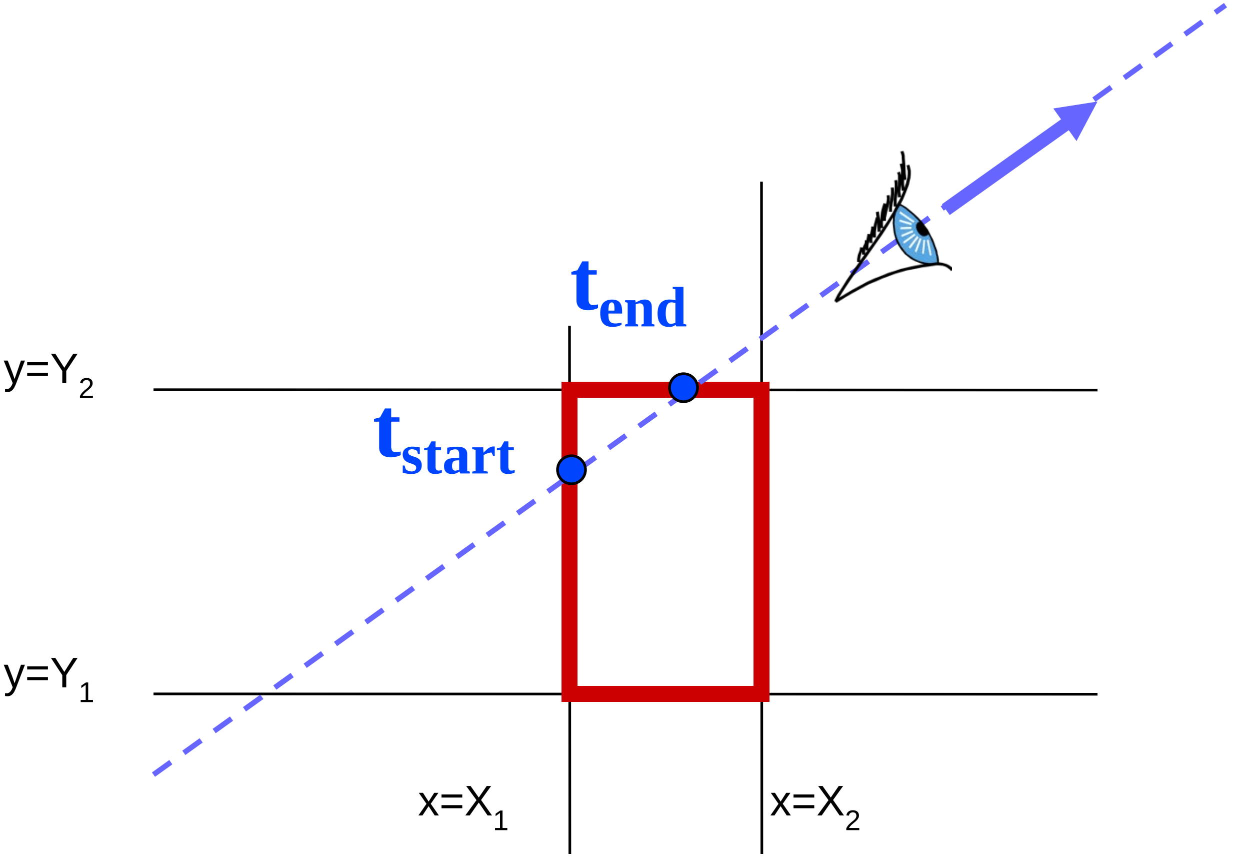
Is there an Intersection?

- If $t_{start} > t_{end}$ → **box is missed**



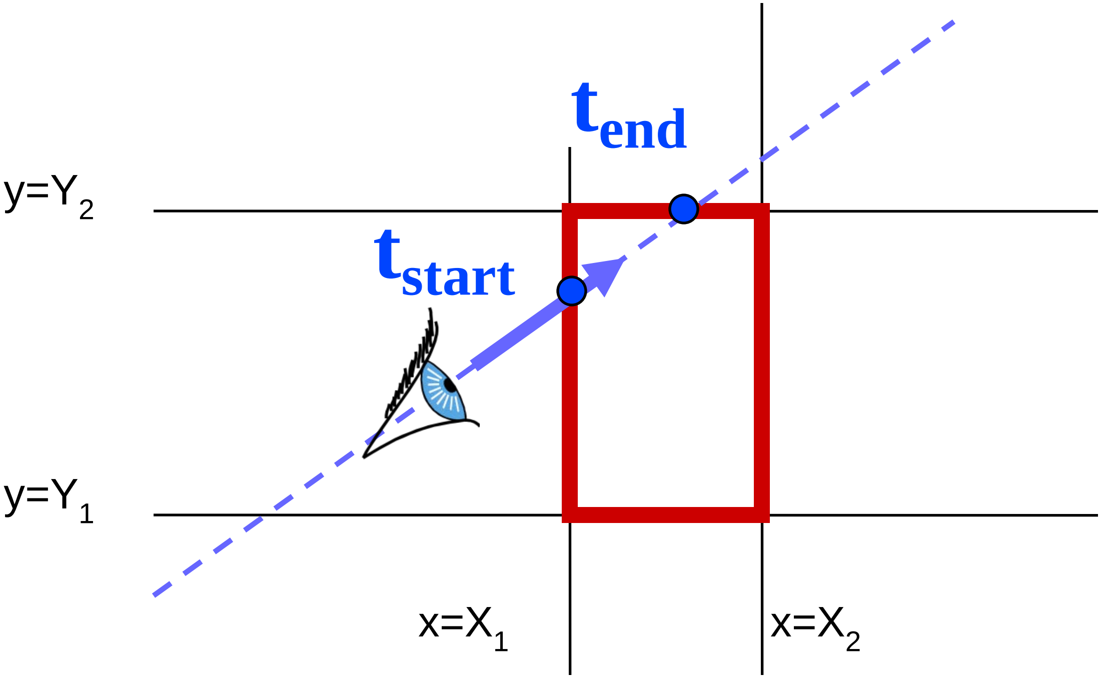
Is the Box Behind the Eyepoint?

- If $t_{end} < t_{min}$ → **box is behind**



Return the Correct Intersection

- If $t_{start} > t_{min}$ → **closest intersection at t_{start}**
- Else → **closest intersection at t_{end}**
  - – Eye is inside box



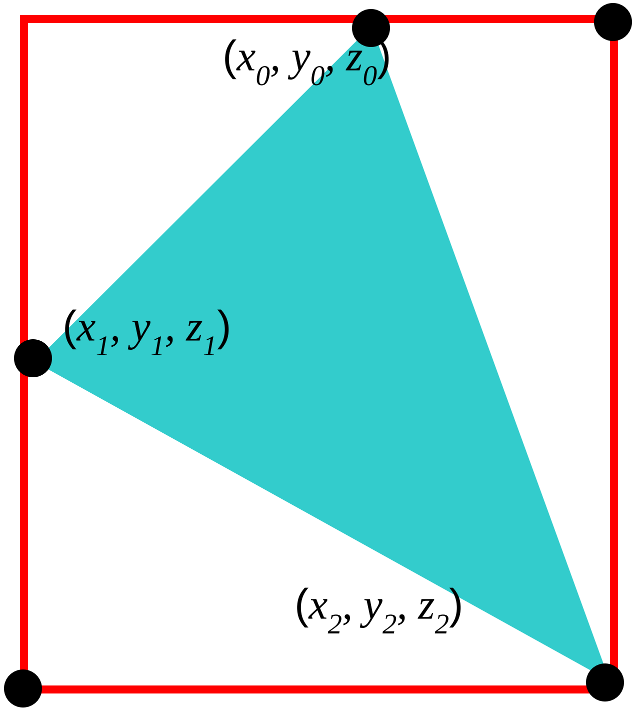
Ray-Box Intersection Summary

- For each dimension,
  - If $R_{dx} = 0$ (ray is parallel) AND
    $R_{ox} < X_1$ or $R_{ox} > X_2$ → **no intersection**
- For each dimension, calculate intersection distances t_1 and t_2
  - $t_1 = (X_1 - R_{ox}) / R_{dx}$ $t_2 = (X_2 - R_{ox}) / R_{dx}$
  - If $t_1 > t_2$, swap
  - Maintain an interval $[t_{start}, t_{end}]$, intersect with current dimension
  - If $t_1 > t_{start}$, $t_{start} = t_1$ If $t_2 < t_{end}$, $t_{end} = t_2$
- If $t_{start} > t_{end}$ → **box is missed**
- If $t_{end} < t_{min}$ → **box is behind**
- If $t_{start} > t_{min}$ → **closest intersection at t_{start}**
- Else → **closest intersection at t_{end}**

Efficiency Issues

- $1/R_{dx}$, $1/R_{dy}$ and $1/R_{dz}$ can be pre-computed and shared for many boxes

Bounding Box of a Triangle



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

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

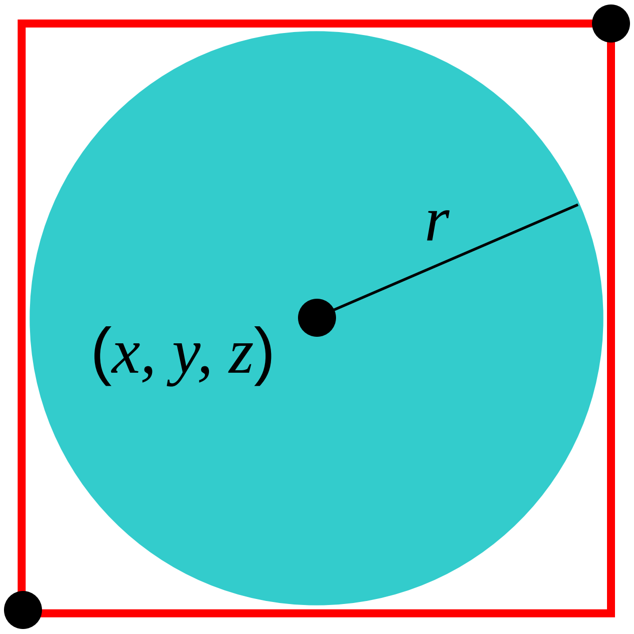
$(x_{min}, y_{min}, z_{min})$

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

Bounding Box of a Sphere

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

$= (x+r,\ y+r,\ z+r)$



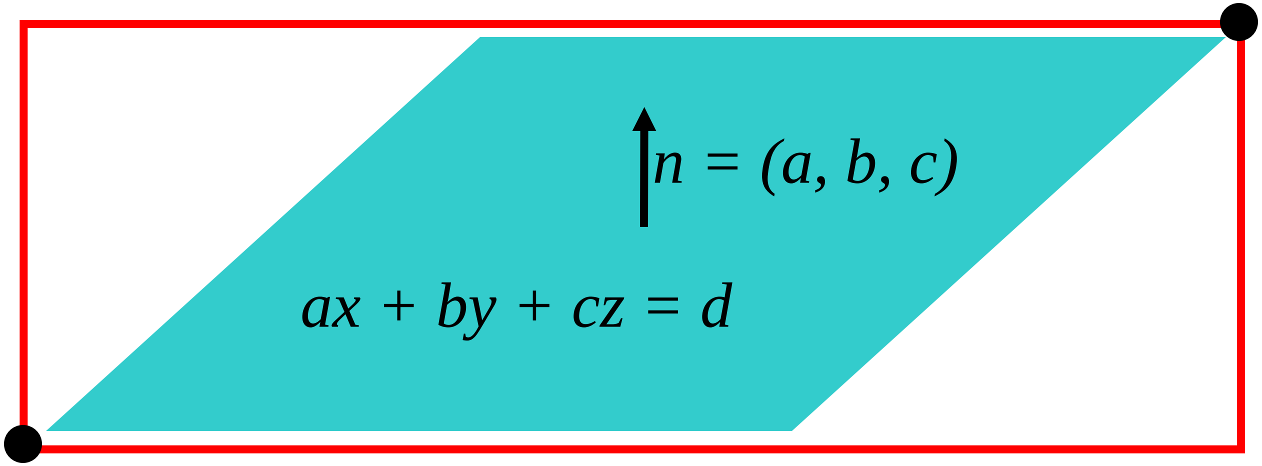
$(x_{min}, y_{min}, z_{min})$

$= (x-r,\ y-r,\ z-r)$

Bounding Box of a Plane

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

$= (+\infty, +\infty, +\infty)$*



$n = (a, b, c)$

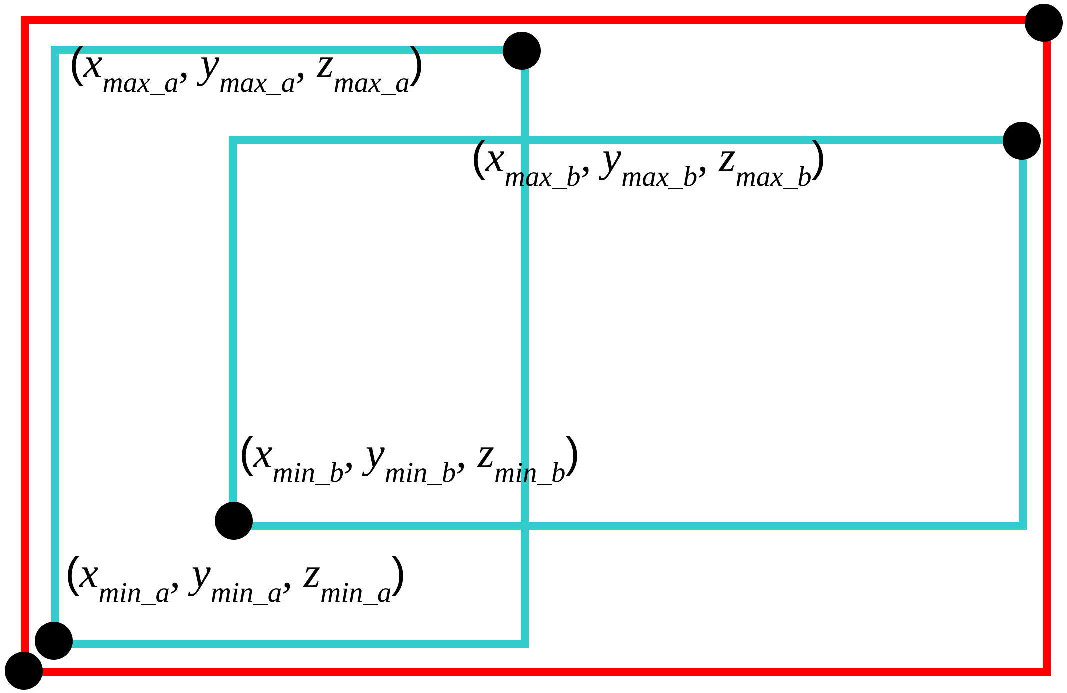
$ax + by + cz = d$

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

$= (-\infty, -\infty, -\infty)$*

* *unless n is exactly perpendicular to an axis*

Bounding Box of a Group


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

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

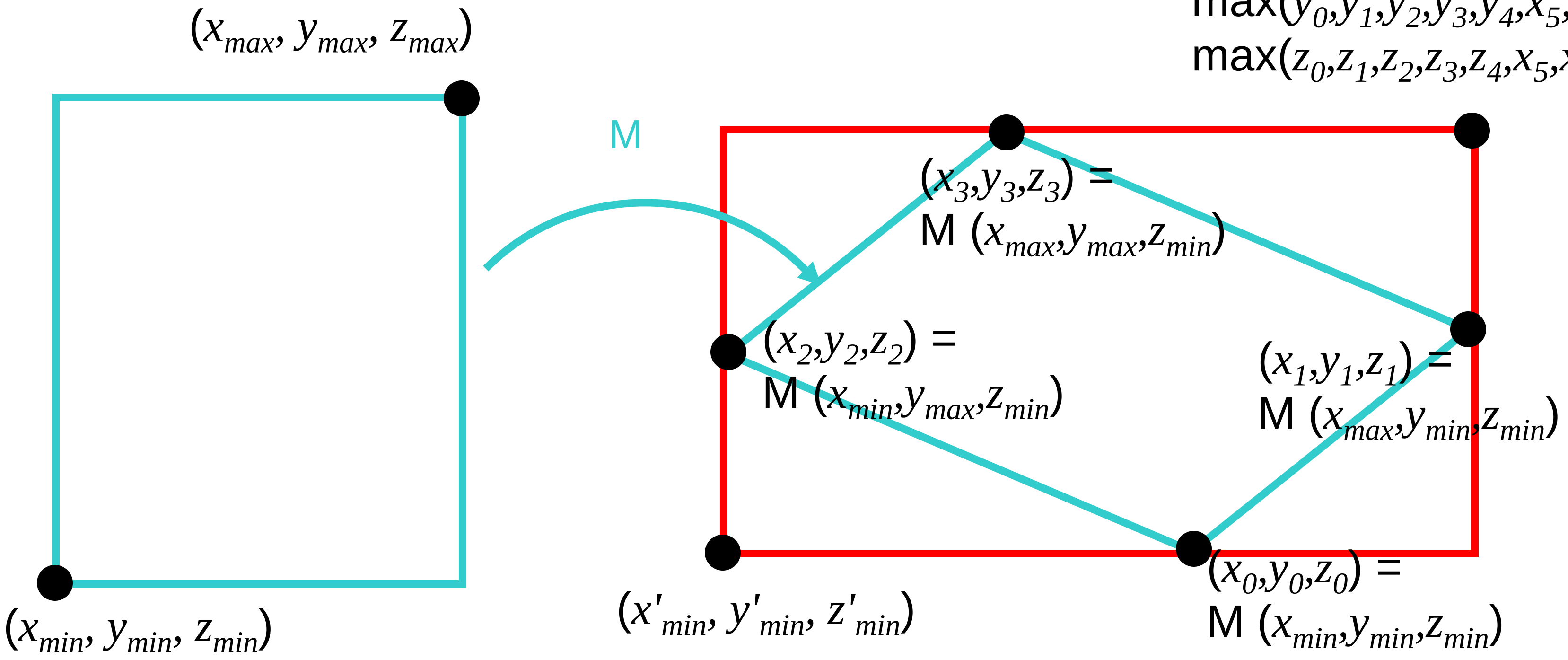
$$(x_{min}, y_{min}, z_{min}) = (\min(x_{min\_a},x_{min\_b}), \min(y_{min\_a},y_{min\_b}), \min(z_{min\_a},z_{min\_b}))$$

Bounding Box of a Transform

Bounding box of transformed object IS NOT the transformation of the bounding box!

$(x'_{max}, y'_{max}, z'_{max})$

$= (\max(x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7),$
$\max(y_0,y_1,y_2,y_3,y_4,x_5,x_6,x_7),$
$\max(z_0,z_1,z_2,z_3,z_4,x_5,x_6,x_7))$



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

M

$(x_3,y_3,z_3) =$
$M\ (x_{max},y_{max},z_{min})$

$(x_2,y_2,z_2) =$
$M\ (x_{min},y_{max},z_{min})$

$(x_1,y_1,z_1) =$
$M\ (x_{max},y_{min},z_{min})$

$(x_0,y_0,z_0) =$
$M\ (x_{min},y_{min},z_{min})$

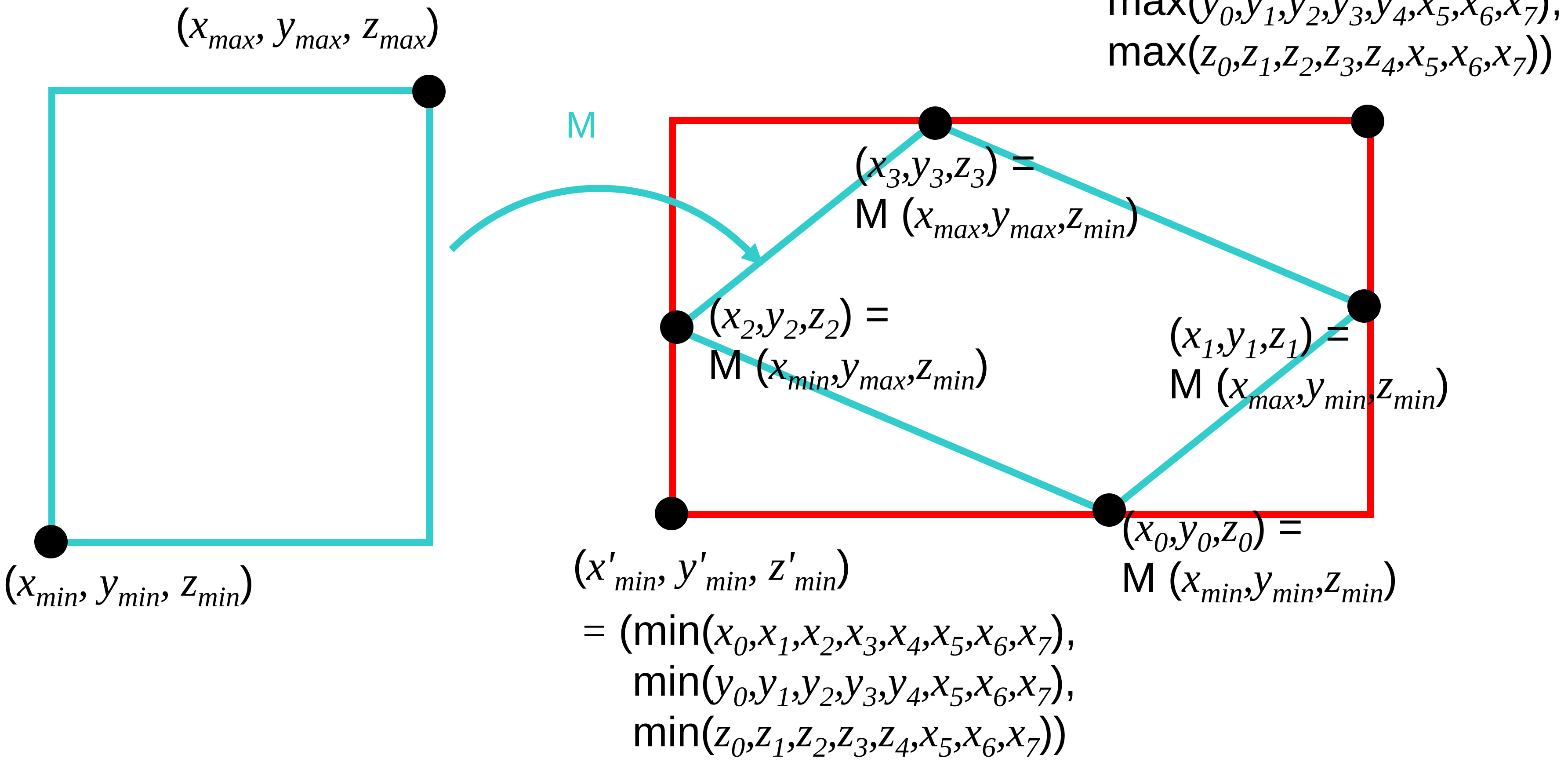
$(x_{min}, y_{min}, z_{min})$

$(x'_{min}, y'_{min}, z'_{min})$

$= (\min(x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7),$
$\min(y_0,y_1,y_2,y_3,y_4,x_5,x_6,x_7),$
$\min(z_0,z_1,z_2,z_3,z_4,x_5,x_6,x_7))$

Bounding Box of a Transform

Bounding box of transformed object IS NOT the transformation of the bounding box!



$(x'_{max}, y'_{max}, z'_{max})$

$= (\max(x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7),$
$\max(y_0,y_1,y_2,y_3,y_4,x_5,x_6,x_7),$
$\max(z_0,z_1,z_2,z_3,z_4,x_5,x_6,x_7))$

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

M

$(x_3,y_3,z_3) =$
$M\ (x_{max},y_{max},z_{min})$

$(x_2,y_2,z_2) =$
$M\ (x_{min},y_{max},z_{min})$

$(x_1,y_1,z_1) =$
$M\ (x_{max},y_{min},z_{min})$

$(x_0,y_0,z_0) =$
$M\ (x_{min},y_{min},z_{min})$

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

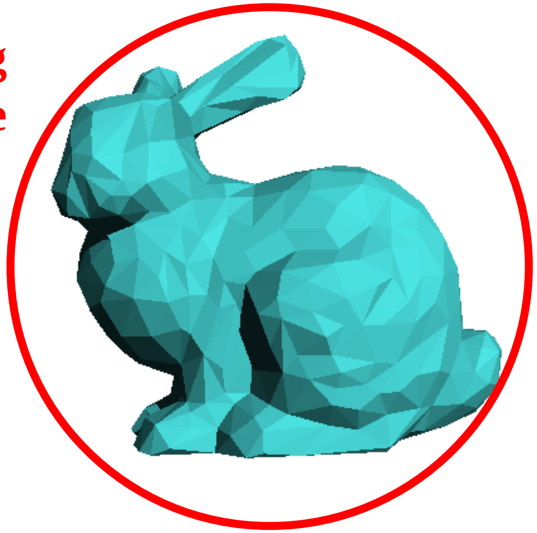
$(x'_{min}, y'_{min}, z'_{min})$

$= (\min(x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7),$
$\min(y_0,y_1,y_2,y_3,y_4,x_5,x_6,x_7),$
$\min(z_0,z_1,z_2,z_3,z_4,x_5,x_6,x_7))$

Questions?

Are Bounding Volumes Enough?

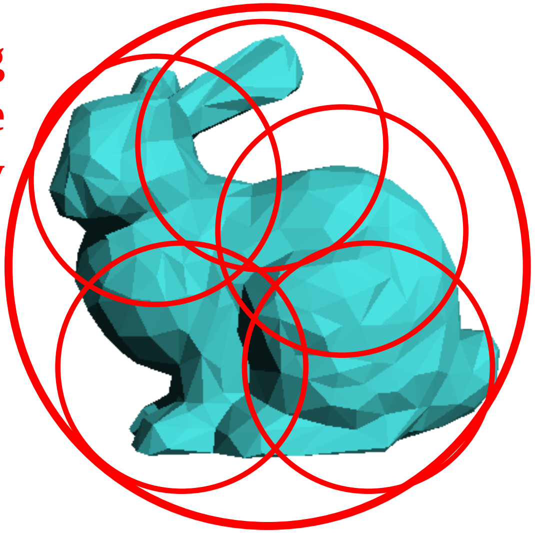
- If ray hits bounding volume,
  must we test all primitives inside it?
  - Lots of work, think of a 1M-triangle mesh



bounding sphere

Bounding Volume Hierarchies

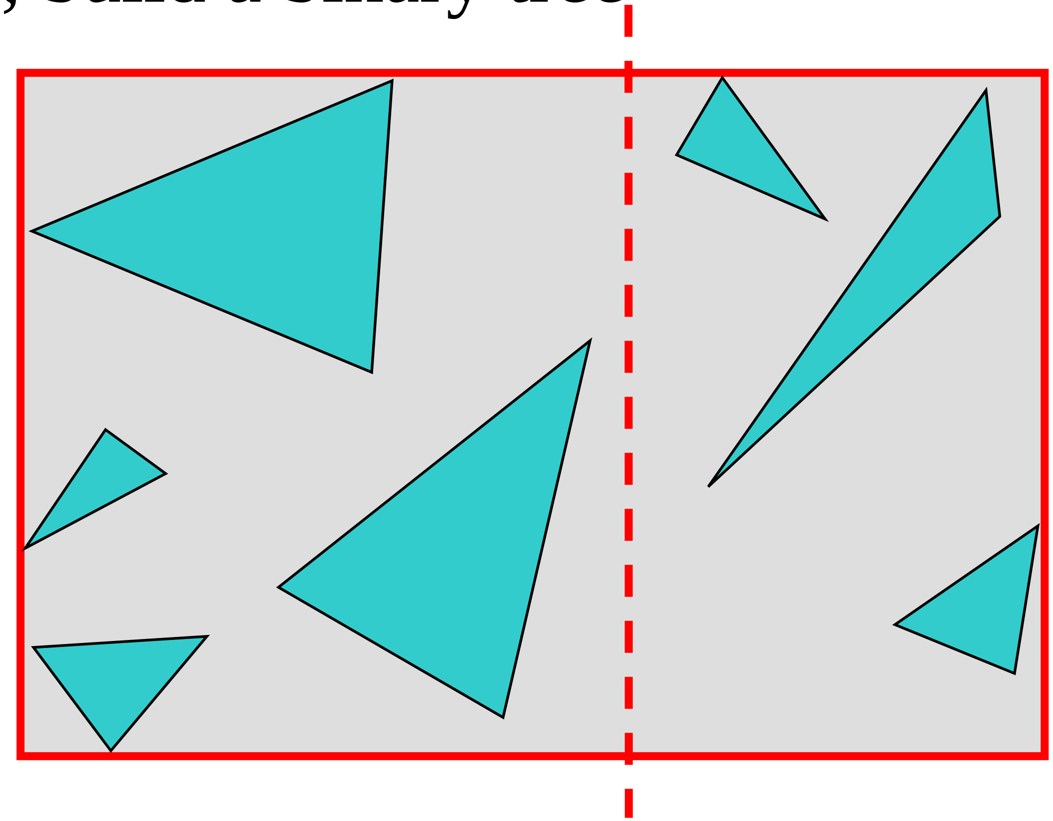
- If ray hits bounding volume,
  must we test all primitives inside it?
  - Lots of work, think of a 1M-triangle mesh
- You guessed it already, we'll split the primitives in groups and build recursive bounding volumes
  - Like collision detection, remember?



bounding sphere hierarchy

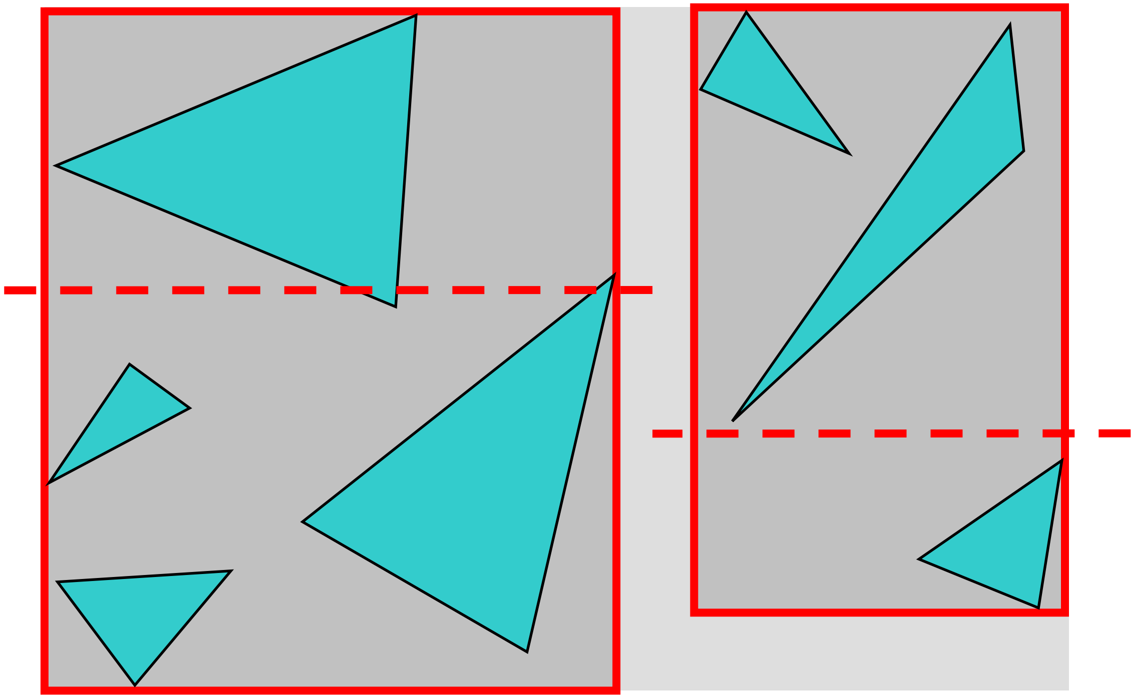
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree



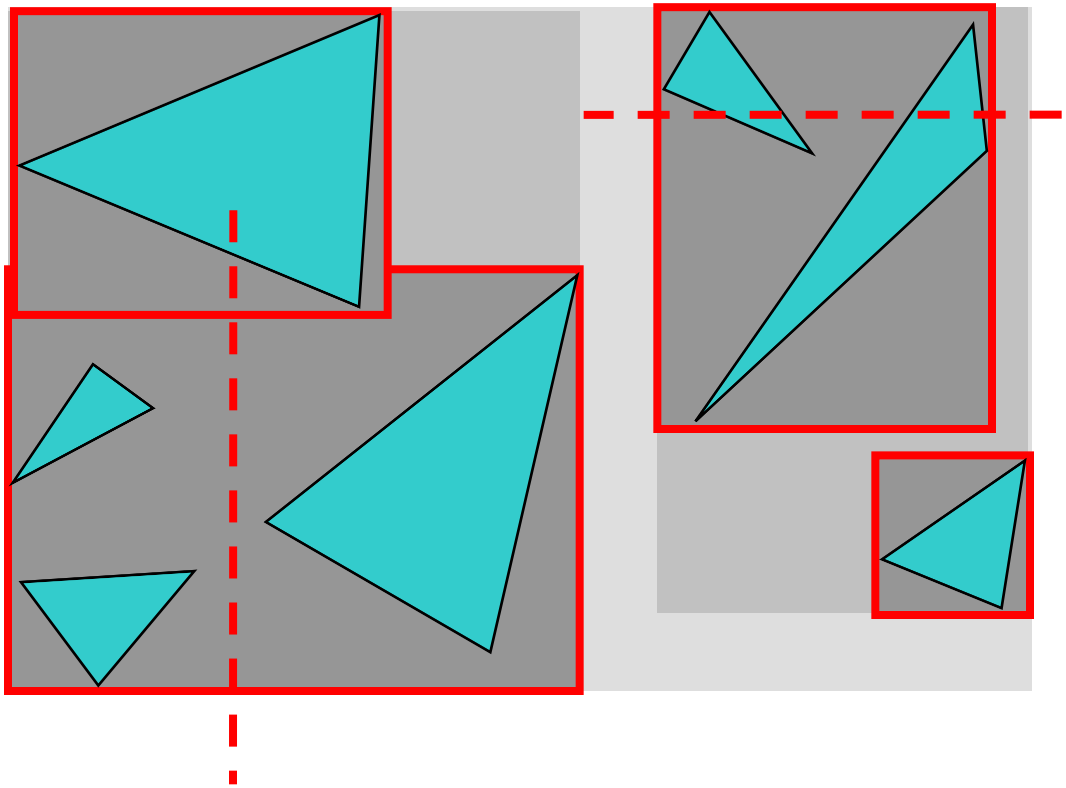
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree



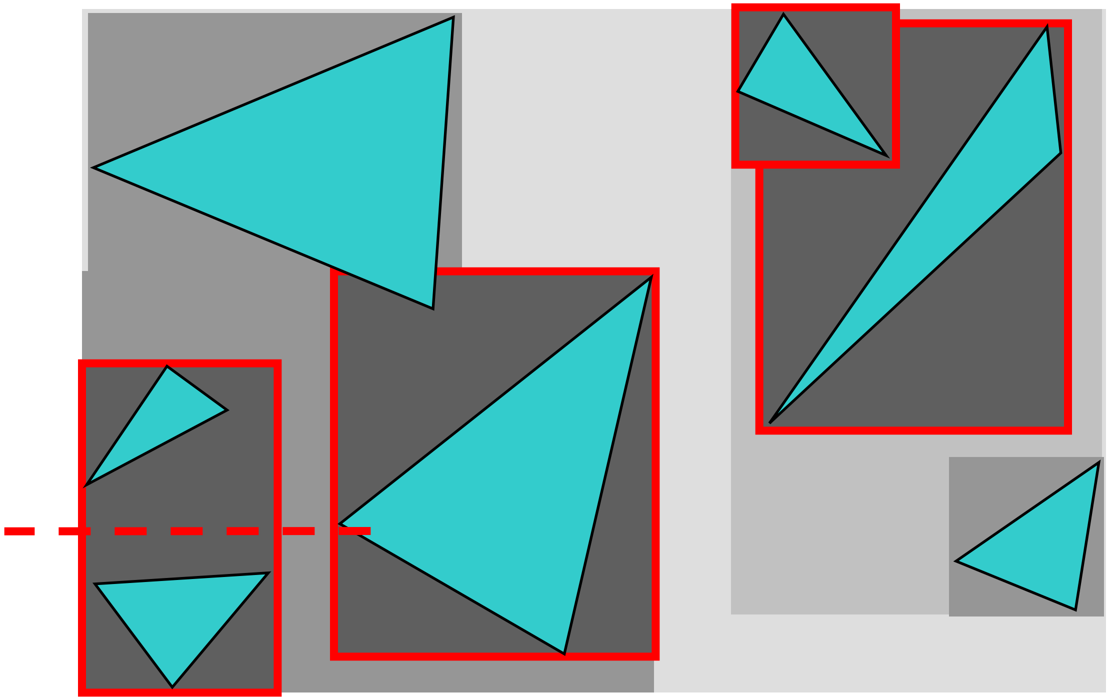
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree



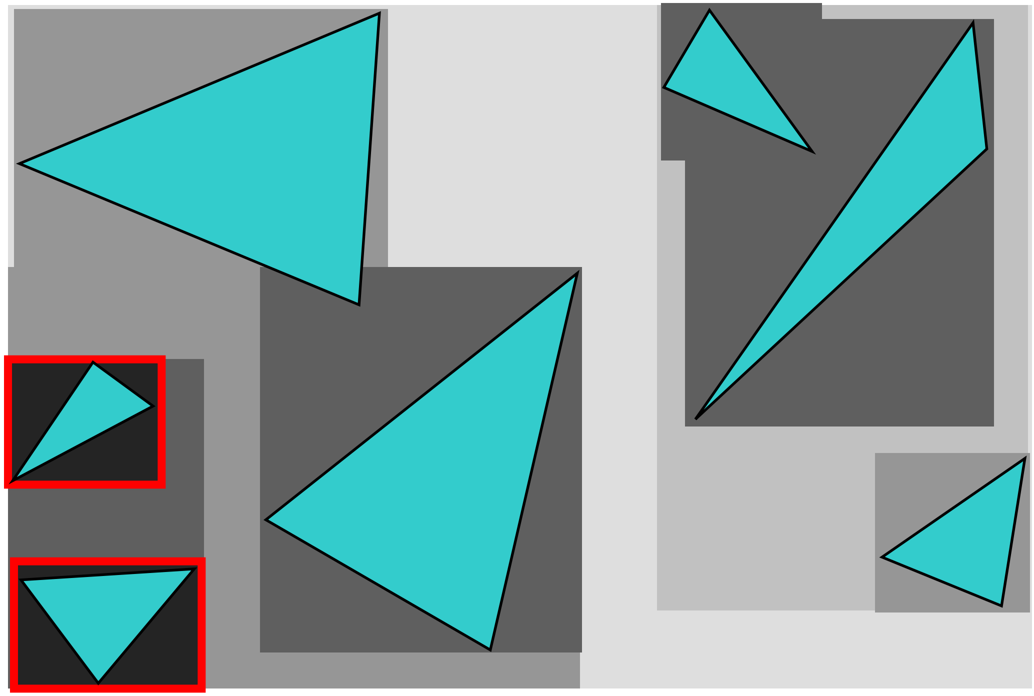
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree



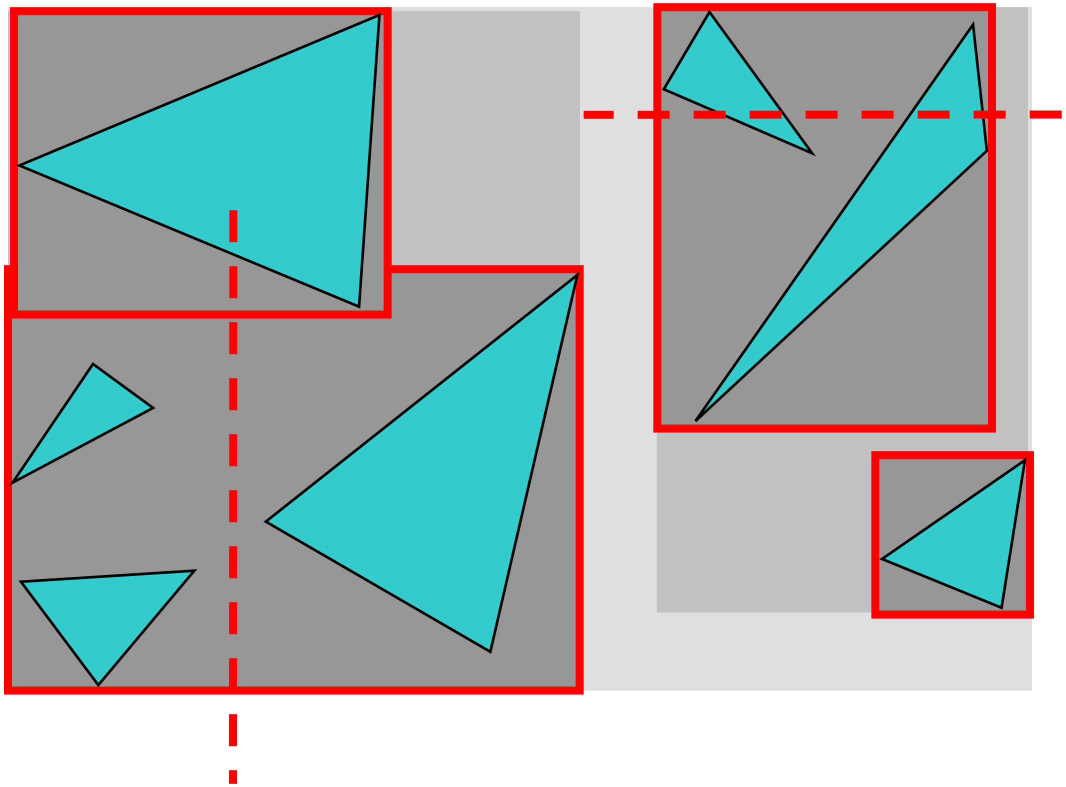
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree



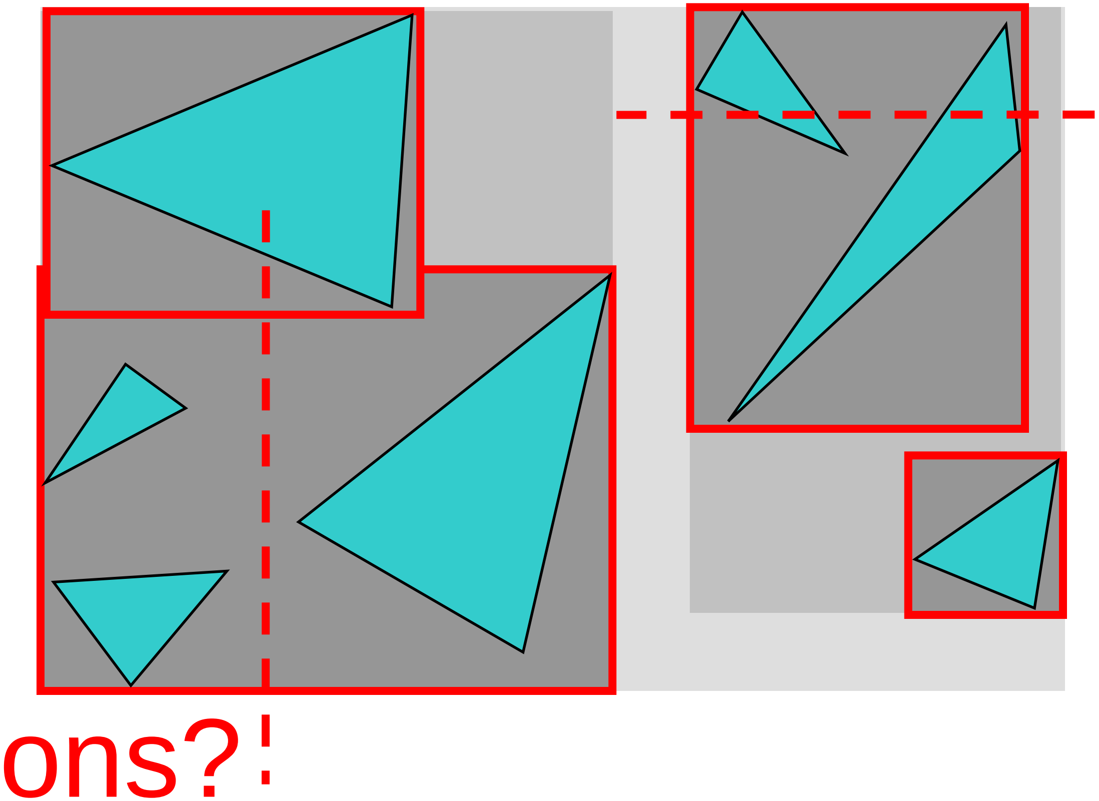
Where to Split Objects?

- At midpoint of current volume *OR*
- Sort, and put half of the objects on each side *OR*
- Use modeling hierarchy



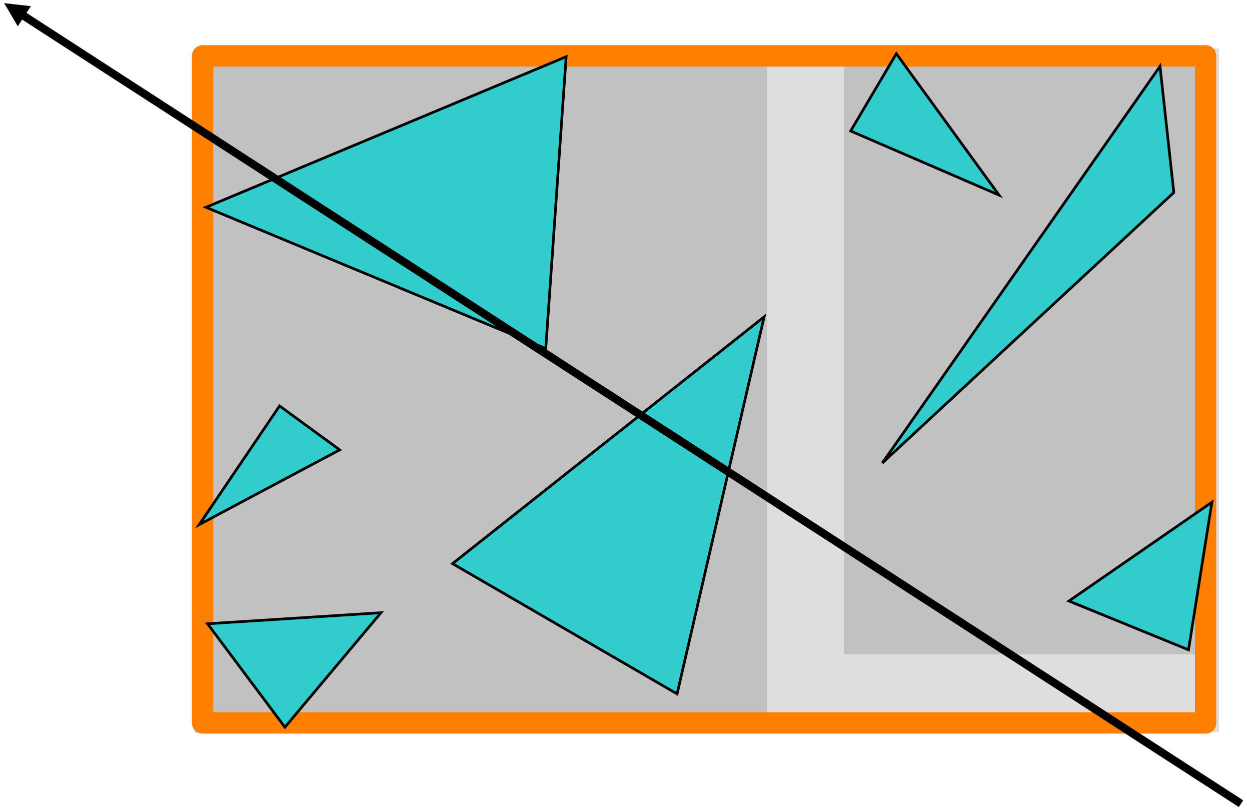
Where to Split Objects?

- At midpoint of current volume *OR*
- Sort, and put half of the objects on each side *OR*
- Use modeling hierarchy

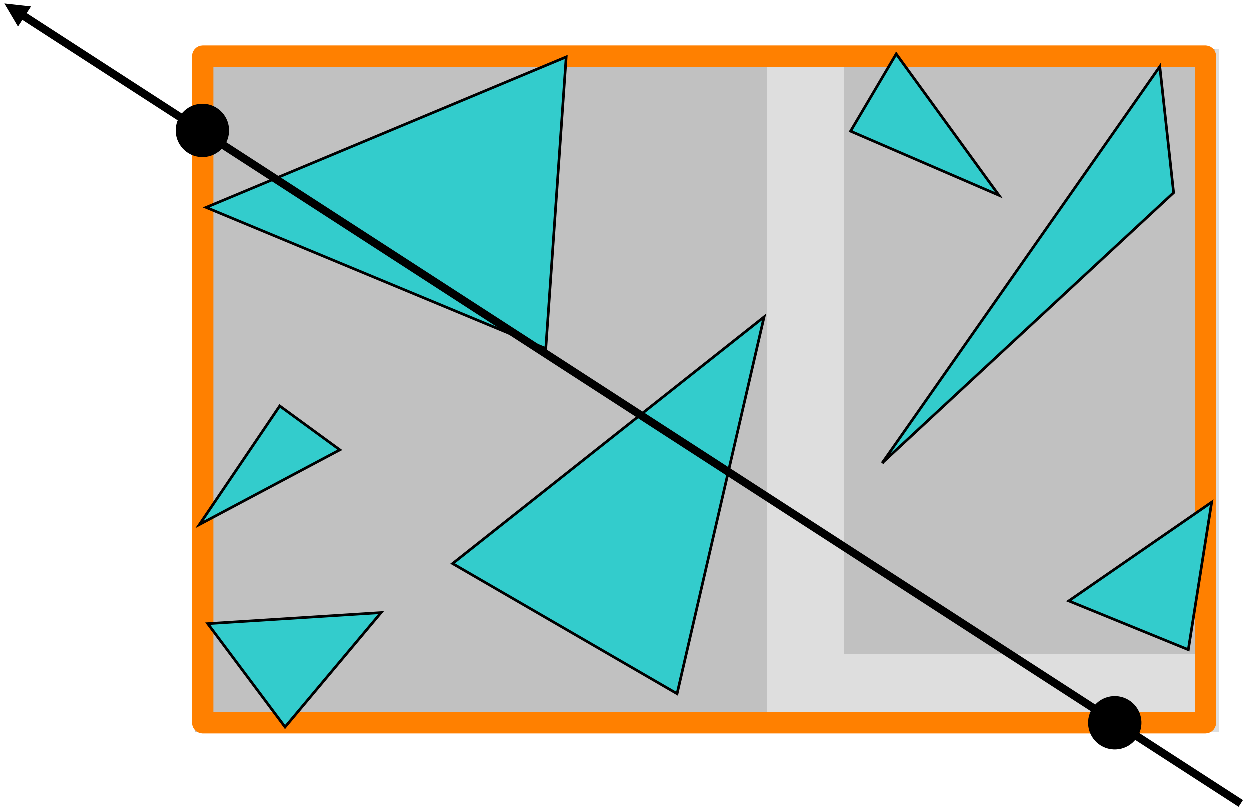


Questions?

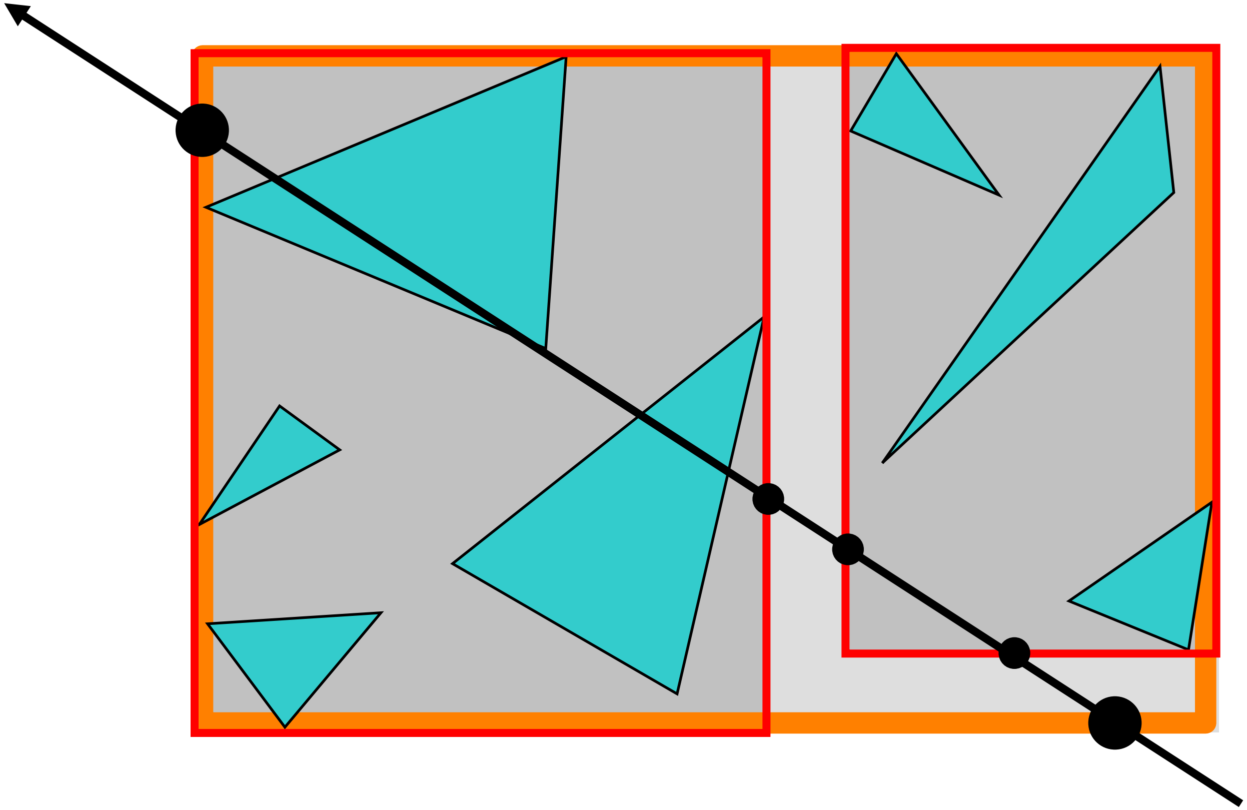
Ray-BVH Intersection



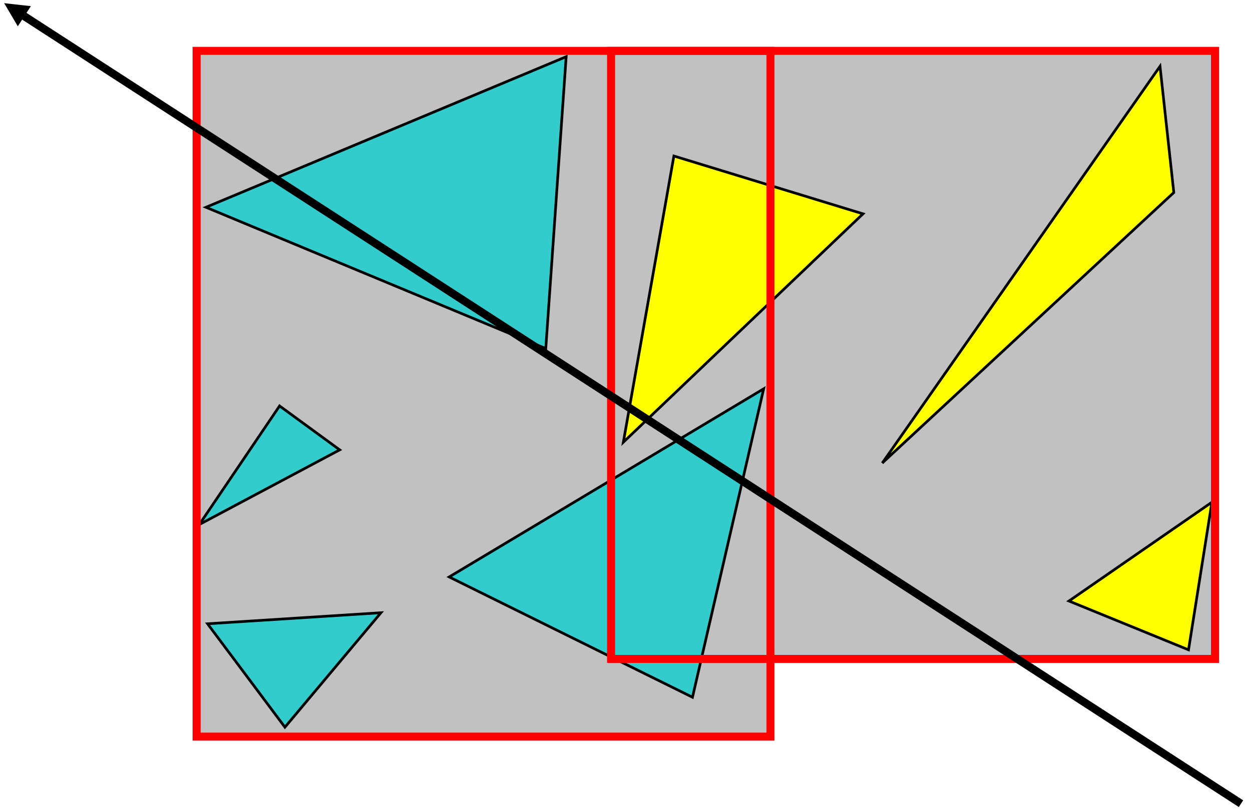
Ray-BVH Intersection



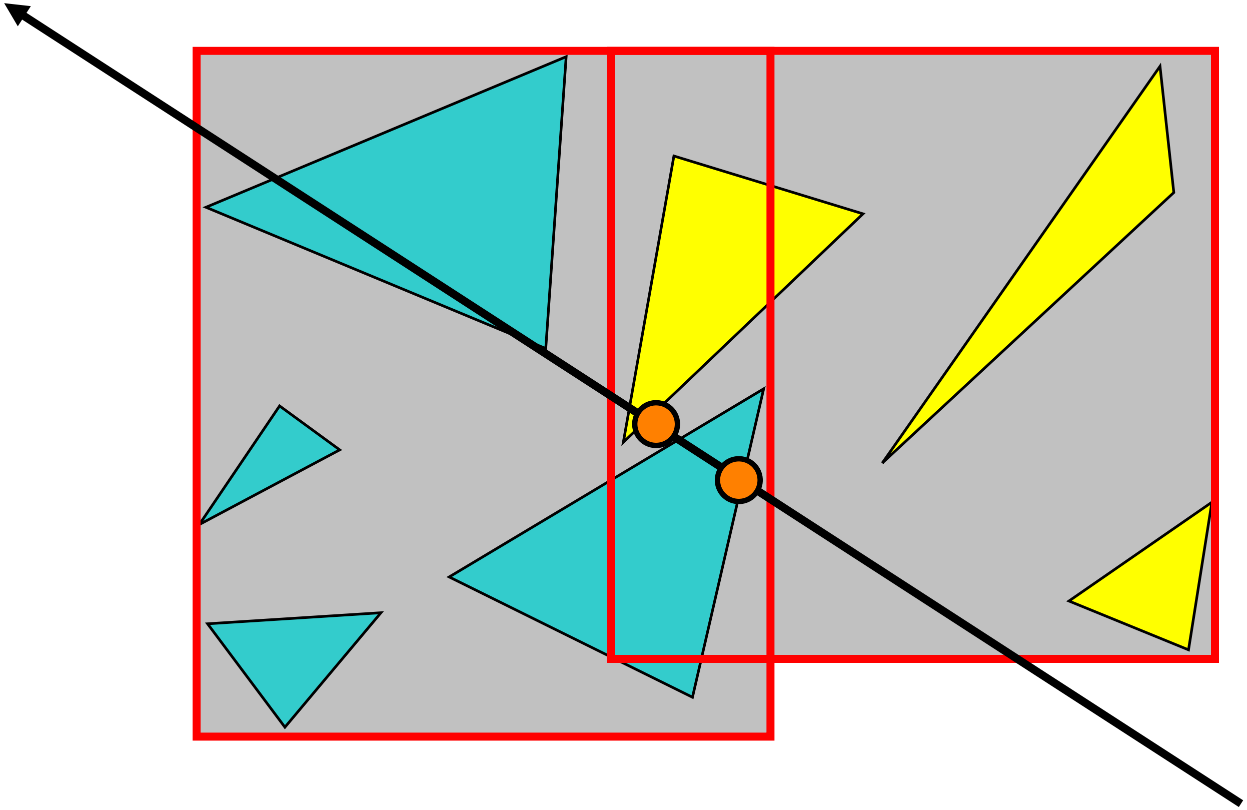
Ray-BVH Intersection



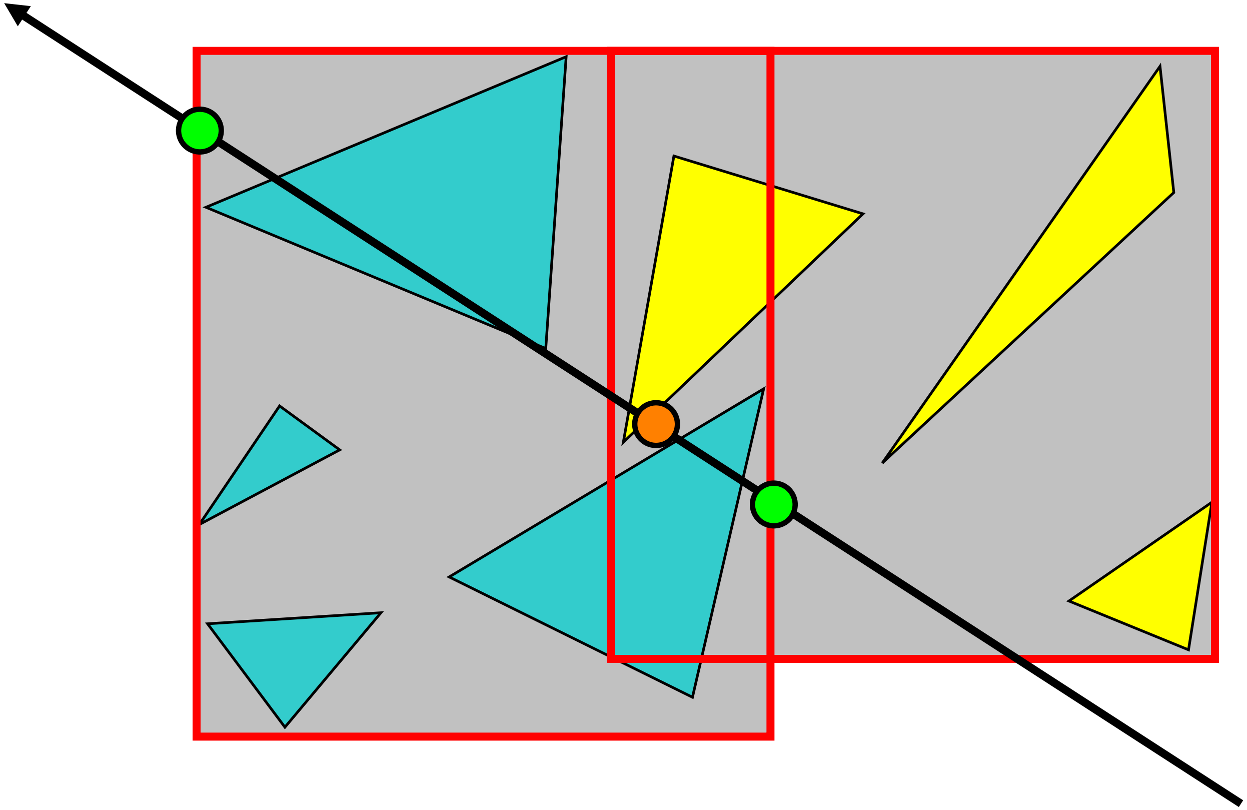
Intersection with BVH



Intersection with BVH



Intersection with BVH



BVH Discussion

- Advantages
  - easy to construct
  - easy to traverse
  - binary tree (=simple structure)

- Disadvantages
  - may be difficult to choose a good split for a node
  - poor split may result in minimal spatial pruning

BVH Discussion

- Advantages
  - easy to construct
  - easy to traverse
  - binary tree (=simple structure)

- Disadvantages
  - may be difficult to choose a good split for a node
  - poor split may result in minimal spatial pruning

- Still one of the best methods
  - **Recommended for your first hierarchy!**

BVH Discussion

Questions?

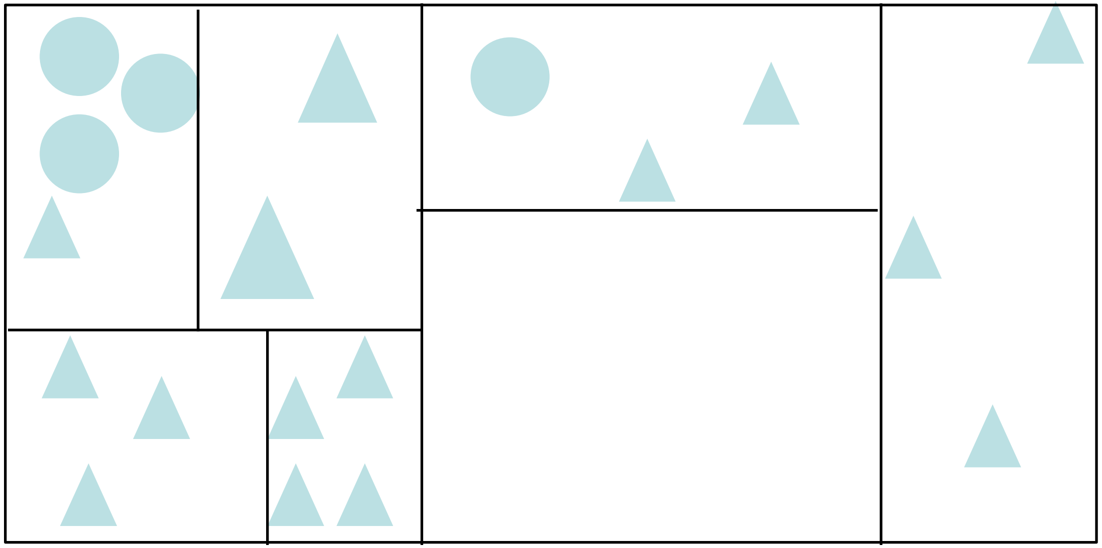
- Advantages
  - easy to construct
  - easy to traverse
  - binary tree (=simple structure)

- Disadvantages
  - may be difficult to choose a good split for a node
  - poor split may result in minimal spatial pruning

- Still one of the best methods
  - **Recommended for your first hierarchy!**

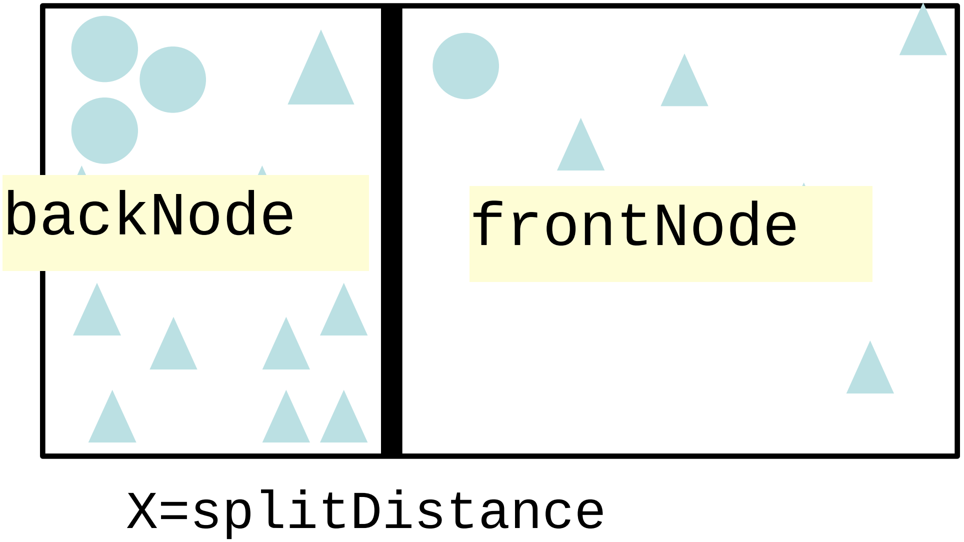
Kd-trees

- Probably most popular acceleration structure
- Binary tree, axis-aligned splits
  - Each node splits space in half along an axis-aligned plane
- A **space partition:** The nodes do not overlap!
  - This is in contrast to BVHs



Data Structure

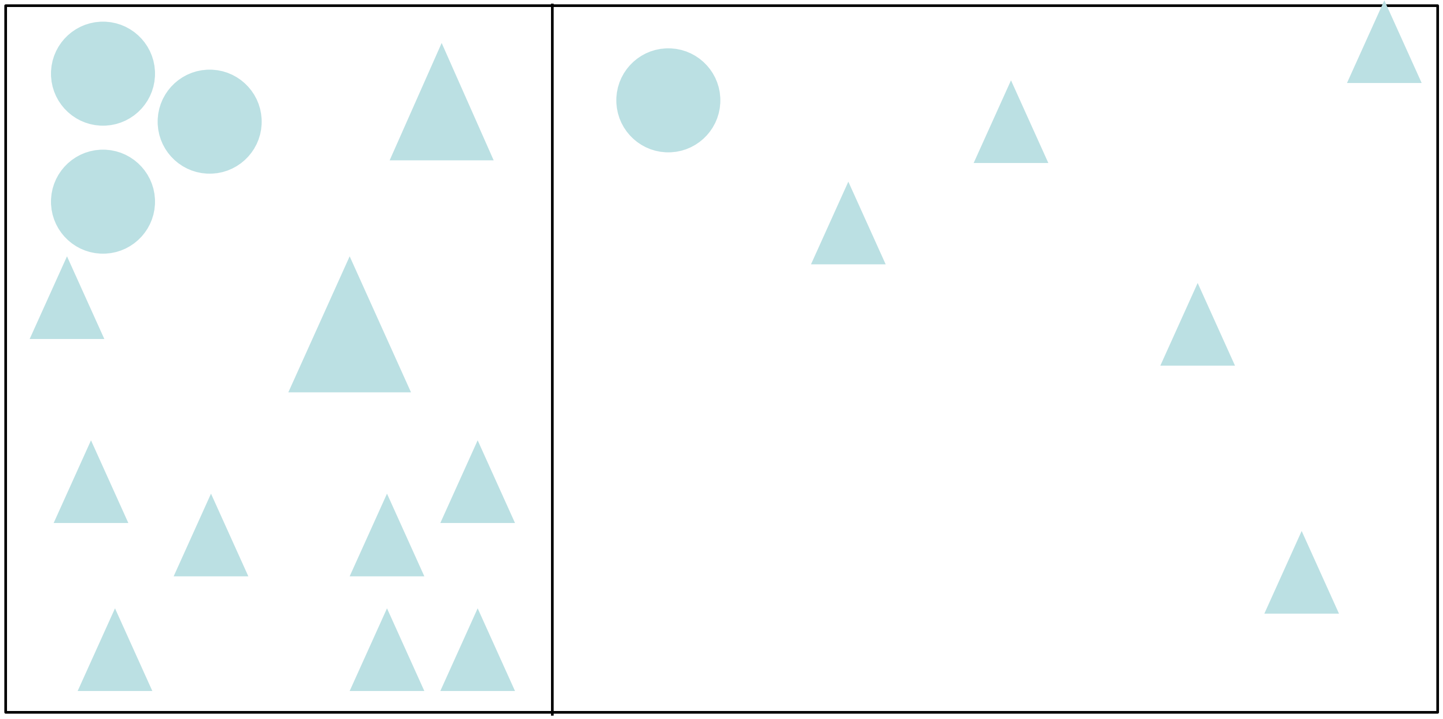
```
KdTreeNode:
      KdTreeNode* backNode, frontNode //children
      int dimSplit // either x, y or z
      float splitDistance
         // from origin along split axis
      boolean isLeaf
      List of triangles //only for leaves
```



here dimSplit = 0 (x axis)

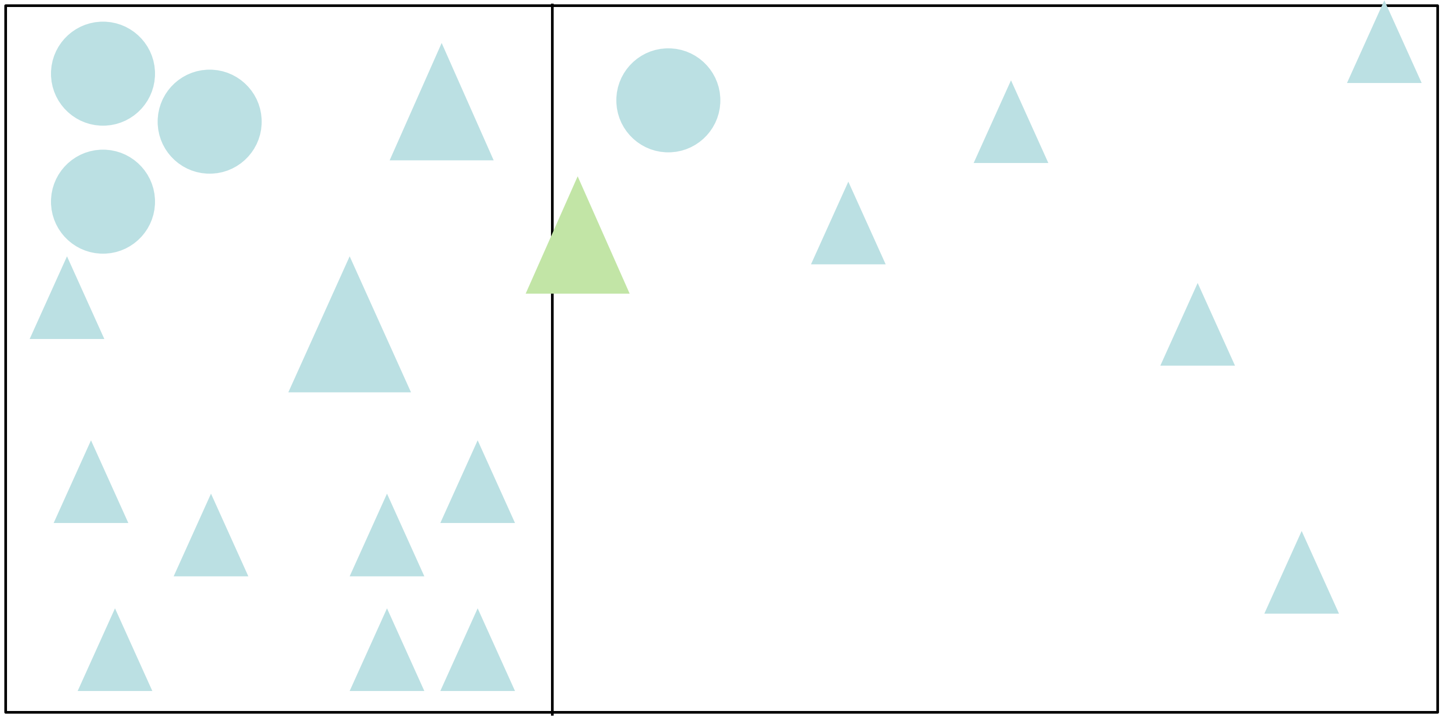
Kd-tree Construction

- Start with scene axis-aligned bounding box
- Decide which dimension to split (e.g. longest)
- Decide at which distance to split (not so easy)



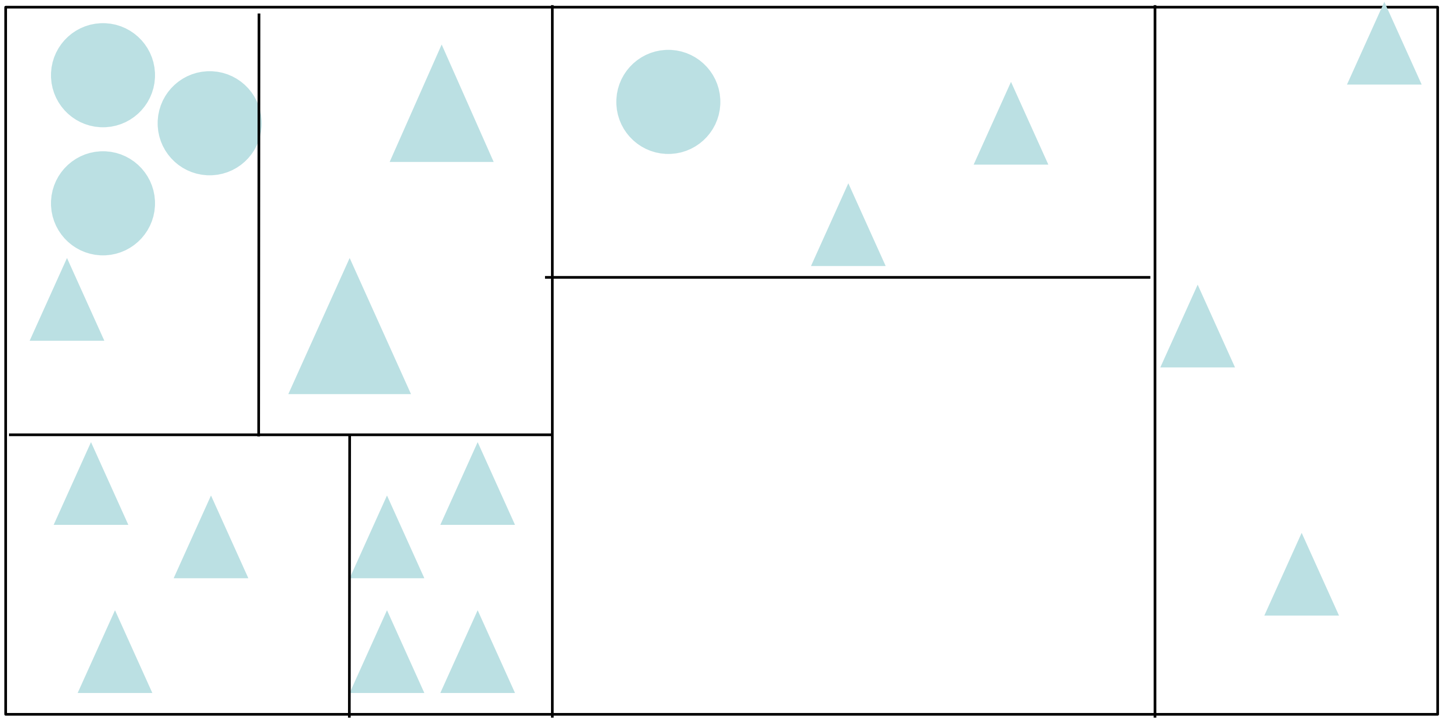
Kd-tree Construction - Split

- Distribute primitives to each side
- If a primitive overlaps split plane, assign to both sides



Kd-tree Construction - Recurse

- Stop when minimum number of primitives reached
- Other stopping criteria possible



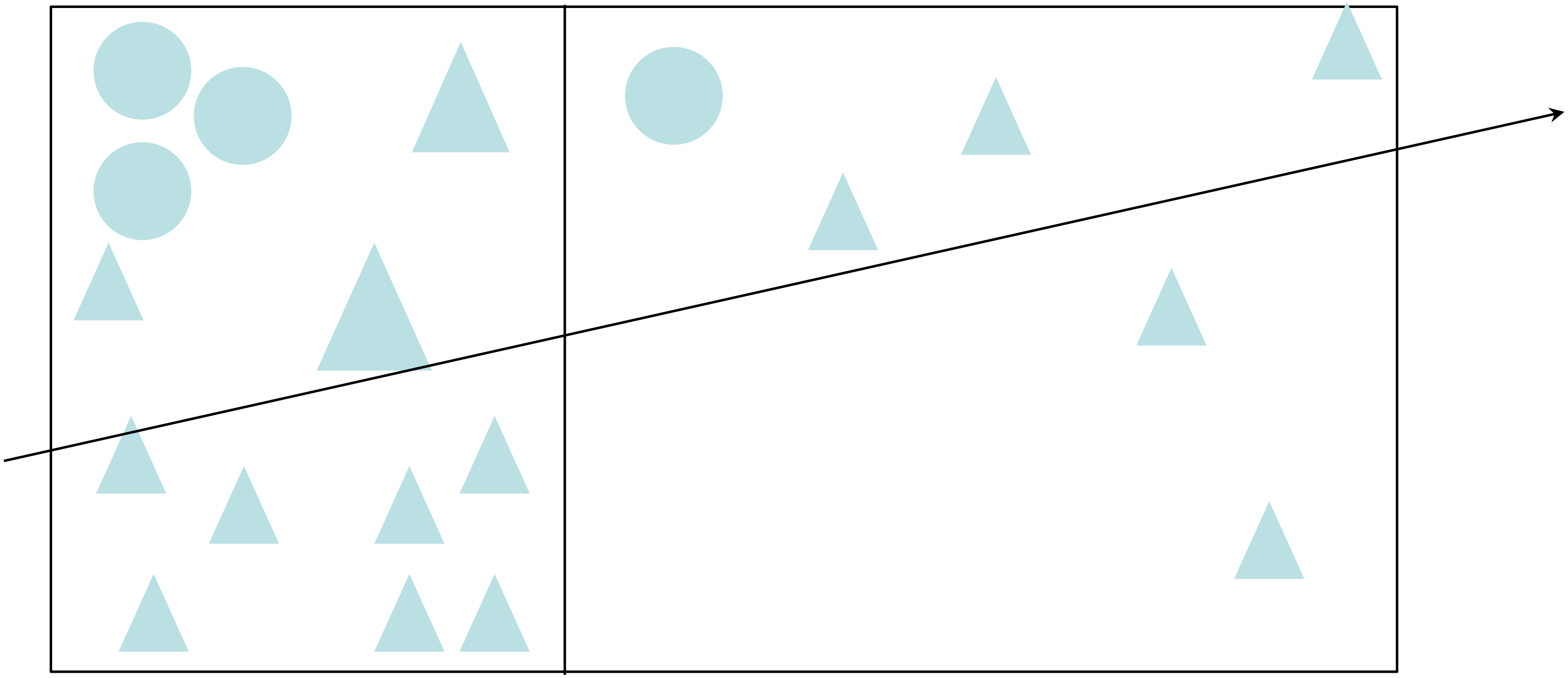
Questions?

- Further reading on efficient Kd-tree construction
  - Hunt, Mark & Stoll, IRT 2006
  - Zhou et al., SIGGRAPH Asia 2008

Zhou et al.

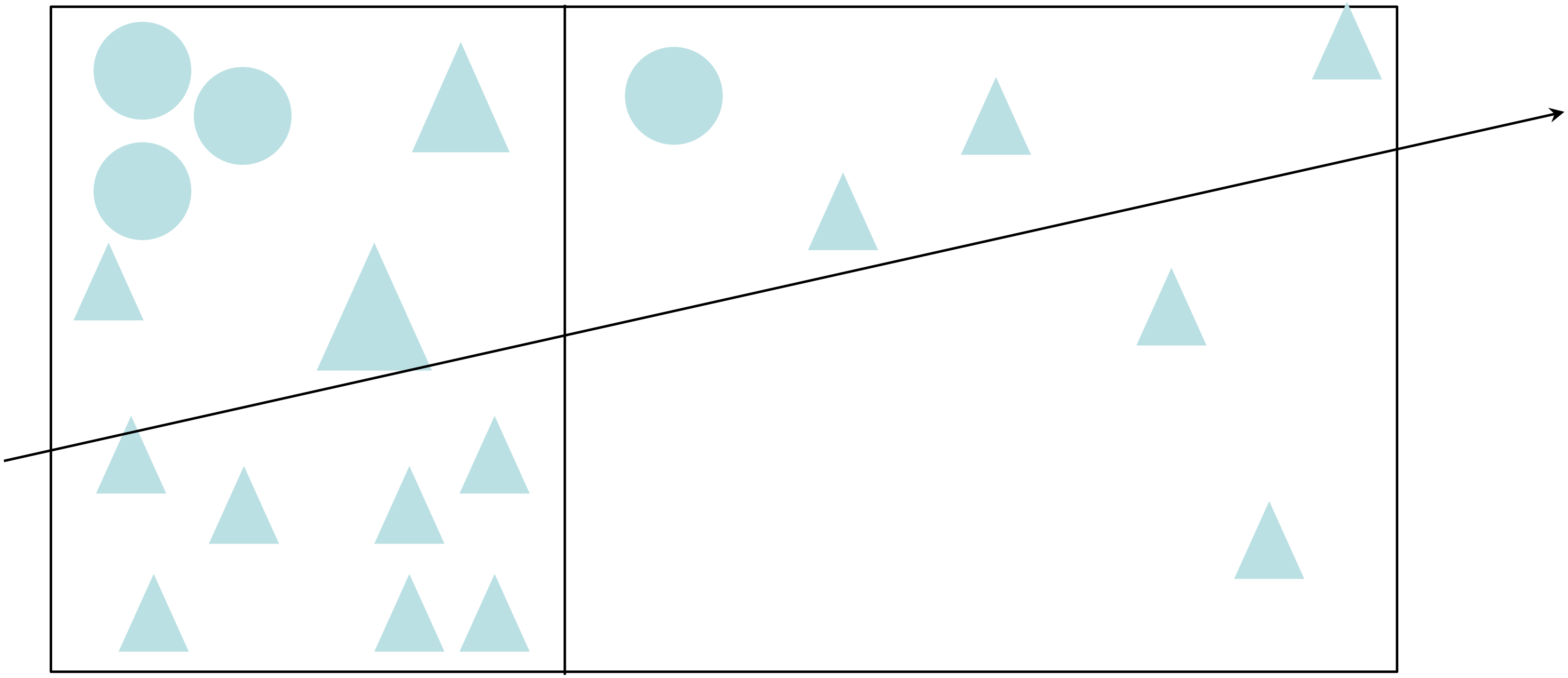
Kd-tree Traversal - High Level

- If leaf, intersect with list of primitives
- If intersects back child, recurse
- If intersects front child, recurse



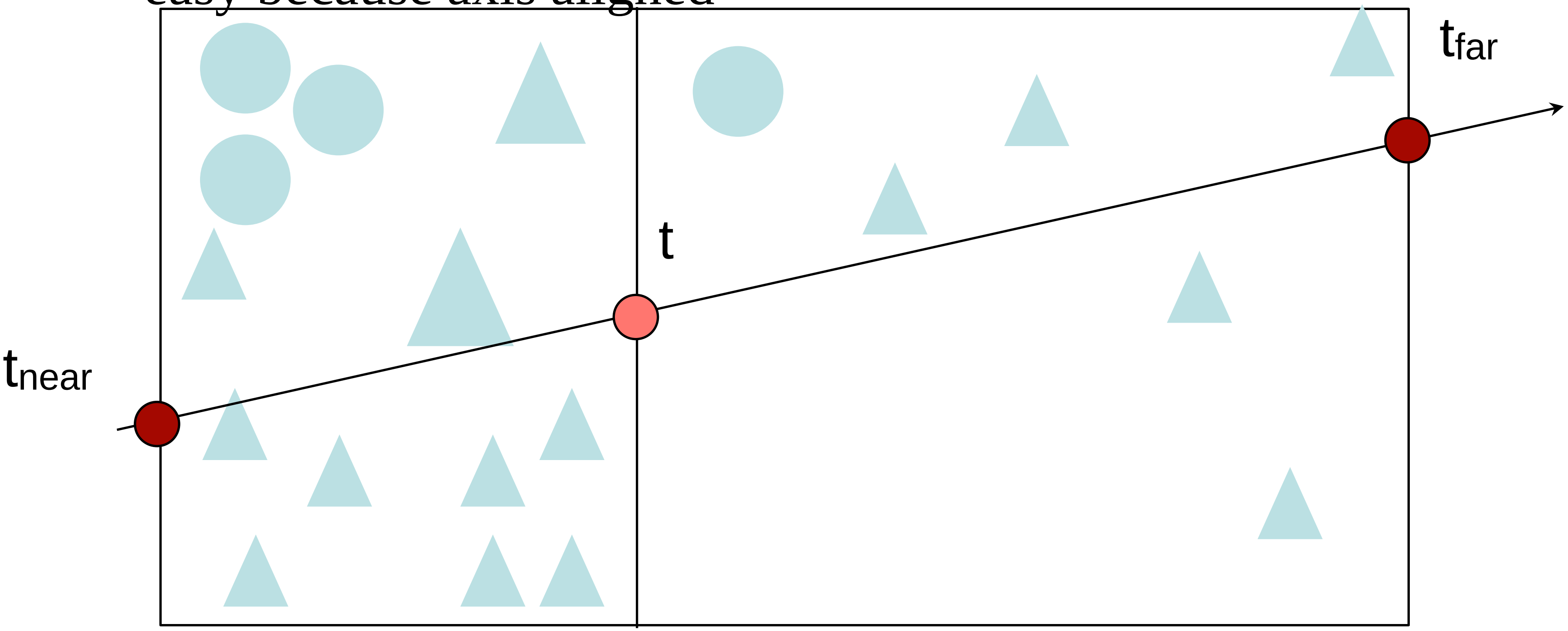
Kd-tree Traversal, Naïve Version

- Could use bounding box test for each child
- But redundant calculation: bbox similar to that of parent node, plus axis aligned, one single split



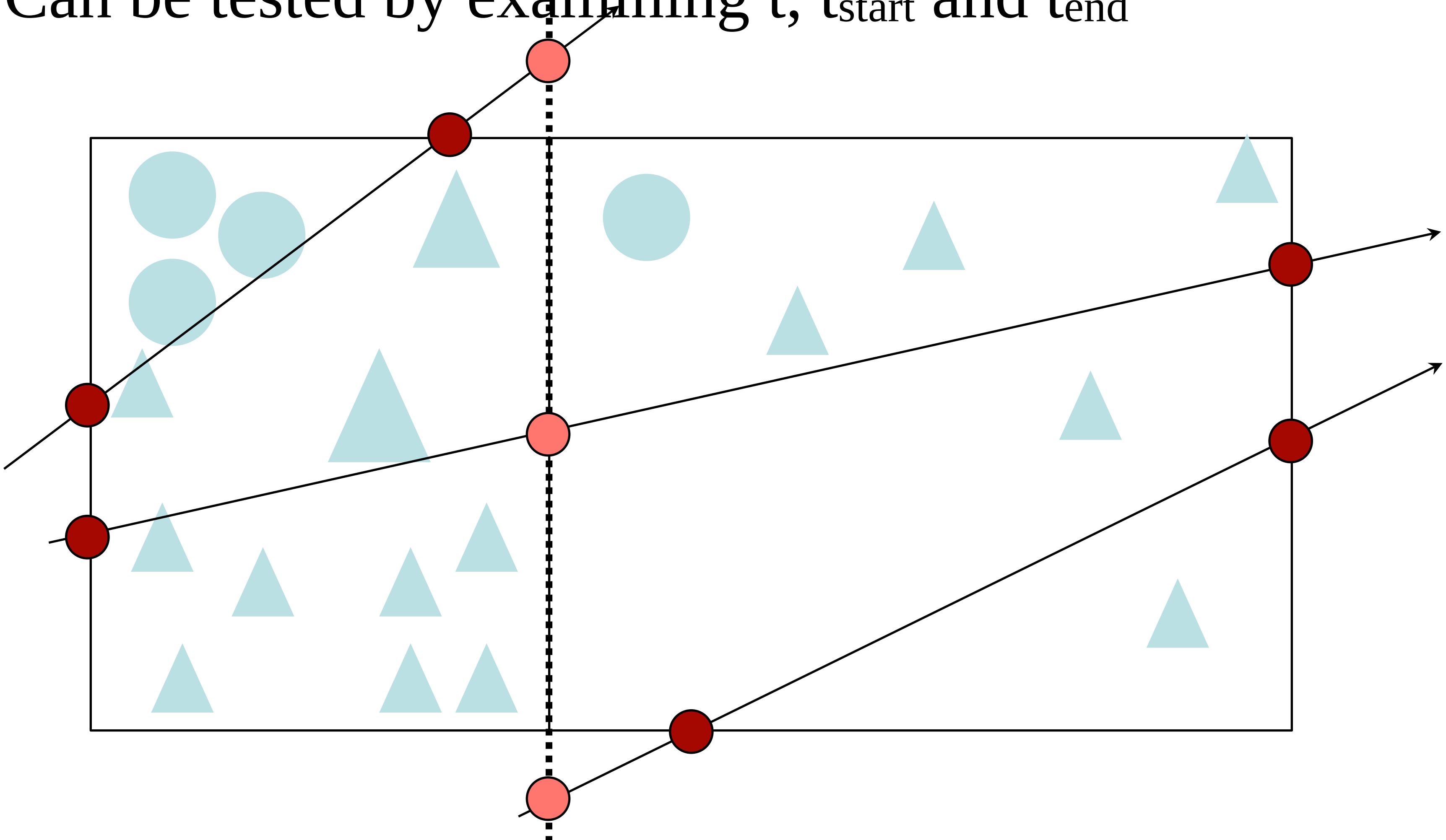
Kd-tree Traversal, Smarter Version

- Get main bbox intersection from parent
  - t_{near}, t_{far}
- Intersect with splitting plane
  - easy because axis aligned



Kd-tree Traversal - Three Cases

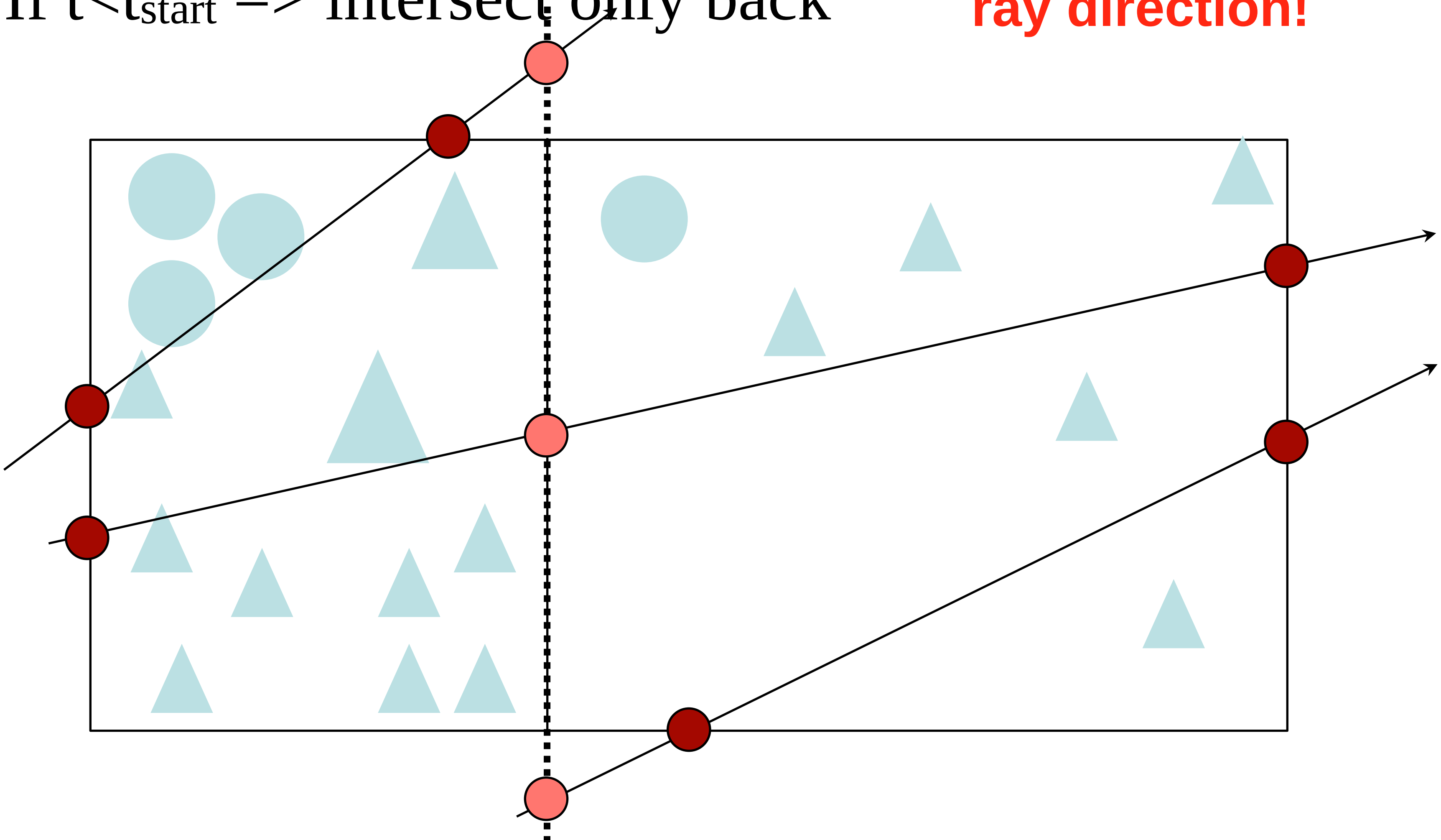
- Intersects only back, only front, or both
- Can be tested by examining t, t_{start} and t_{end}



Kd-tree traversal - three cases

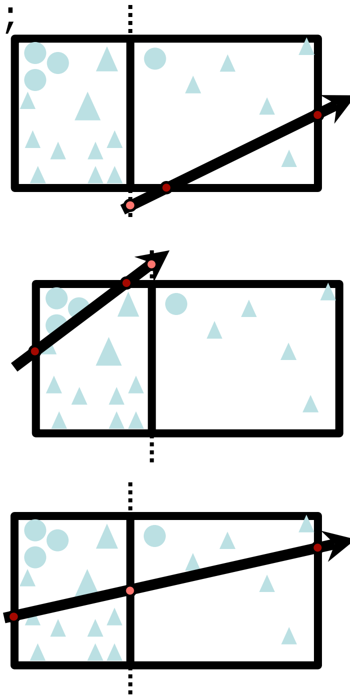
- If $t>t_{end}$ => intersect only front
- If $t<t_{start}$ => intersect only back

Note: "Back" and "Front" depend on ray direction!



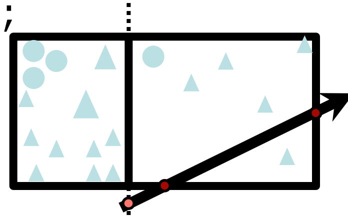
Kd-tree Traversal Pseudocode

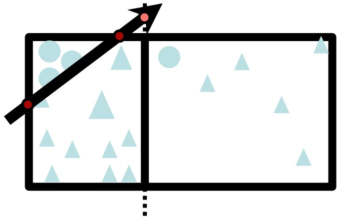
```
travers(orig, dir, t_start, t_end):
    #adapted from Ingo Wald’s thesis
    #assumes that dir[self.dimSplit] >0
    if self.isLeaf:
            return intersect(self.listOfTriangles, orig, dir, t_start, t_end)
    t = (self.splitDist - orig[self.dimSplit]) / dir[self.dimSplit];
    if t <= t_start:
        # case one, t <= t_start <= t_end -> cull front side
        return self.backSideNode.traverse(orig, dir,t_start,t_end)
    elif t >= t_end:
        # case two, t_start <= t_end <= t -> cull back side
        return self.frontSideNode.traverse(orig, dir,t_start,t_end)
    else:
        # case three: traverse both sides in turn
        t_hit = self.frontSideNode.traverse(orig, dir, t_start, t)
        if t_hit <= t: return t_hit; # early ray termination
        return self.backSideNode.traverse(orig, dir, t, t_end)
```

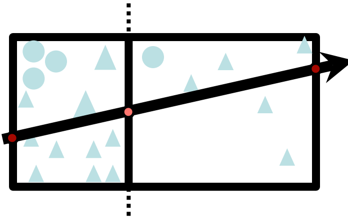


Important!

```
travers(orig, dir, t_start, t_end):
    #adapted from Ingo Wald’s thesis
    #assumes that dir[self.dimSplit] >0
    if self.isLeaf:
            return intersect(self.listOfTriangles, orig, dir, t_start, t_end)
    t = (self.splitDist - orig[self.dimSplit]) / dir[self.dimSplit];
    if t <= t_start:
        # case one, t <= t_start <= t_end -> cull front side
        return self.backSideNode.traverse(orig, dir,t_start,t_end)
    elif t >= t_end:
        # case two, t_start <= t_end <= t -> cull back side
        return self.frontSideNode.traverse(orig, dir,t_start,t_end)
    else:
        # case three: traverse both sides in turn
        t_hit = self.frontSideNode.traverse(orig, dir, t_start, t)
        if t_hit <= t: return t_hit; # early ray termination
        return self.backSideNode.traverse(orig, dir, t, t_end)
```

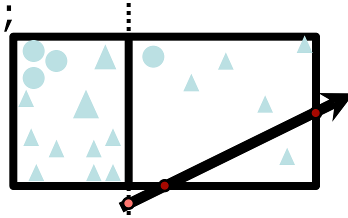


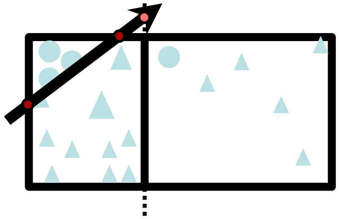


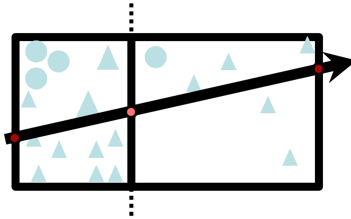


Early termination is powerful!

```
travers(orig, dir, t_start, t_end):
    #adapted from Ingo Wald’s thesis
    #assumes that dir[self.dimSplit] >0
    if self.isLeaf:
            return intersect(self.listOfTriangles, orig, dir, t_start, t_end)
    t = (self.splitDist - orig[self.dimSplit]) / dir[self.dimSplit];
    if t <= t_start:
        # case one, t <= t_start <= t_end -> cull front side
        return self.backSideNode.traverse(orig, dir,t_start,t_end)
    elif t >= t_end:
        # case two, t_start <= t_end <= t -> cull back side
        return self.frontSideNode.traverse(orig, dir,t_start,t_end)
    else:
        # case three: traverse both sides in turn
        t_hit = self.frontSideNode.traverse(orig, dir, t_start, t)
        if t_hit <= t: return t_hit; # early ray termination
        return self.backSideNode.traverse(orig, dir, t, t_end)
```

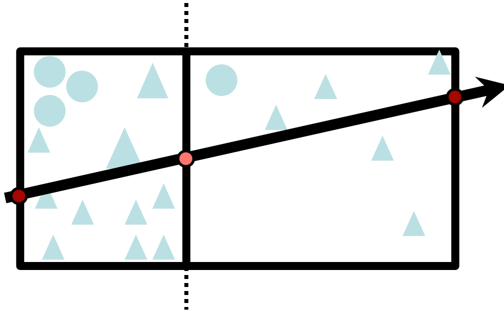






Early termination is powerful

- If there is an intersection in the first node, don’t visit the second one
- Allows ray casting to be reasonably independent of scene depth complexity



Recap: Two main gains

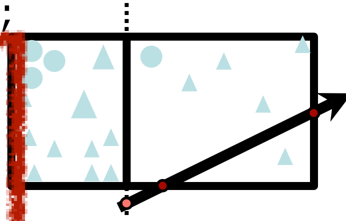
- Only intersect with triangles “near” the line
- Stop at the first intersection

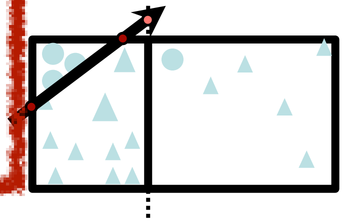
Two main gains

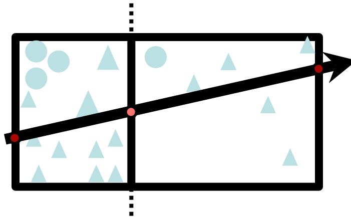
```
travers(orig, dir, t_start, t_end):
    #adapted from Ingo Wald's thesis
    #assumes that dir[self.dimSplit] >0
    if self.isLeaf:
            return intersect(self.listOfTriangles, orig, dir, t_start, t_end)
    t = (self.splitDist - orig[self.dimSplit]) / dir[self.dimSplit];
    if t <= t_start:
        # case one, t <= t_start <= t_end -> cull front side
        return self.backSideNode.traverse(orig, dir,t_start,t_end)
    elif t >= t_end:
        # case two, t_start <= t_end <= t -> cull back side
        return self.frontSideNode.traverse(orig, dir,t_start,t_end)
    else:
        # case three: traverse both sides in turn
        t_hit = self.frontSideNode.traverse(orig, dir, t_start, t)
        if t_hit <= t: return t_hit; # early ray termination
        return self.backSideNode.traverse(orig, dir, t, t_end)
```

Only near line

stop at first intersection







Important Details

- For leaves, do NOT report intersection if t is not in [t_{near}, t_{far}].
  - Important for primitives that overlap multiple nodes!

- Need to take direction of ray into account
  - Reverse back and front if the direction has negative coordinate along the split dimension
- Degeneracies when ray direction is parallel to one axis

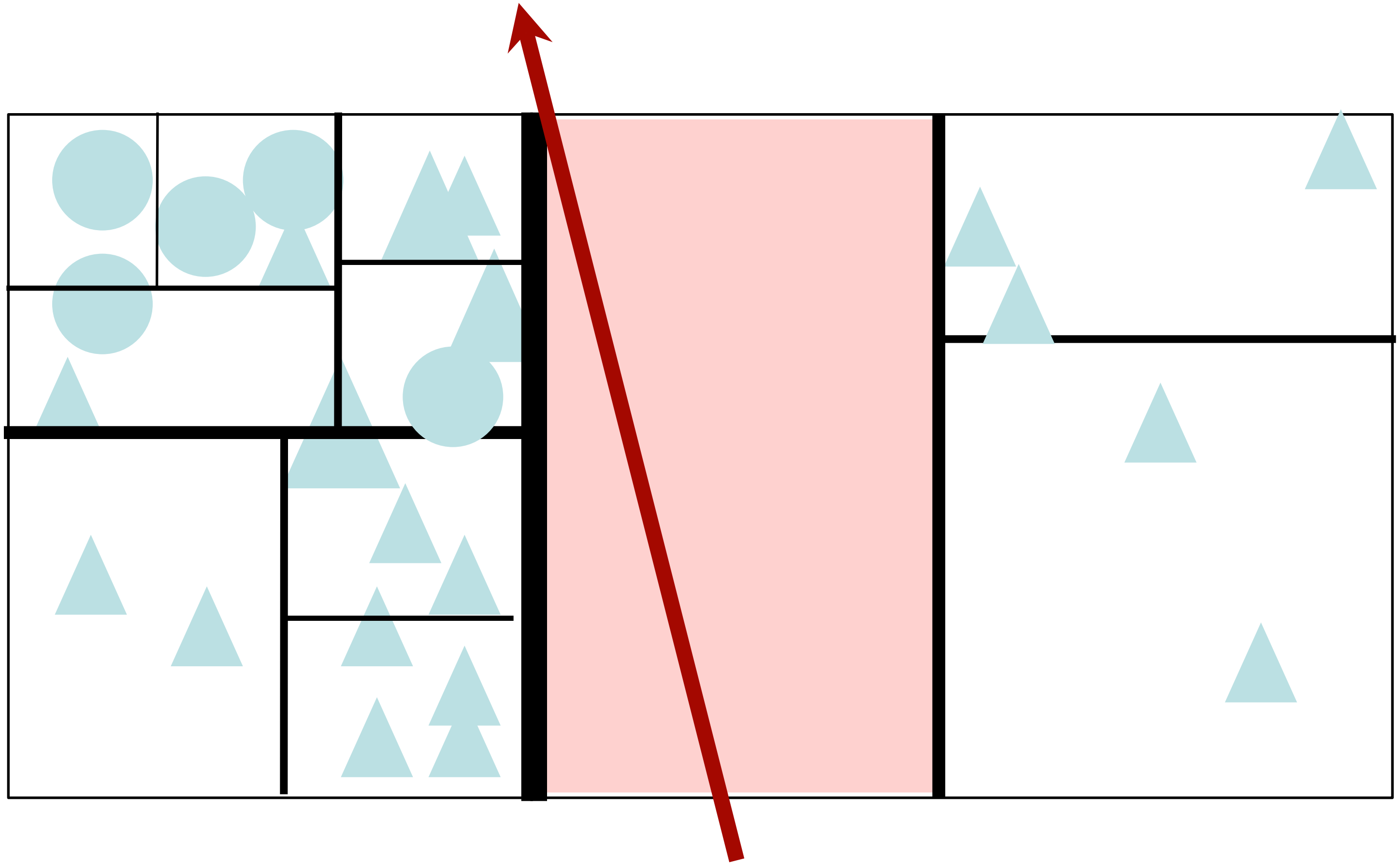
Important Details Questions?

- For leaves, do NOT report intersection if t is not in [t_{near}, t_{far}].
  - Important for primitives that overlap multiple nodes!

- Need to take direction of ray into account
  - Reverse back and front if the direction has negative coordinate along the split dimension
- Degeneracies when ray direction is parallel to one axis

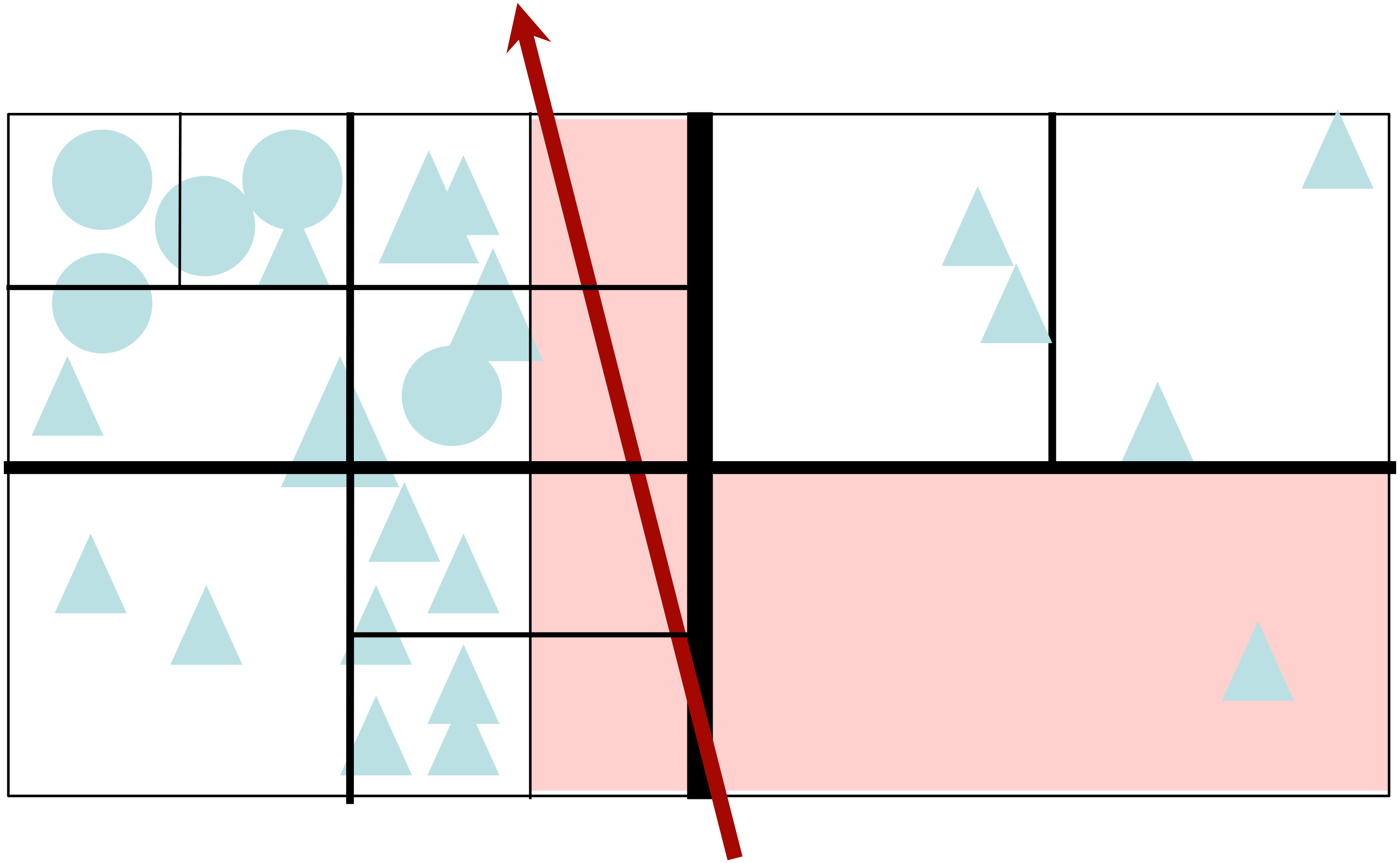
Where to split for construction?

- Example for baseline
- Note how this ray traverses easily: one leaf only



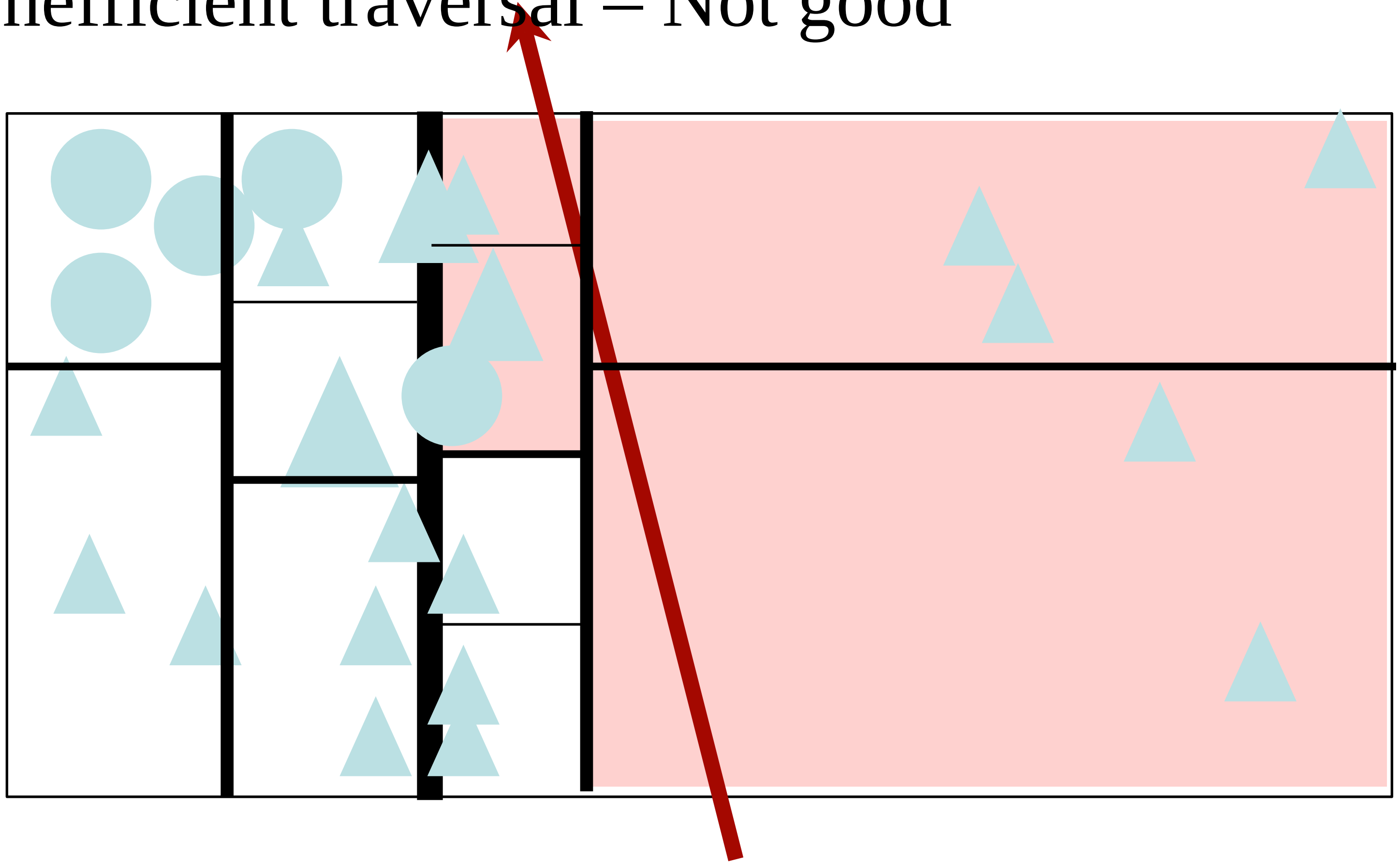
Split in the Middle

- Does not conform to empty vs. dense areas
- Inefficient traversal – Not so good!



Split in the Median

- Tries to balance tree, but does not conform to empty vs. dense areas
- Inefficient traversal – Not good

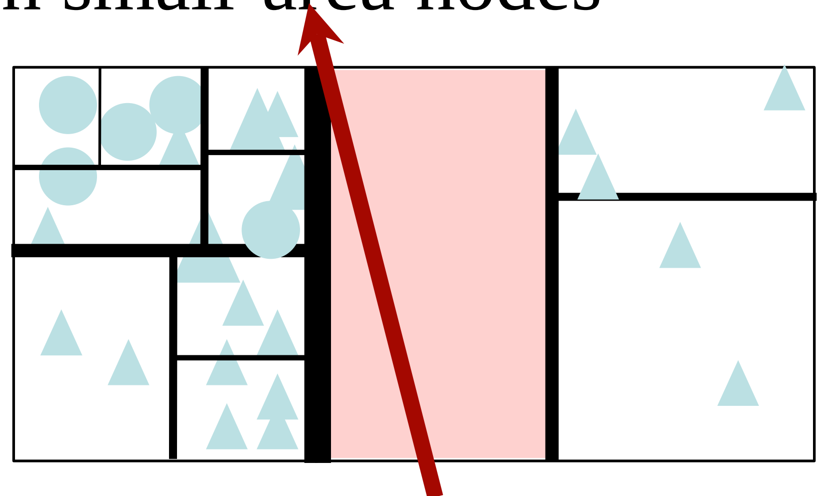


Optimizing Splitting Planes

- Most people use the Surface Area Heuristic (SAH)
  - MacDonald and Booth 1990, “Heuristic for ray tracing using space subdivision”, Visual Computer
- Idea: simple probabilistic prediction of traversal cost based on split distance
- Then try different possible splits and keep the one with lowest cost
- Further reading on efficient Kd-tree construction
  - Hunt, Mark & Stoll, IRT 2006
  - Zhou et al., SIGGRAPH Asia 2008

Surface Area Heuristic

- Probability that we need to intersect a child
  - Area of the bbox of that child
    (exact for uniformly distributed rays)
- Cost of the traversal of that child
  - number of primitives (simplistic heuristic)
- This heuristic likes to put big densities of primitives in small-area nodes



Is it Important to Optimize Splits?

- Given the same traversal code, the quality of Kd-tree construction can have a big impact on performance, e.g. a factor of 2 compared to naive middle split
  - But then, you should consider carefully if you need that extra performance
  - Could you optimize something else for bigger gain?

Hard-core efficiency considerations

- See e.g. Ingo Wald’s PhD thesis
  - http://www.mpi-inf.mpg.de/~wald/PhD/
- Calculation
  - Optimized barycentric ray-triangle intersection
- Memory
  - Make kd-tree node as small as possible (dirty bit packing, make it 8 bytes)
- Parallelism
  - SIMD extensions, trace 4 rays at a time, mask results where they disagree

Pros and Cons of Kd trees

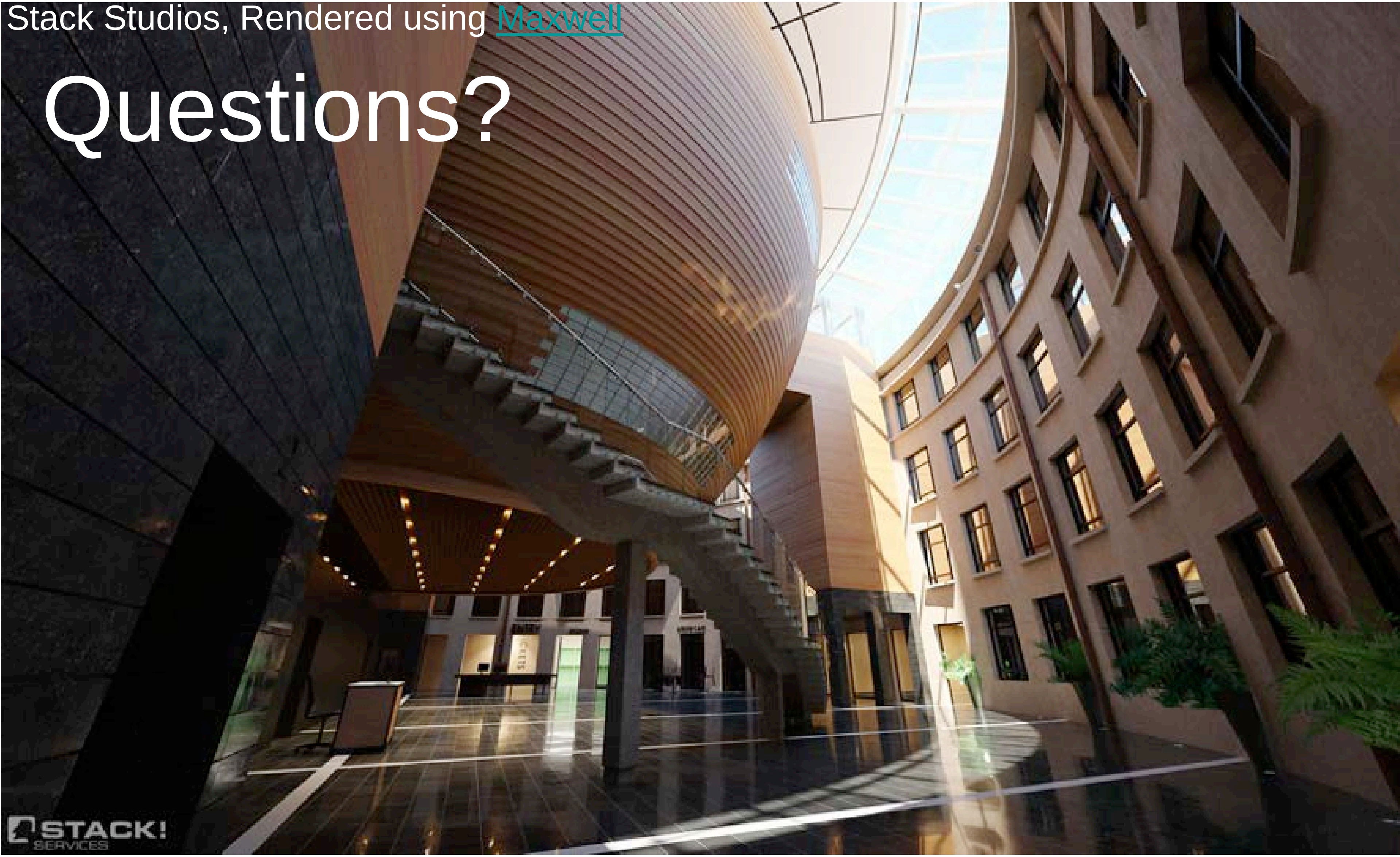
- Pros
  - Simple code
  - Efficient traversal
  - Can conform to data

- Cons
  - costly construction, not great if you work with moving objects

Questions?

- For extensions to moving scenes, see Real-Time KD-Tree Construction on Graphics Hardware, Zhou et al., SIGGRAPH 2008

Questions?



Stack Studios, Rendered using Maxwell