# Acceleration Structures for Ray Tracing

Real-time ray tracing in *Control* by Remedy Entertainment

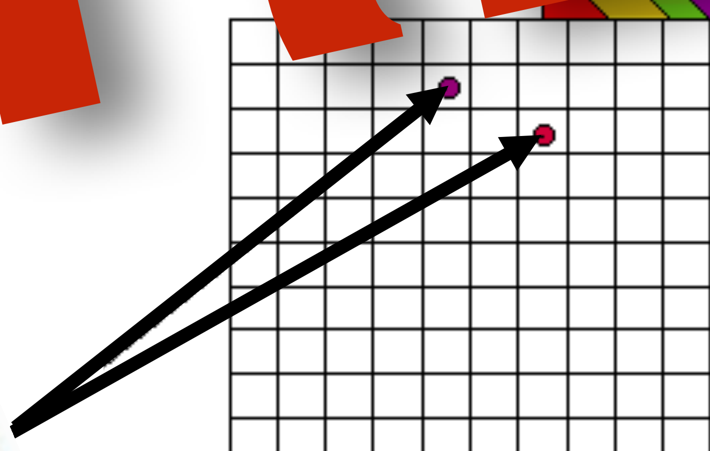
# Ray Tracing vs. Rasterization

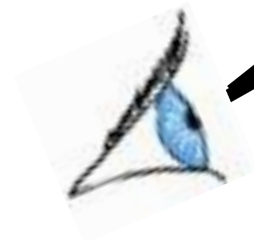
## Ray Tracing

```
For each pixel (ray)
  For each object
    Does ray hit object?
```

**Keep closest hit**

RECAP





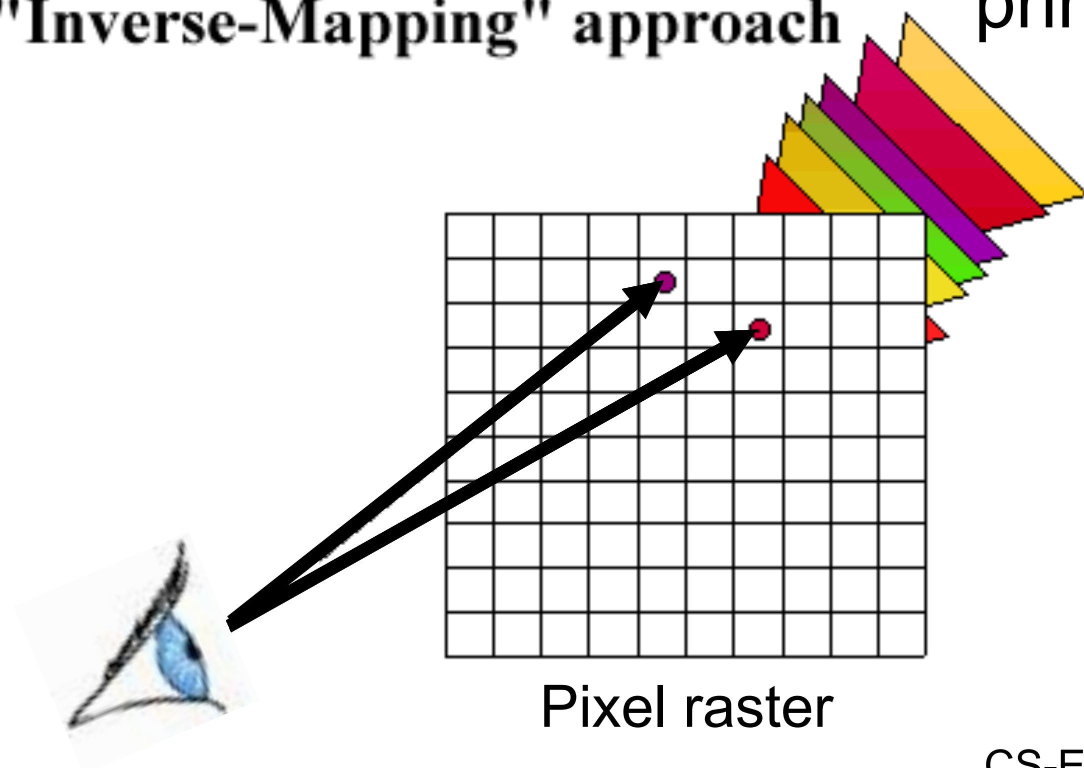
Pixel raster

# Ray Tracing vs. Rasterization

## Ray Tracing

```
For each pixel (ray)
  For each object
    Does ray hit object?
```

**Keep closest hit**



# Ray Tracing vs. Rasterization

Ray Tracing

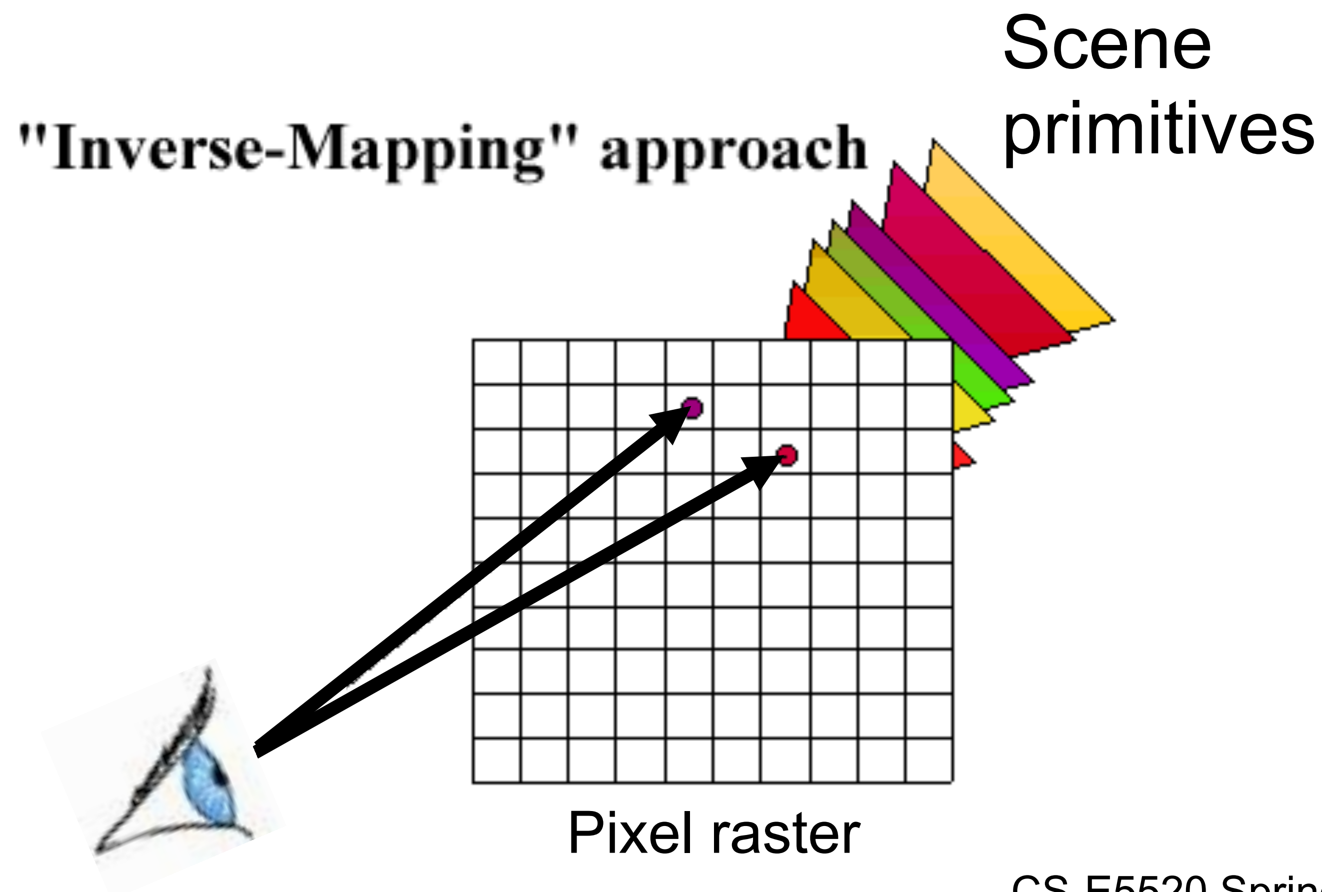
```
For each pixel (ray)
  For each object
    Does ray hit object?
```

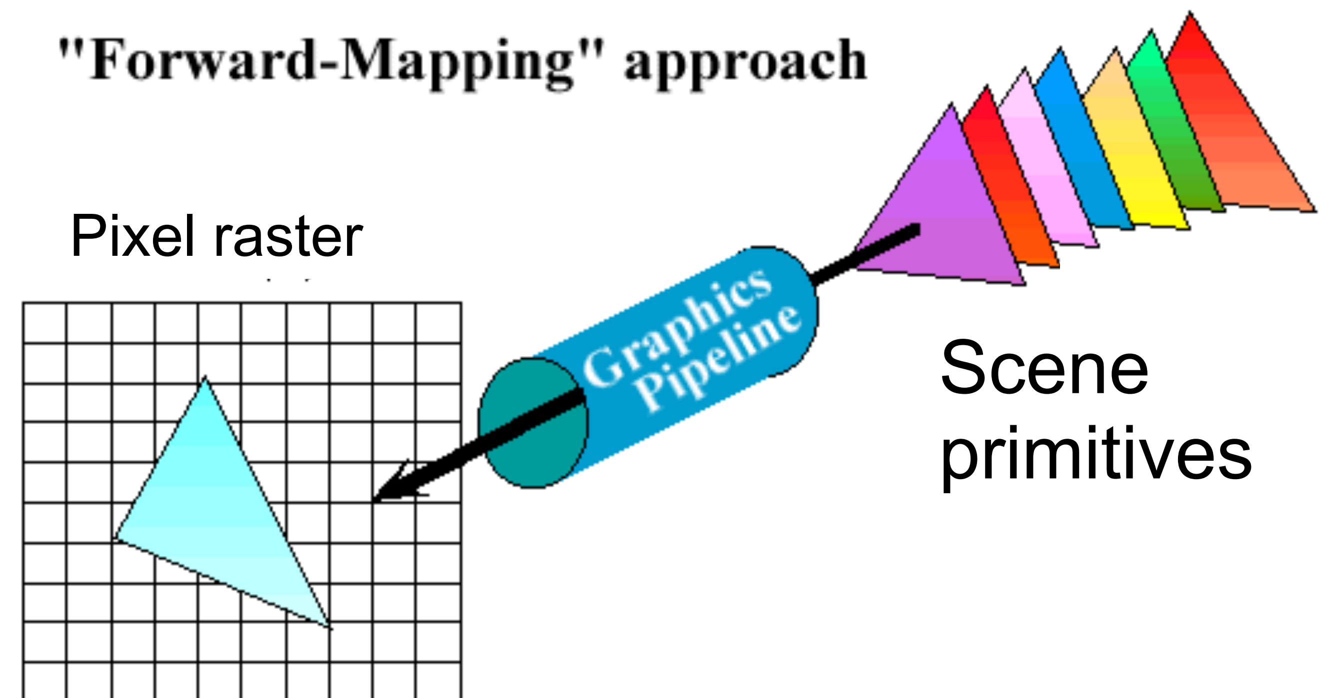
**Keep closest hit**

Rasterization

```
For each triangle
  For each pixel
    Does triangle cover pixel?
```

**Keep closest hit**



# Ray Tracing vs. Rasterization

## Ray Tracing

```
For each pixel (ray)
  For each object
    Does ray hit object?
```

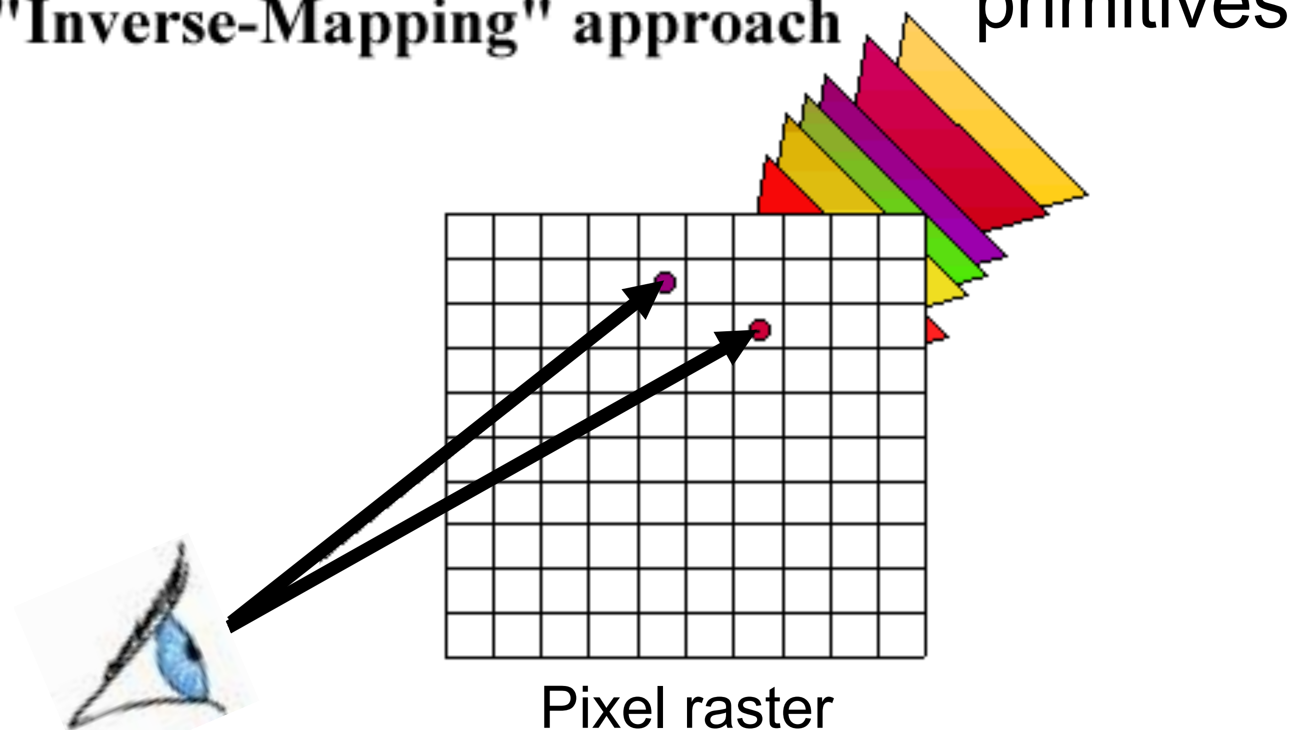
**Keep closest hit**

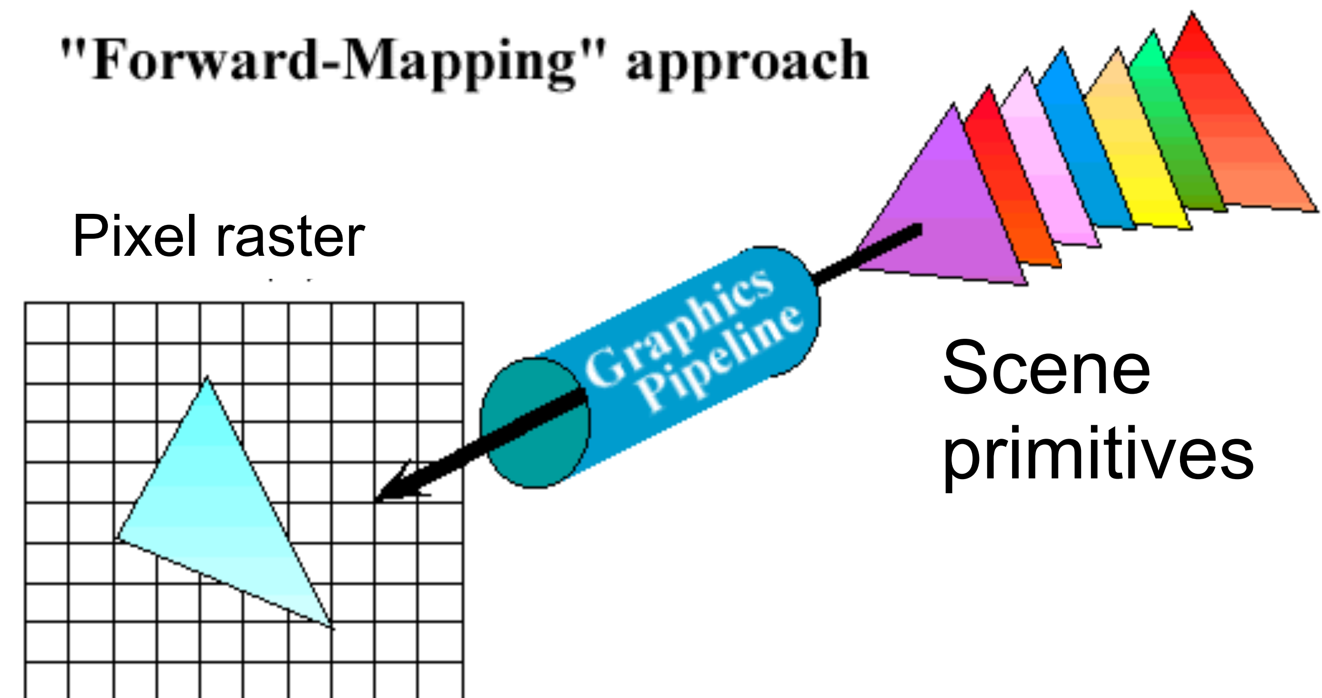
## Rasterization

```
For each triangle
  For each pixel
    Does triangle cover pixel?
```

**Keep closest hit**

**It's just a different order of the loops!**



# Ray Tracing

- Advantages
  - Generality: can render anything that can be intersected with a ray
  - Easily allows recursion (shadows, reflections, etc.)
- Disadvantages
  - ***Much harder*** to implement in hardware (less computation coherence, scene must fit memory, worse memory access…)
  - Was, for long, too slow for interactive applications…*but today, interactive ray tracing is reality!*
    - GPU software tracers from ~2008 onwards
    - NVIDIA GeForce RTX hardware introduced in 2018
    - Other vendors (AMD, Apple, etc.) since

# Ray Tracing

- Advantages
  - Generality: can render anything that can be intersected with a ray
  - Easily allows recursion (shadows, reflections, etc.)

**Our focus in this class**

- Disadvantages
  - *Much harder* to implement in hardware (less computation coherence, scene must fit memory, worse memory access…)
  - Was, for long, too slow for interactive applications…*but today, interactive ray tracing is reality!*
    - GPU software tracers from ~2008 onwards
    - NVIDIA GeForce RTX hardware introduced in 2018
    - Other vendors (AMD, Apple, etc.) since

**“Marbles at Night” (YouTube)**

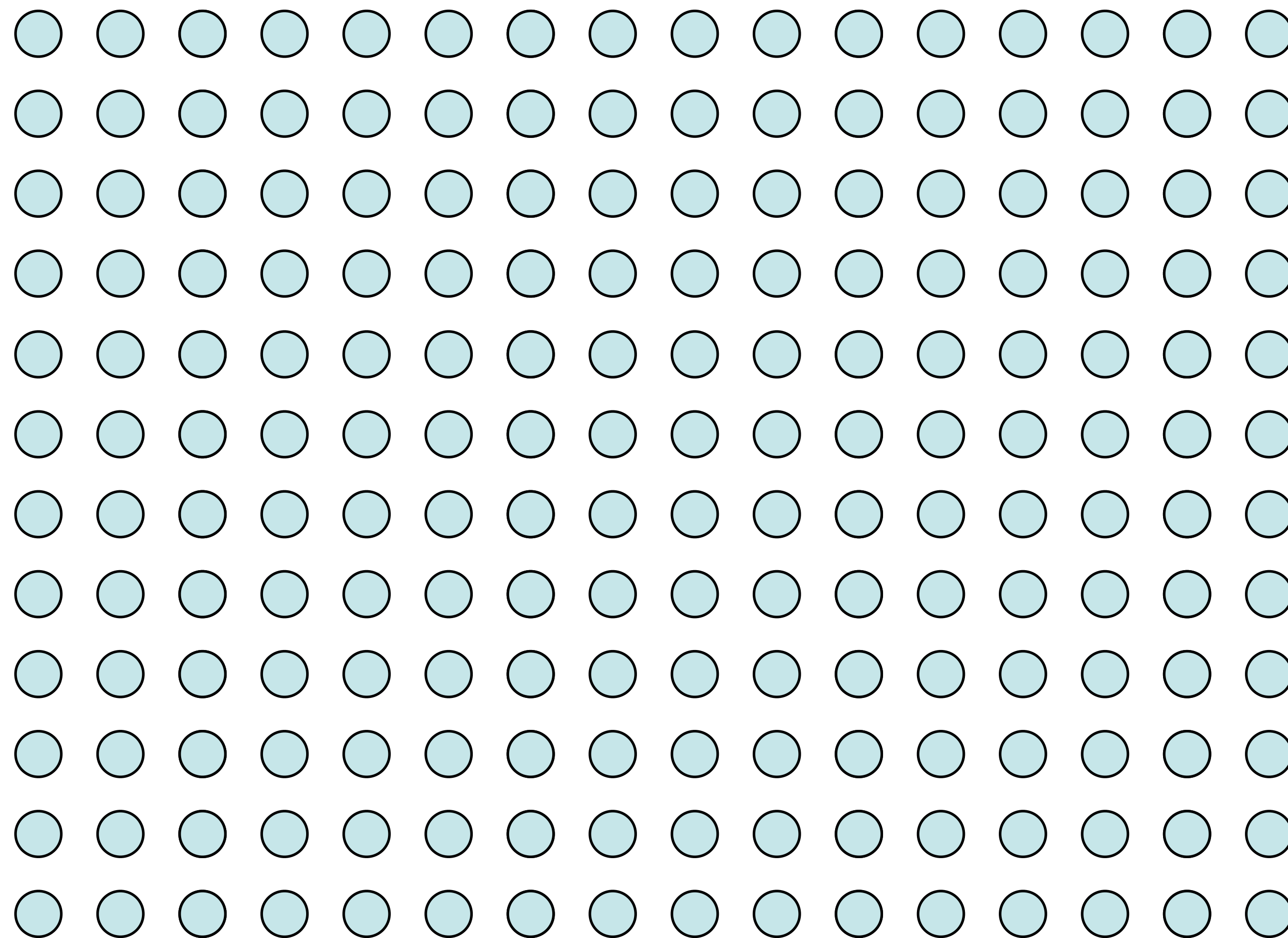
Real-time ray tracing on an NVIDIA RTX GPU

# On to Today’s Topic

```
For each pixel (ray)
  For each triangle
    Does ray hit triangle?
```

○ = Intersection test

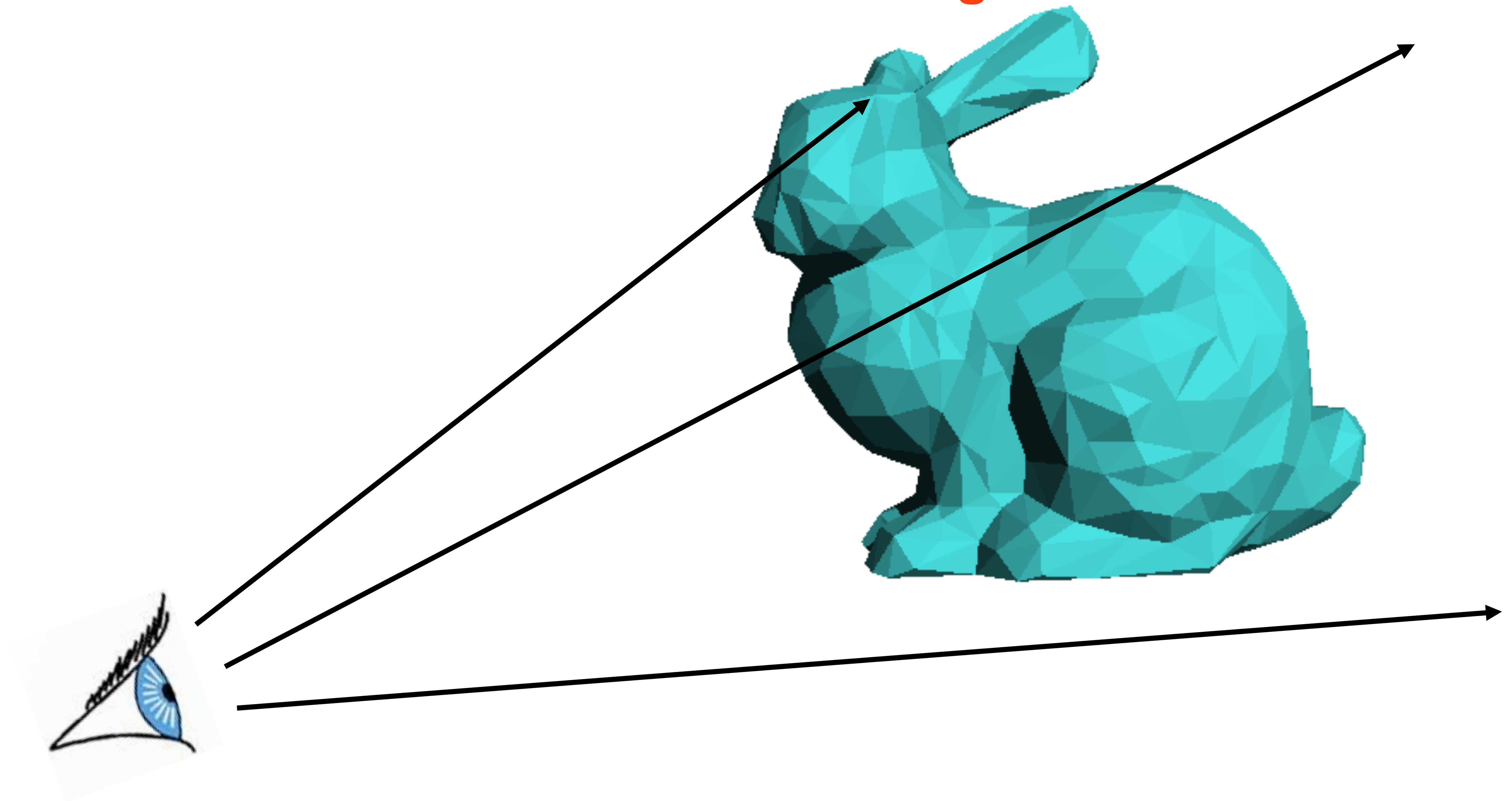
Rays (pixels)



Primitives (triangles, …)

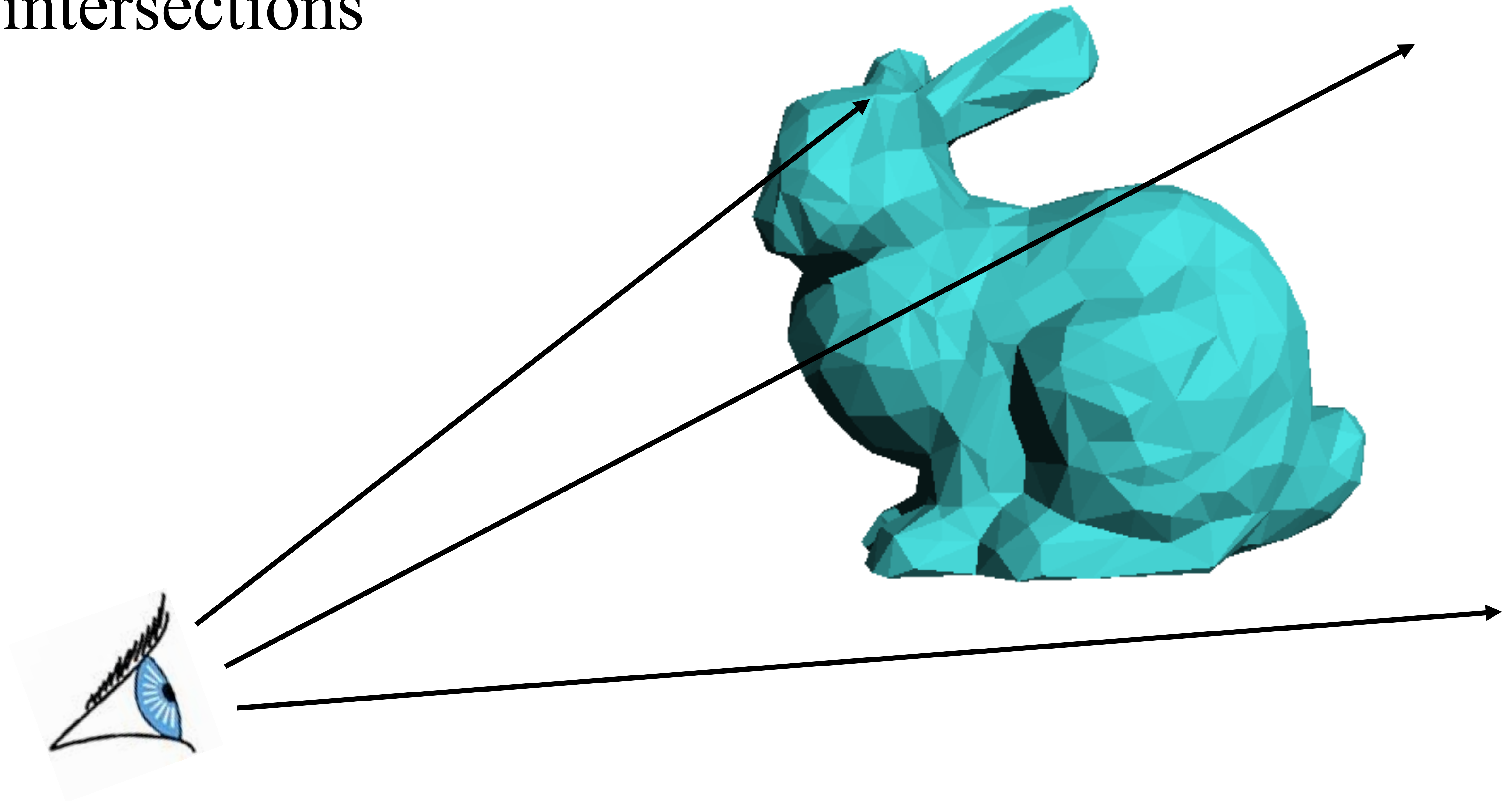
# Accelerating Ray Casting

**What if this thing had 1B triangles and your ray tracer just walked through all of them?**



# Accelerating Ray Casting

- Goal: Reduce the number of ray/primitive intersections



WETA Digital / New Line Cinema

**No, you're NOT going to get the image any time soon if you test every triangle!**

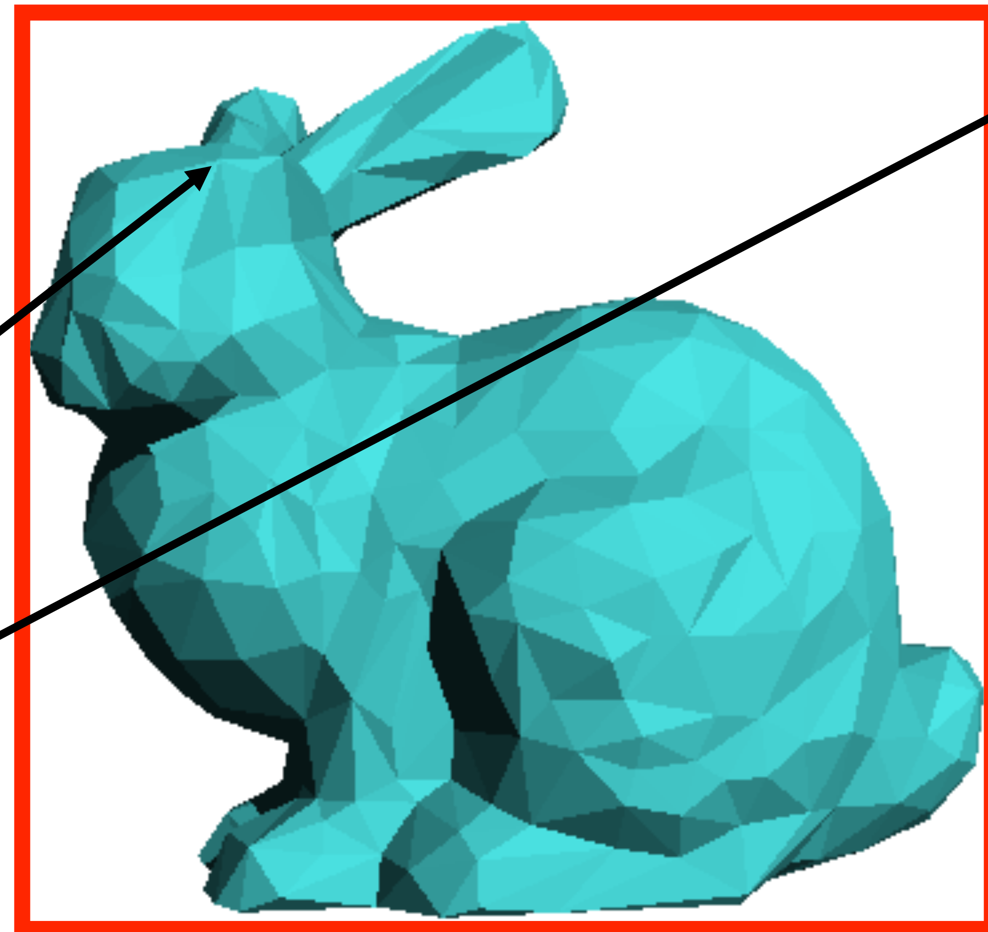
WETA Digital / New Line Cinema

**No, you're NOT going to get the image any time soon if you test every triangle!**

# What to do about it?

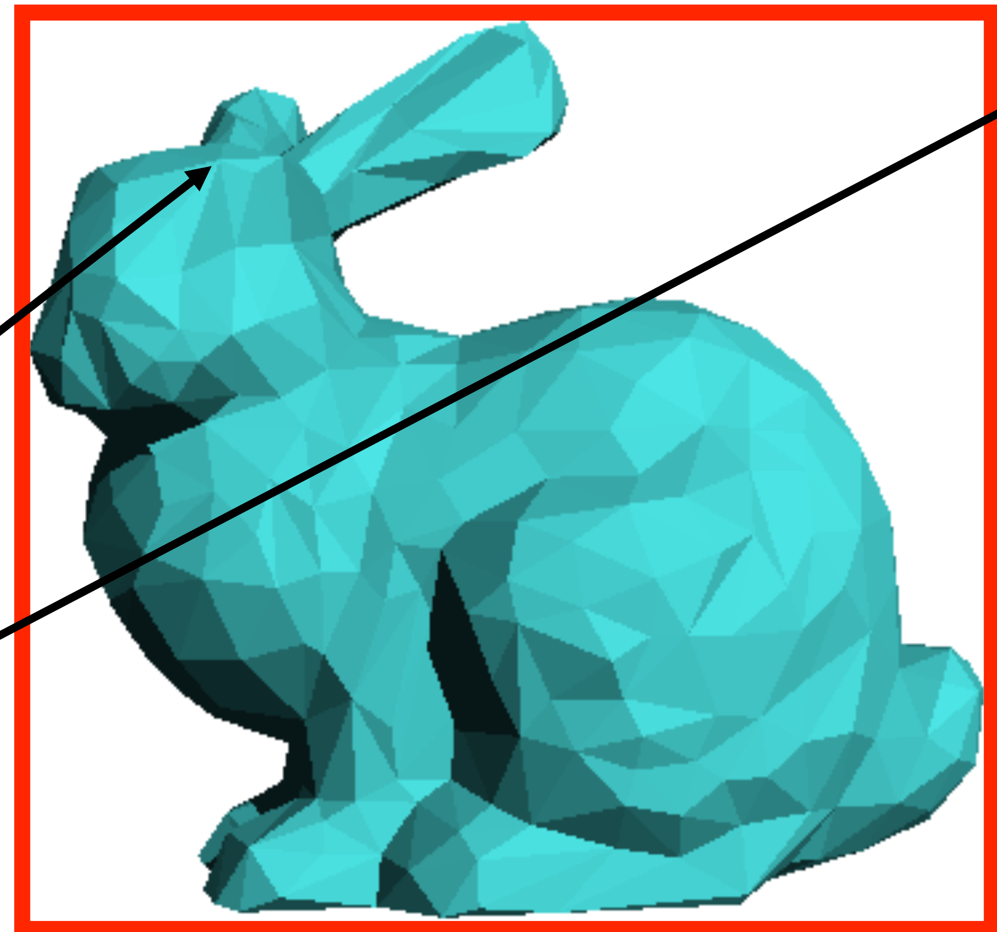
# Conservative Bounding Volume

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



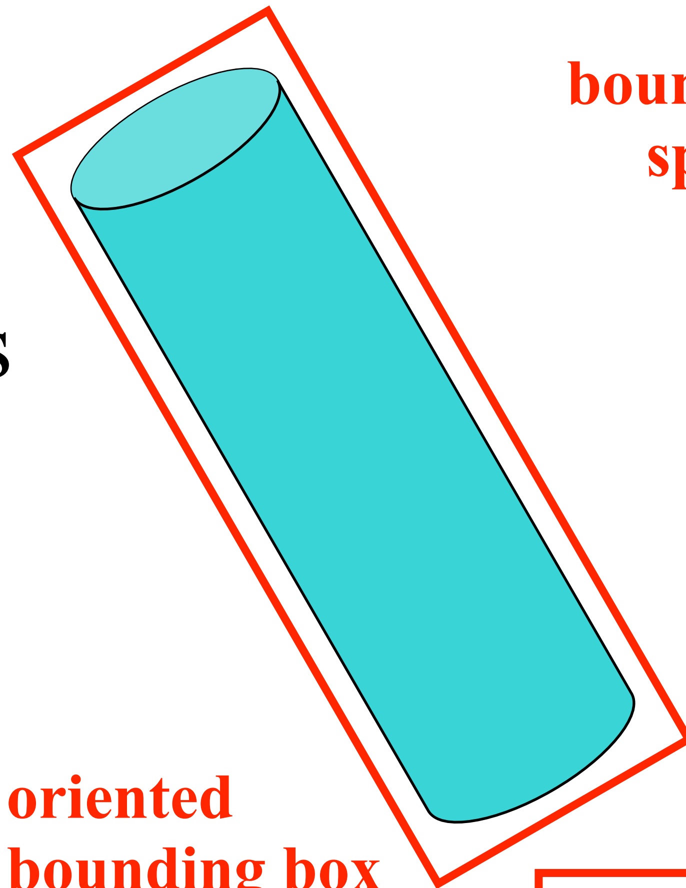
# Conservative Bounding Volume

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

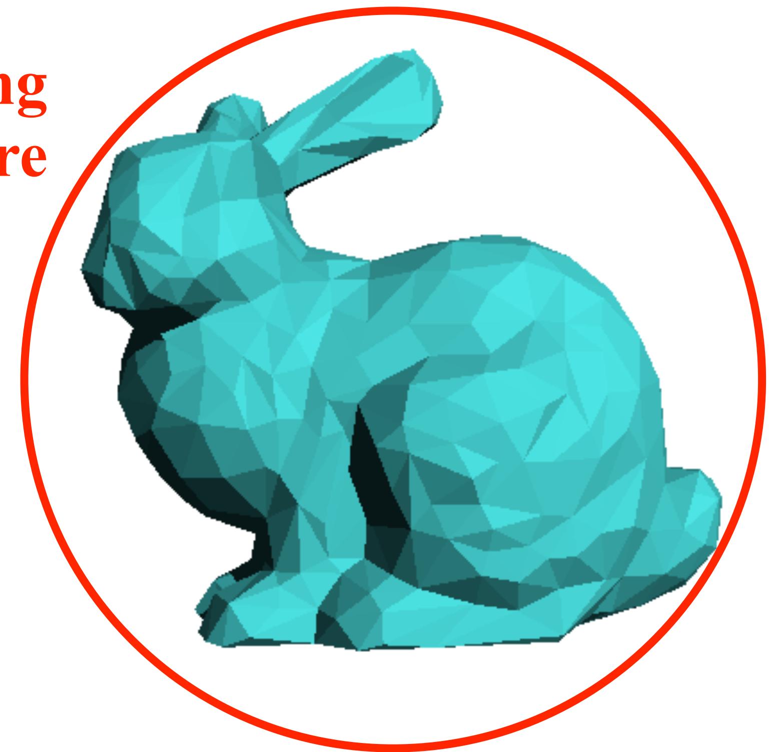


# Conservative Bounding Volume

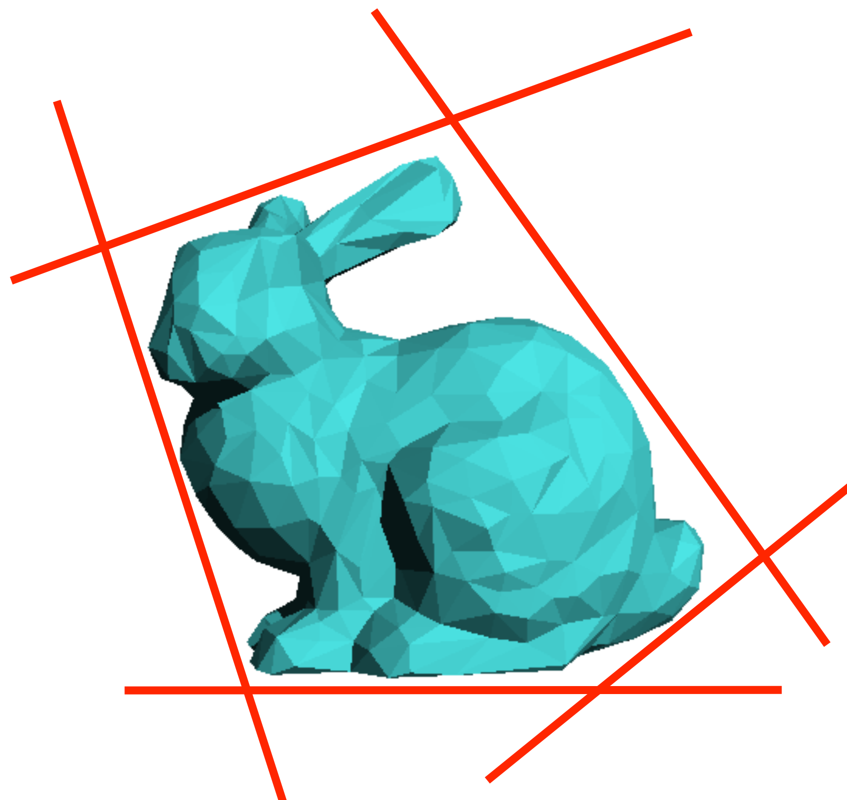
- Desiderata
  - Tight →
    avoid false positives
  - Fast to intersect
  - Fast to construct



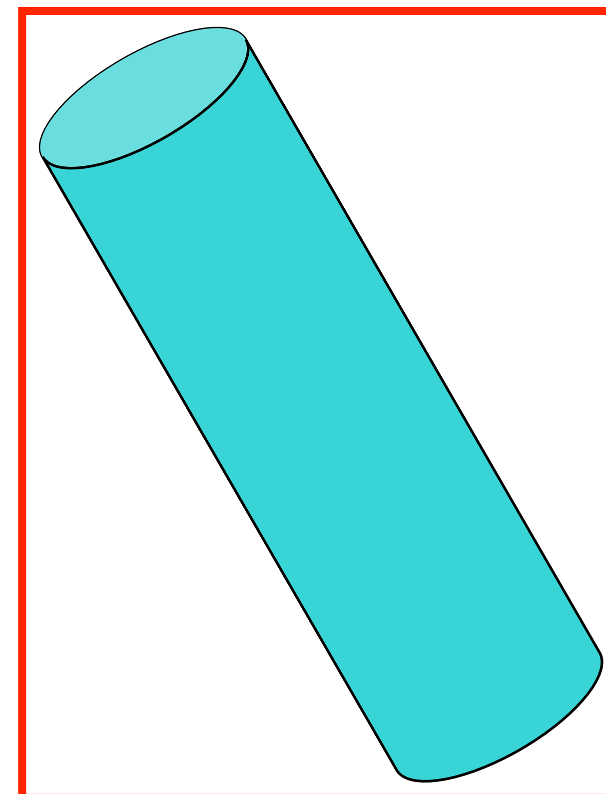
oriented bounding box (OBB)



bounding sphere



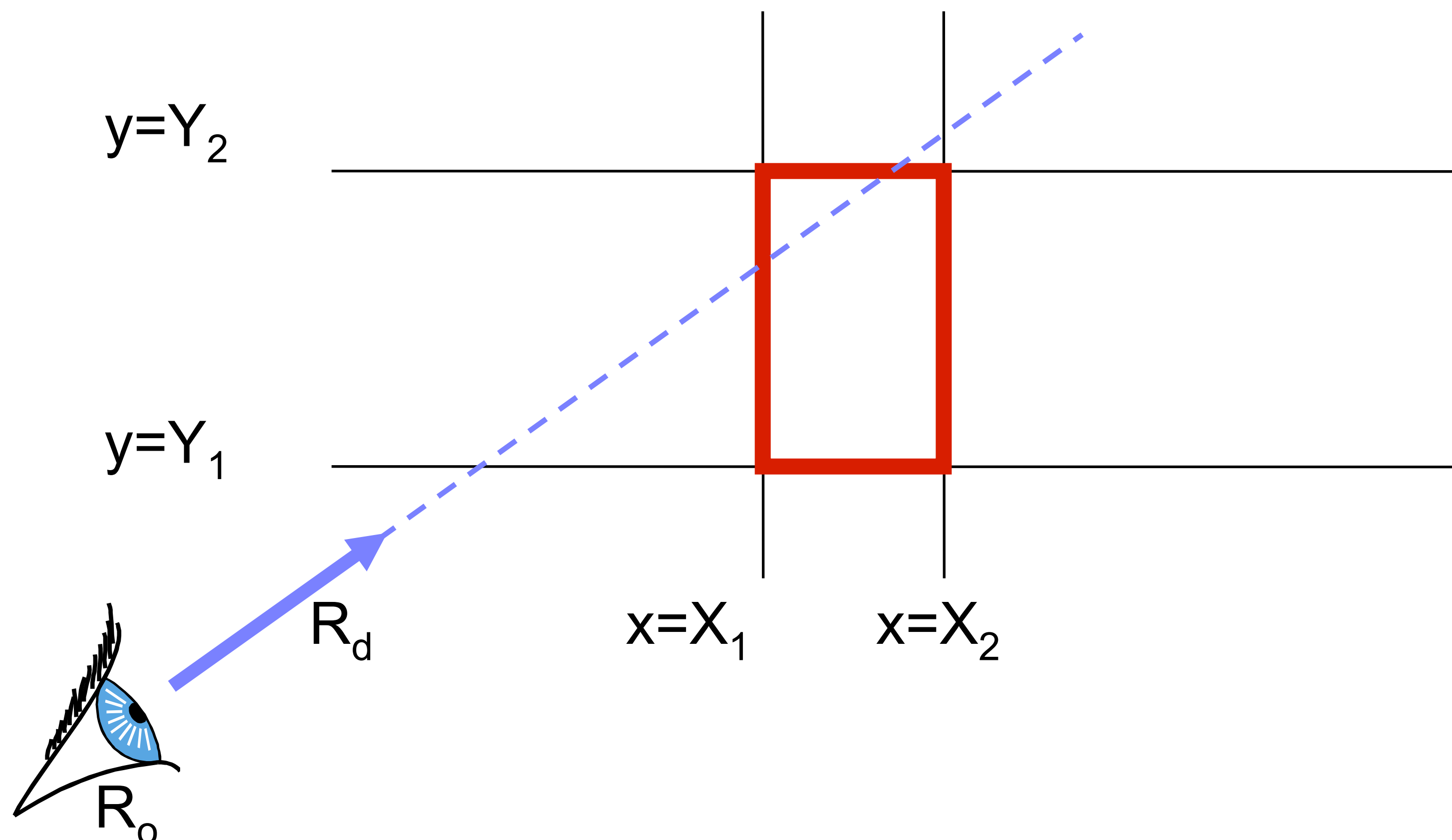
arbitrary convex region (bounding half-spaces)



axis-aligned bounding box (AABB)

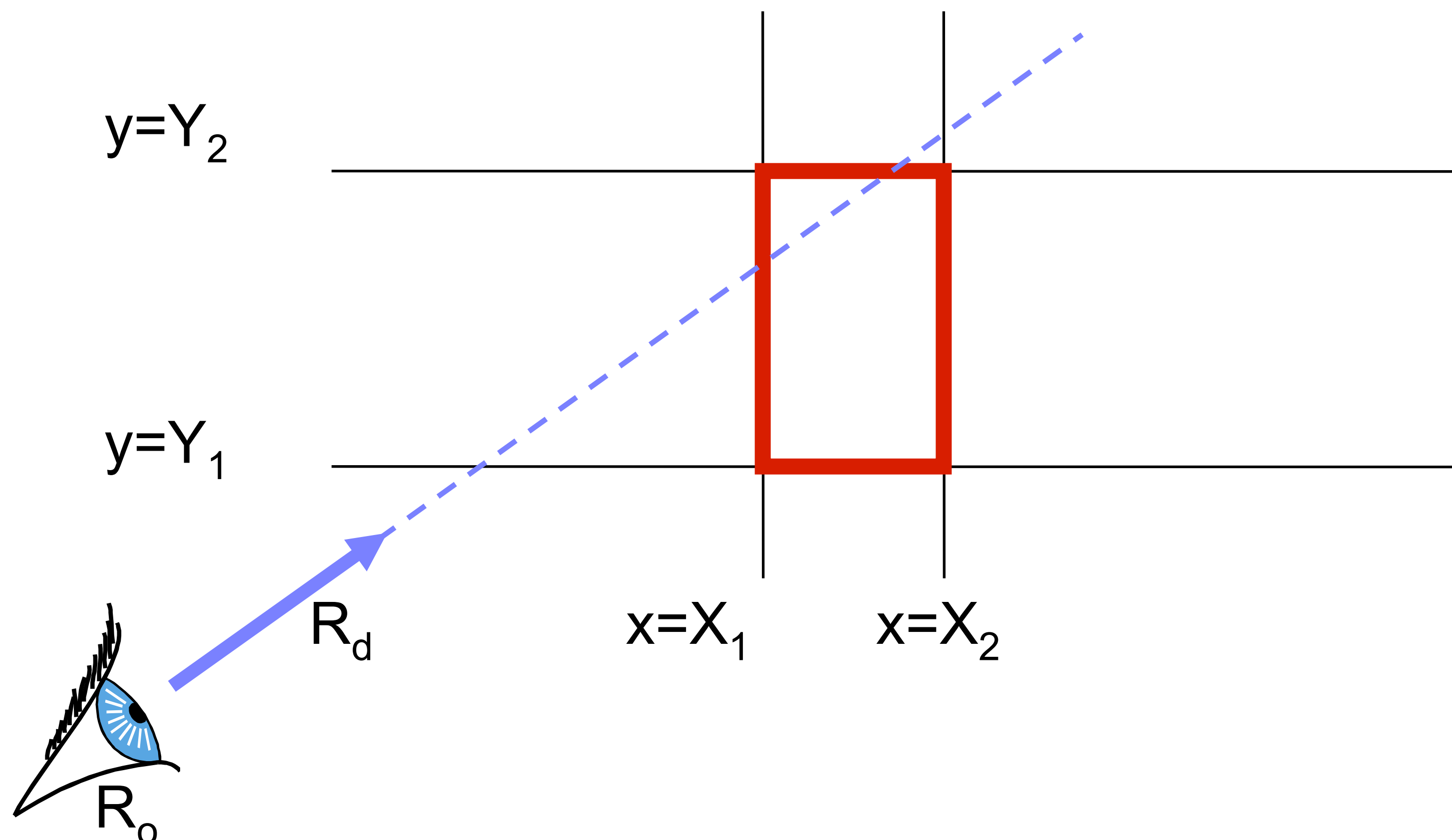
# Ray-Box Intersection

- Axis-aligned box
- Box: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- Ray: $P(t) = R_o + tR_d$
  - Remember? $R_o$ is ray origin, $R_d$ is direction vector



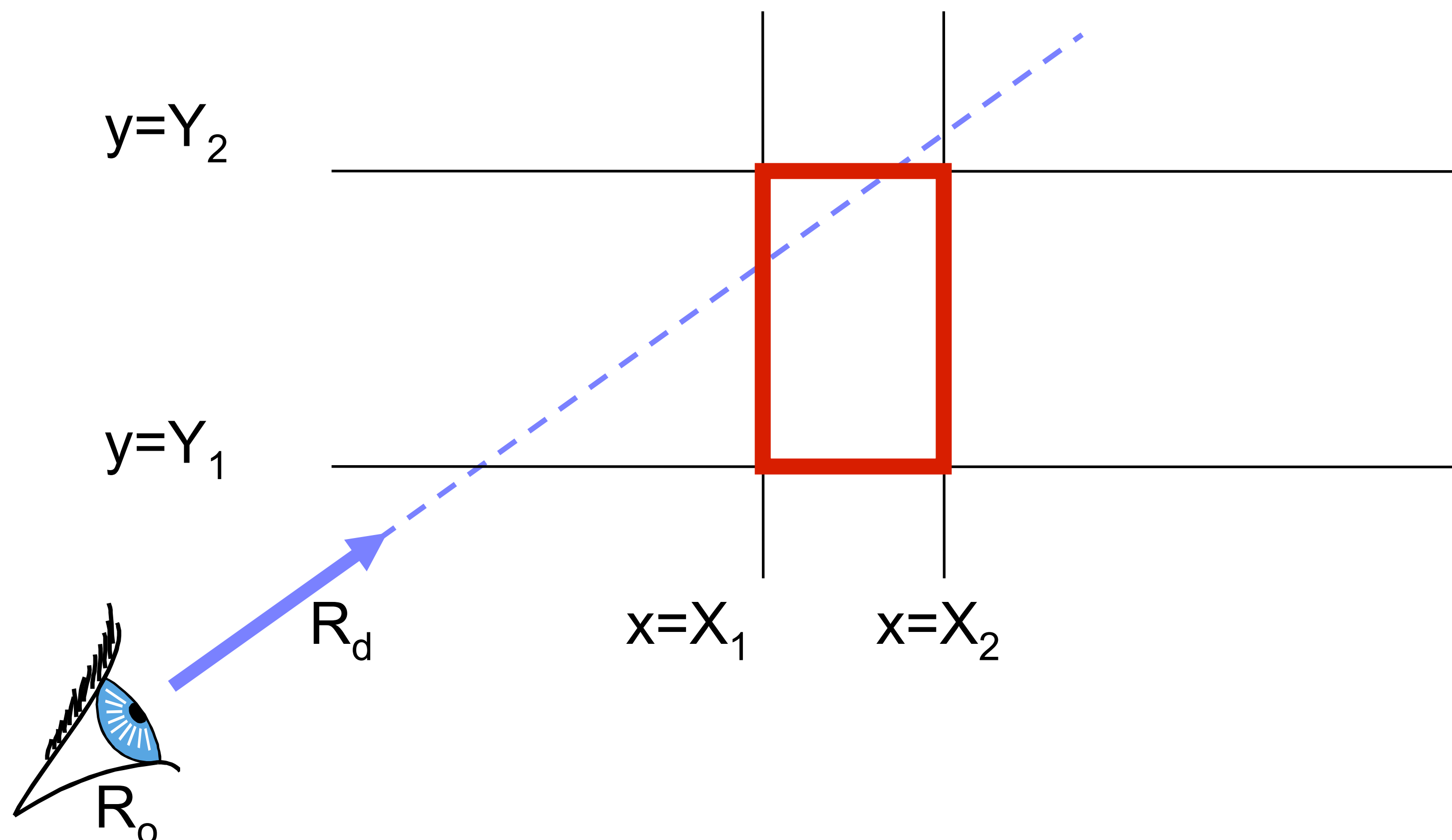
# Naïve Ray-Box Intersection

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



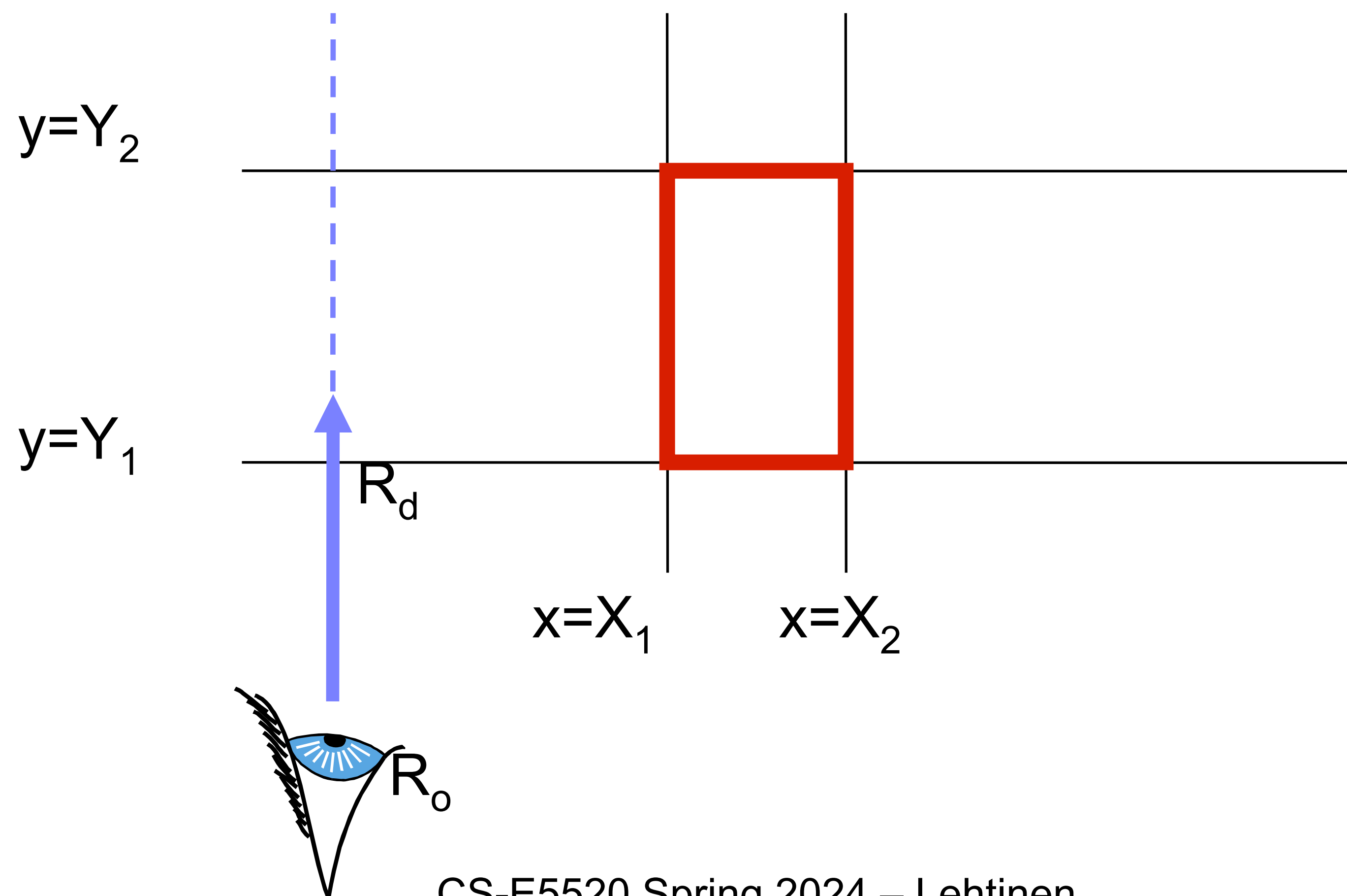
# Reducing Total Computation

- Pairs of planes have the same normal
- Normals have only one non-0 component
- Do computations one dimension at a time



# Test if Parallel

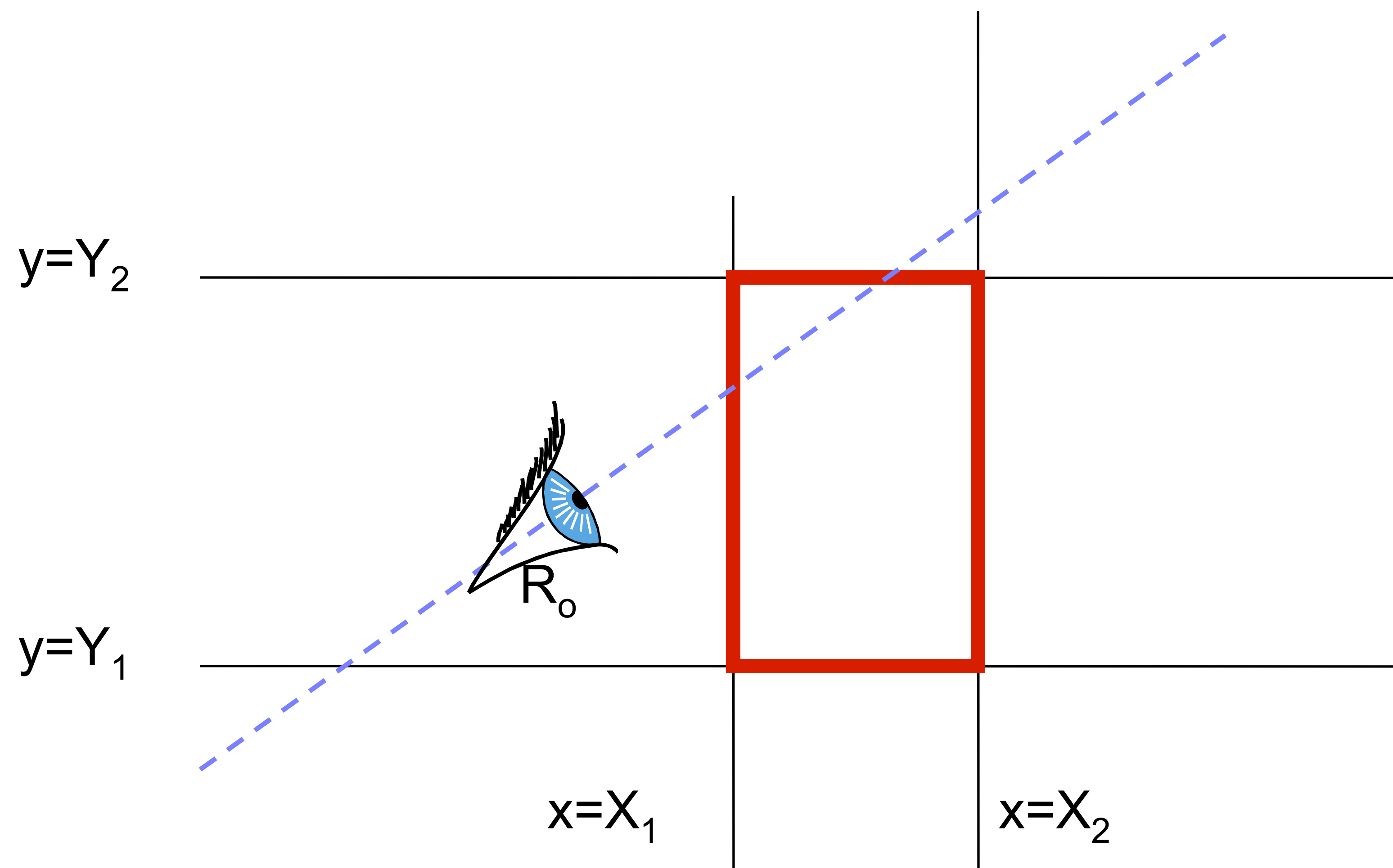
- If $R_{dx} = 0$ (ray is parallel) AND
  $R_{ox} < X_1$ or $R_{ox} > X_2$ $\rightarrow$ **no intersection**




**(The same for Y and Z, of course)**

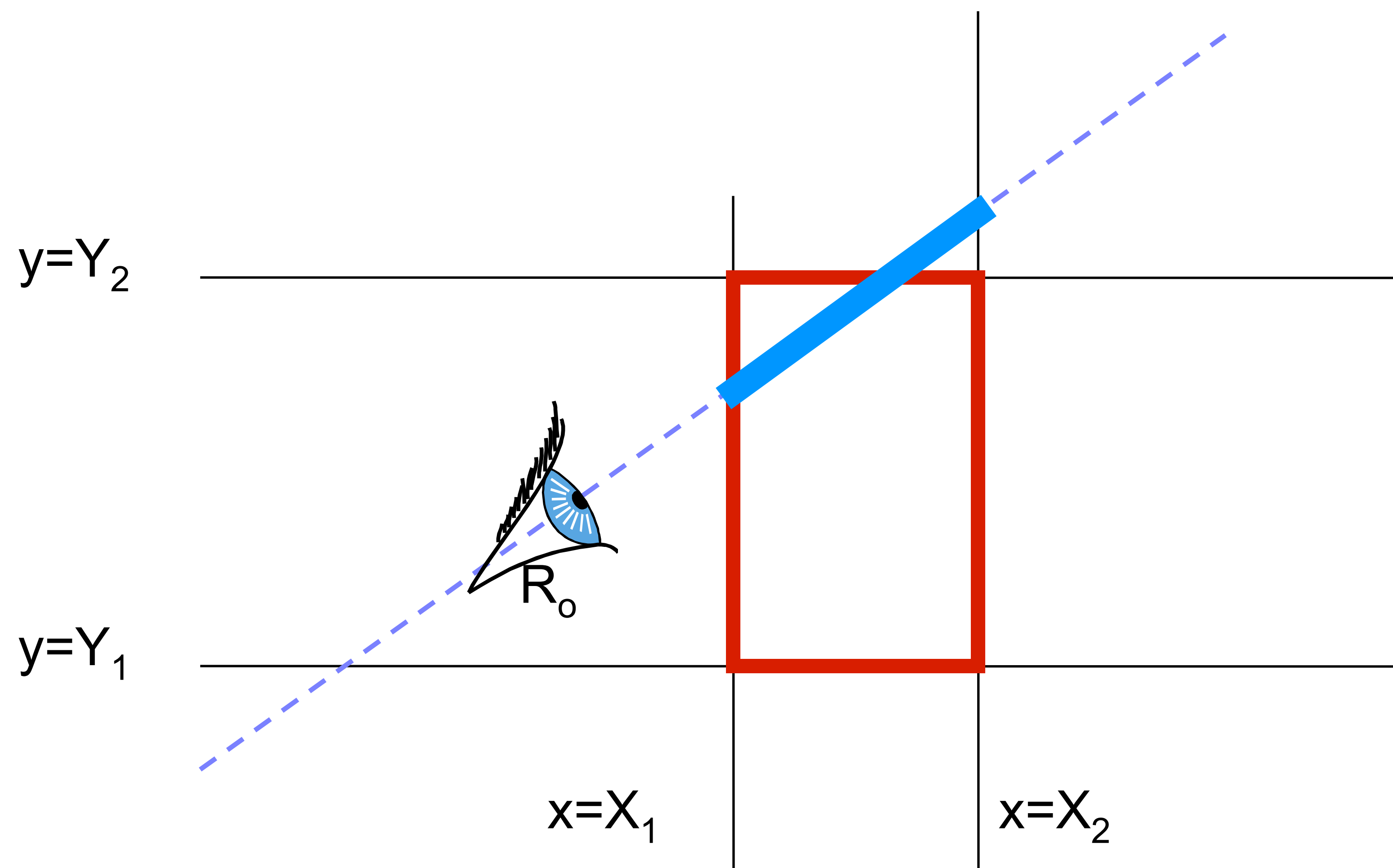
# Find Intersections Per Dimension

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



# Find Intersections Per Dimension

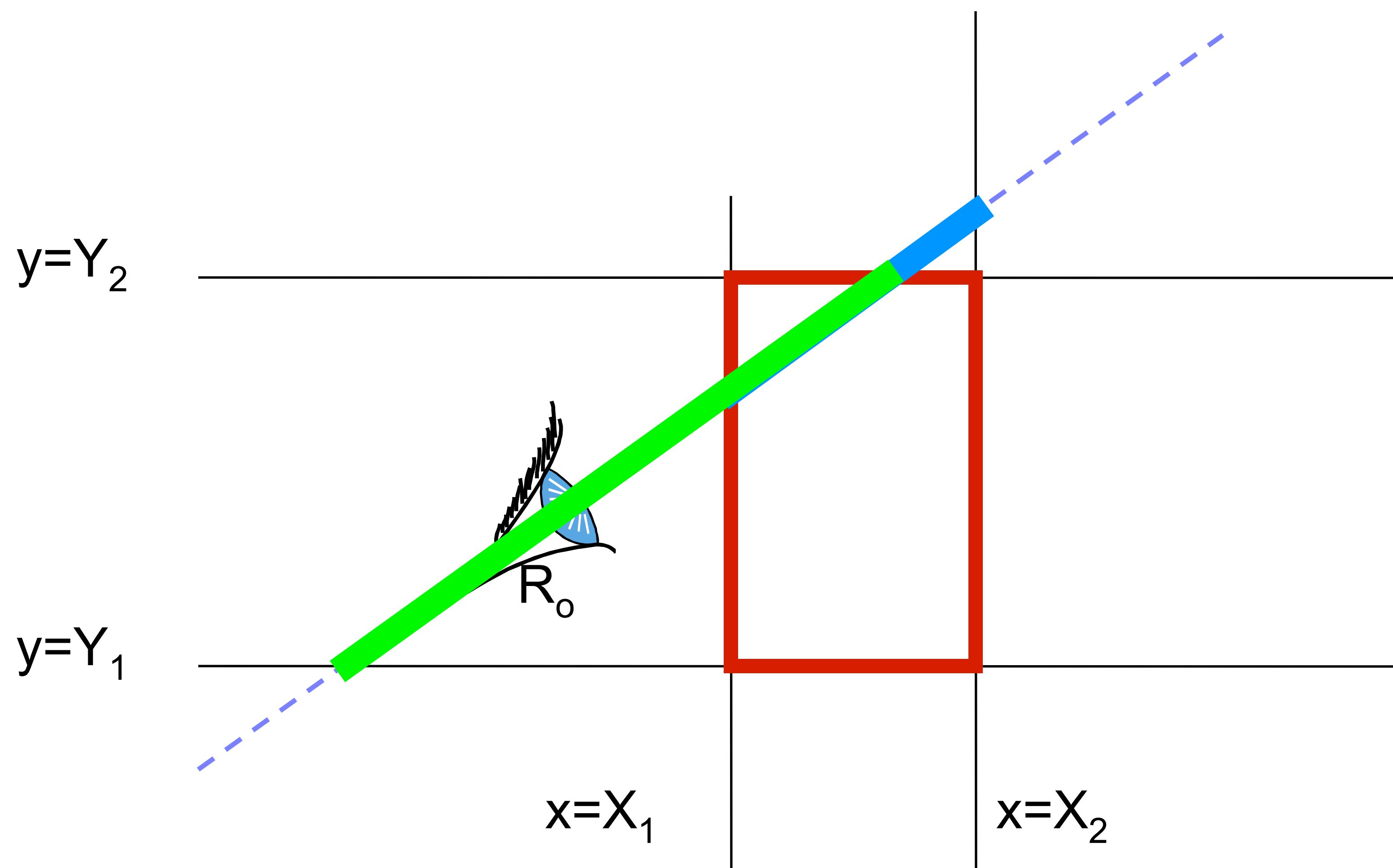
- Basic idea
  - Determine an interval along the ray for each dimension
  - The intersect these 1D intervals (remember CSG!)
  - Done!




**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!

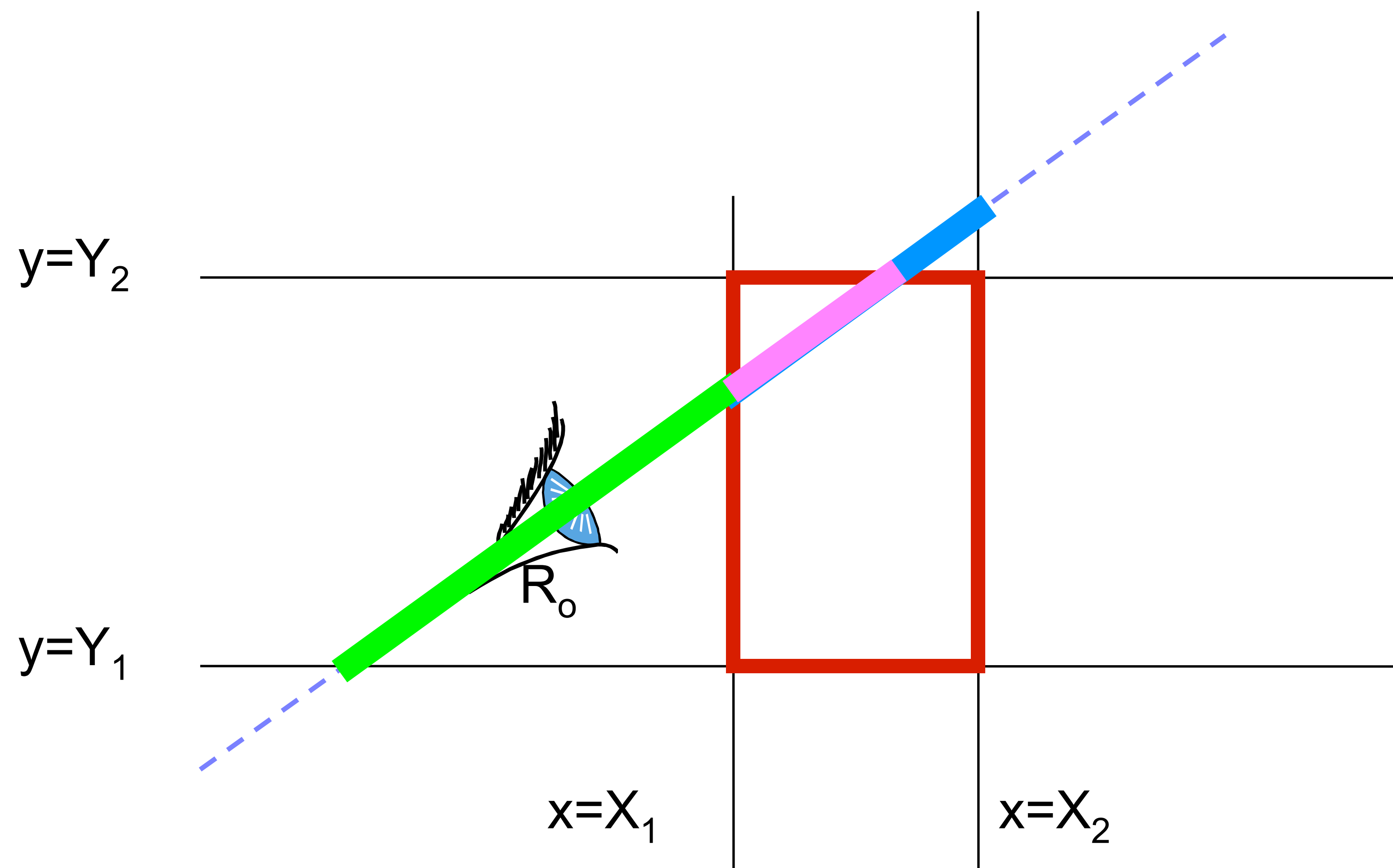



**Interval between $X_1$ and $X_2$**

**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!

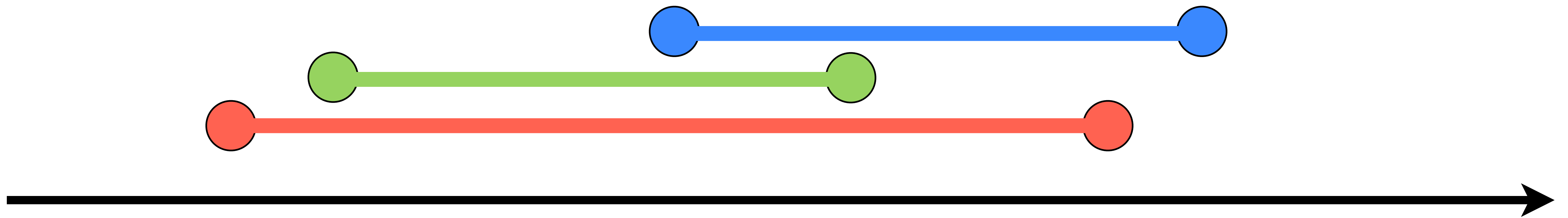



Interval between $X_1$ and $X_2$

Interval between $Y_1$ and $Y_2$

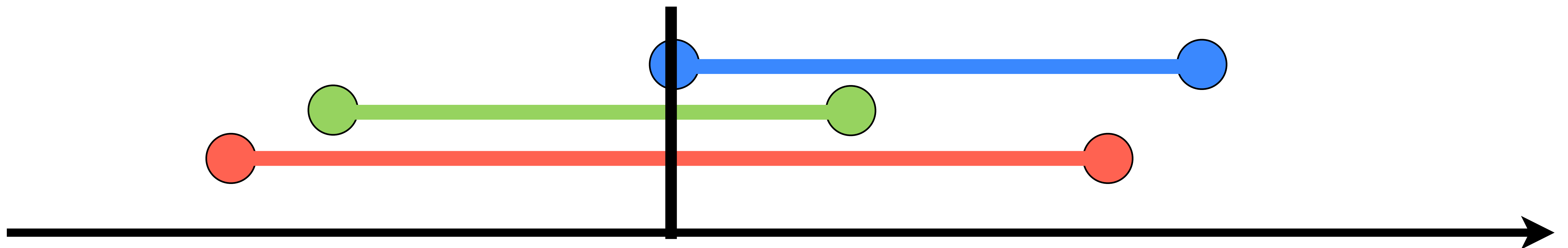
Intersection

# Intersecting 1D Intervals

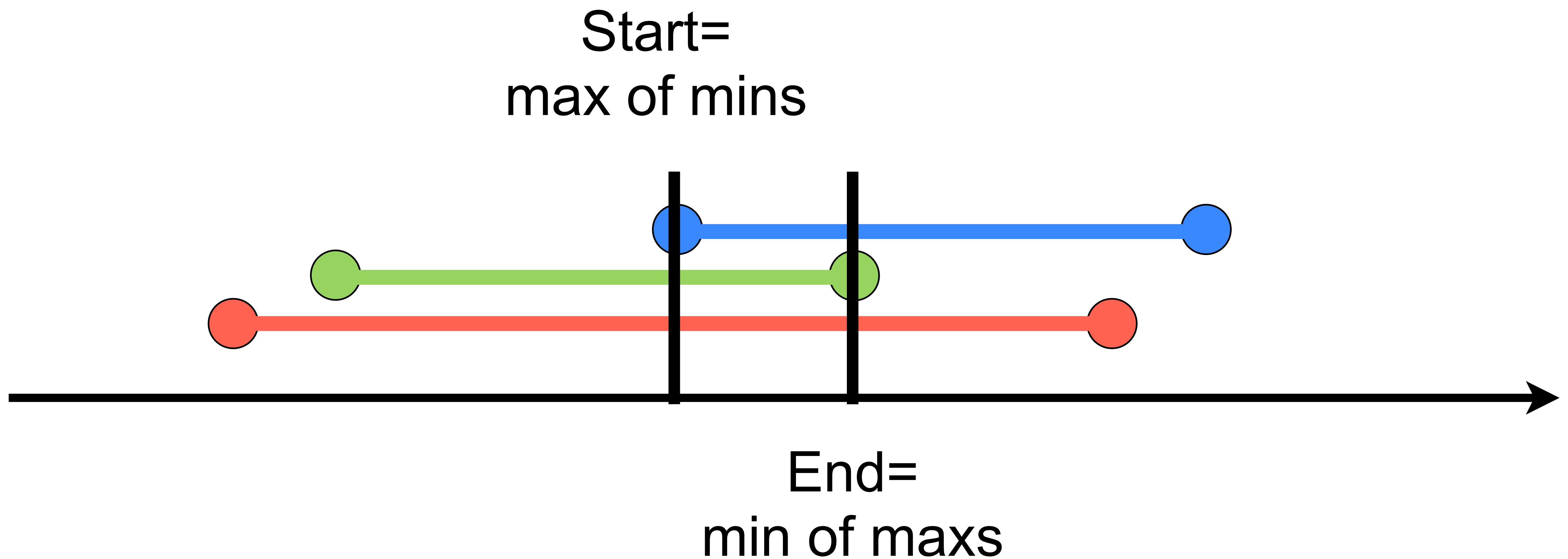


# Intersecting 1D Intervals

Start=
max of mins

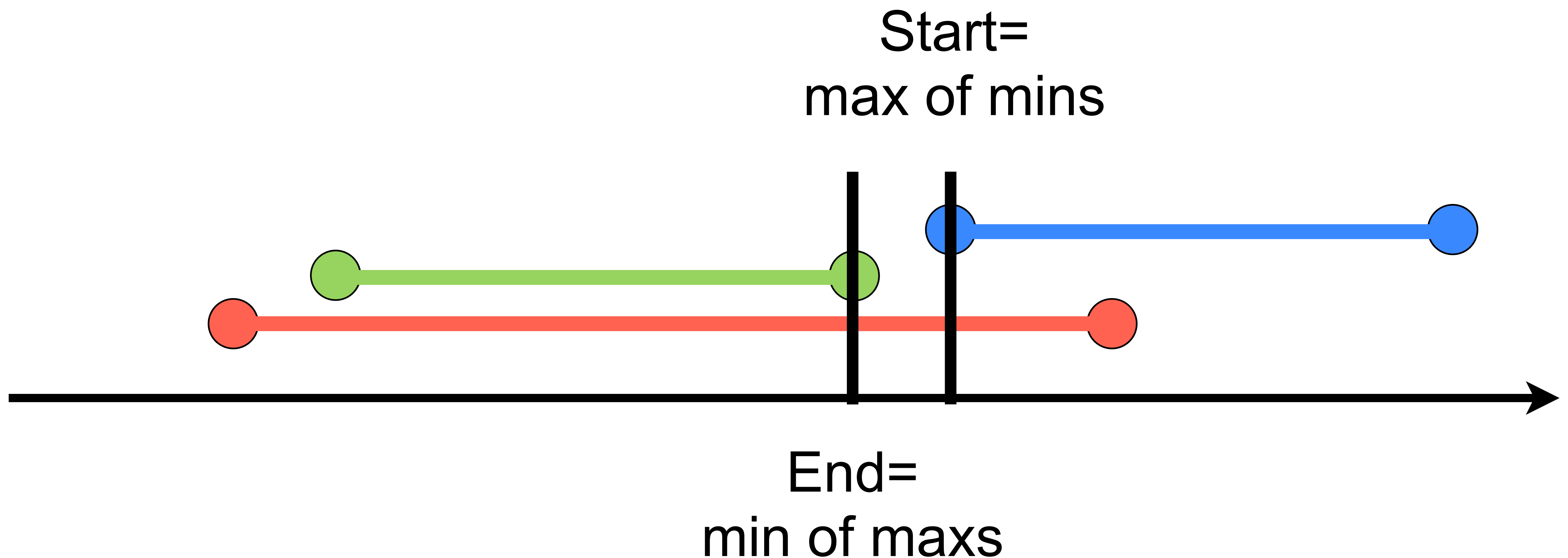


# Intersecting 1D Intervals



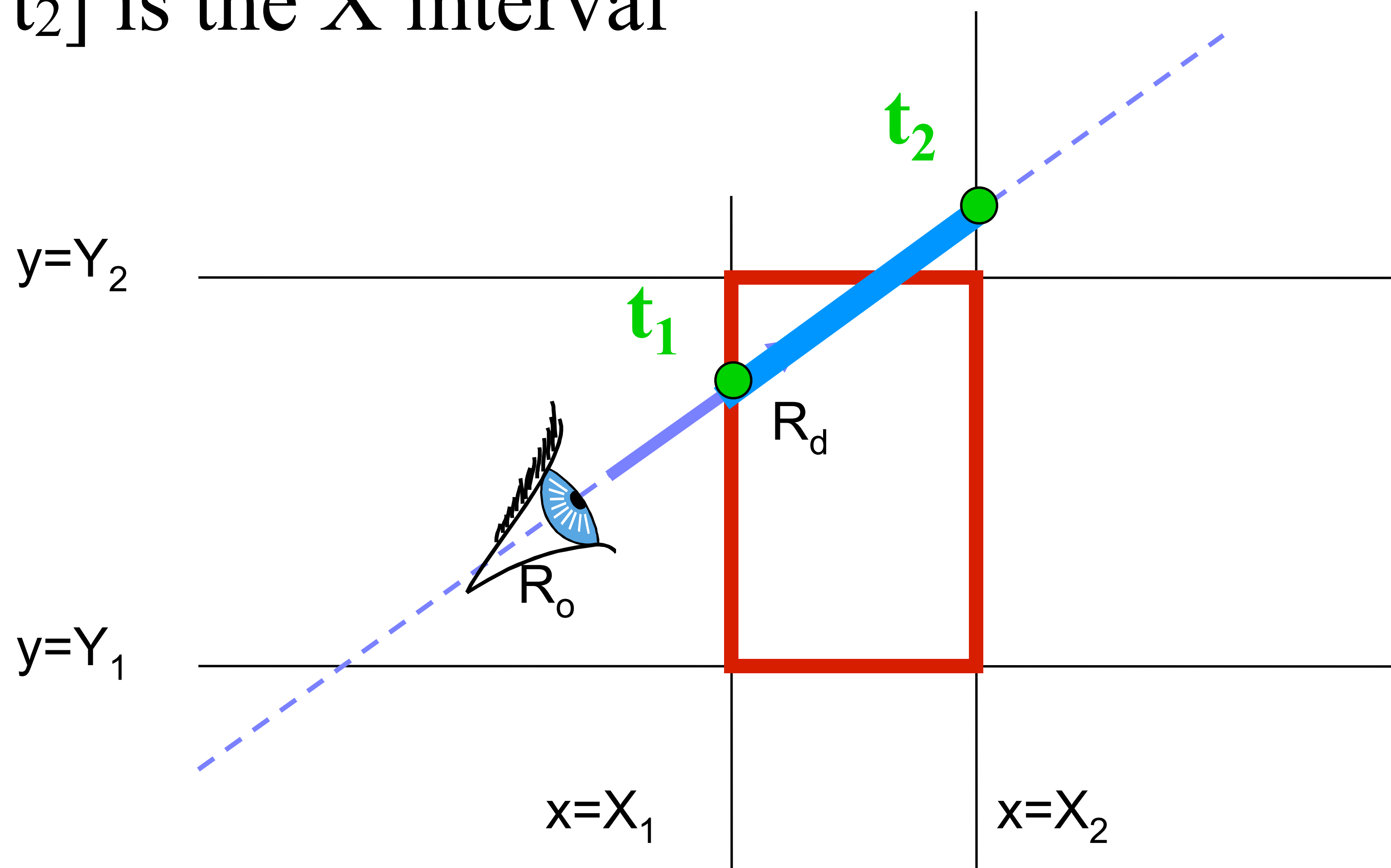
# Intersecting 1D Intervals

If Start > End, the intersection is empty!



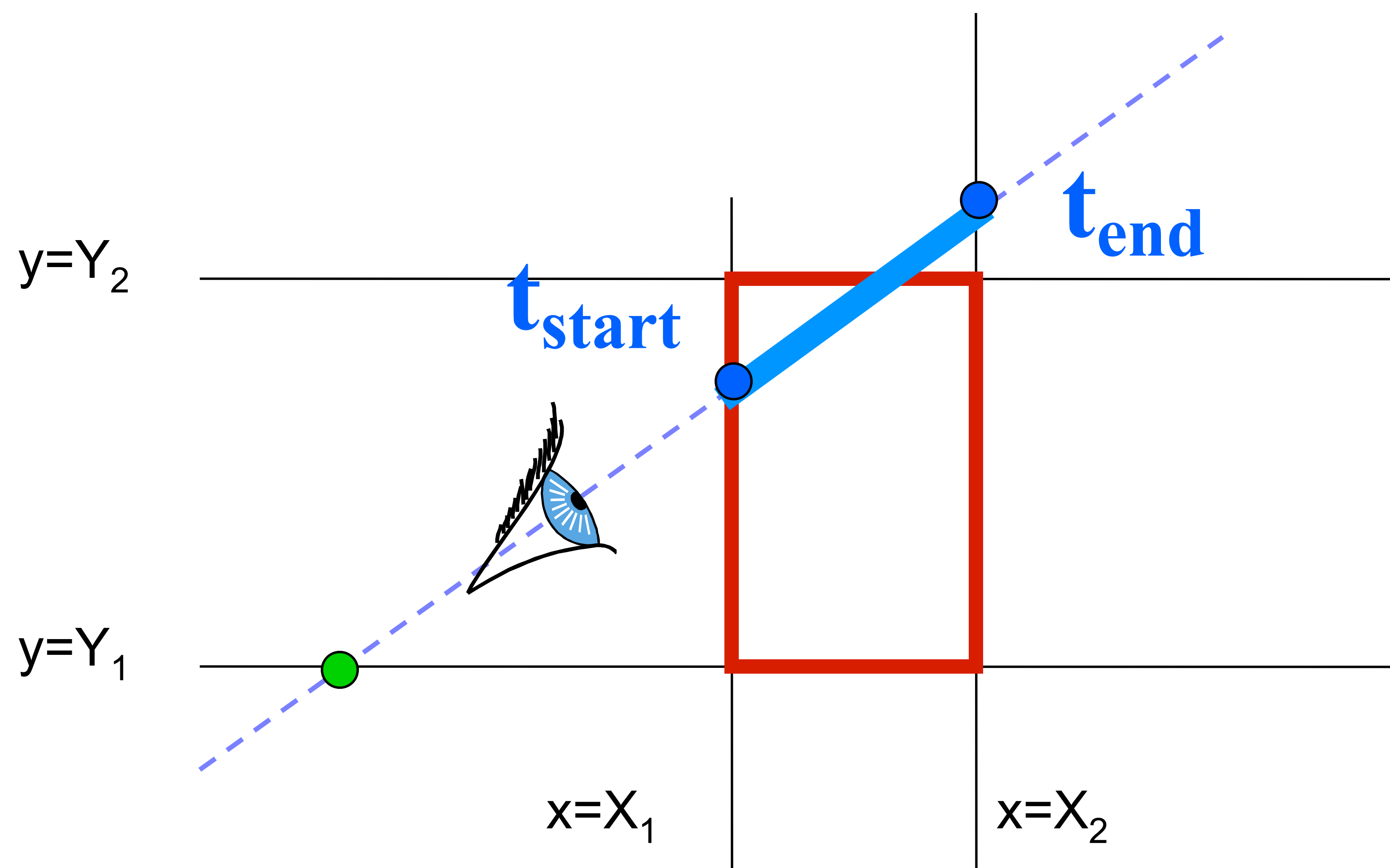
# Find Intersections Per Dimension

- 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



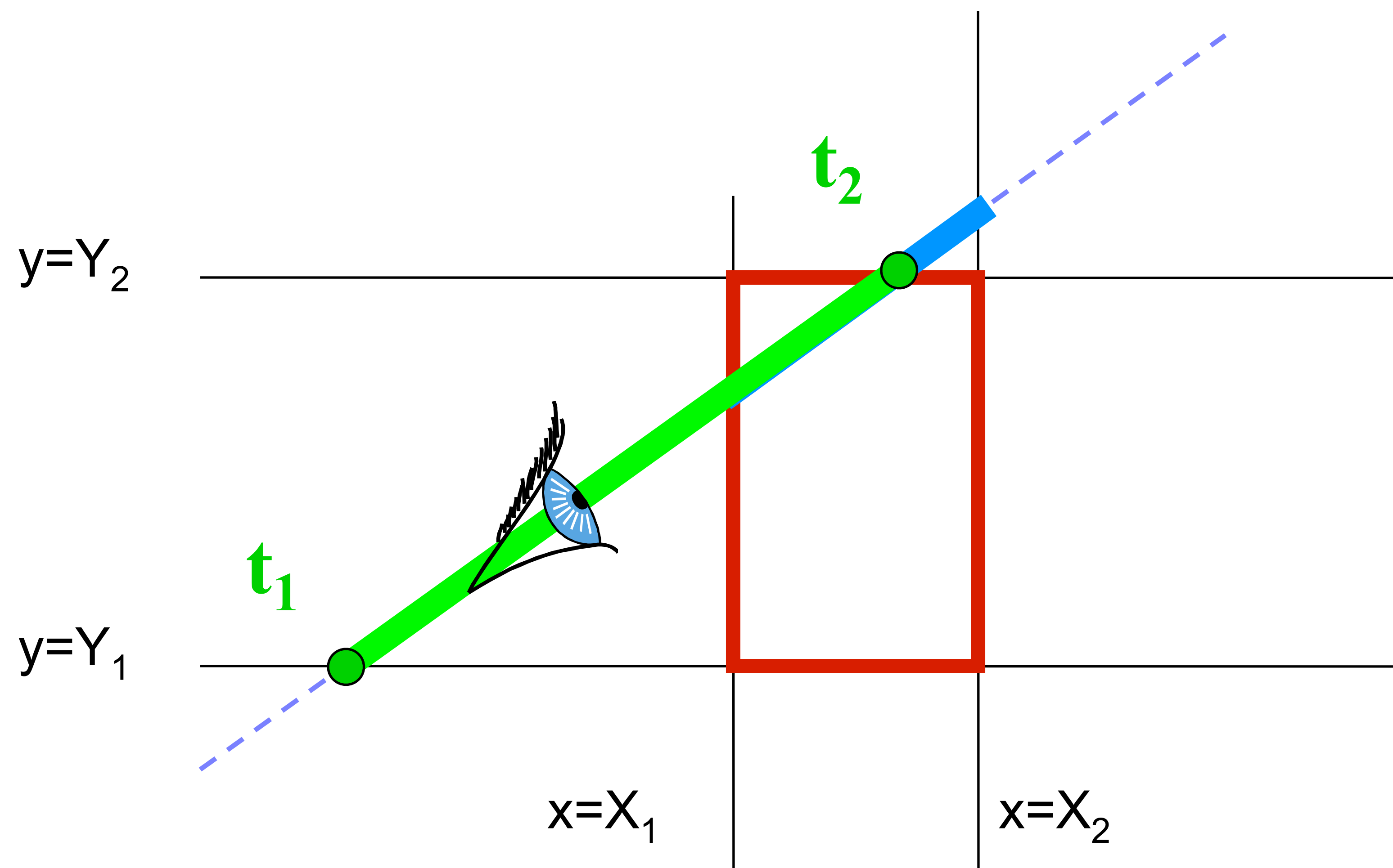
# Then Intersect Intervals

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



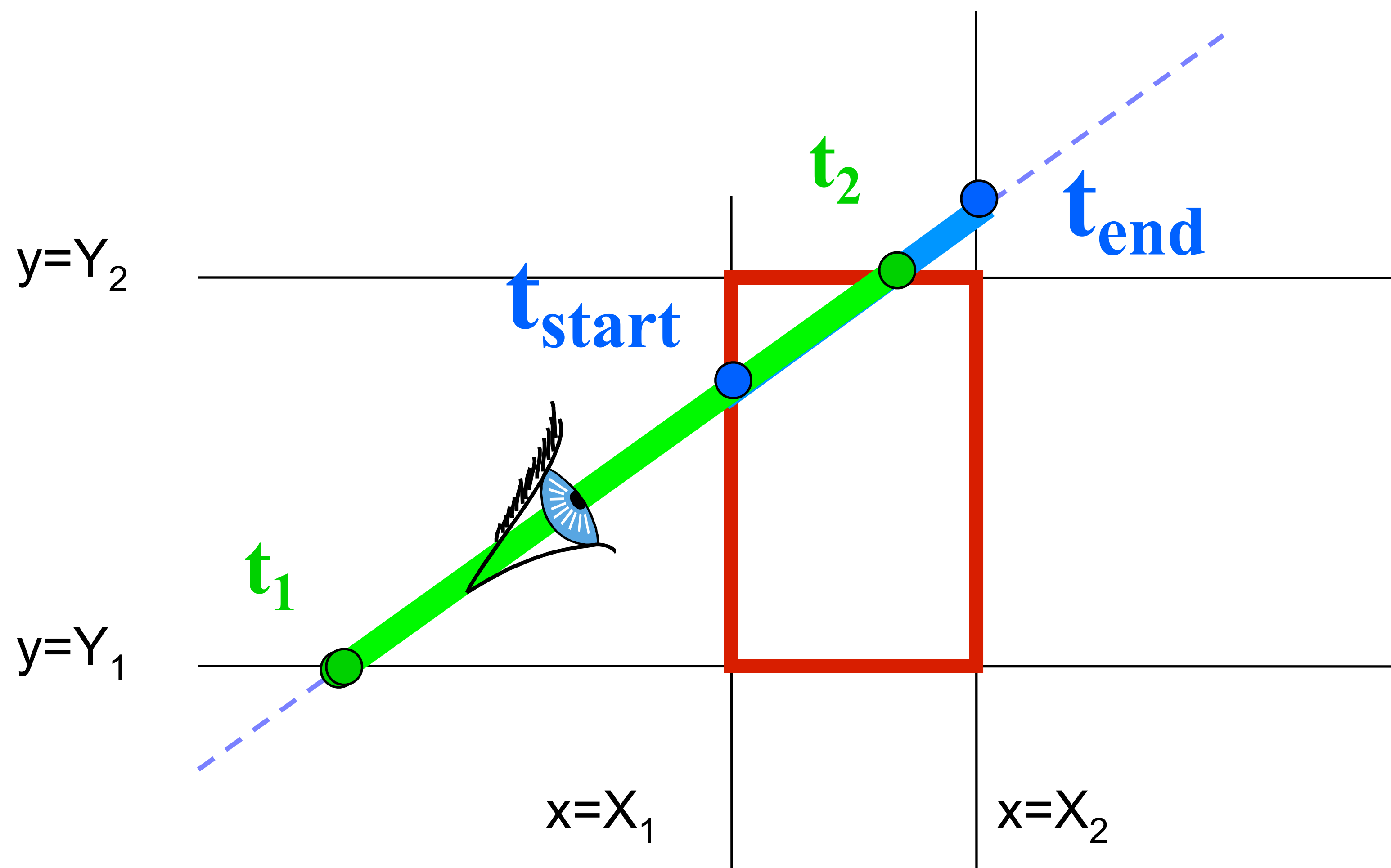
# Then Intersect Intervals

- Compute $t_1$ and $t_2$ for Y...



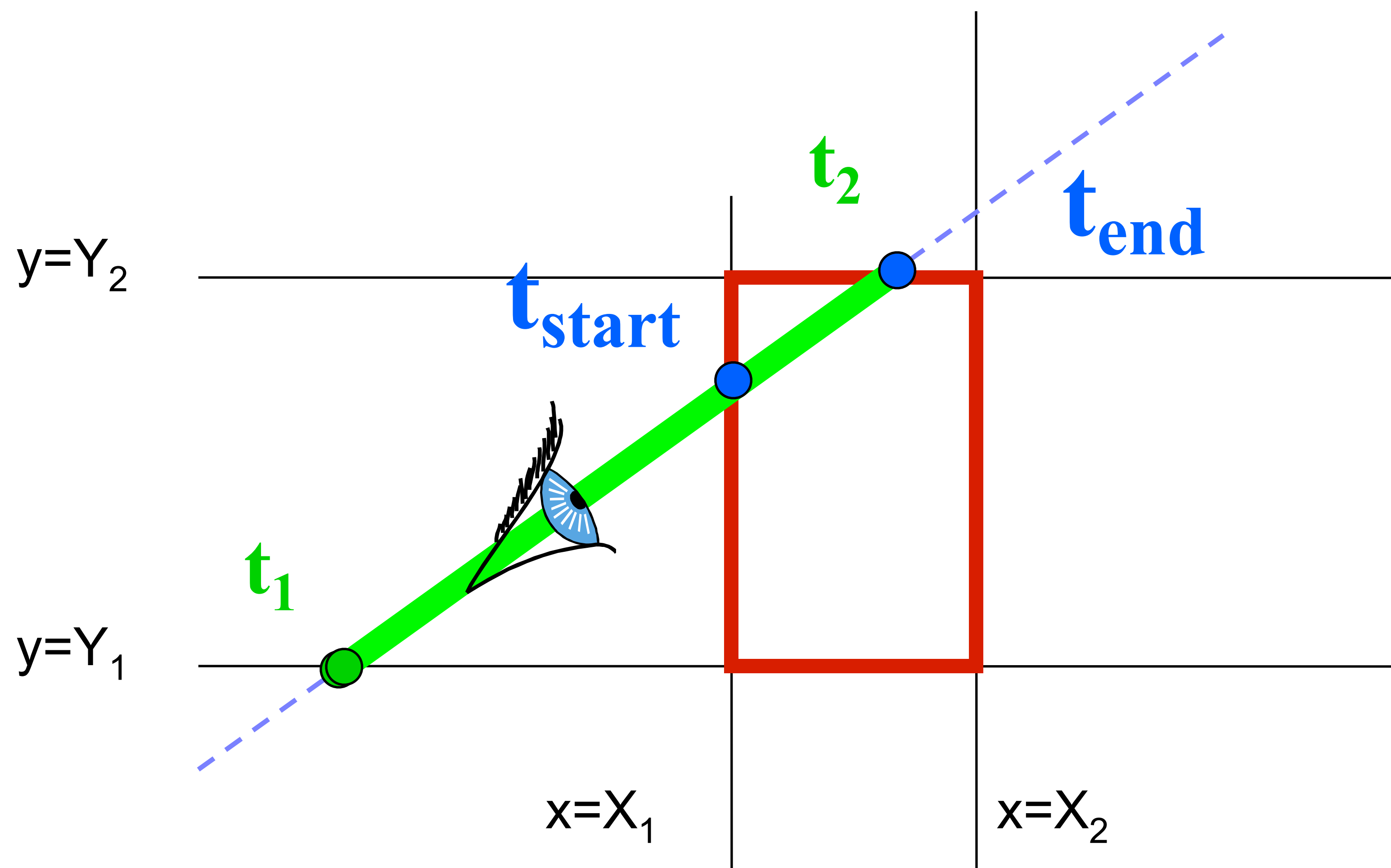
# Then Intersect Intervals

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



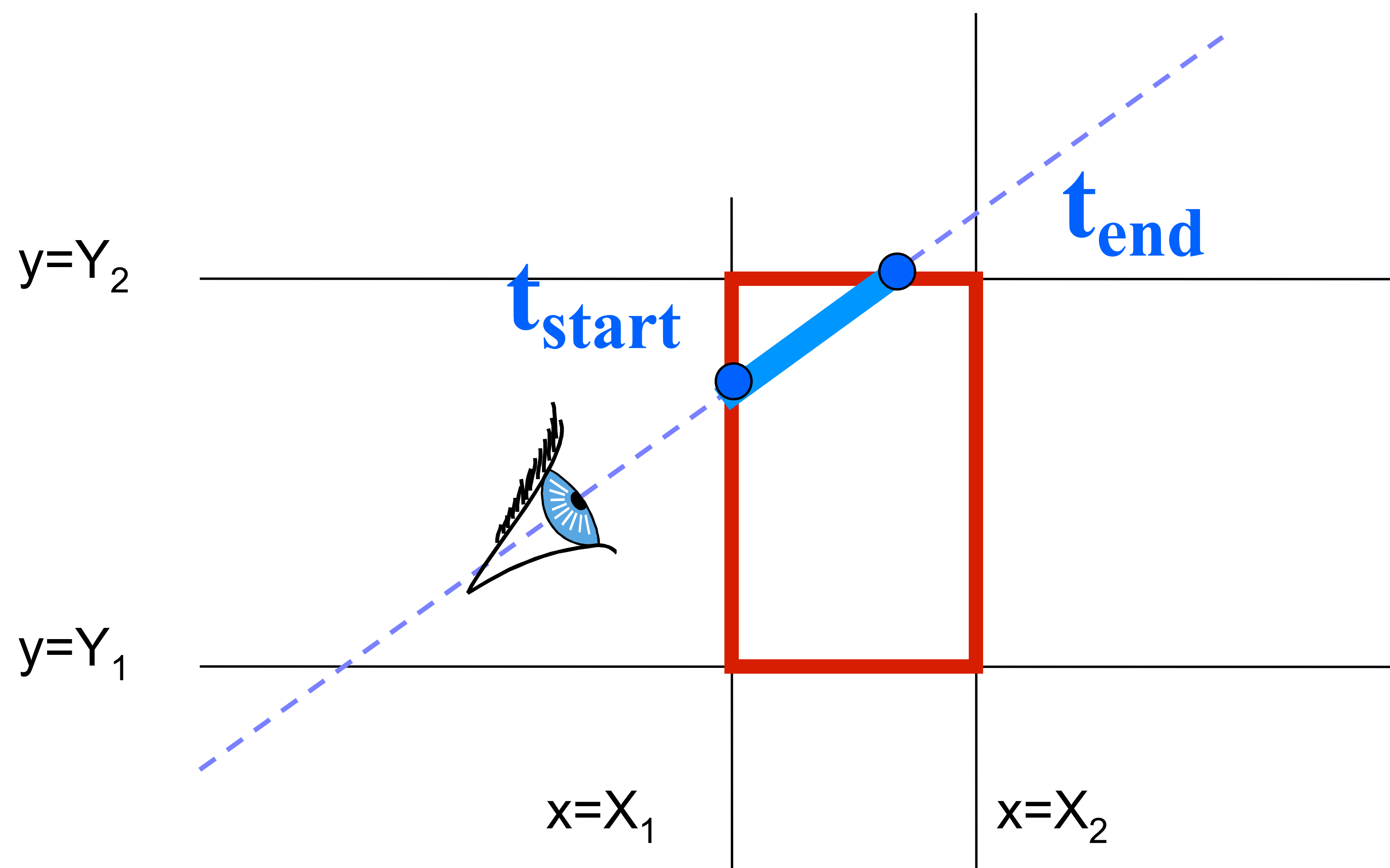
# Then Intersect Intervals

- 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

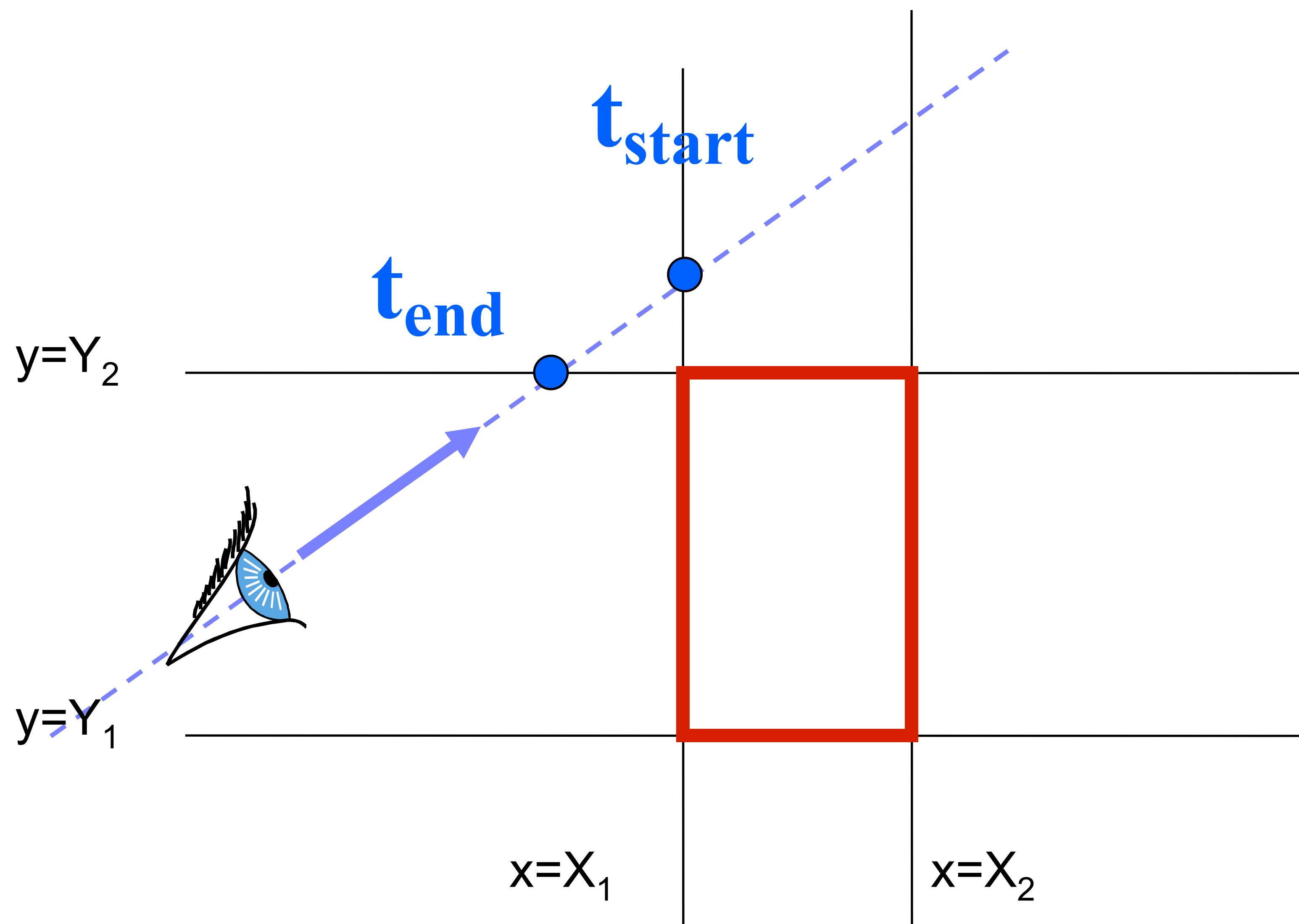
- Update $t_{start}$ & $t_{end}$ for each subsequent dimension
  - If $t_1 > t_{start}$, $t_{start} = t_1$
  - If $t_2 < t_{end}$, $t_{end} = t_2$




:-)

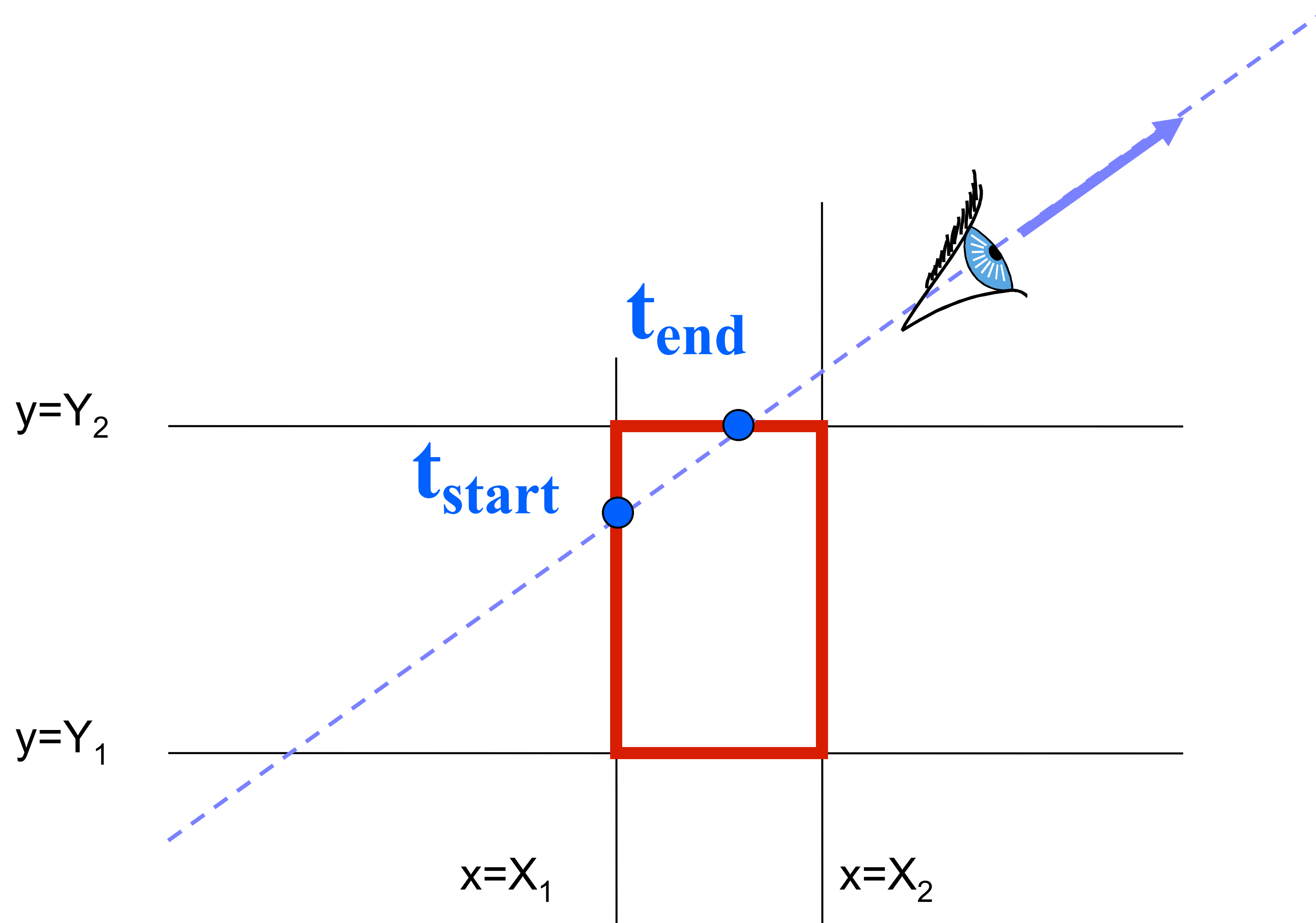
# Is there an Intersection?

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



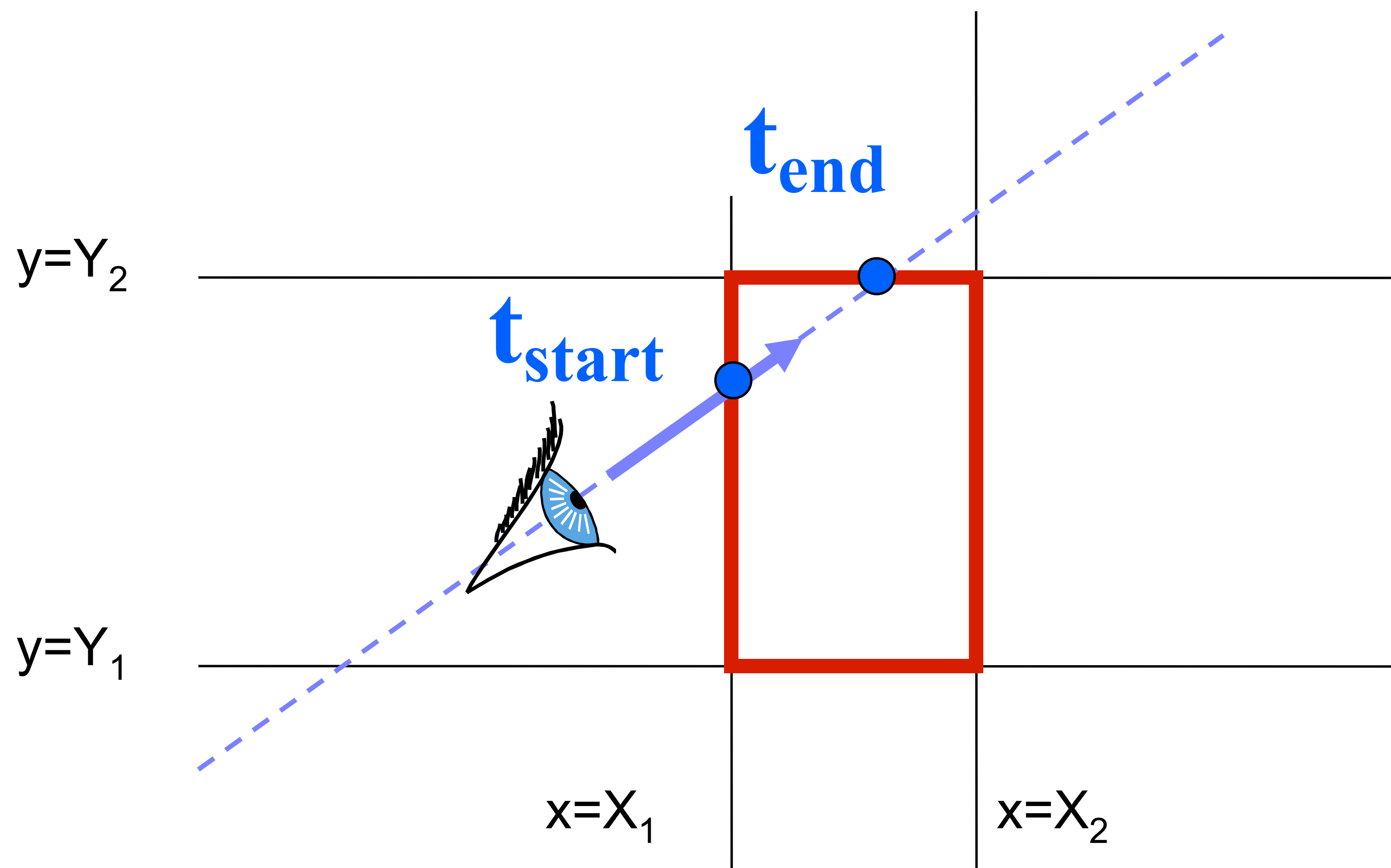
# Is the Box Behind the Eyepoint?

- 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



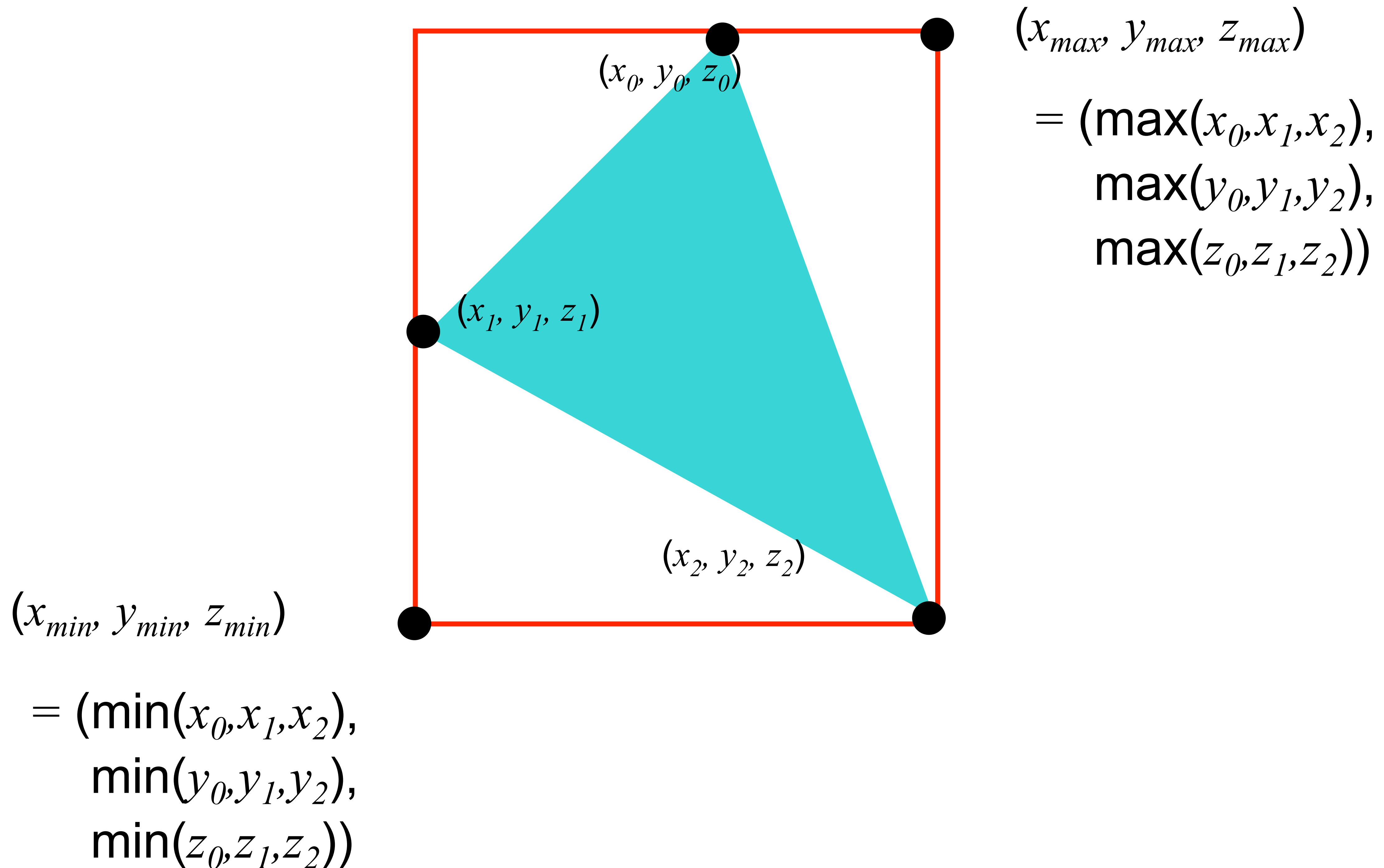
# Ray-Box Intersection Summary

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

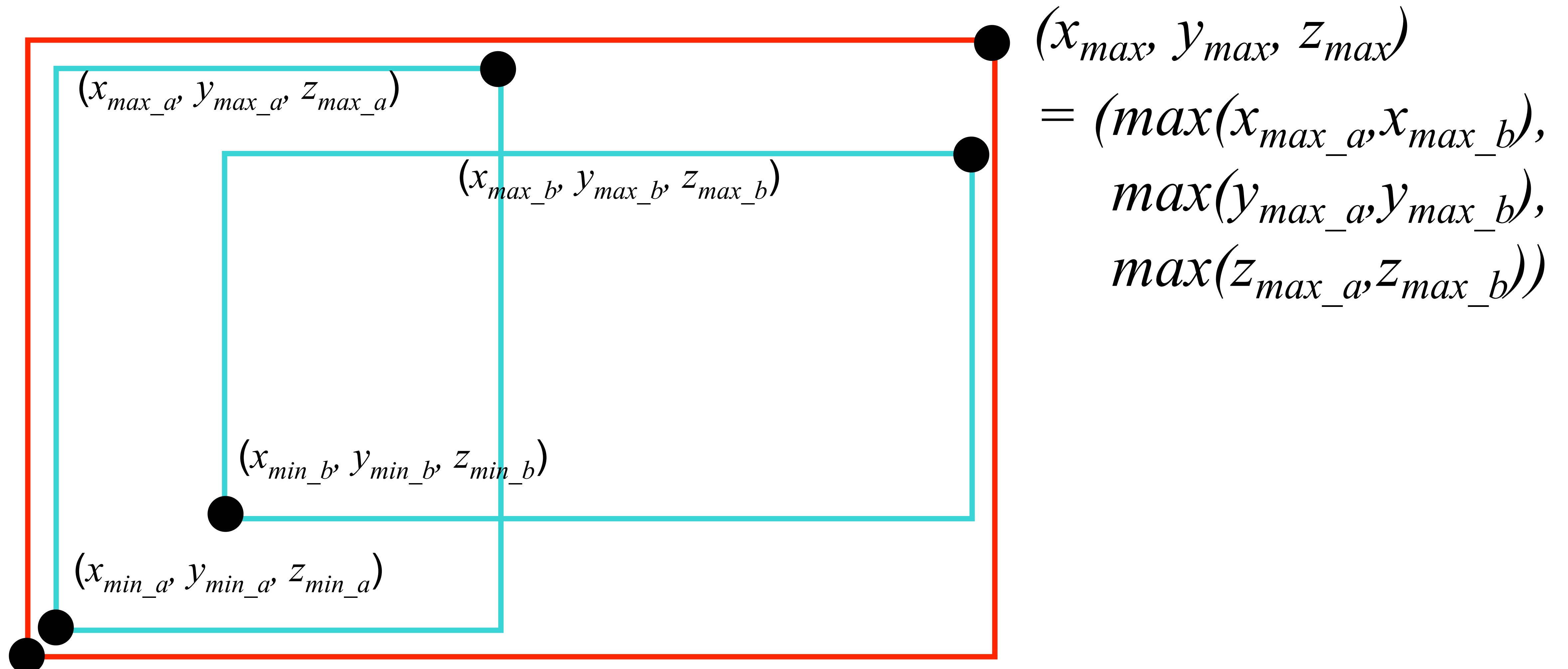
# Efficiency Issues

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

# AA Bounding Box of a Triangle



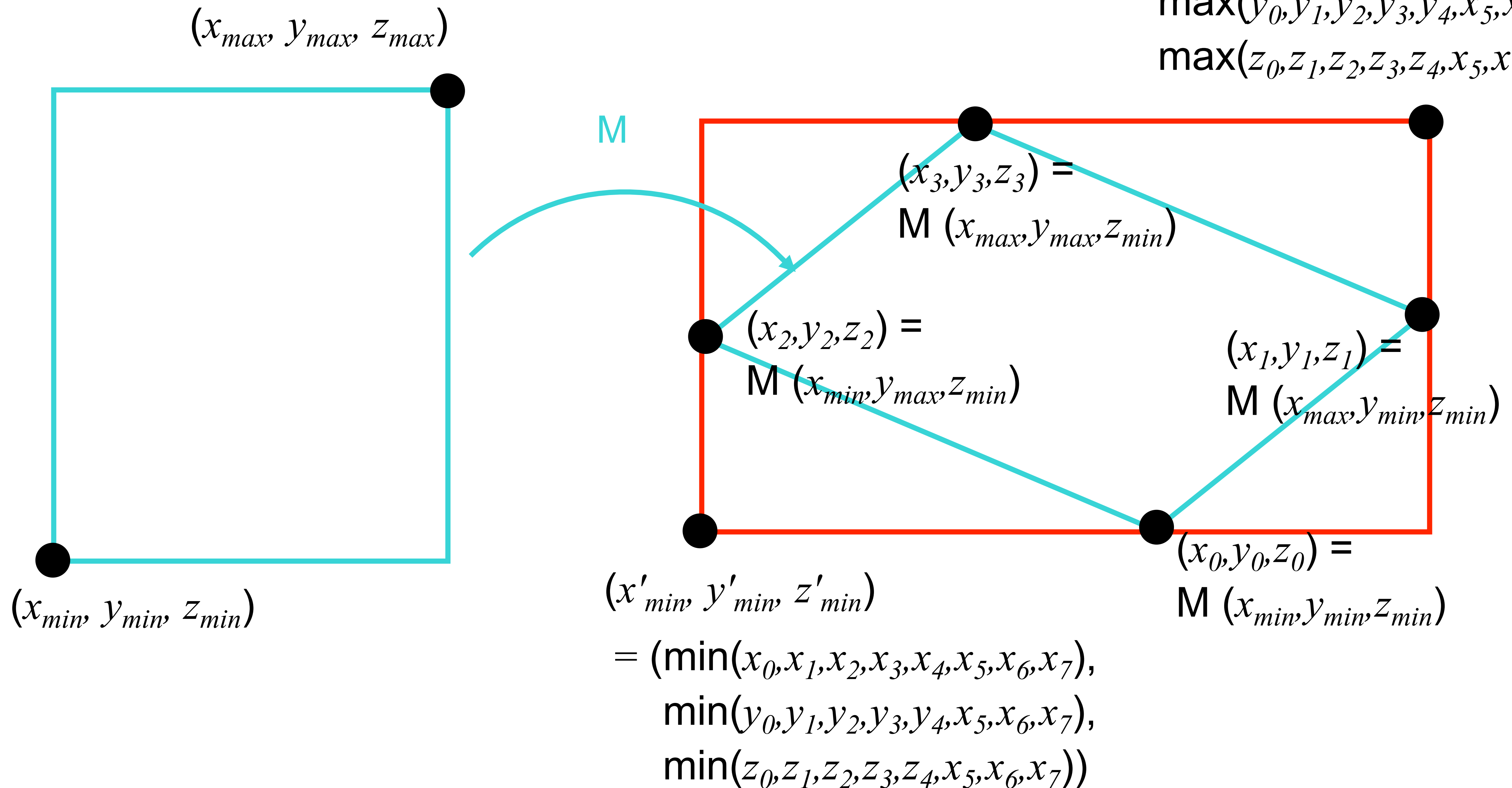
# AA Bounding Box of a Group




$$(x_{min}, y_{min}, z_{min}) = (min(x_{min_a}, x_{min_b}), \; min(y_{min_a}, y_{min_b}), \; min(z_{min_a}, z_{min_b}))$$

# AA Bounding Box After Transform

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



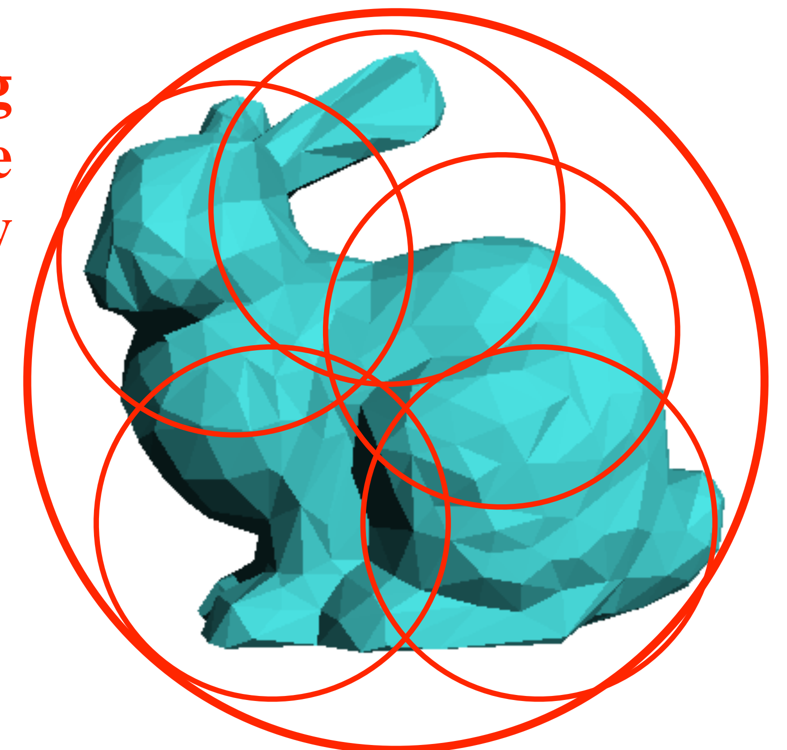
# Questions?

# Are Bounding Volumes Enough?

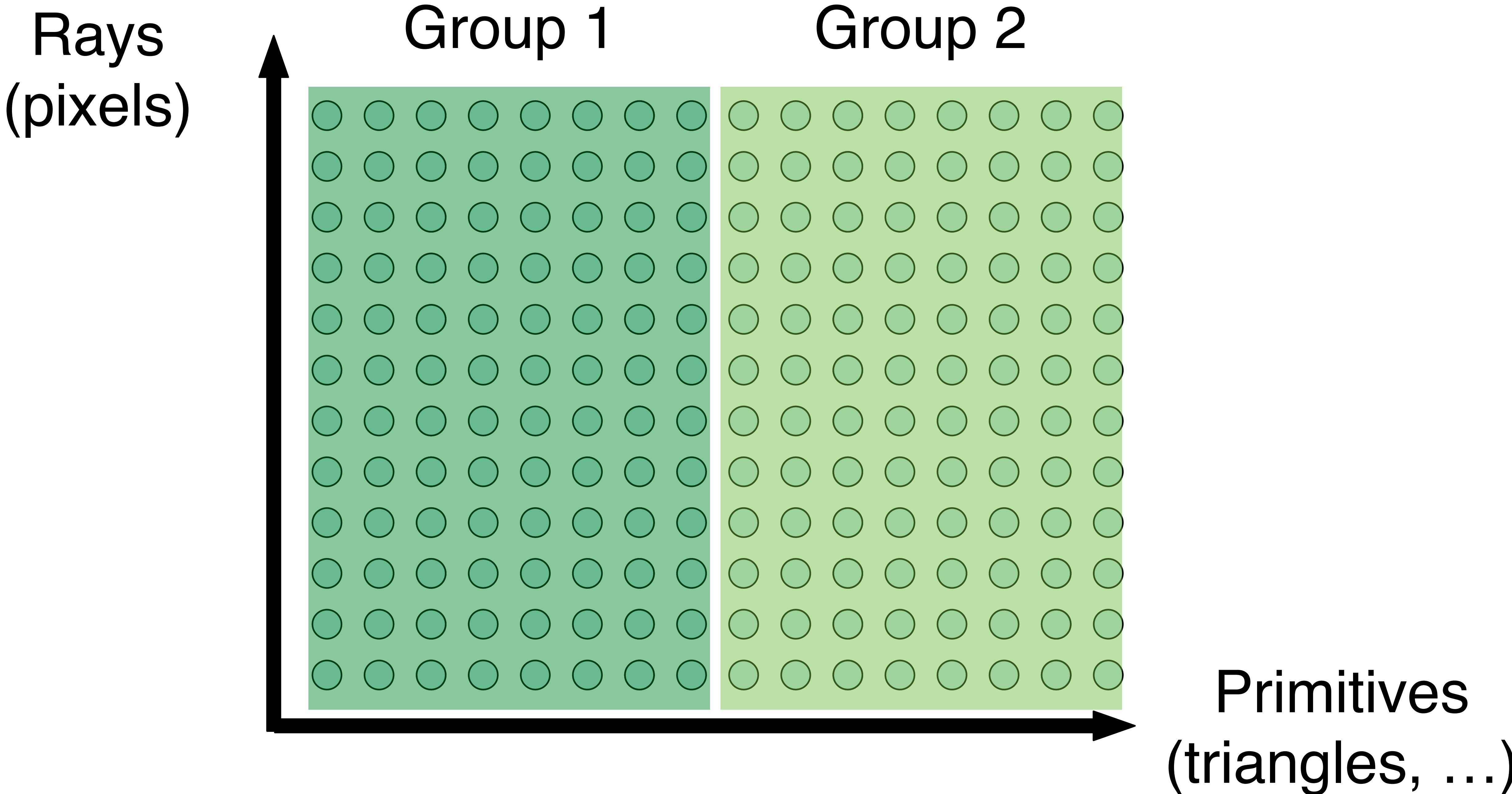
- If ray hits bounding volume, test all contained primitives?
  - Lots of work, think of a 10M-triangle mesh

# Bounding Volume Hierarchies

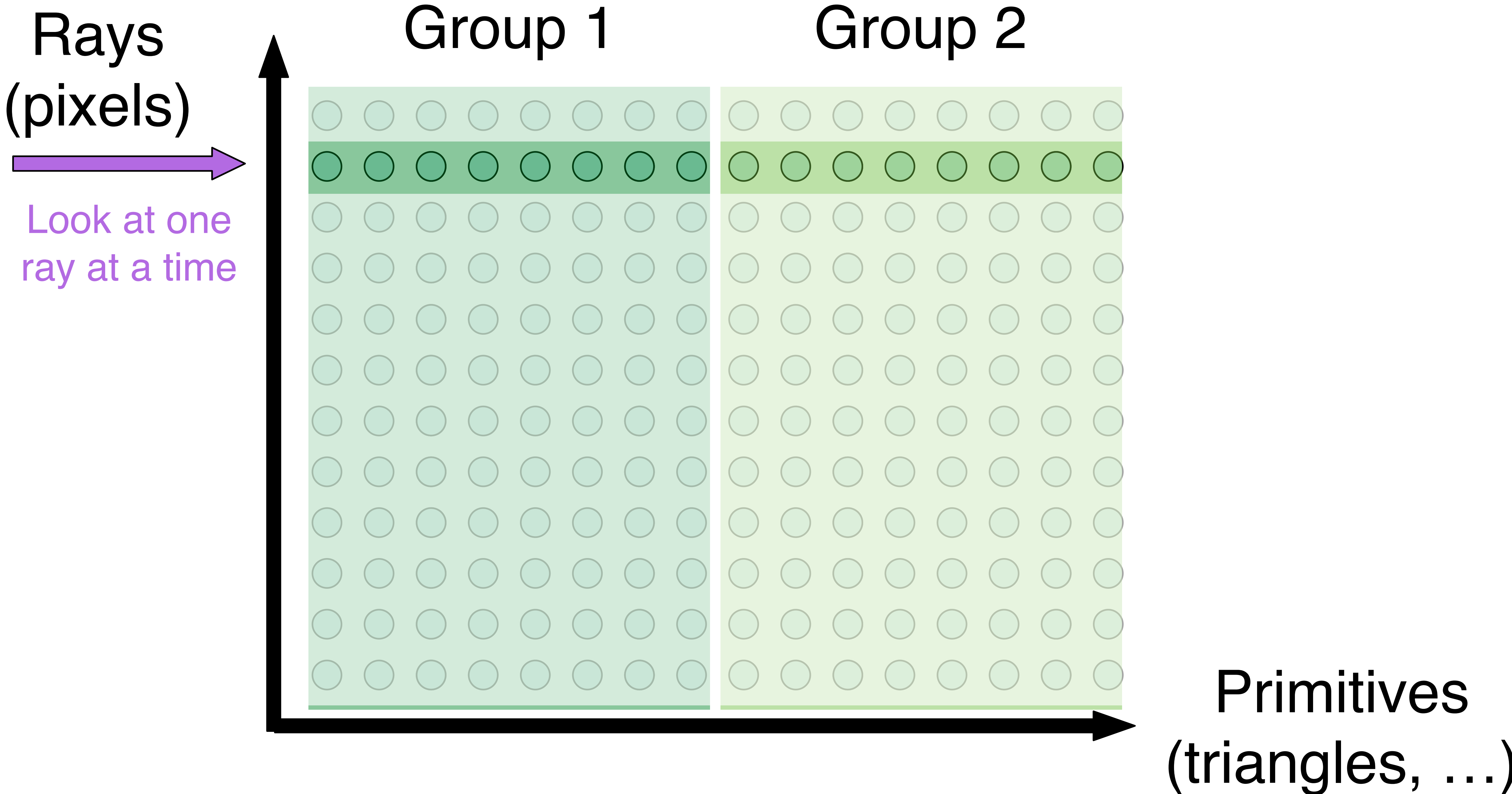
- If ray hits bounding volume, test all contained primitives?
  - Lots of work, think of a 10M-triangle mesh
- You guessed it already, we'll split the primitives in groups and build recursive bounding volumes



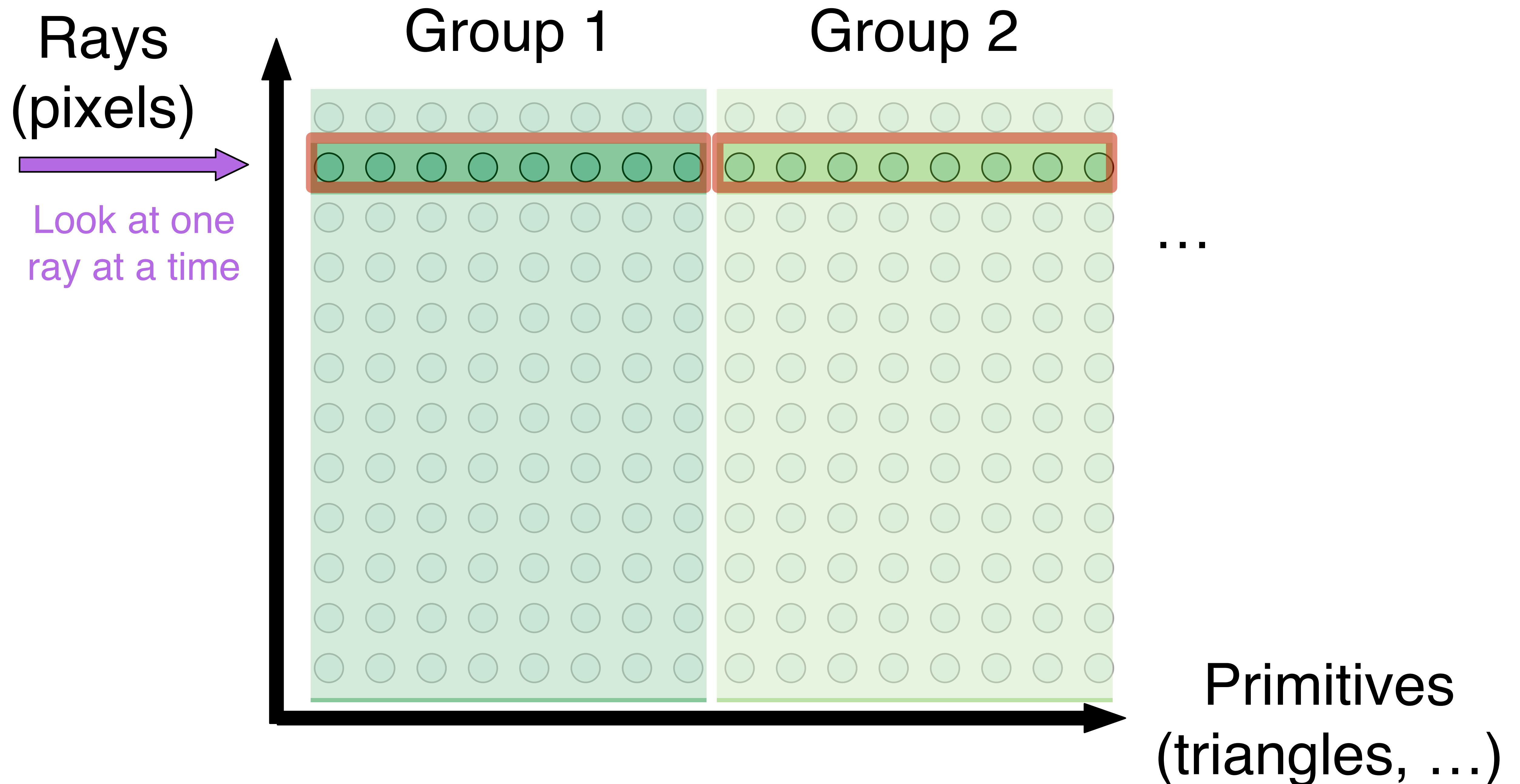
# Hierarchical Subdivision



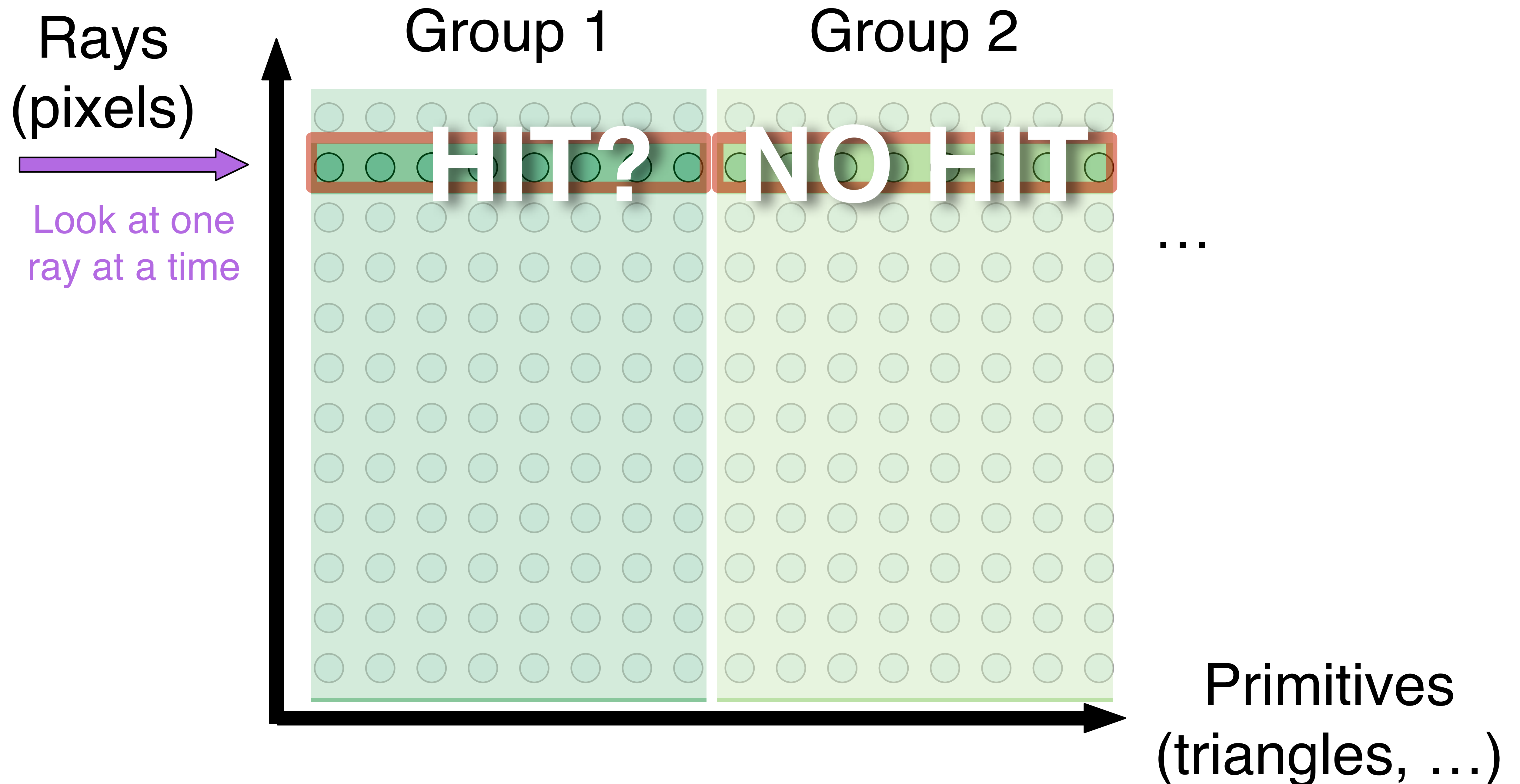
# Hierarchical Subdivision



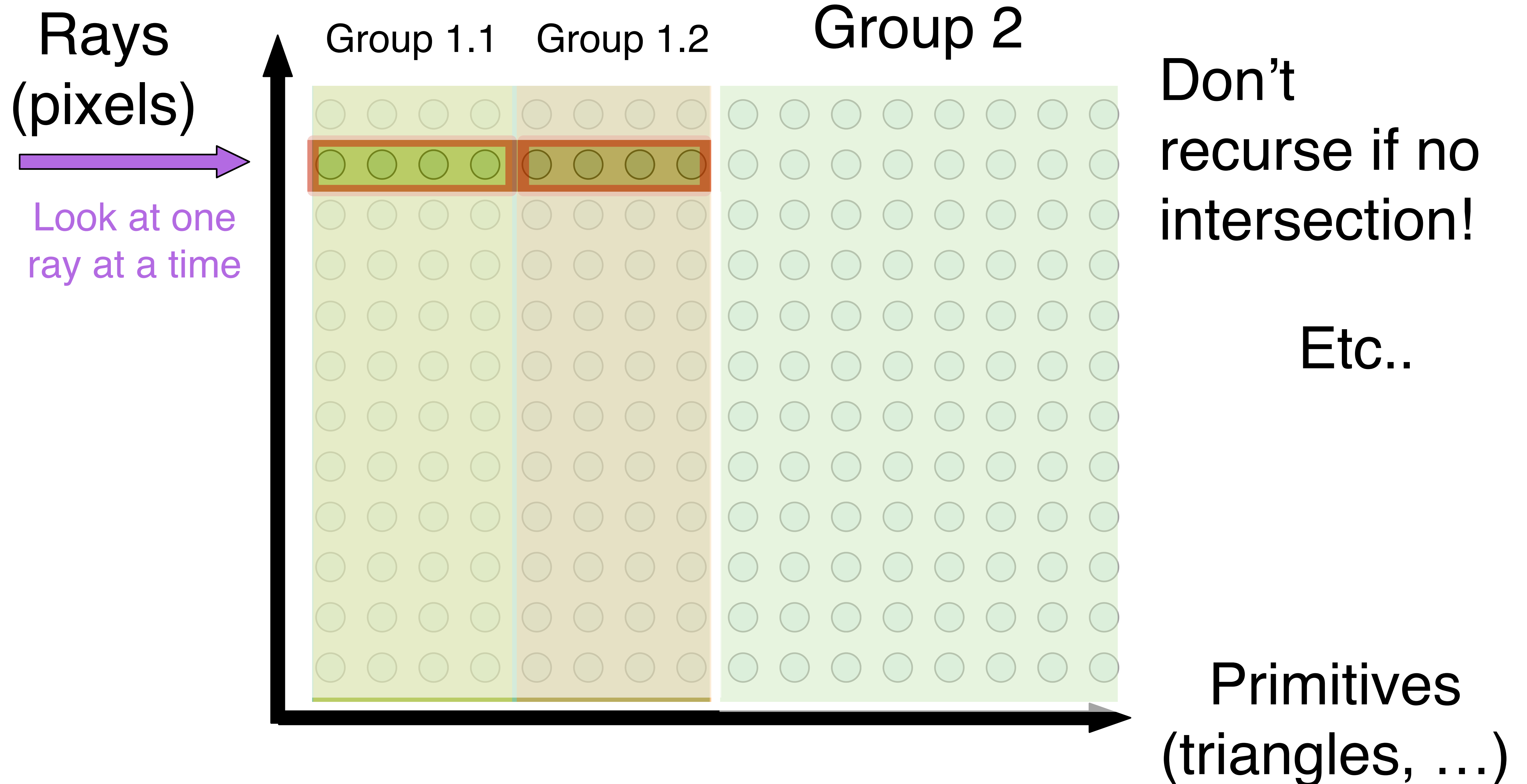
# Hierarchical Subdivision



# Hierarchical Subdivision

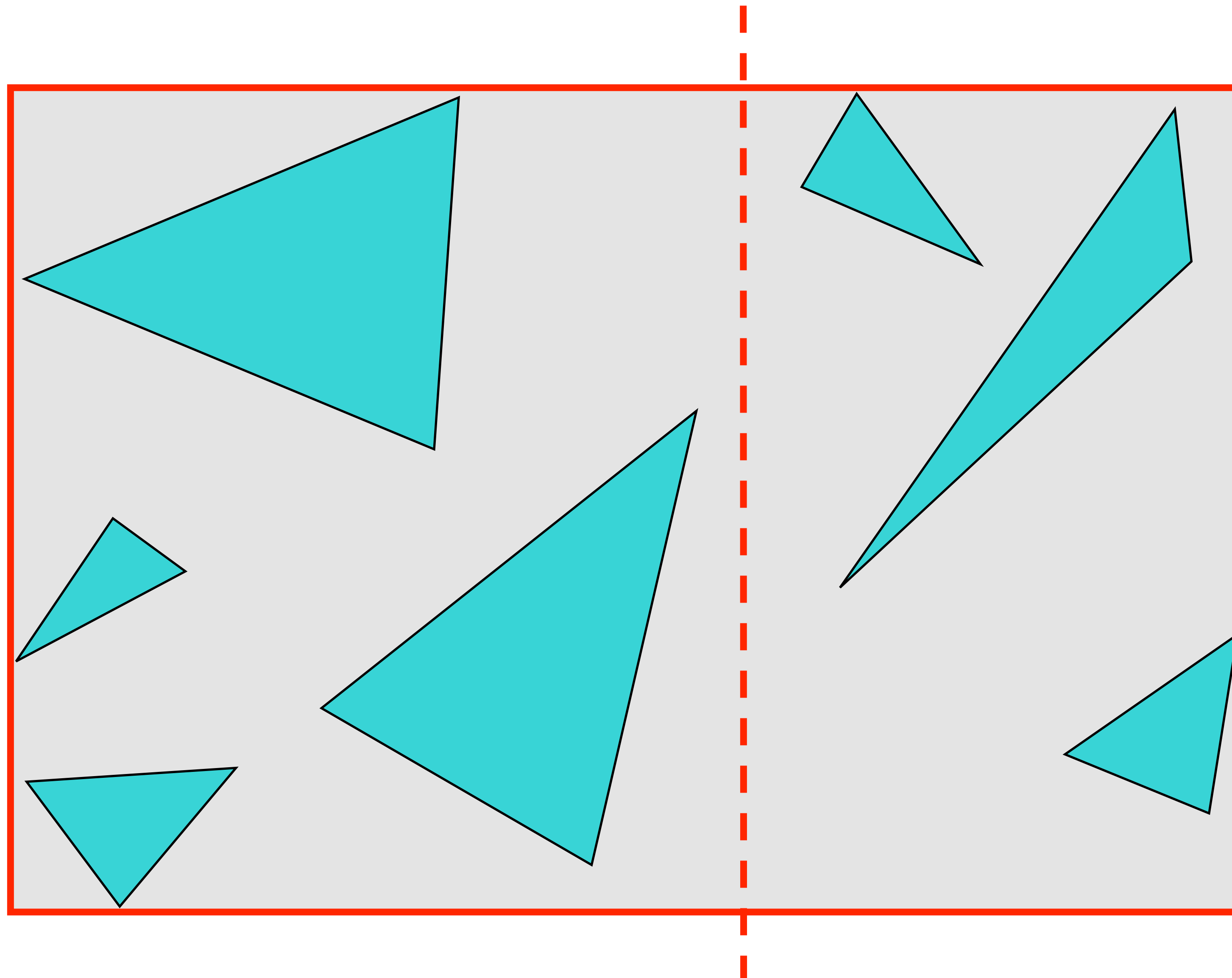


# Hierarchical Subdivision…



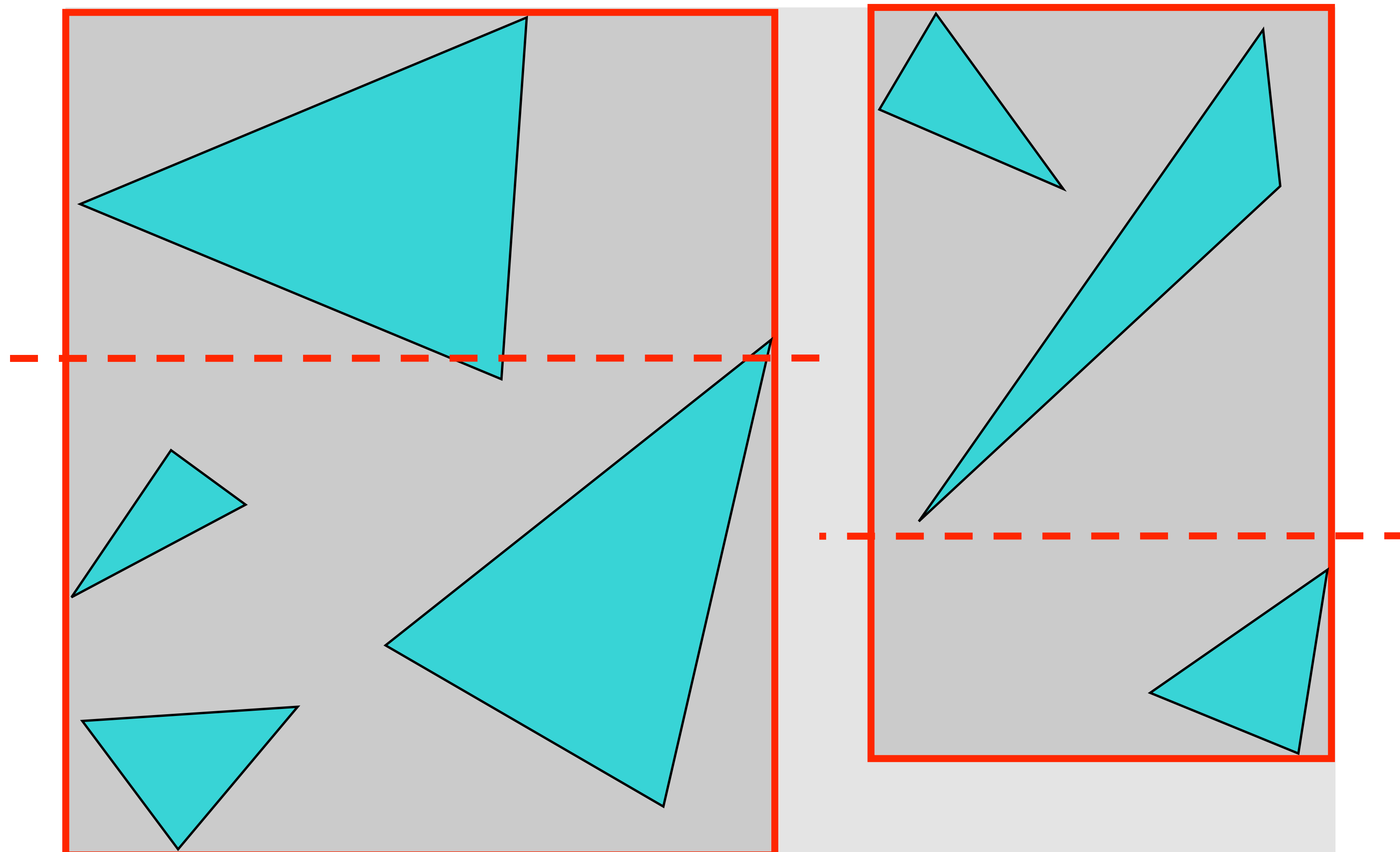
# Bounding Volume Hierarchy (BVH)

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



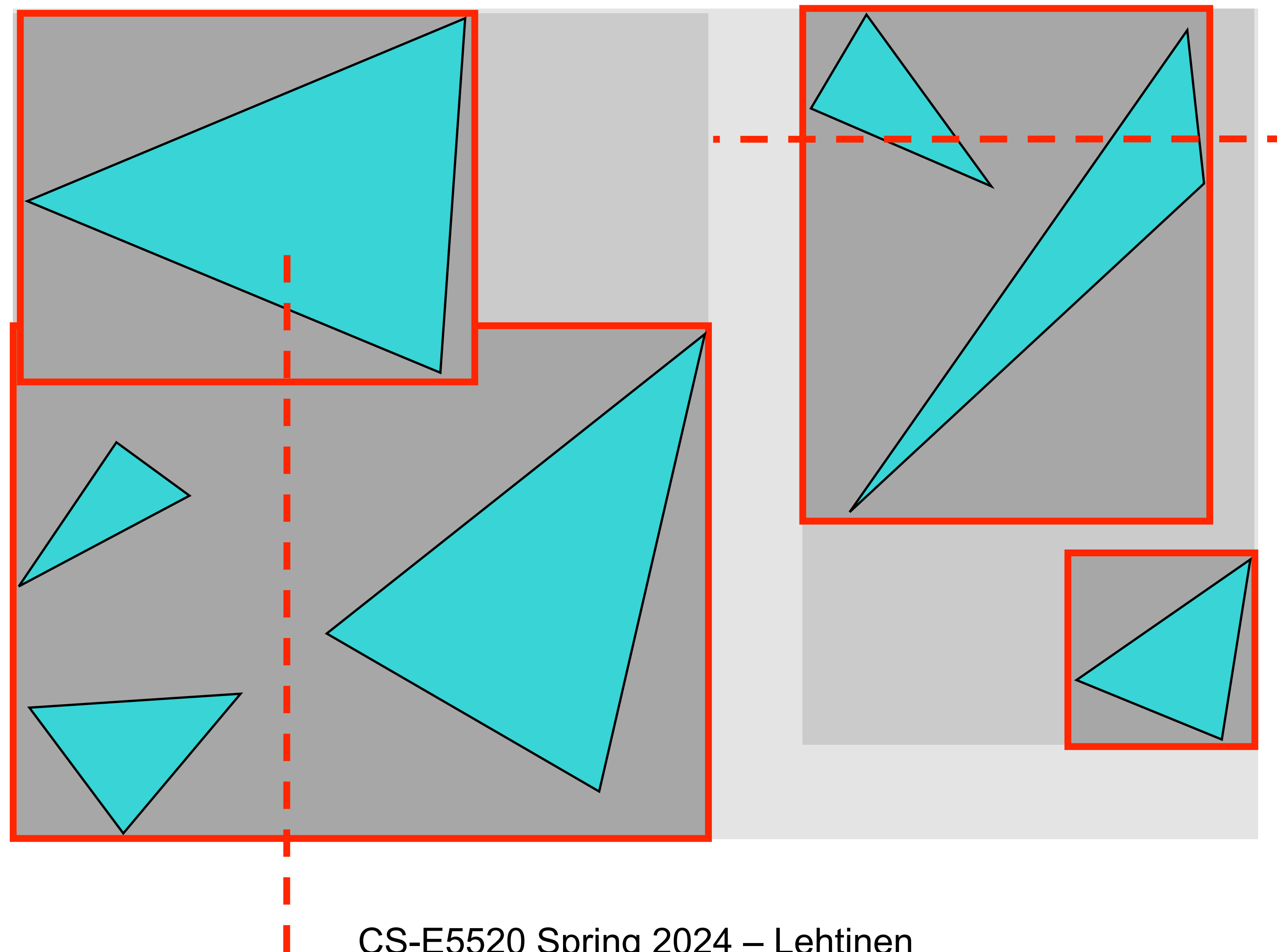
# Bounding Volume Hierarchy (BVH)

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



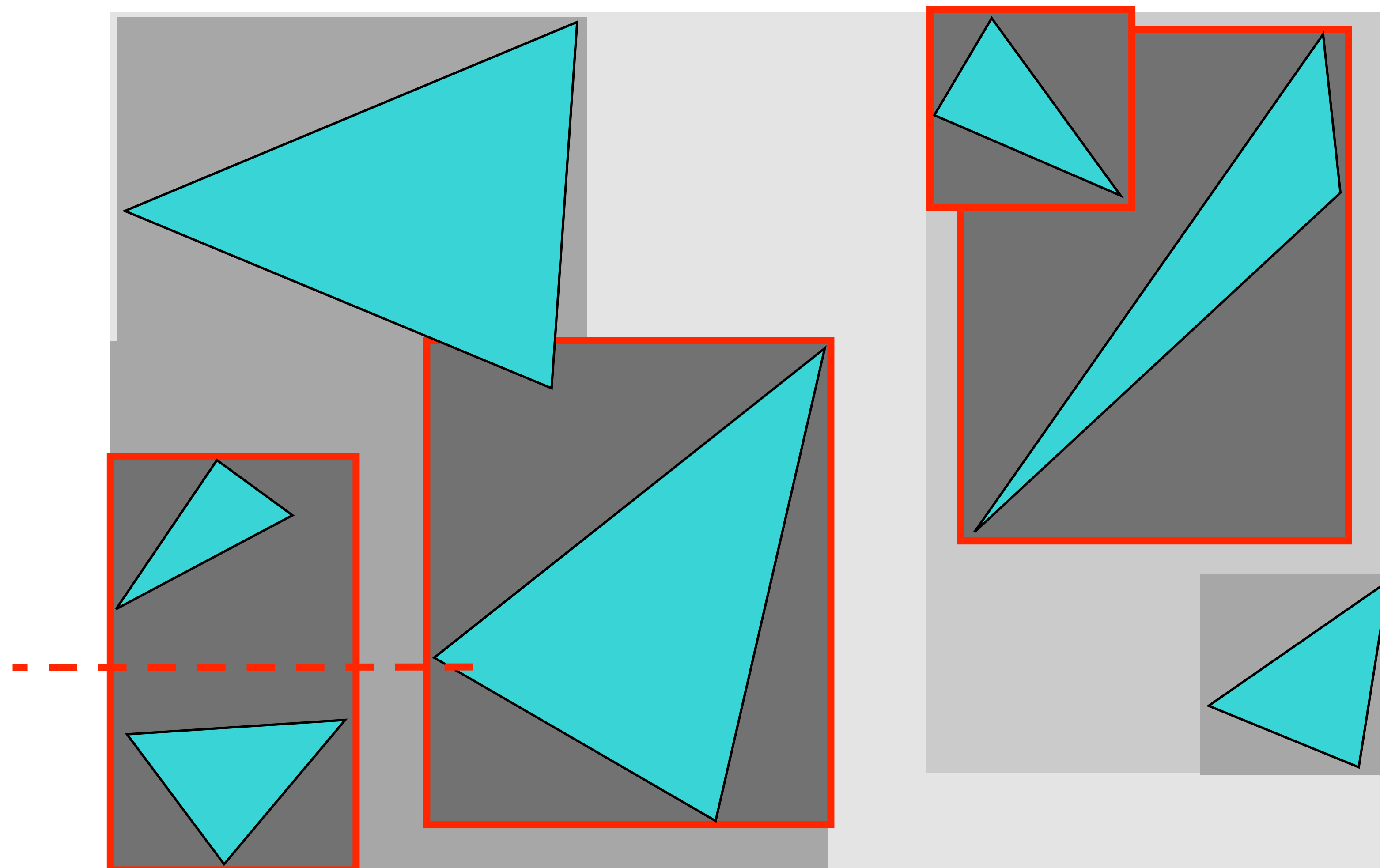
# Bounding Volume Hierarchy (BVH)

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



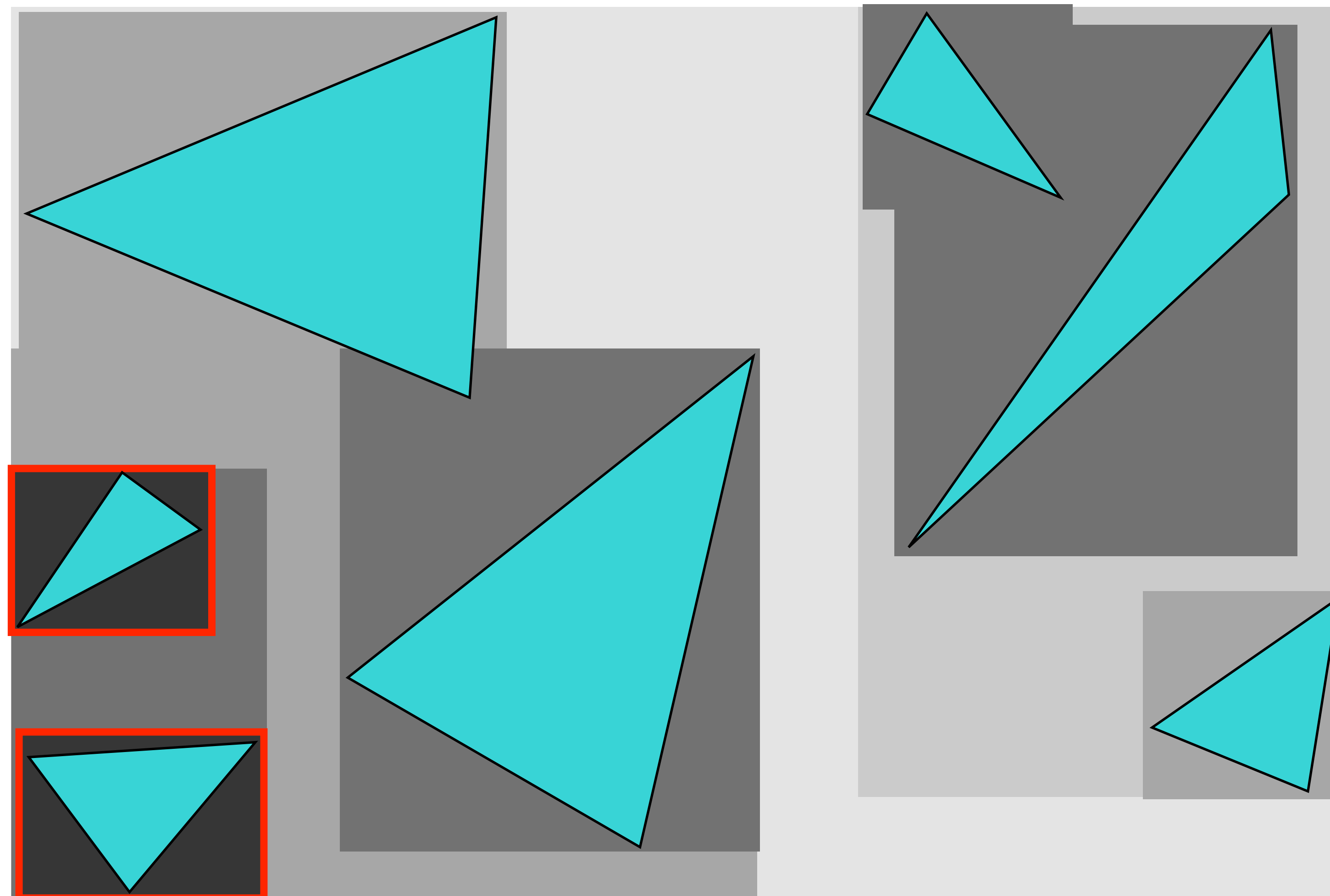
# Bounding Volume Hierarchy (BVH)

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



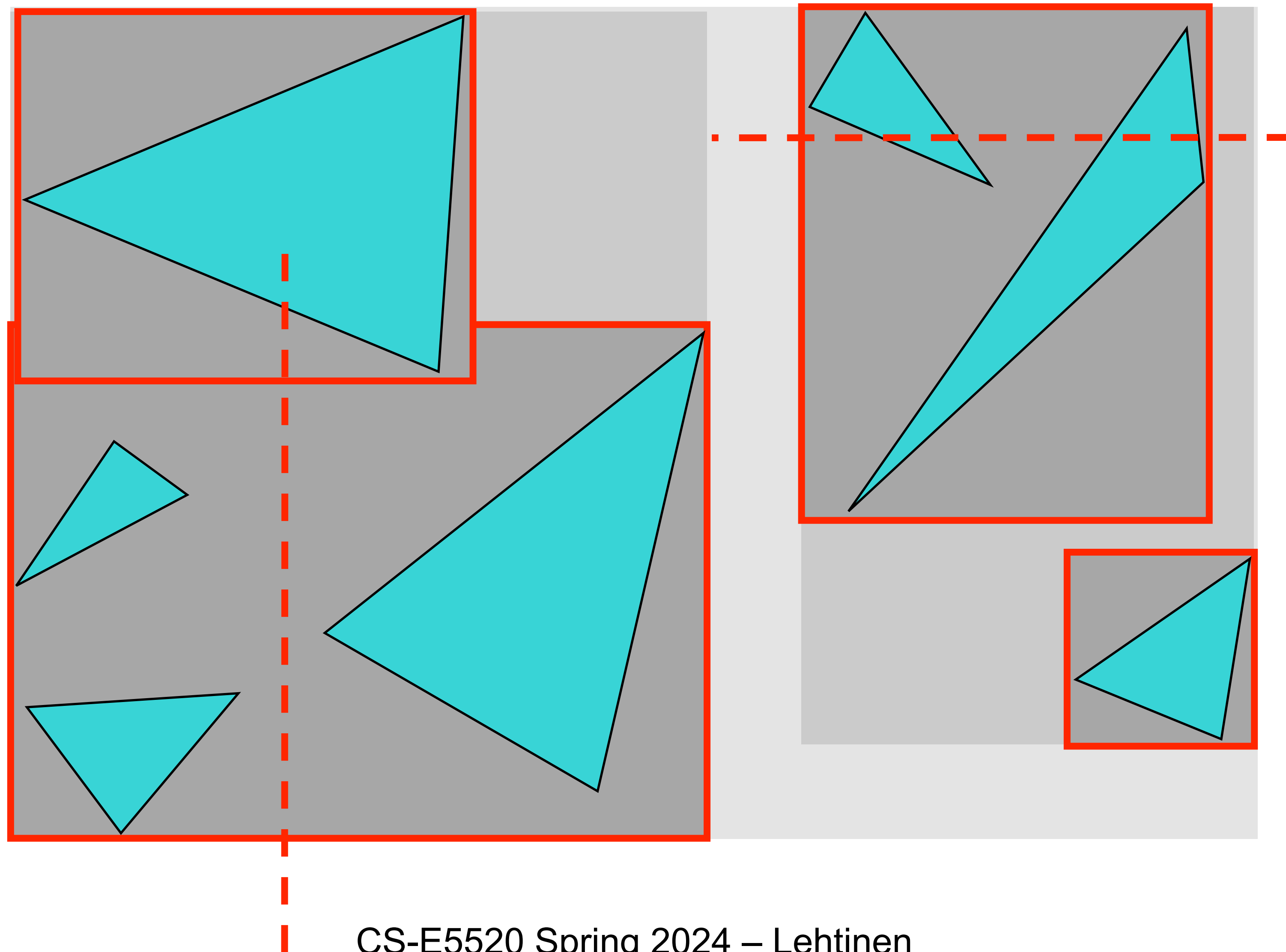
# Bounding Volume Hierarchy (BVH)

- 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 along longest axis, put half of objects on each side *OR*
- Use modeling hierarchy
  - (Actually, don't. You're relying on the artist for speed.)



# Interlude

- The Path to Path Traced Movies

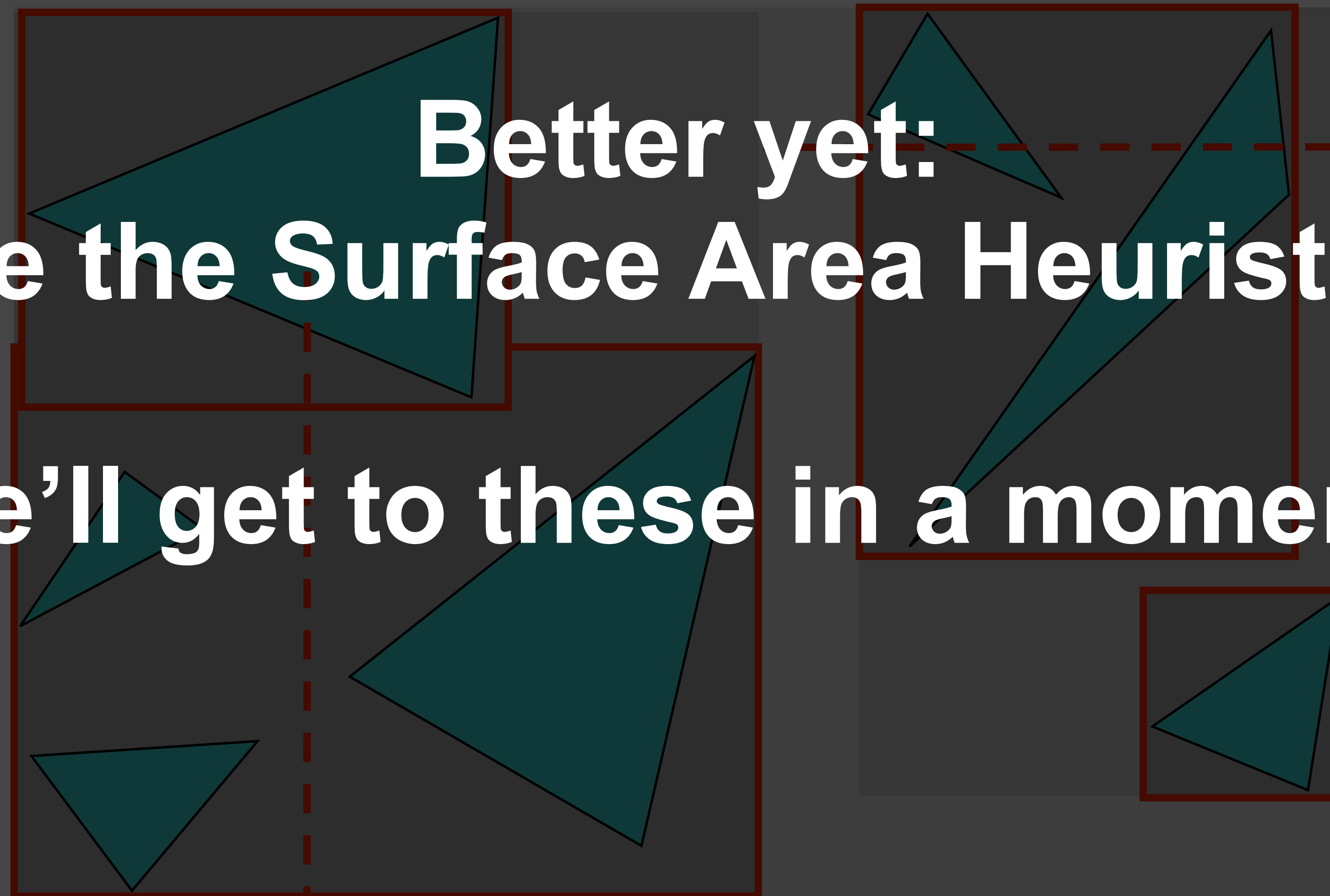
**The Path to Path-Traced Movies**

Per H. Christensen
Pixar Animation Studios
per@pixar.com

Wojciech Jarosz
Dartmouth College
wojciech.k.jarosz@dartmouth.edu

# Where to Split Objects?

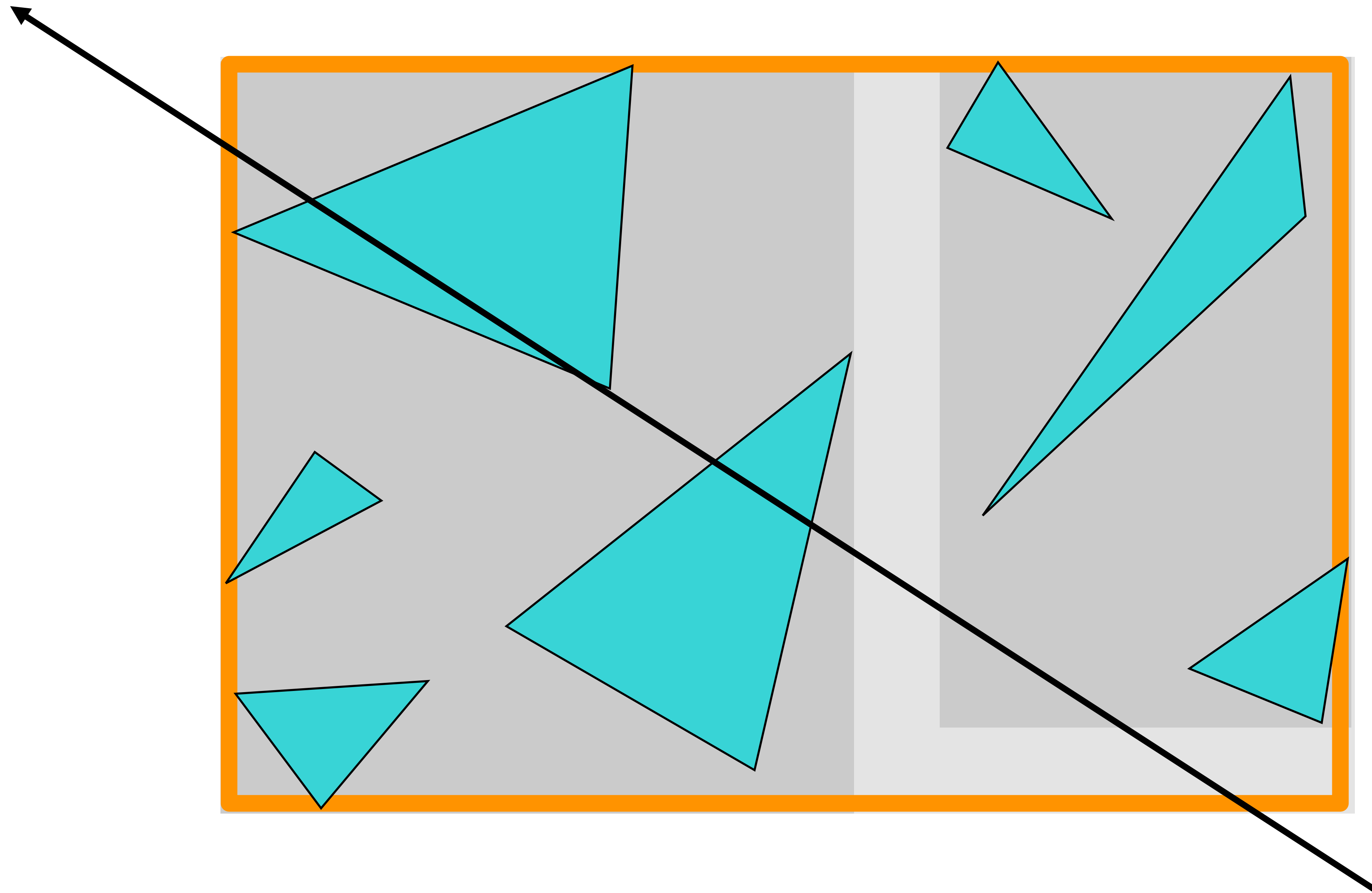
- At midpoint of current volume *OR*
- Sort, and put half of the objects on each side *OR*
- Use modeling hierarchy
  – Actually, don't. You're relying on the artist for speed.



**Better yet:**
**optimize the Surface Area Heuristic (SAH)**

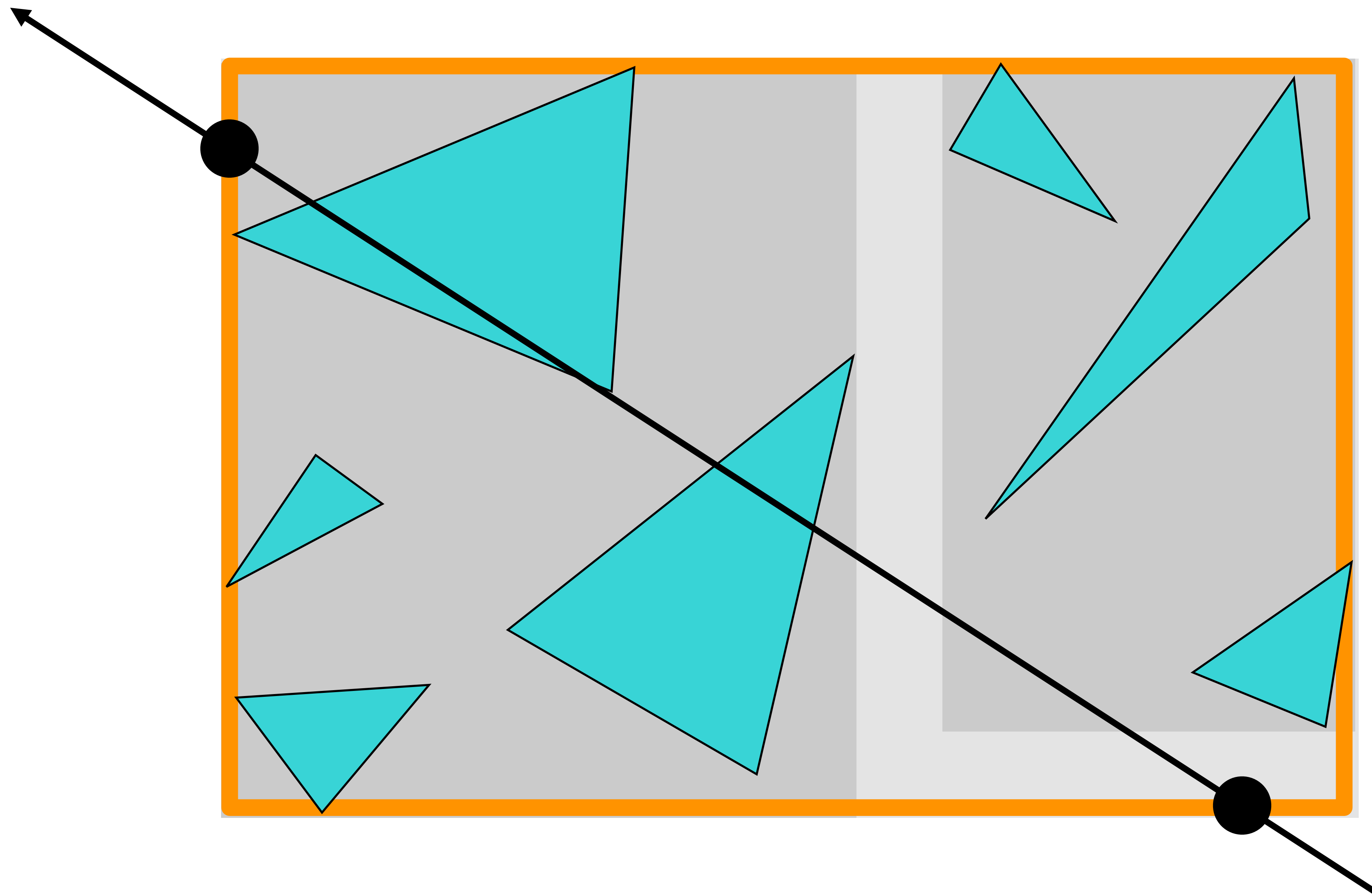
**We'll get to these in a moment..**

# Ray-BVH Intersection



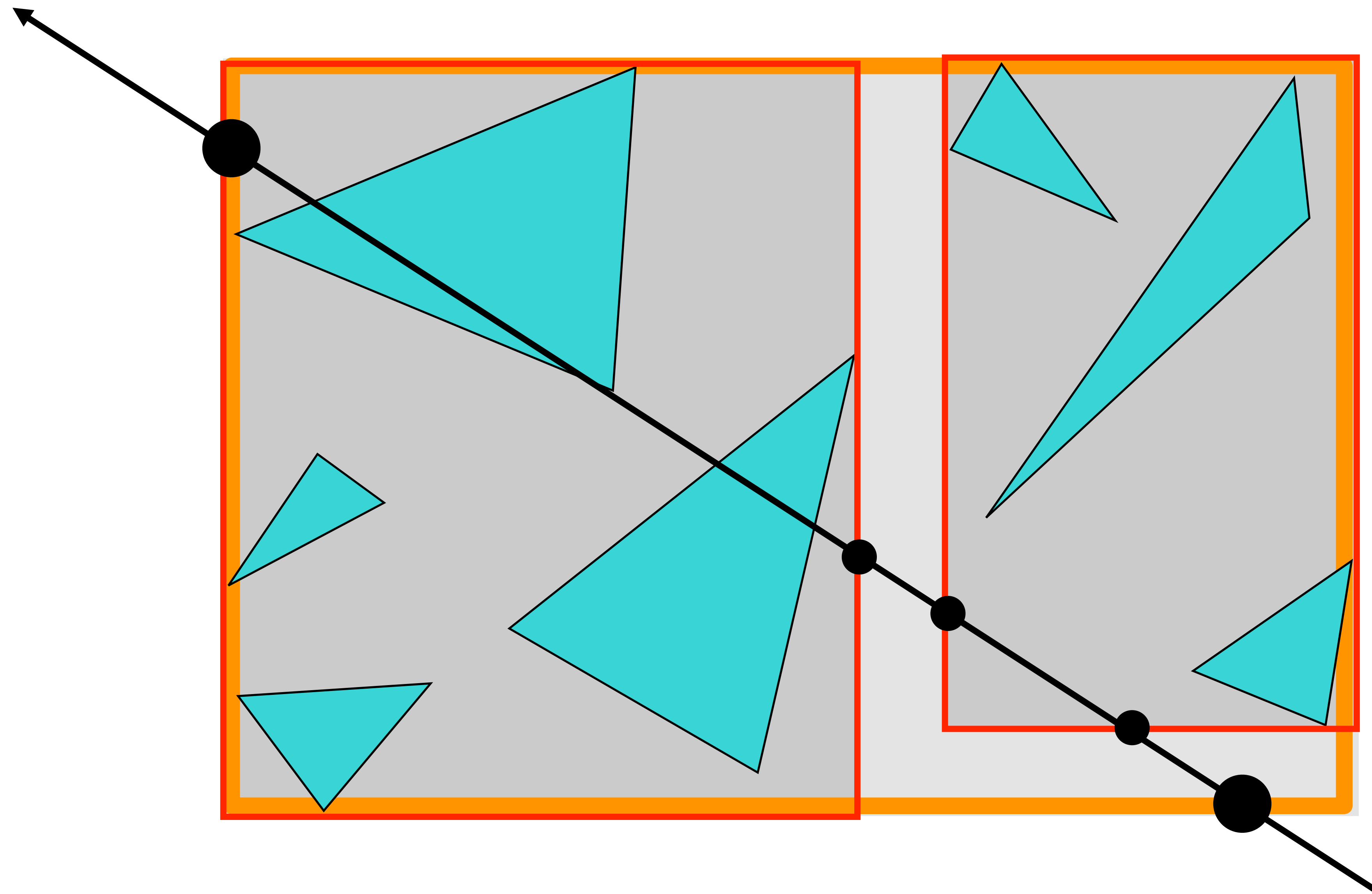
# Ray-BVH Intersection

- Find $t_{start}$ and $t_{end}$ for node
  - If no hit, return



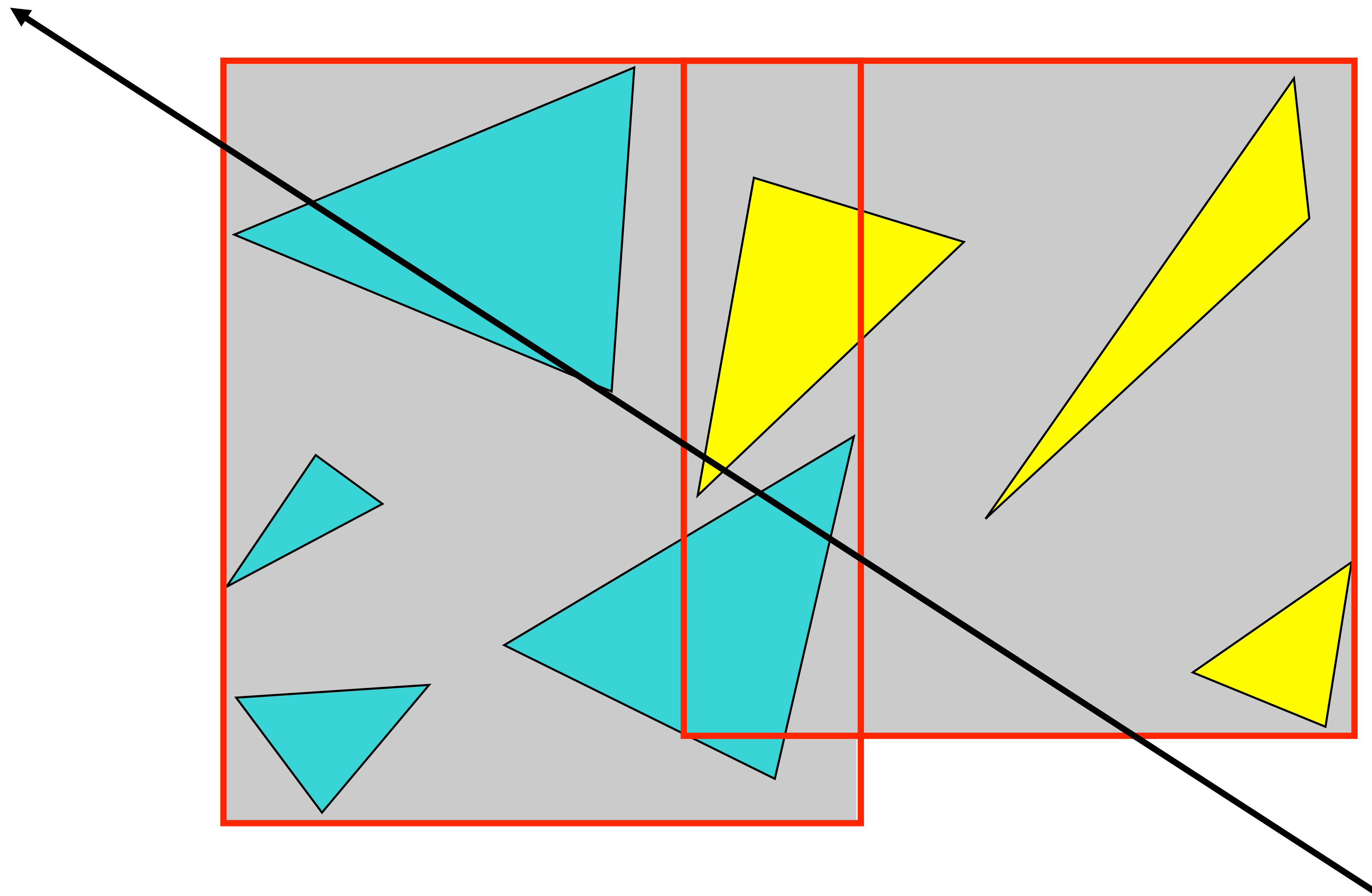
# Ray-BVH Intersection

- Compute $t_{start}$, $t_{end}$ for child nodes
  - Recursively check sub-volume with closer intersection first



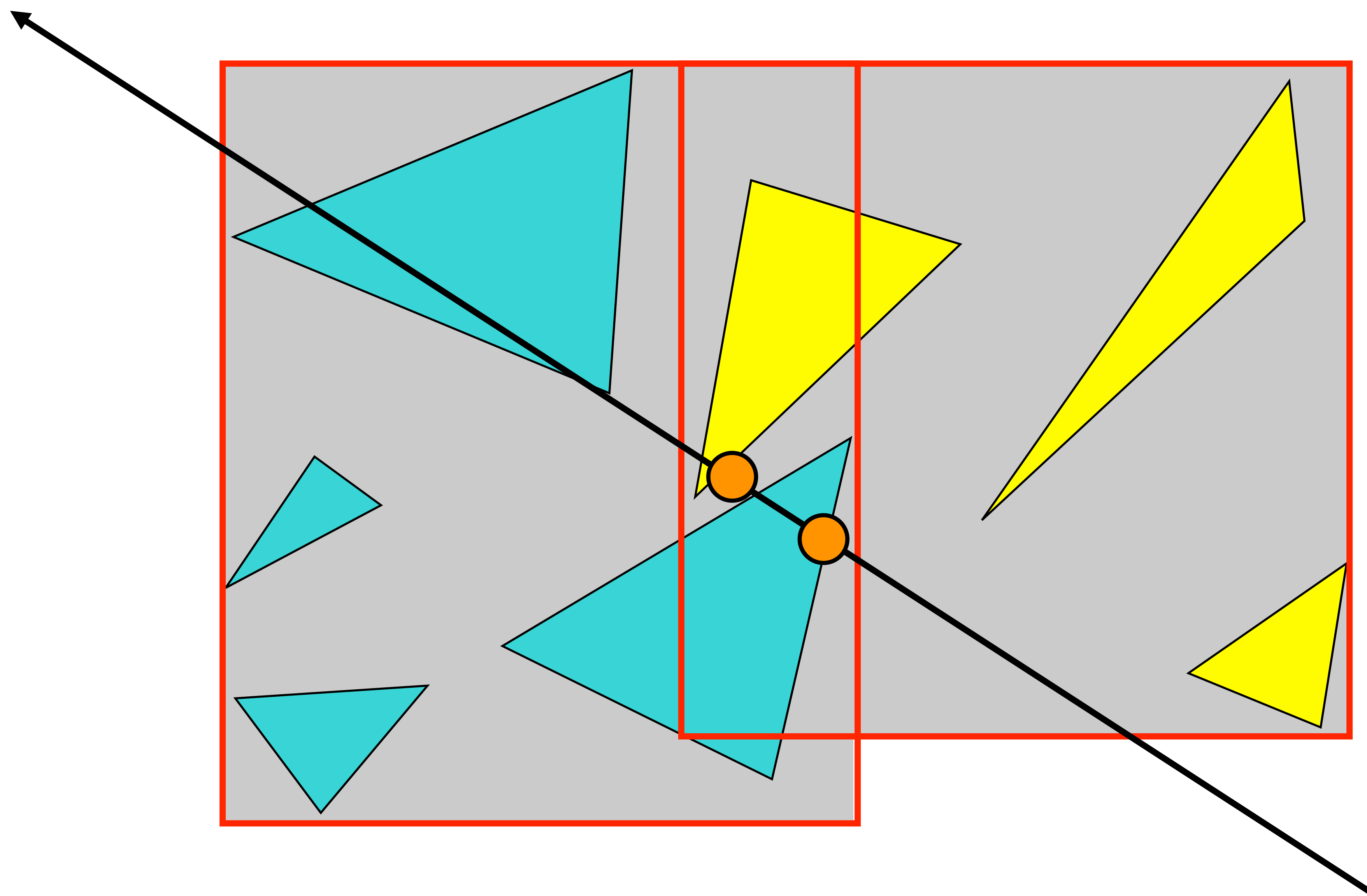
# Intersection with BVH

- Don't return intersection immediately if the other subvolume may have a closer intersection
  - Nodes can and *will* overlap!



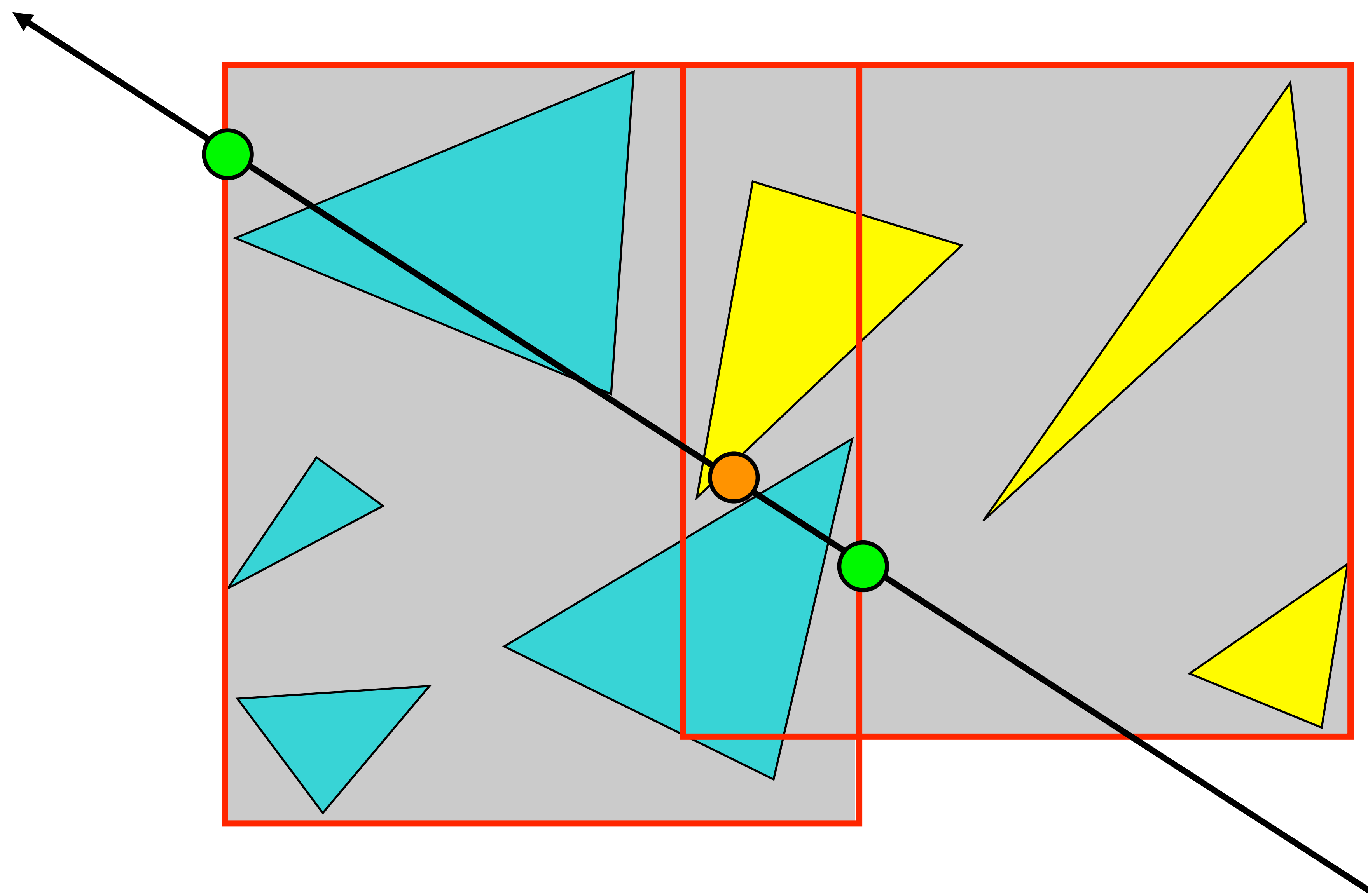
# Intersection with BVH

- Don't return intersection immediately if the other subvolume may have a closer intersection
  - Nodes can and *will* overlap!



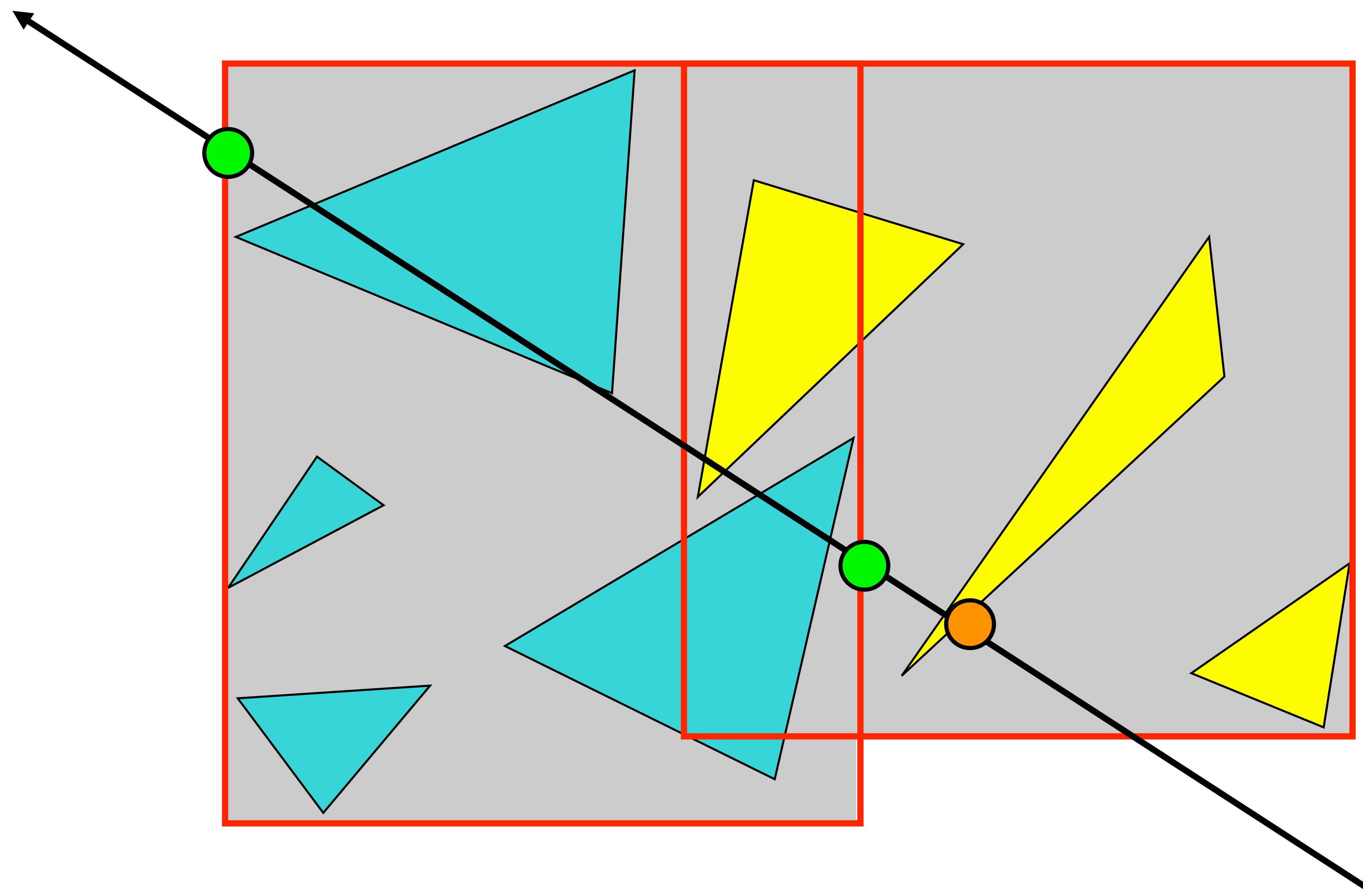
# Intersection with BVH

- Must also check the farther child node if closest hit so far (●) is inside its bounding volume



# Intersection with BVH

- Must also check the farther child node if closest hit so far (●) is inside its bounding volume



In this case checking not necessary!

# Questions?

[Lehtinen et al. SIGGRAPH 2012]

Input (8 spp)

Our reconstruction

Path tracing (512 spp)

**(It pays off to be smart, but not only in how fast your tracer is)**

# BVH Details

- In ray tracing, a BVH node contains
  - The node's bounding volume
    - e.g. axis-aligned box
  - Internal nodes: Pointers to left and right children
  - Leaf nodes: List of primitives inside the node

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
```

```
main()
{
  Node root = new Node();
  constructTree( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;
```

```
main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;

    if ( L.numElements() > MAX_TRIS_PER_LEAF )   // continue recursion?
    {











}

main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;

    if ( L.numElements() > MAX_TRIS_PER_LEAF )   // continue recursion?
    {
        // decide how to split primitives
        plane S = chooseSplit( L );









}

main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;

    if ( L.numElements() > MAX_TRIS_PER_LEAF )   // continue recursion?
    {
        // decide how to split primitives
        plane S = chooseSplit( L );
        // perform actual split: partition L into two disjoint sets
        list leftChild, rightChild;
        partitionPrimitives( L, S, leftChild, rightChild );









}

main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;

    if ( L.numElements() > MAX_TRIS_PER_LEAF )   // continue recursion?
    {
        // decide how to split primitives
        plane S = chooseSplit( L );
        // perform actual split: partition L into two disjoint sets
        list leftChild, rightChild;
        partitionPrimitives( L, S, leftChild, rightChild );
        // construct left child node, recurse
        N.leftChild = new Node();
        constructTree( leftChild, N.leftChild );




}

main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

# Constructing a BVH, Pseudocode

```
function constructTree( list L, node N )
{
    // must know node size, be it internal or leaf...
    N.boundingBox = computeBB( L );
    N.leftChild = N.rightChild = NULL;

    if ( L.numElements() > MAX_TRIS_PER_LEAF )   // continue recursion?
    {
        // decide how to split primitives
        plane S = chooseSplit( L );
        // perform actual split: partition L into two disjoint sets
        list leftChild, rightChild;
        partitionPrimitives( L, S, leftChild, rightChild );
        // construct left child node, recurse
        N.leftChild = new Node();
        constructTree( leftChild, N.leftChild );
        // construct left child node, recurse
        N.rightChild = new Node();
        constructTree( rightChildList, N.rightChild );
    }
    else N.primitives = L;  // no: store which primitives are in this leaf
}

main()
{
  Node root = new Node();
  construct( allPrimitives, root );    // recursively builds entire tree
}
```

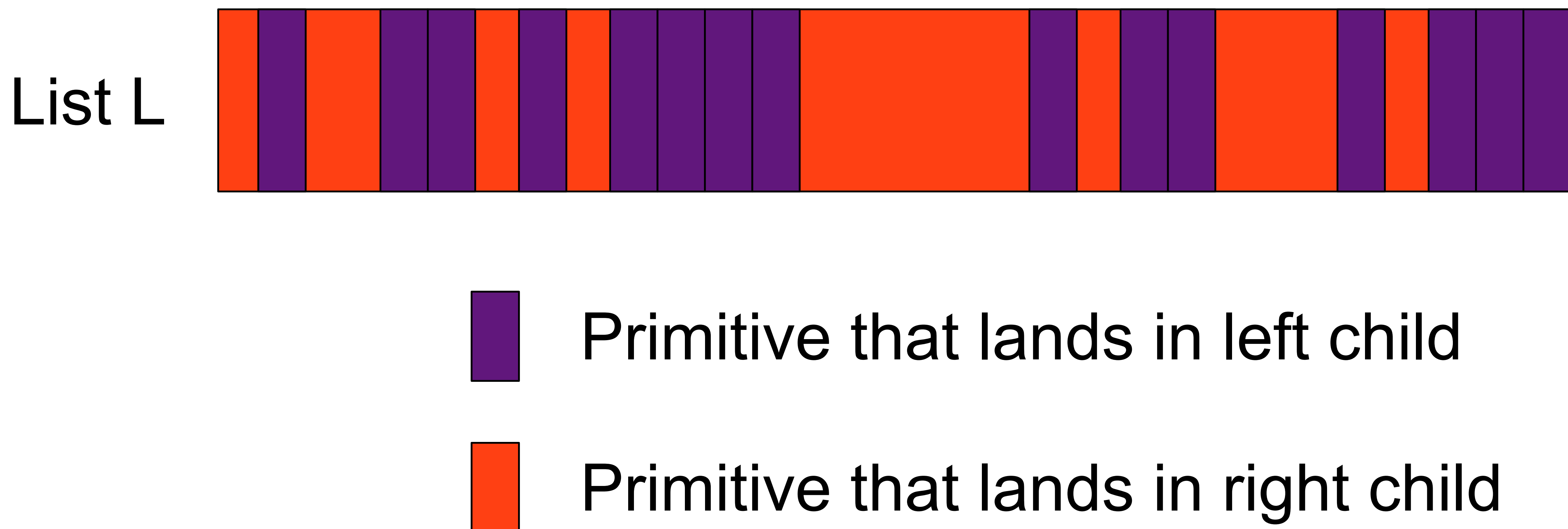
# Neat Implementation Trick

- In a BVH, the split always divides the primitives (triangles) to two *non-overlapping* sets
  - The bounding boxes of the child nodes will most often overlap *in space*, but each *primitive* always lands in exactly one of the children

- In contrast, *space-partitioning trees*, such as kD-trees or BSP trees, have non-overlapping spatial nodes, but primitives that straddle the split plane land in both children

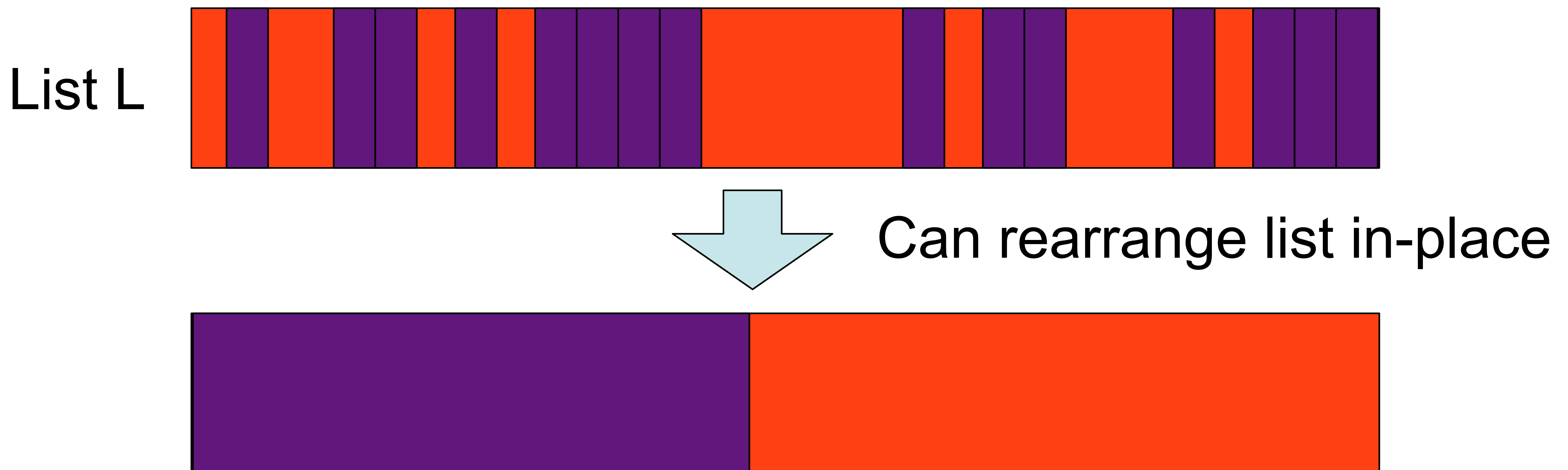
# Neat Implementation Trick

- In a BVH, the split always divides the primitives (triangles) to two *non-overlapping* sets
  - The bounding boxes of the child nodes will most often overlap *in space*, but each *primitive* always lands in exactly one of the children



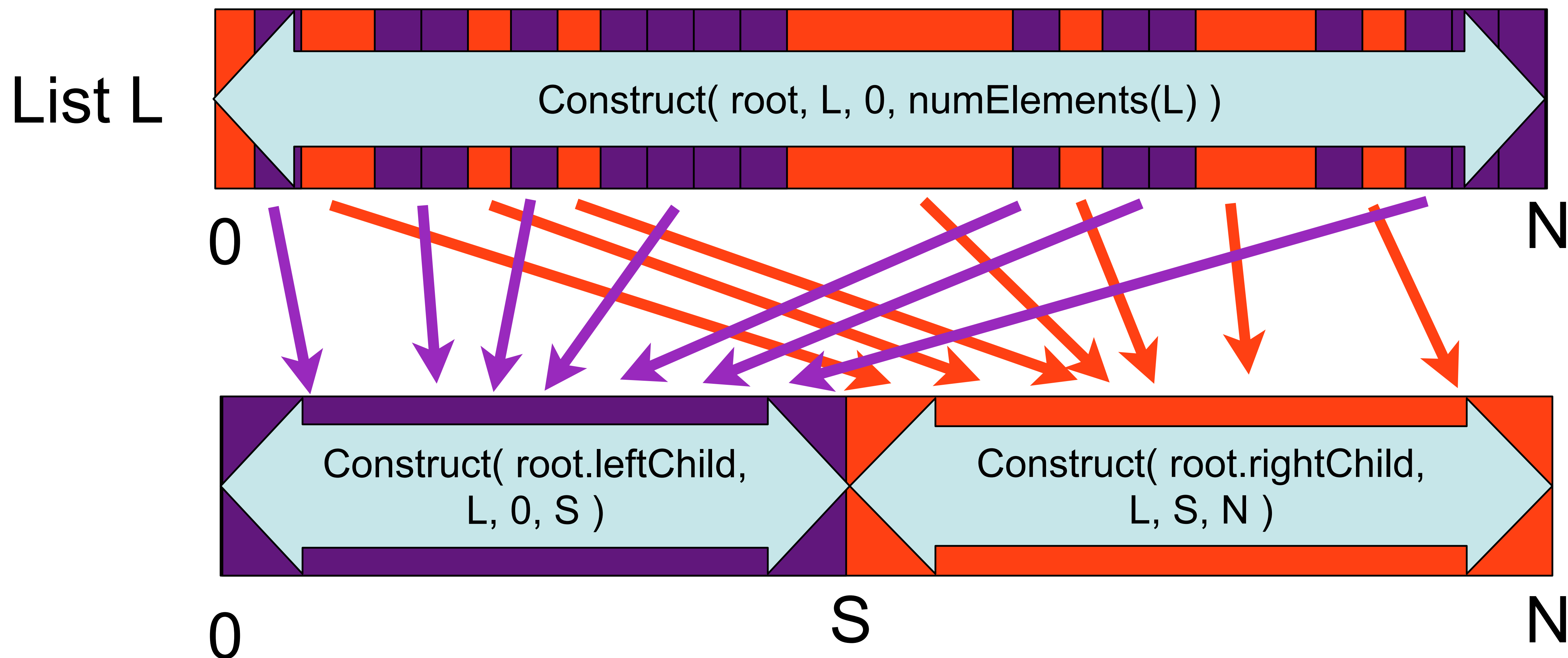
# Neat Implementation Trick

- In a BVH, the split always divides the primitives (triangles) to two *non-overlapping* sets
  - The bounding boxes of the child nodes will most often overlap *in space*, but each *primitive* always lands in exactly one of the children



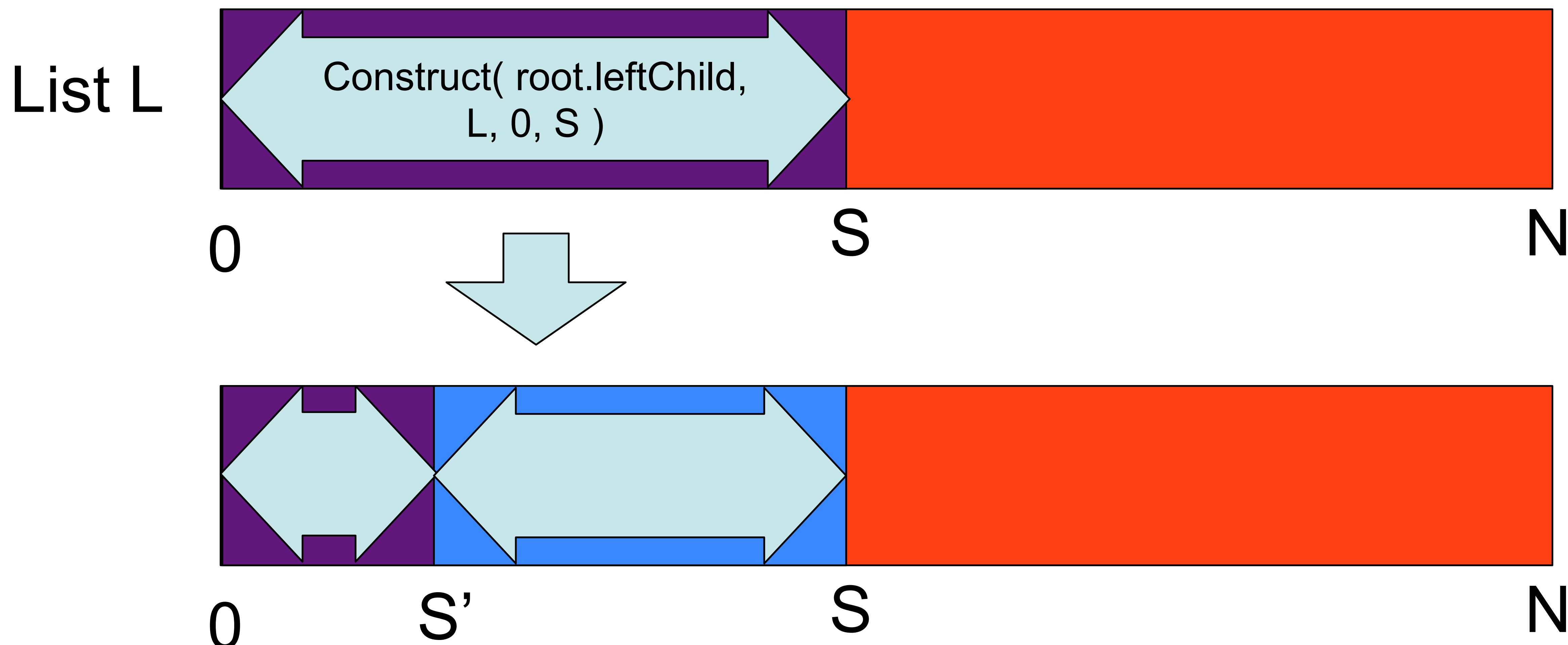
# Neat Implementation Trick

- Each nodes doesn't actually hold its entire own list of primitives
  - It suffices to mark the start and end indices in a global index list



# Neat Implementation Trick

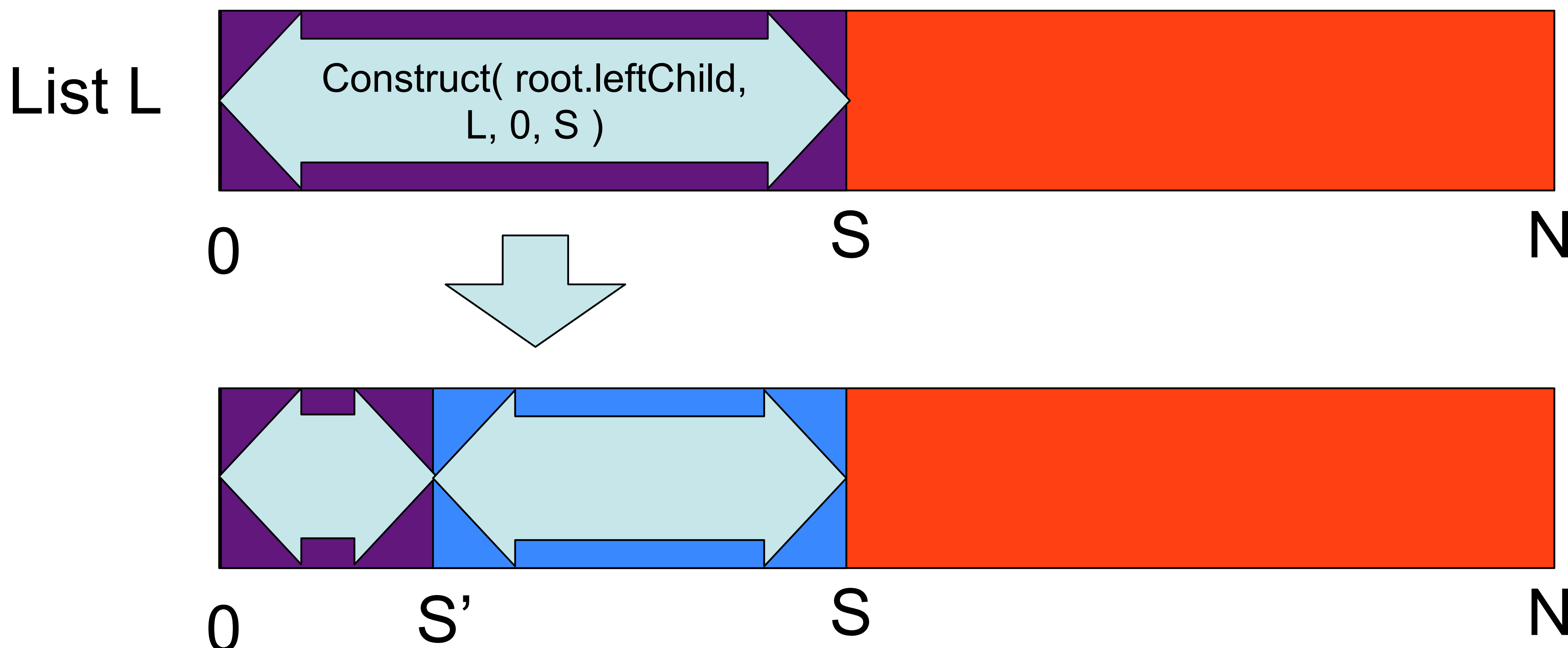
- Each nodes doesn’t actually hold its entire own list of primitives
  - It suffices to mark the start and end indices in a global index list



# Neat Implementation Trick

```
struct Node
{
    Vec3f bbMin, bbMax;          // axis-aligned BB
    int   startPrim, endPrim;    // these are indices in the global list
    Node* pLeft, pRight;         // these are NULL if node is leaf
};
```

(Lots of room for optimizations!)



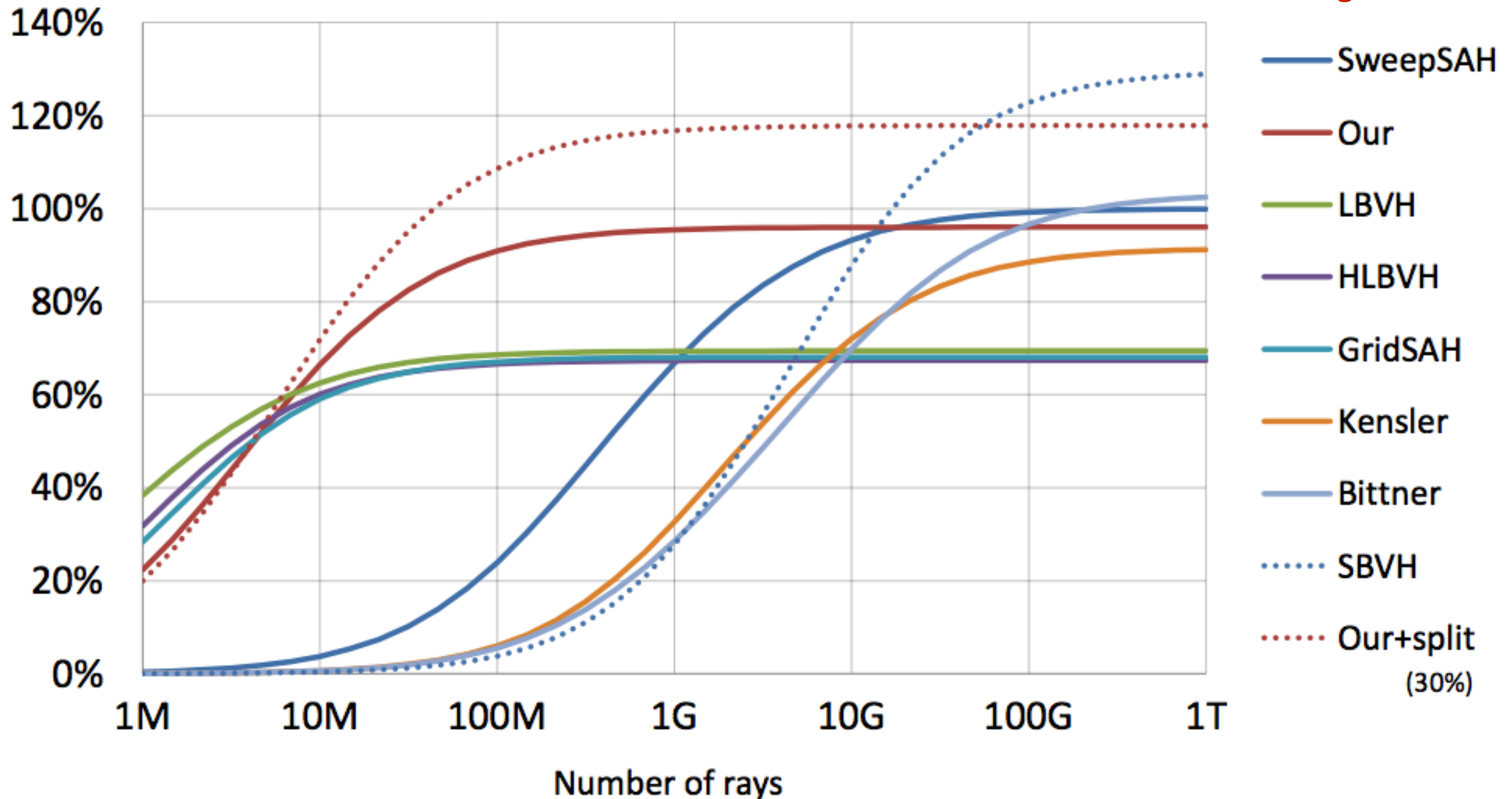
# In practice

- You can either
  - Keep one global list of triangles in their original order, and maintain an index list during traversal
    - **Recommended**
    - Index list initialized to {0, 1, 2, ..., n} before calling construct for root
    - Index list is shuffled in-place by the tree construction code
    - Final index list gives the nodes' as indices to the global triangle list
  - You can also just shuffle the global list of triangles without the indirection
  - **Not recommended** to use lists of pointers to triangles, will make save/load more complicated
    - Pointer references an actual memory location, will vary between runs!

# Is it Important to Optimize Splits?

- Given the same traversal code, the quality of the tree may 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
  - And constructing better trees is slower!
  - Could you optimize something else for bigger gain?

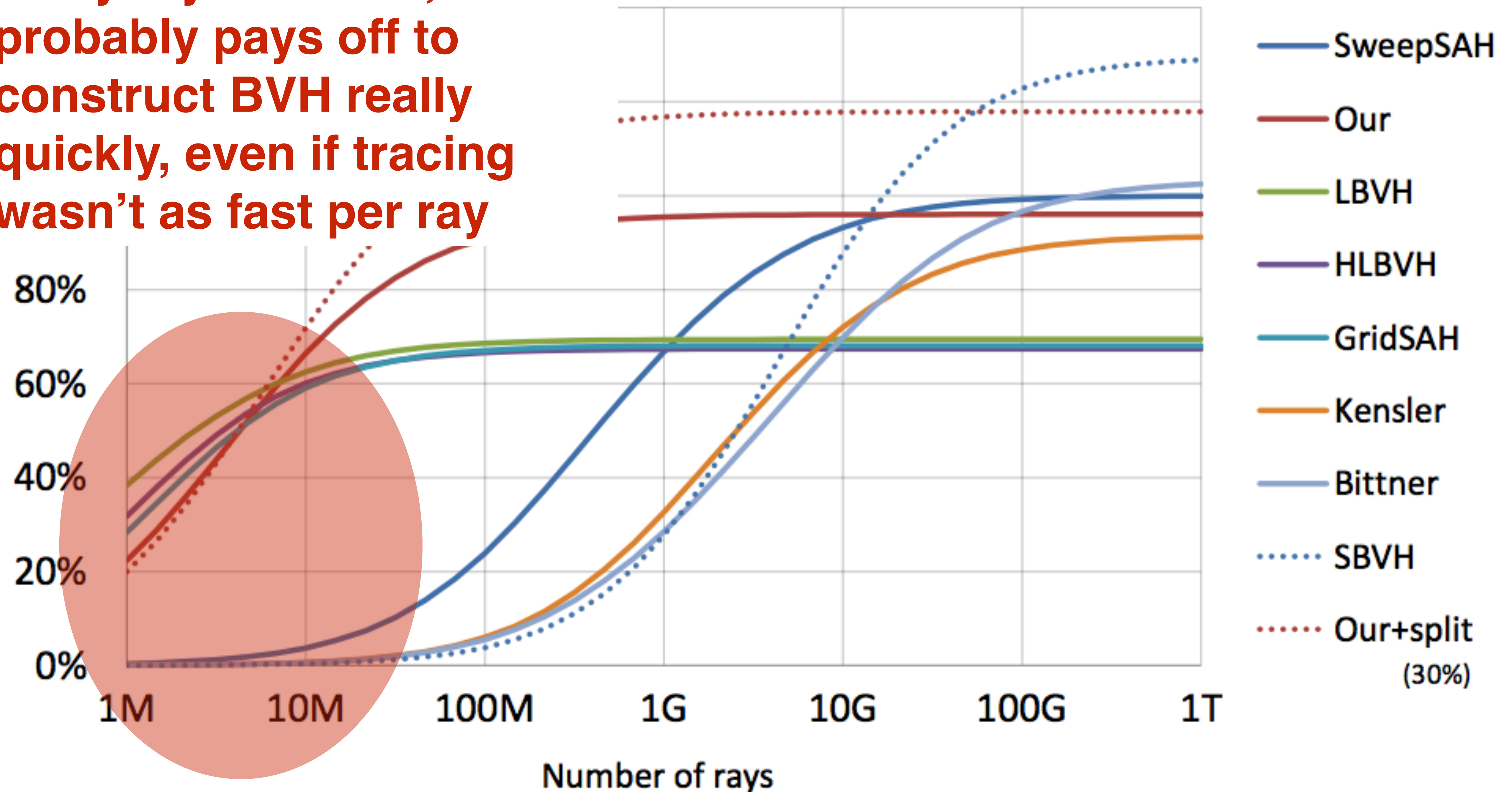
# Karras, Aila HPG 2013 (underline means link!)



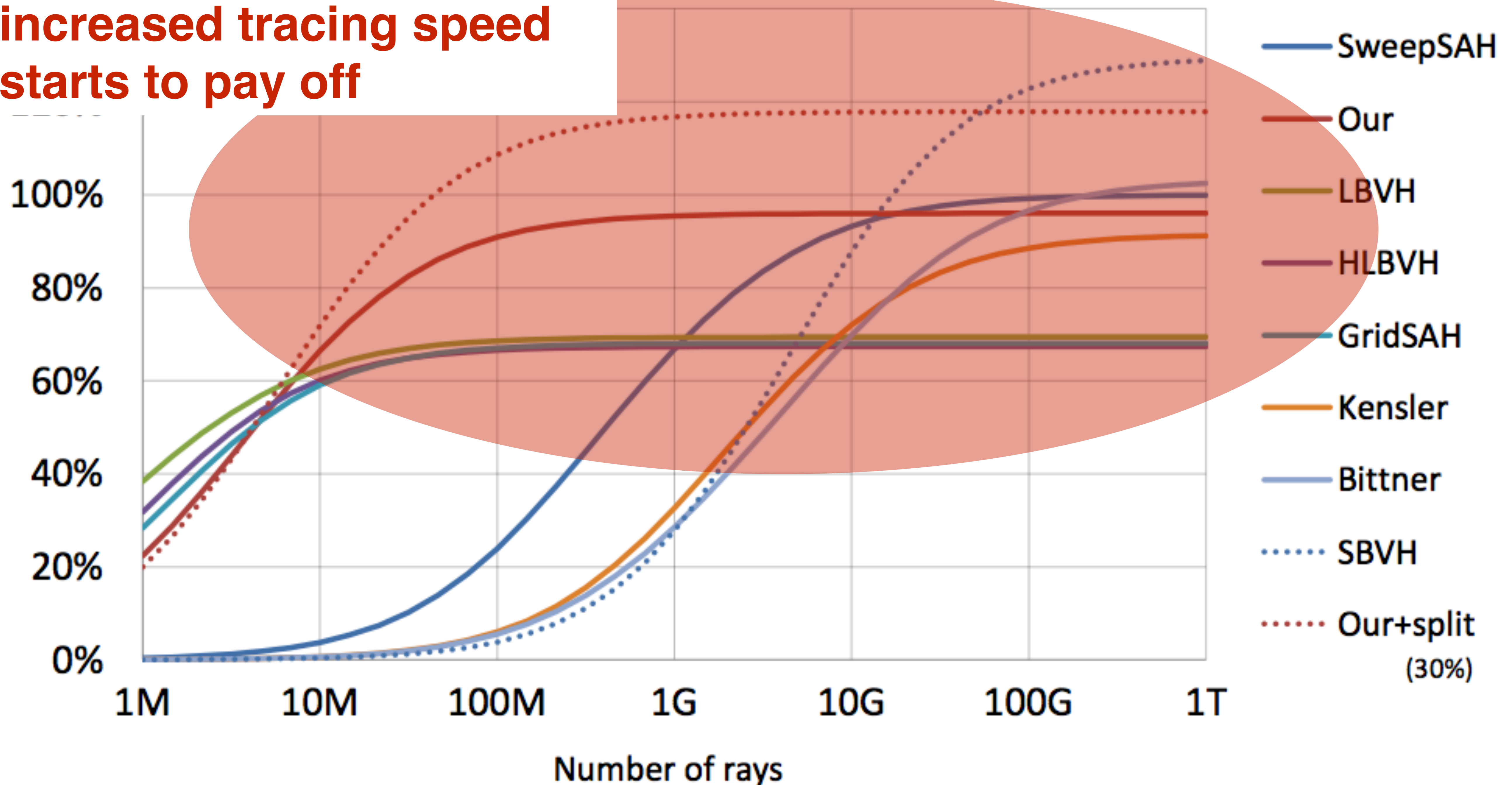
# Karras, Aila HPG 2013 (underline means link!)

**Efficiency measured as a function of TOTAL WALLCLOCK TIME PER RAY, taking into account both BVH construction and actual tracing.**

**If you don’t have too many rays to trace, it probably pays off to construct BVH really quickly, even if tracing wasn’t as fast per ray**

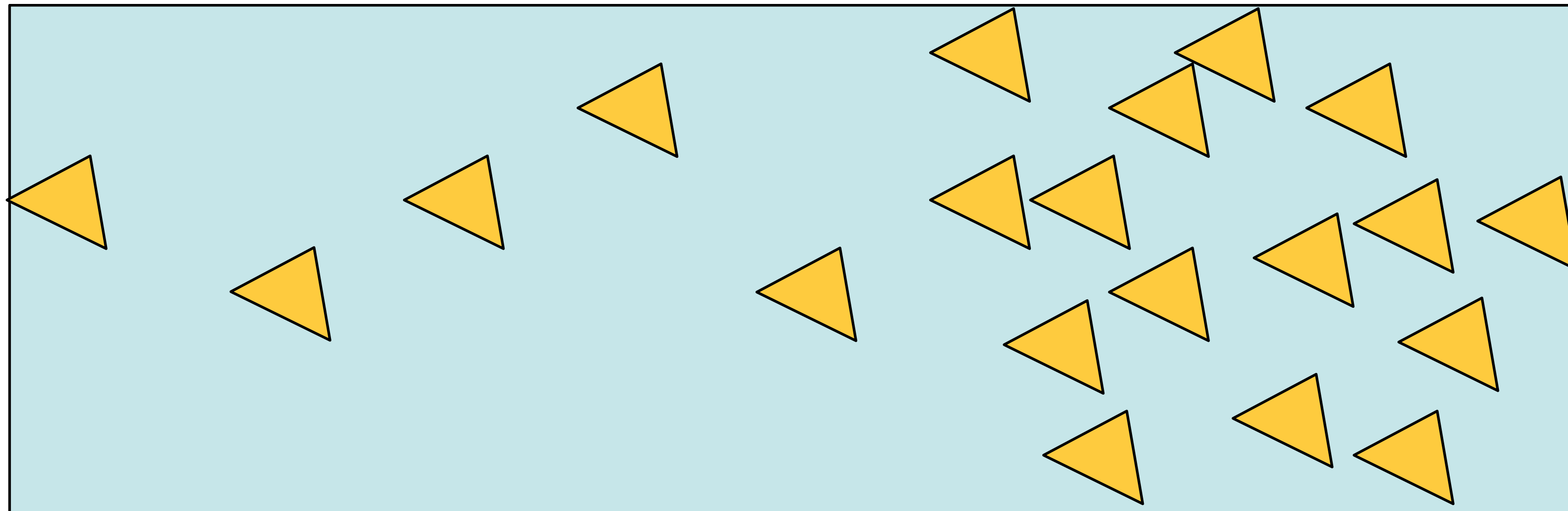


# Karras, Aila HPG 2013 (underline means link!)



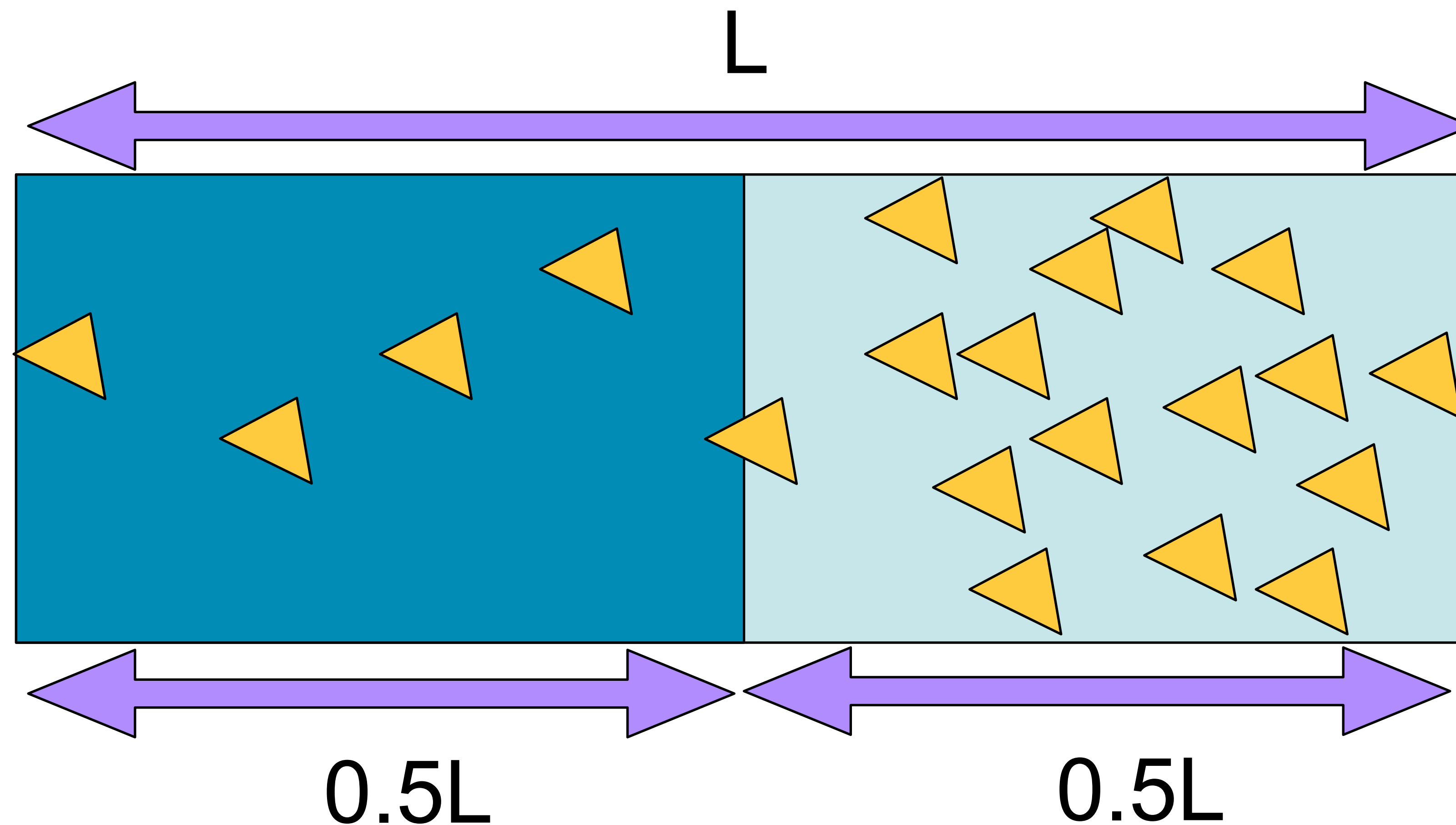
# Ways of Choosing Splits

- Spatial median
  - Pick the longest axis (x/y/z) of current node's bounding box
  - Distribute primitives to positive/negative based on which side the primitive's centroid lies



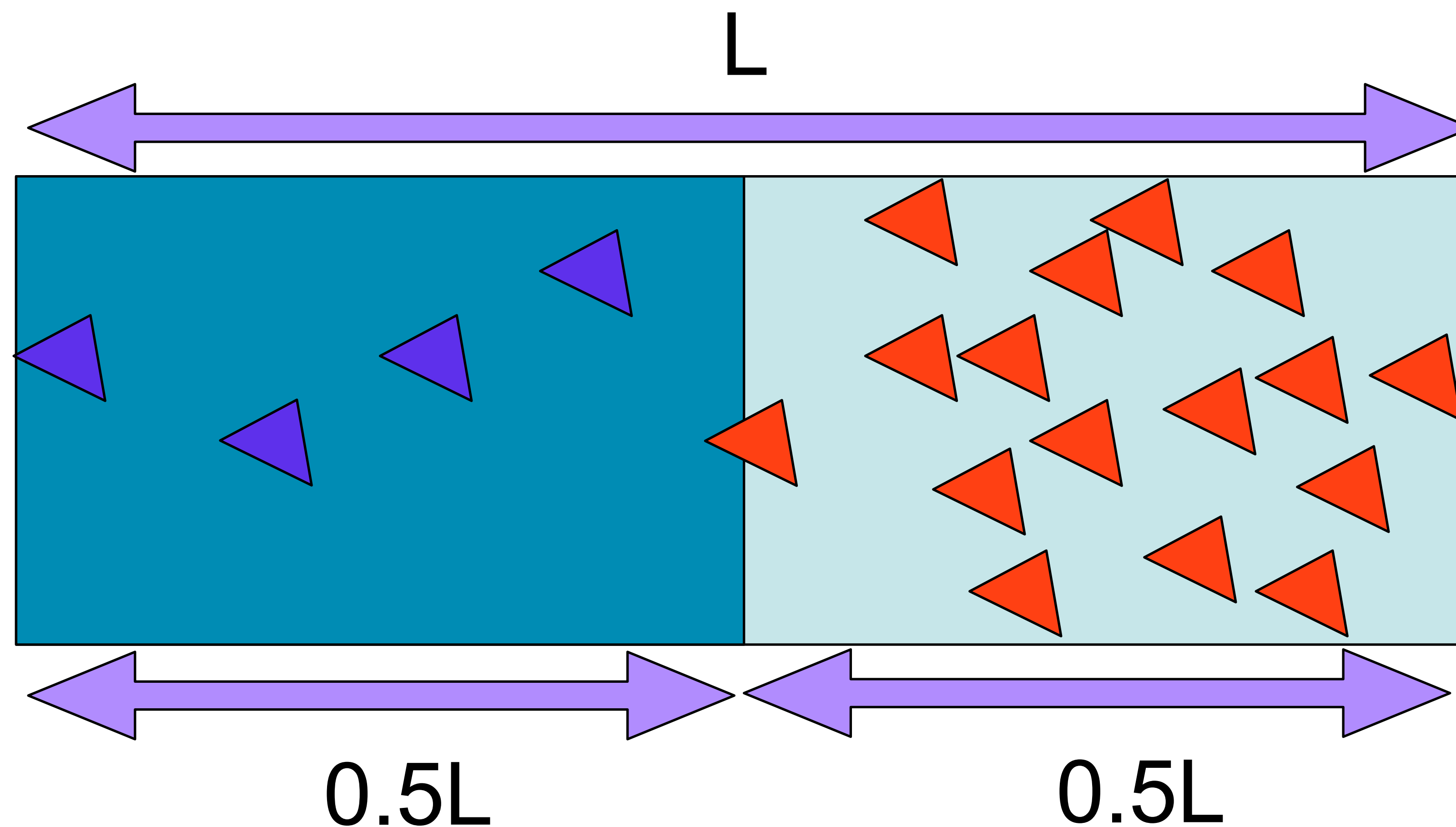
# Ways of Choosing Splits

- Spatial median
  - Pick the longest axis (x/y/z) of current node's bounding box
  - Distribute primitives to positive/negative based on which side the primitive's centroid lies



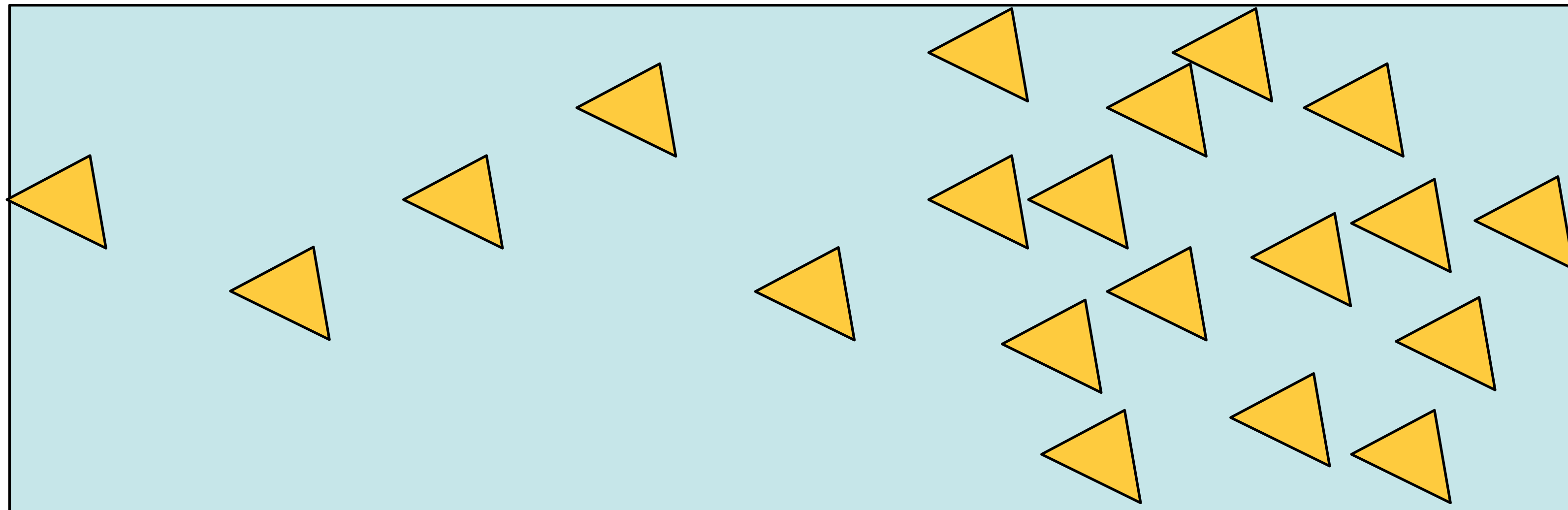
# Ways of Choosing Splits

- Spatial median
  - Pick the longest axis (x/y/z) of current node's bounding box
  - Distribute primitives to positive/negative based on which side the primitive's centroid lies



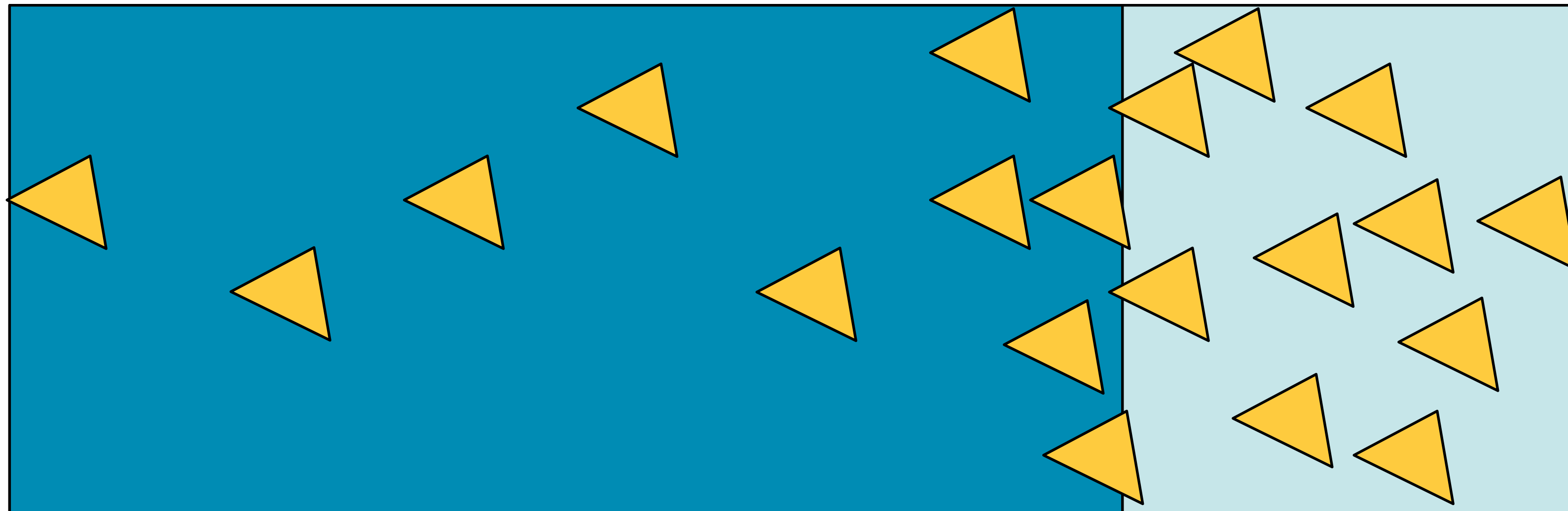
# Ways of Choosing Splits

- Object median
  - Pick an axis (you can try them all)
  - Sort primitives along the axis
  - Assign half of the primitives to left, half to right



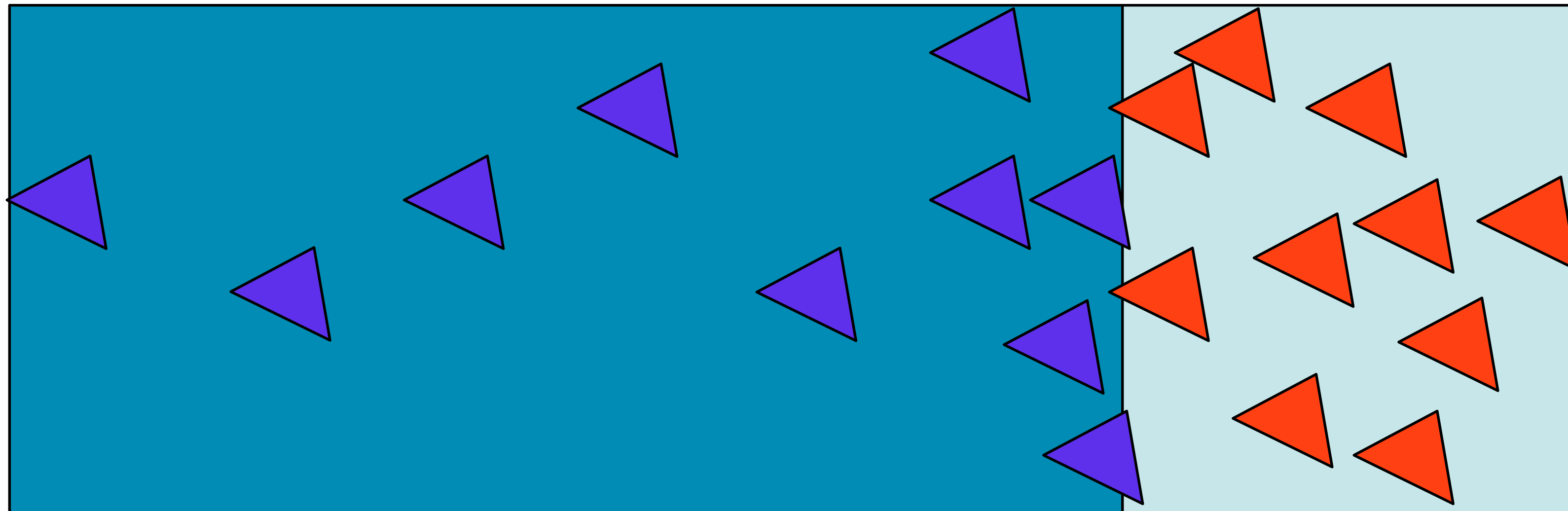
# Ways of Choosing Splits

- Object median
  - Pick an axis (you can try them all)
  - Sort primitives along the axis
  - Assign half of the primitives to left, half to right



# Ways of Choosing Splits

- Object median
  - Pick an axis (you can try them all)
  - Sort primitives along the axis
  - Assign half of the primitives to left, half to right



**Which one is better...?**

# Traversal Cost

- Spatial and object median are heuristics to prune efficiently either
  - space (spatial median)
    - The volumes of the nodes shrink as fast as possible
  - tree depth (object median)
    - The number of primitives in each node shrinks as fast as possible
    - *Builds a balanced tree. That's a good thing – right…?*

- Do these actually determine how fast the tree is able to answer ray queries?
  - What determines ray tracing cost?

# Surface Area Heuristic (SAH)

- MacDonald, Booth 1990: Heuristics for ray tracing using space subdivision, Visual Computer (6) 153-166.
- Rather simple, but surprisingly effective heuristic to choose split planes. Main ideas:
  1. Assume rays are uniformly distributed in space.
  2. Then, the probability of a ray hitting a BVH node is directly proportional to the node's surface area.
  3. The *cost* of traversing a node is proportional to the number of primitives (triangles) it contains, plus some fixed constant, plus the cost of the subtree rooted at the node
  4. Ergo: a good split is one that minimizes the *expected cost* (Note! Greedy algorithm. Global optimum over entire tree is unattainable.)

# SAH cont'd

- MacDonald and Booth suggest choosing split planes by minimizing

$$f(b) = LSA(b) \cdot L(b) + RSA(b) \cdot (n - L(b))$$

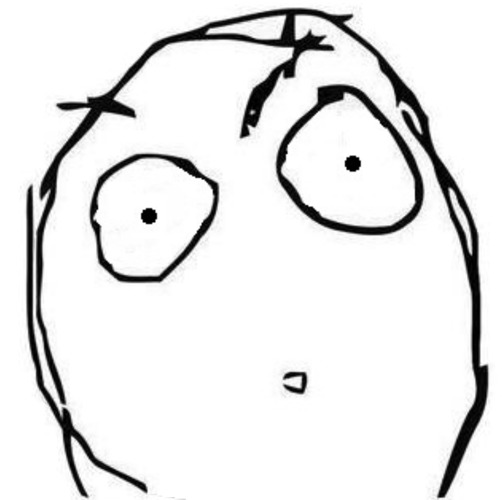
- where $b$ is the position of the split along an axis, normalized to [0,1] along the extents of the node,
- LSA/RSA is the surface area of the resulting left/right child nodes,
- L($b$) is the number of primitives in the left child, and
- $n$ is the number of primitives in the node being split.

# SAH Notes

- It's a top-down greedy heuristic. Does not guarantee any sort of global optimality.
- However, works surprisingly well in practice
  - *The research community was long confused as to why ;)*
- Can also be done bottom-up, but harder
  - Walter et al. 2008, Fast agglomerative clustering for rendering, Proc. IRT 2008
- **Strongly** suggested for extra credit in your first assignment
  - Try object median, spatial median, SAH; which one gives you the fastest tracer? The more test cases you have and the fancier you describe results, the more points!

# “The research community was long confused as to why ;)”

- Not true any more!
- Timo Aila, Tero Karras & Samuli Laine (NVResearch) finally shed some light on this in 2013 “On Quality Metrics of Bounding Volume Hierarchies”
  - See also slides from Timo’s homepage
  - Highly recommended reading!
  - *Turns out the above doesn’t actually optimize what it promises to optimize..*



# Questions?

# BVH Discussion

- Advantages
  - easy to construct
  - easy to traverse
  - binary tree (=simple structure)
- Disadvantages
  - Choosing splits requires care

- Current fastest tracers use BVHs
  - You should, too.
  - The other choice: kD-trees
    - Not going to cover here.

# Elephant in the Room

- Optimizing SAH makes sense in a static context where the frame is going to take long to render anyway

- What about dynamic scenes?
  - Greedy SAH way too slow
  - Research in BVH construction for dynamic scenes advancing rapidly these days

# Fast BVH Construction

- “**Recent” (~2009) Hot Idea**: use space-filling curves
  - Treat triangles as points (just take their centroid)
  - Sort them along a Morton curve (a kind of space-filling curve)
  - Morton sorting order implicitly corresponds to a hierarchy in a uniform spatial median split
  - Emit corresponding tree nodes
  - Done!
- ~State of the Art pushed by Finns
  - Tero Karras, 2012: Maximizing Parallelism in the Construction of BVHs, Octrees and *k*-d trees. Proc. High Performance Graphics 2012
  - (Tero’s magic happens in hierarchy emission)

# Absolute Top Perf

- Build bad tree fast using space-filling curves; then perform exhaustive local optimization on tree topology.
  - See Tero & Timo's paper (HPG 2013)

## Fast Parallel Construction of High-Quality Bounding Volume Hierarchies


Tero Karras Timo Aila

NVIDIA



**Abstract**

We propose a new massively parallel algorithm for constructing high-quality bounding volume hierarchies (BVHs) for ray tracing. The algorithm is based on modifying an existing BVH to improve its quality, and executes in linear time at a rate of almost 40M triangles/sec on NVIDIA GTX Titan. We also propose an improved approach for parallel splitting of triangles prior to tree construction. Averaged over 20 test scenes, the resulting trees offer over 90% of the ray tracing performance of the best offline construction method (SBVH), while previous fast GPU algorithms offer only about 50%. Compared to state-of-the-art, our method offers a significant improvement in the majority of practical workloads that need to construct the BVH for each frame. On the average, it gives the best overall performance when tracing between 7 million and 60 billion rays per frame. This covers most interactive applications, product and architectural design, and even movie rendering.



**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

**Keywords:** ray tracing, bounding volume hierarchies

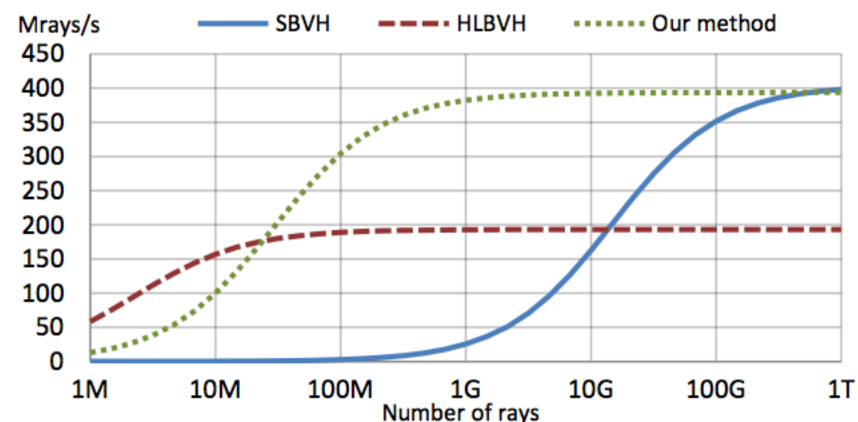
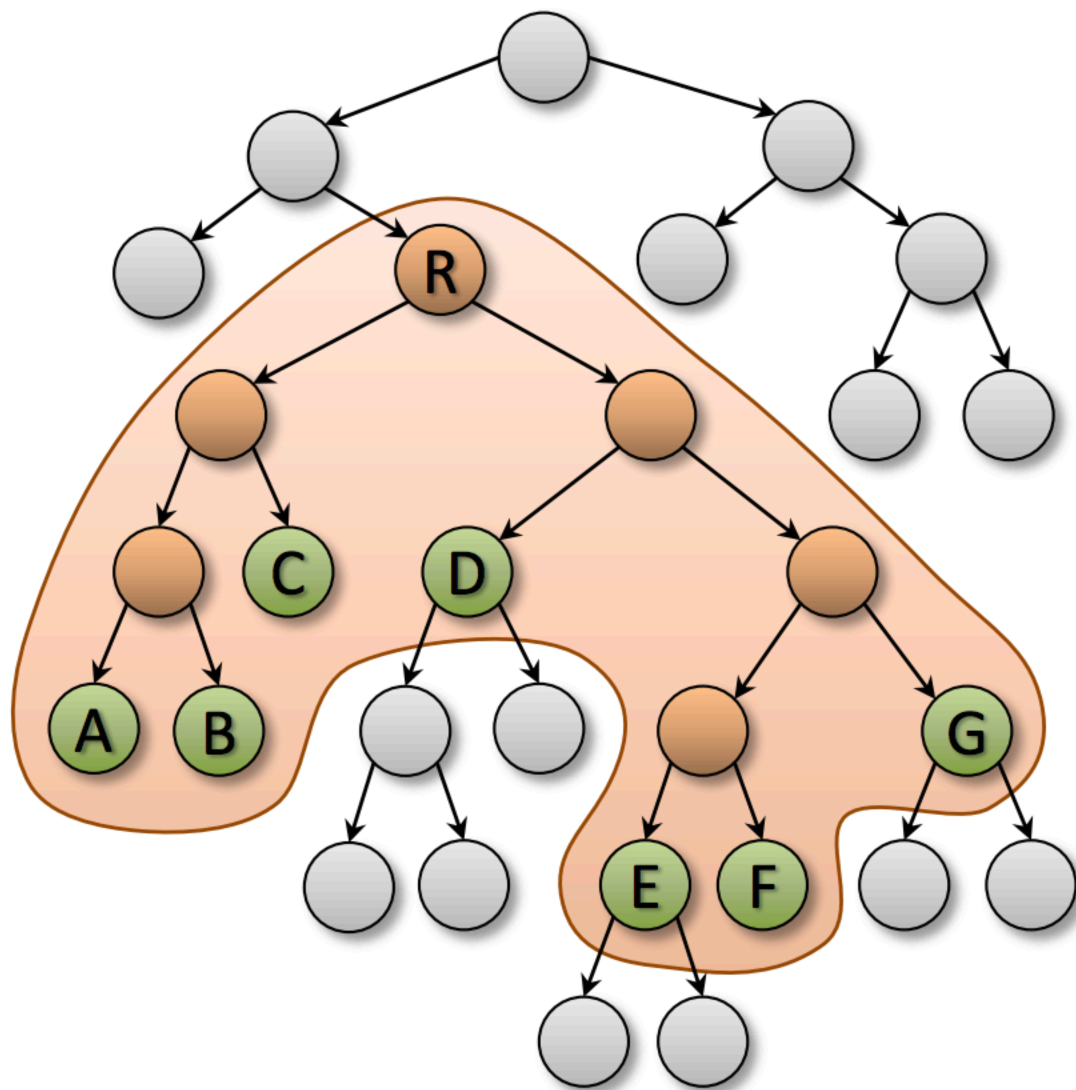
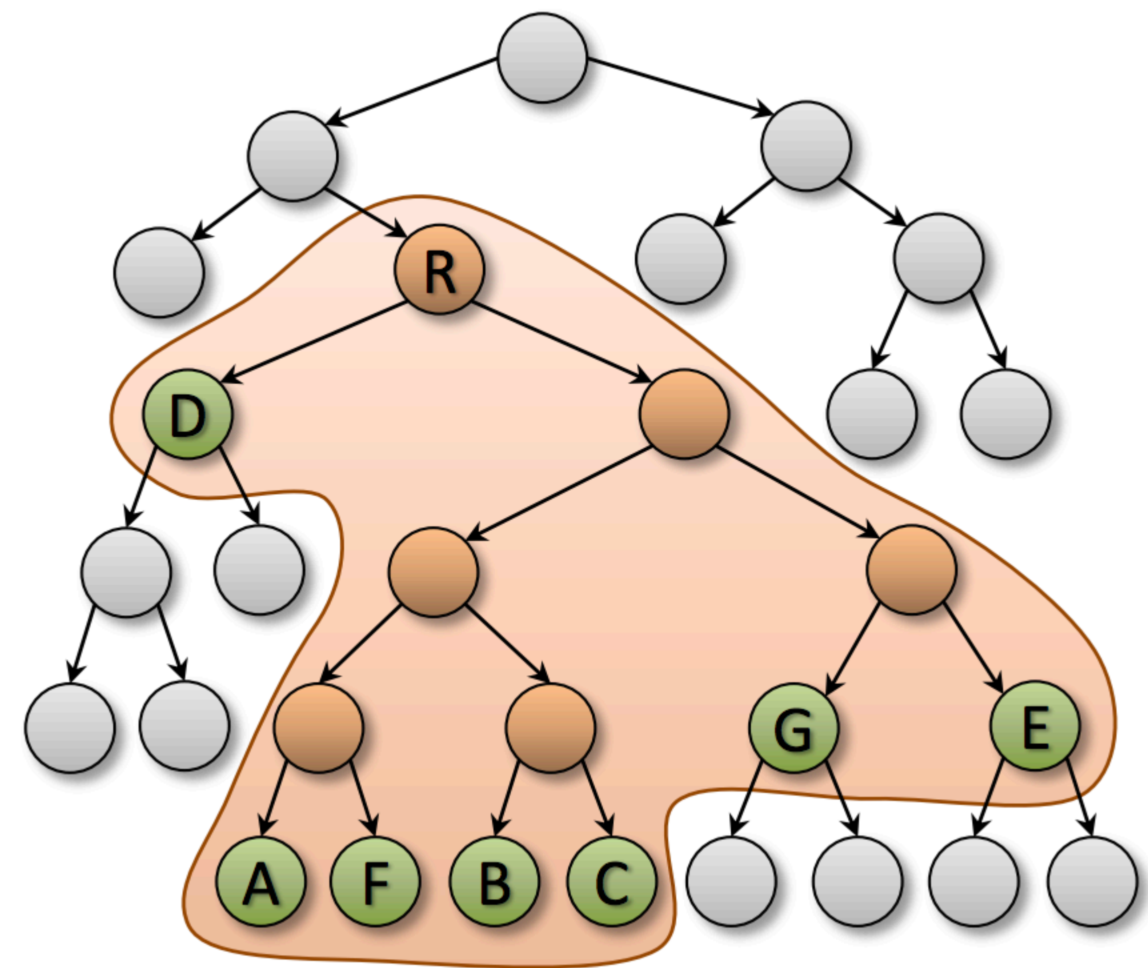




**Figure 1:** *Performance of constructing a BVH and then casting a number of diffuse rays with NVIDIA GTX Titan in SODA (2.2M triangles). SBVH [Stich et al. 2009] yields excellent ray tracing performance, but suffers from long construction times. HLBVH [Garanzha et al. 2011a] is very fast to construct, but reaches only about 50% of the performance of SBVH. Our method is able to reach 97% while still being fast enough to use in interactive applications. In this particular scene, it offers the best quality–speed tradeoff for workloads ranging from 30M to 500G rays per frame.*

# Main Idea

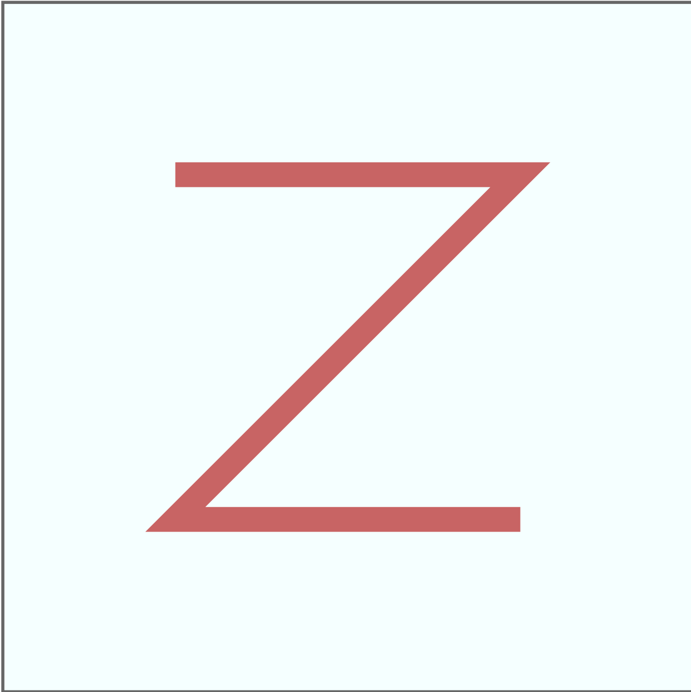
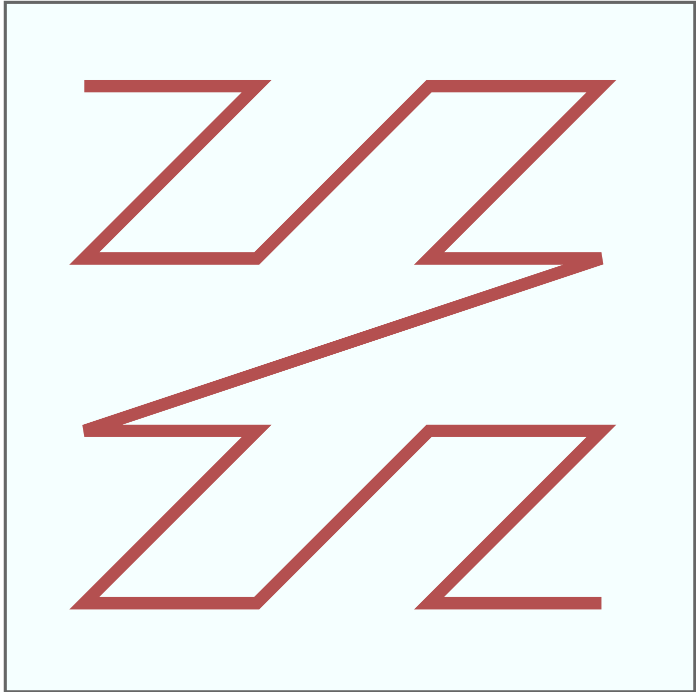
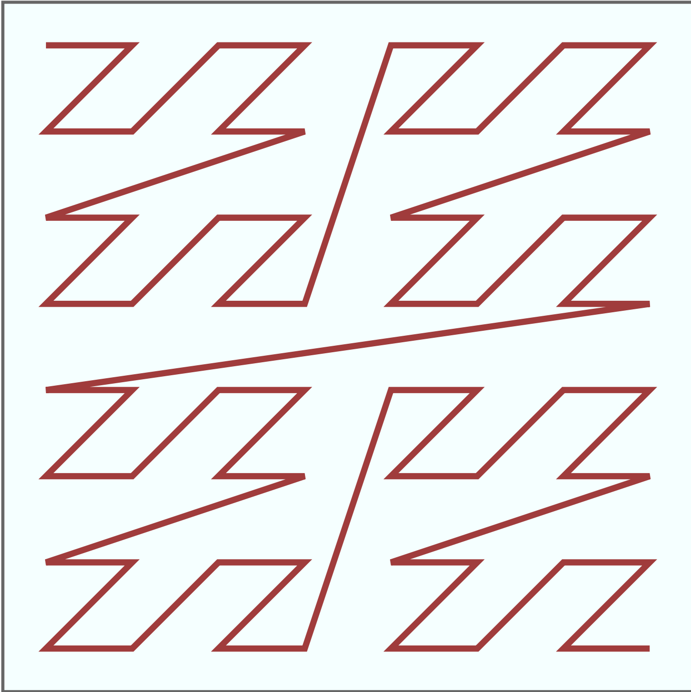
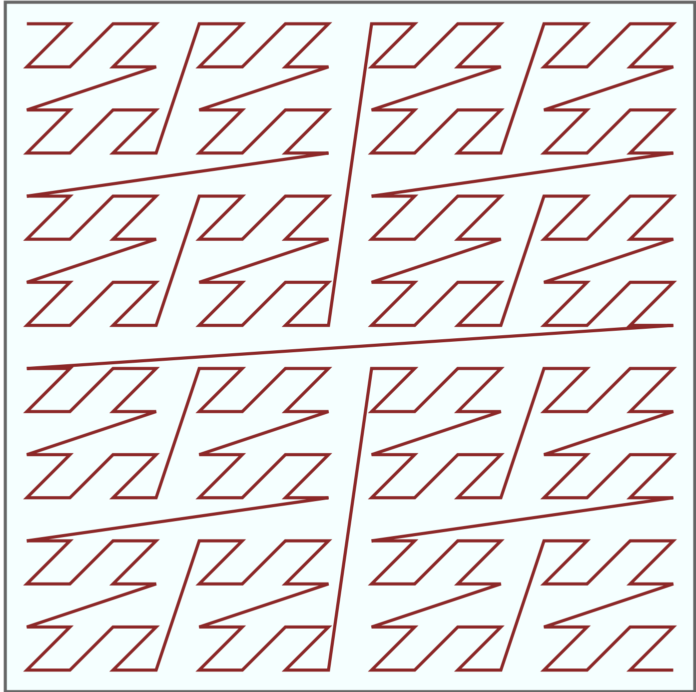
- First build bad tree using space-filling curves, then locally reorganise subtrees *optimally*



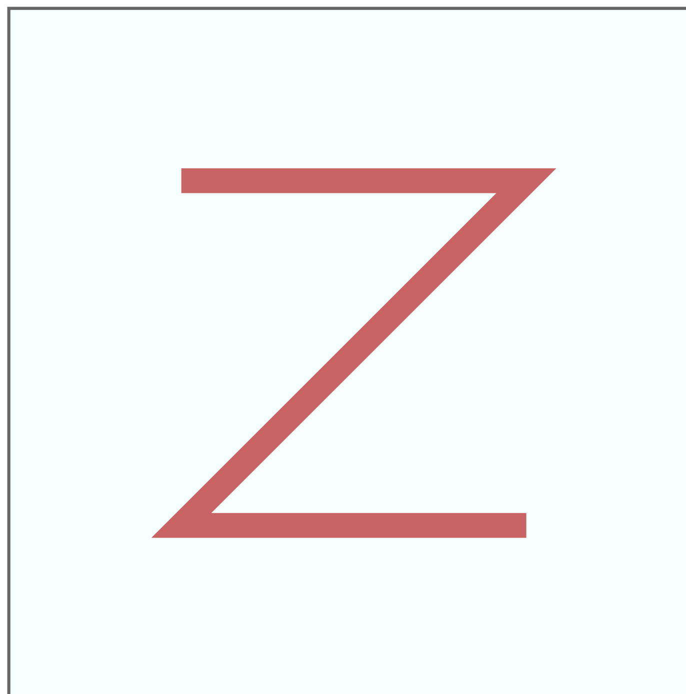
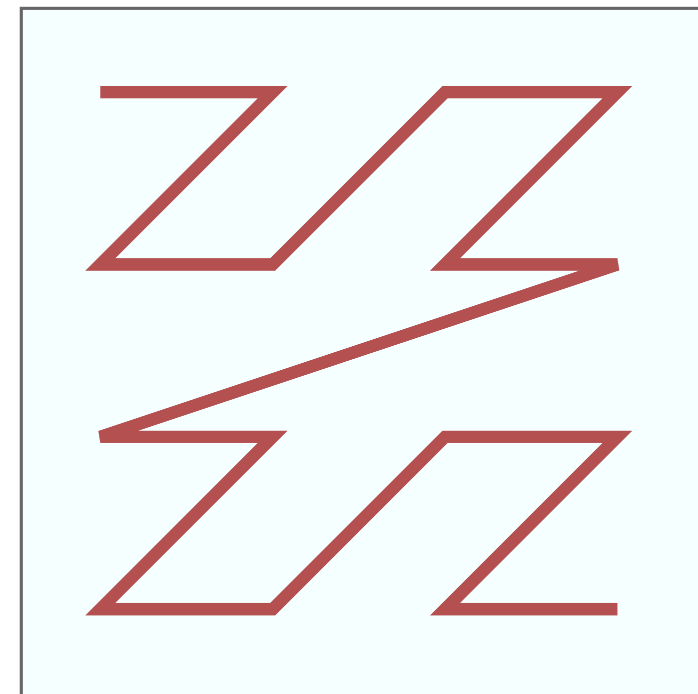
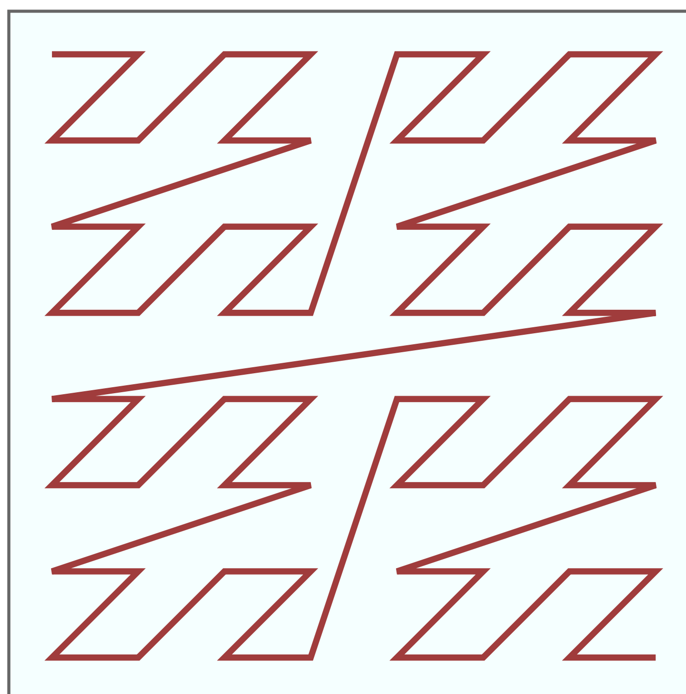
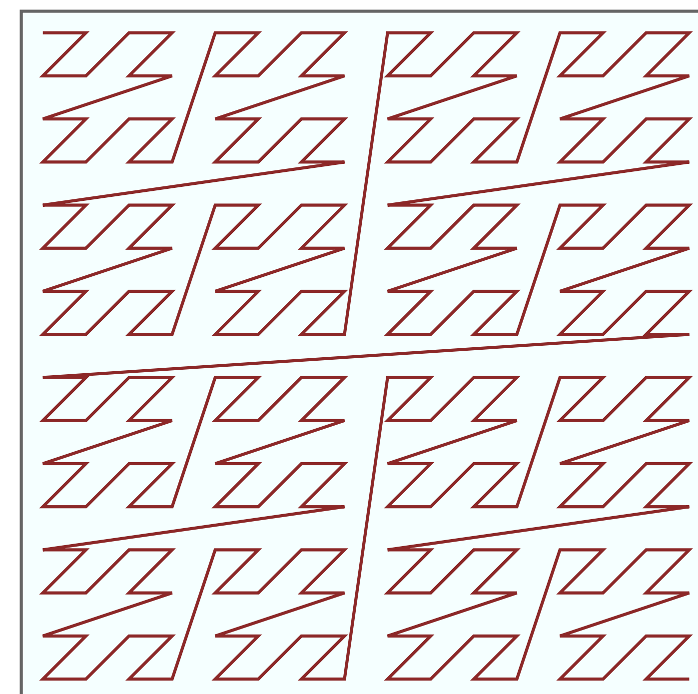
# Morton Curves

- 2D Morton
  known as *Z-curve*

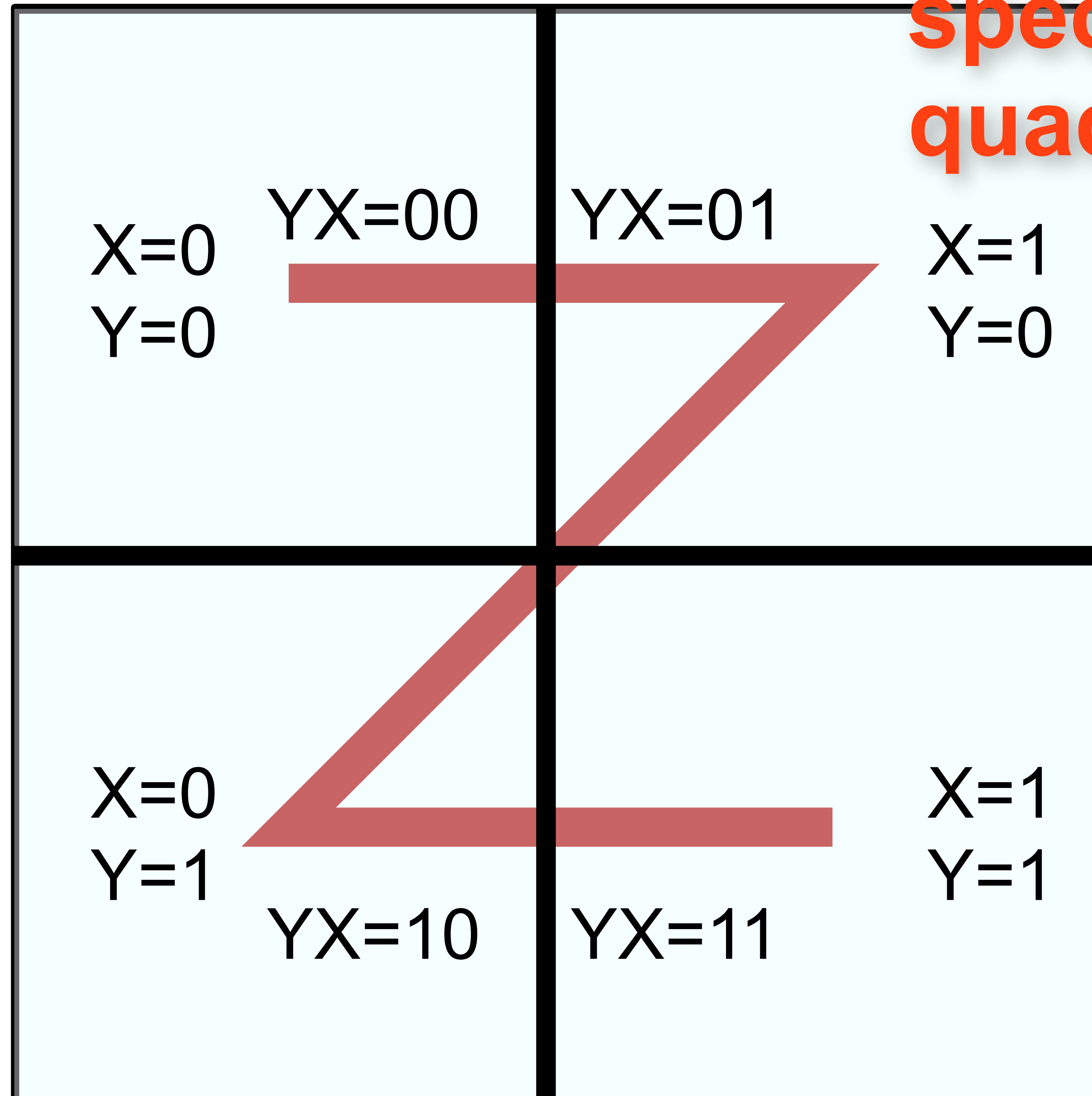
# Morton Curves

- 2D Morton known as *Z-curve*
- Obtained by scaling the x, y, z coordinates to some convenient integer range and then interleaving bits
  - E.g. 21-bit integer coordinates result in 63-bit Morton code, still fits in register

# Why Does This Work?

**Two one-bit coordinates specify a quadrant**



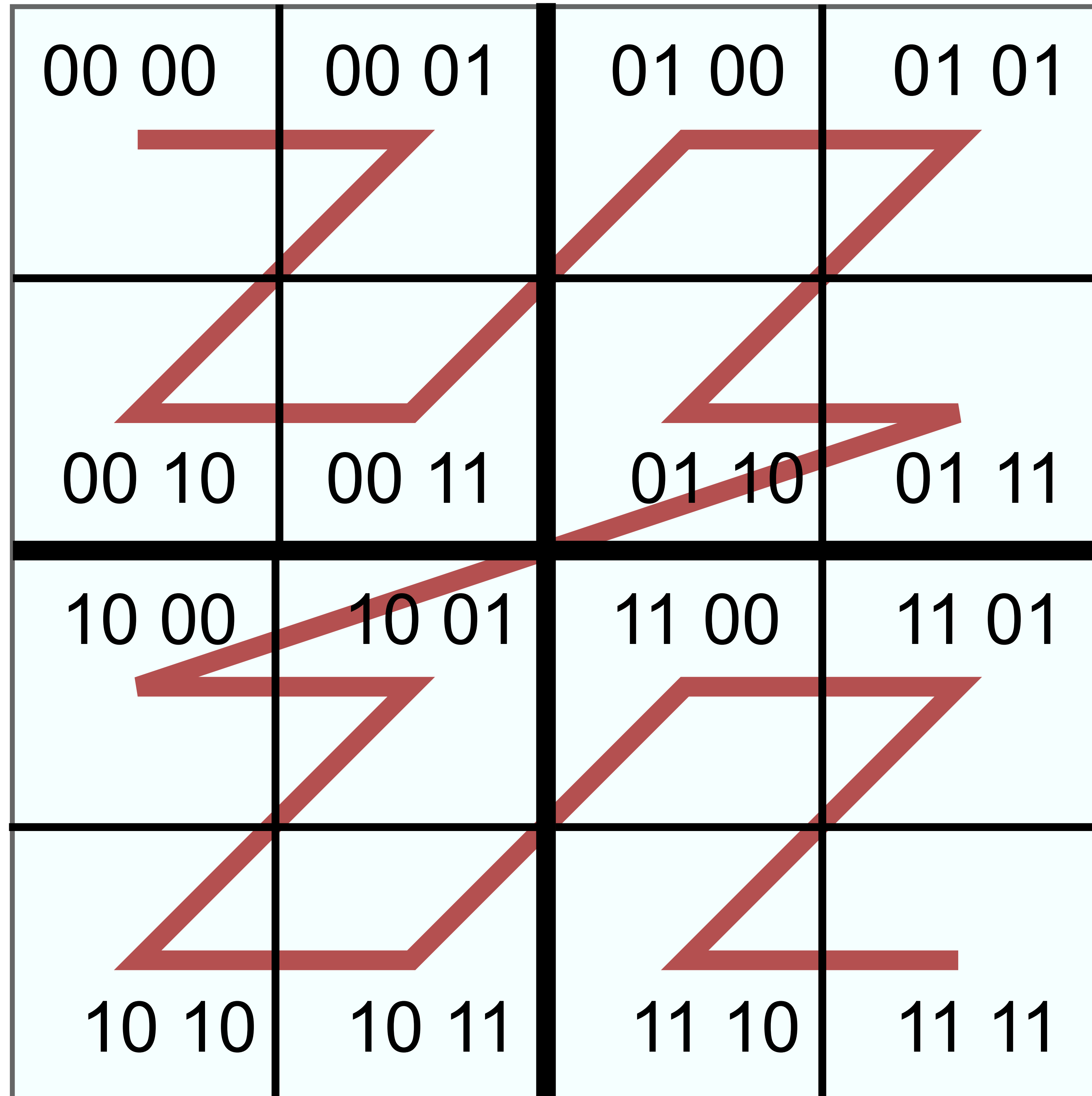

**Interpreting the 2 bits as a binary number gives the one-level Z order**

# Why Does This Work?

Let's group the bits!

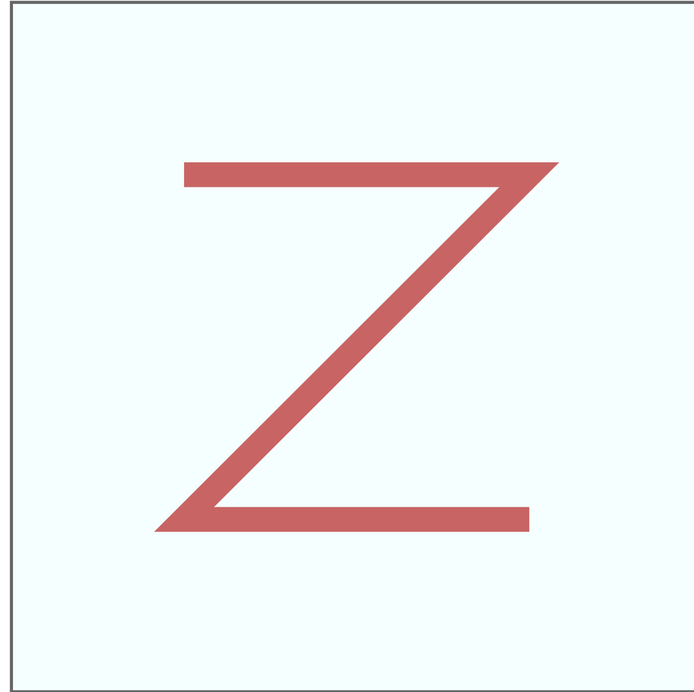
First level: first two bits

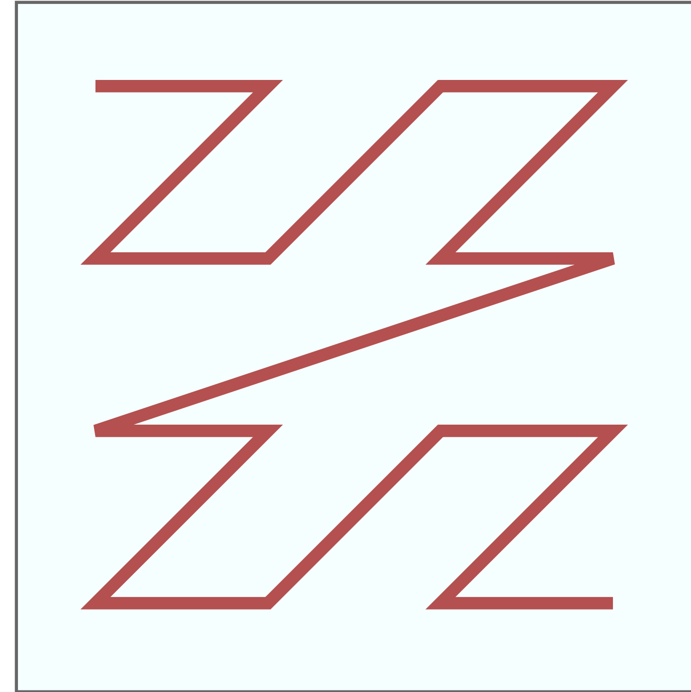
Second level: 2nd two bits



# ..and so on

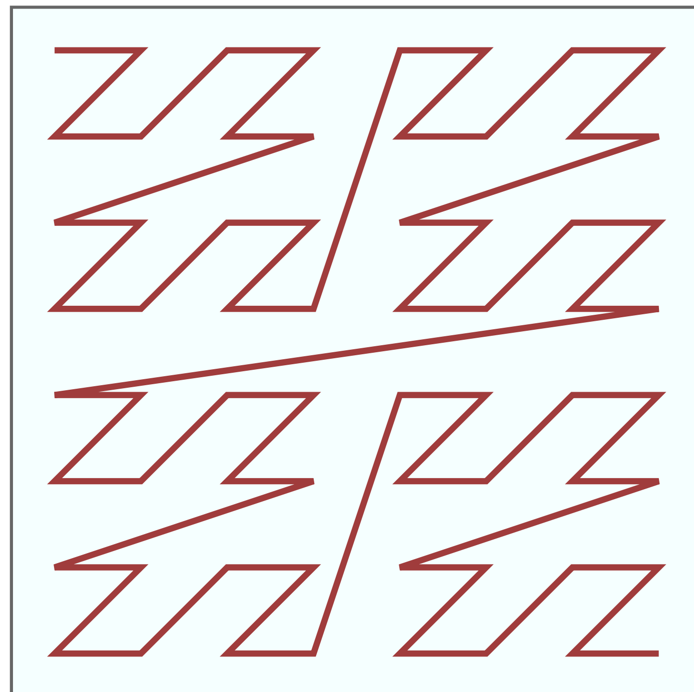
Level 1

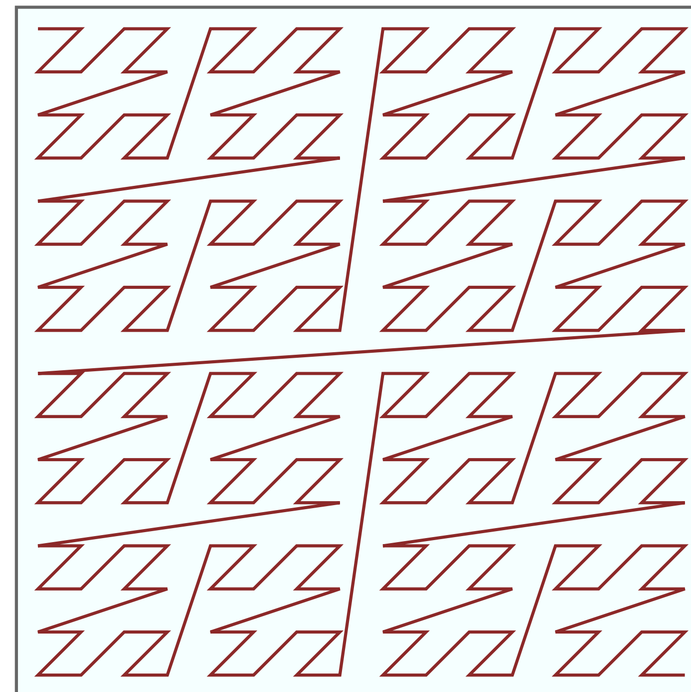




Level 2

Level 3

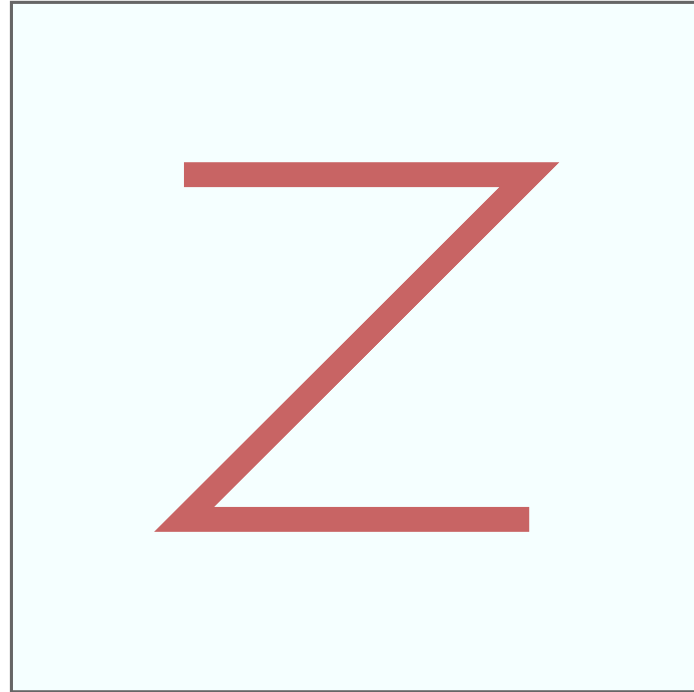


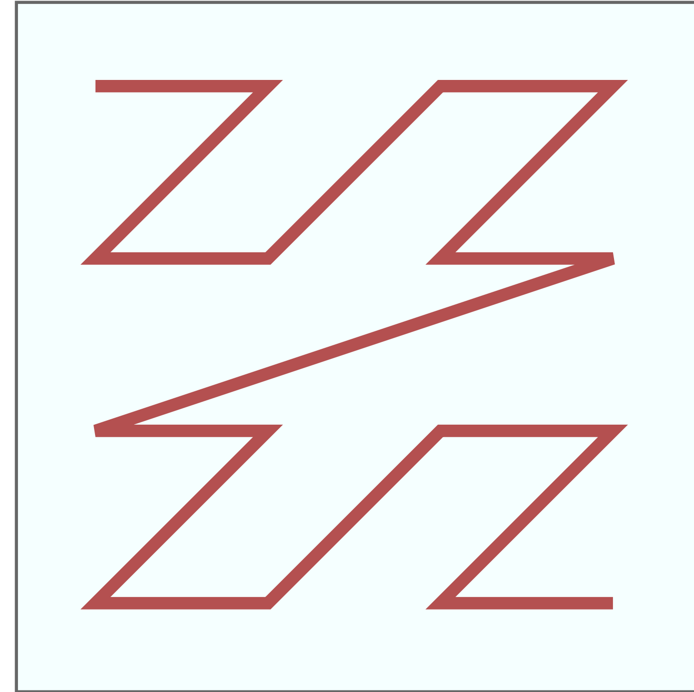


Level 4...

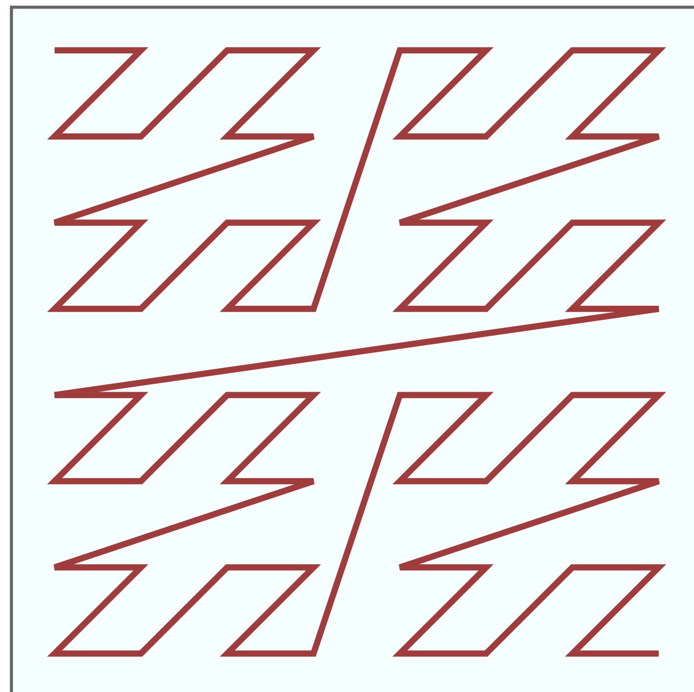
# ..and so on

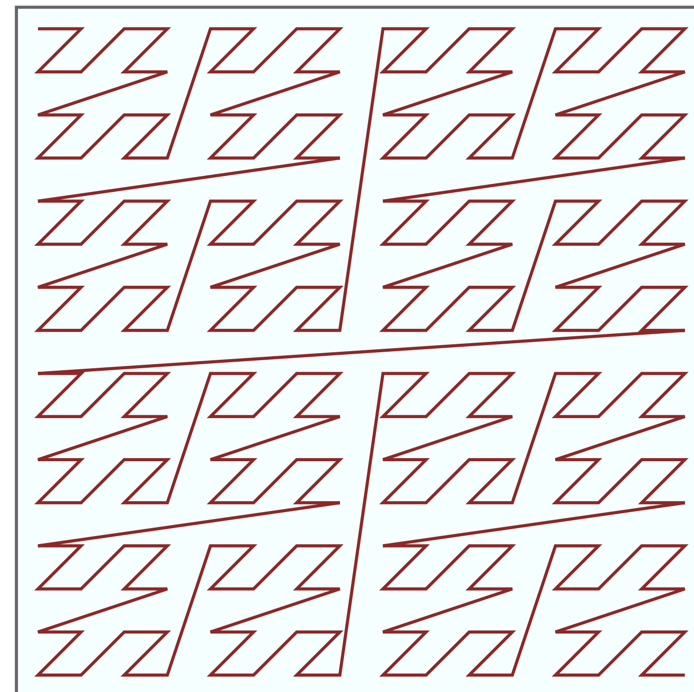
Level 1





Level 3





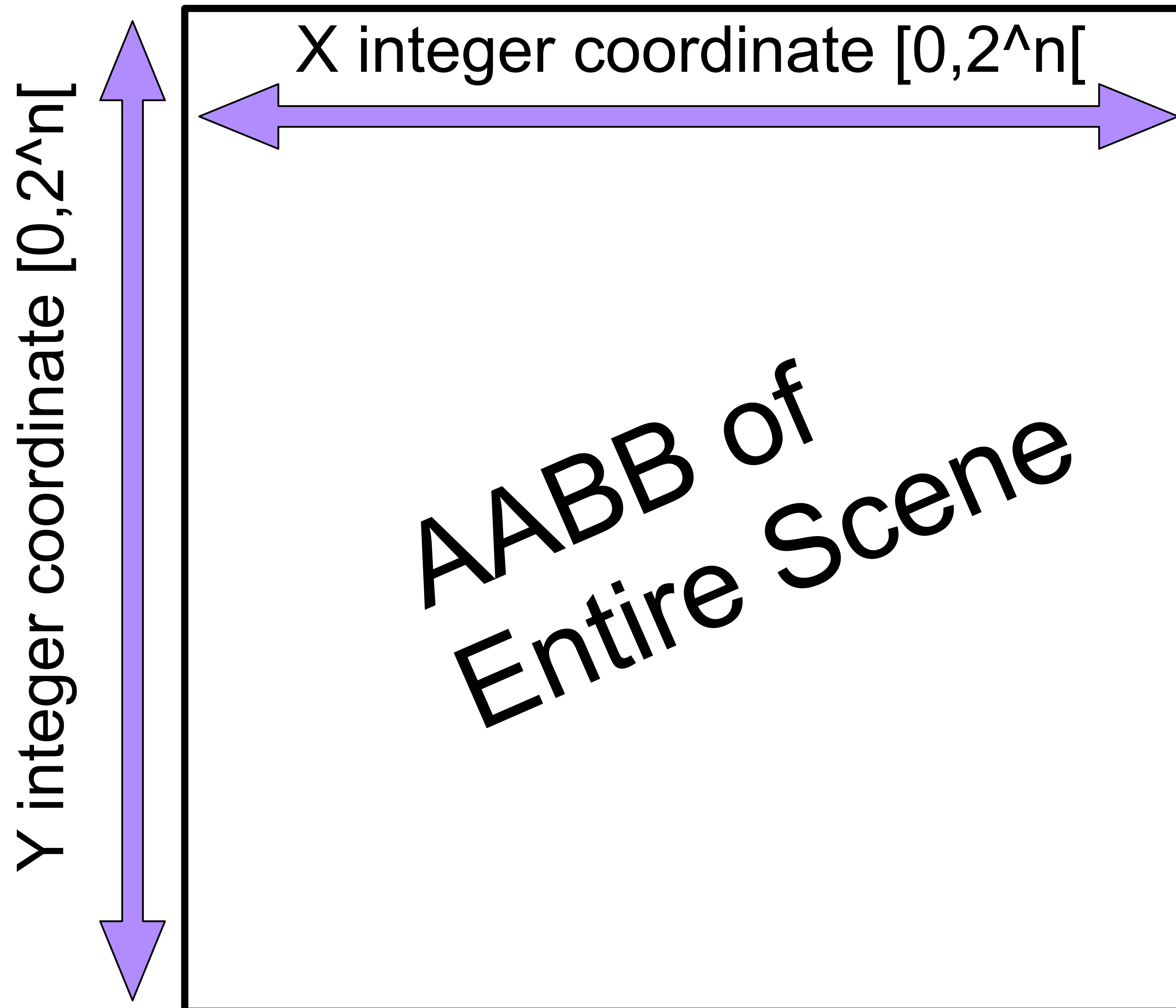
**Key ideas:**

**1.** Short 1D distance along this curve usually corresponds to spatial proximity

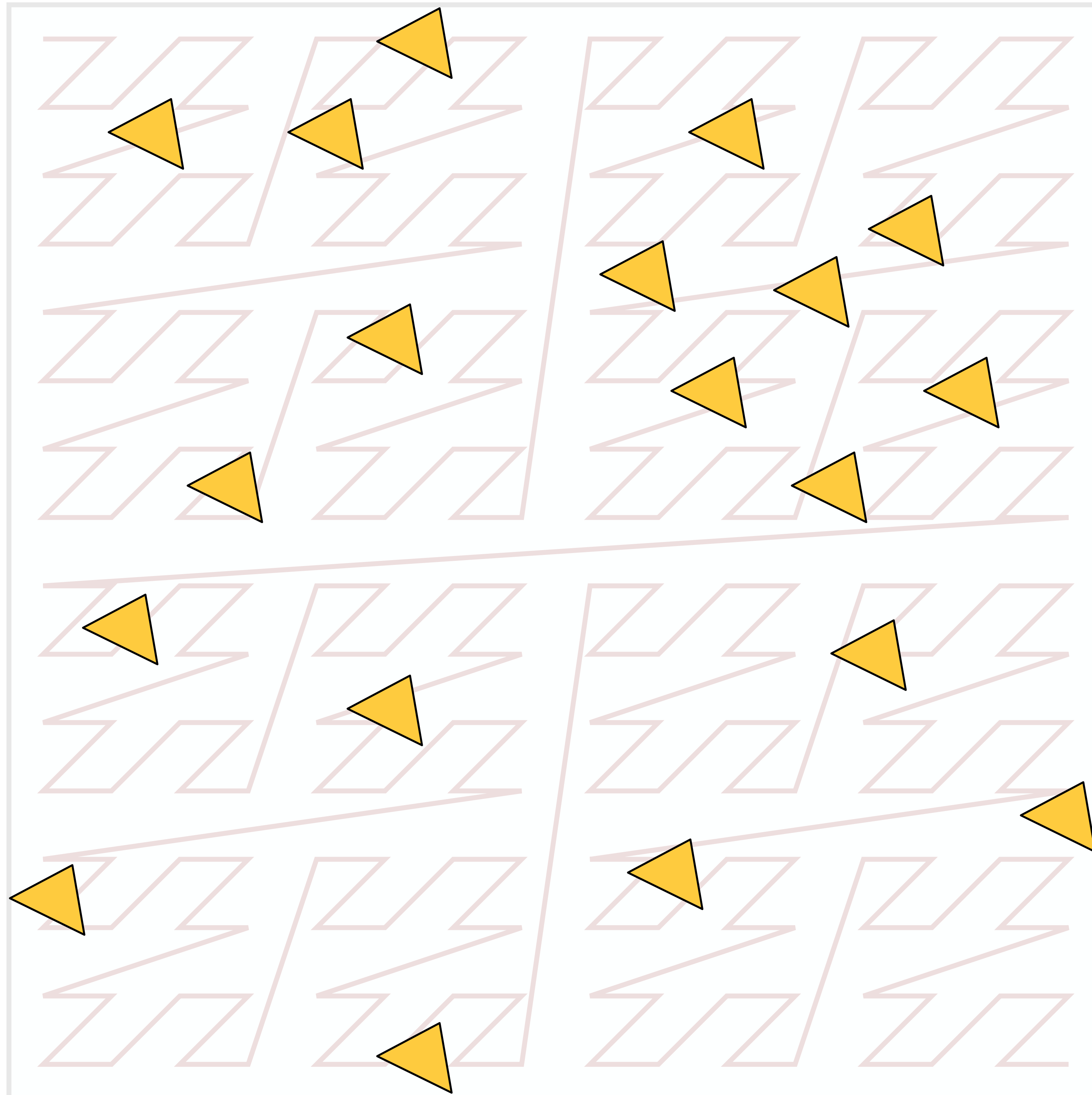
**2.** It's hierarchical by construction

# Scaling

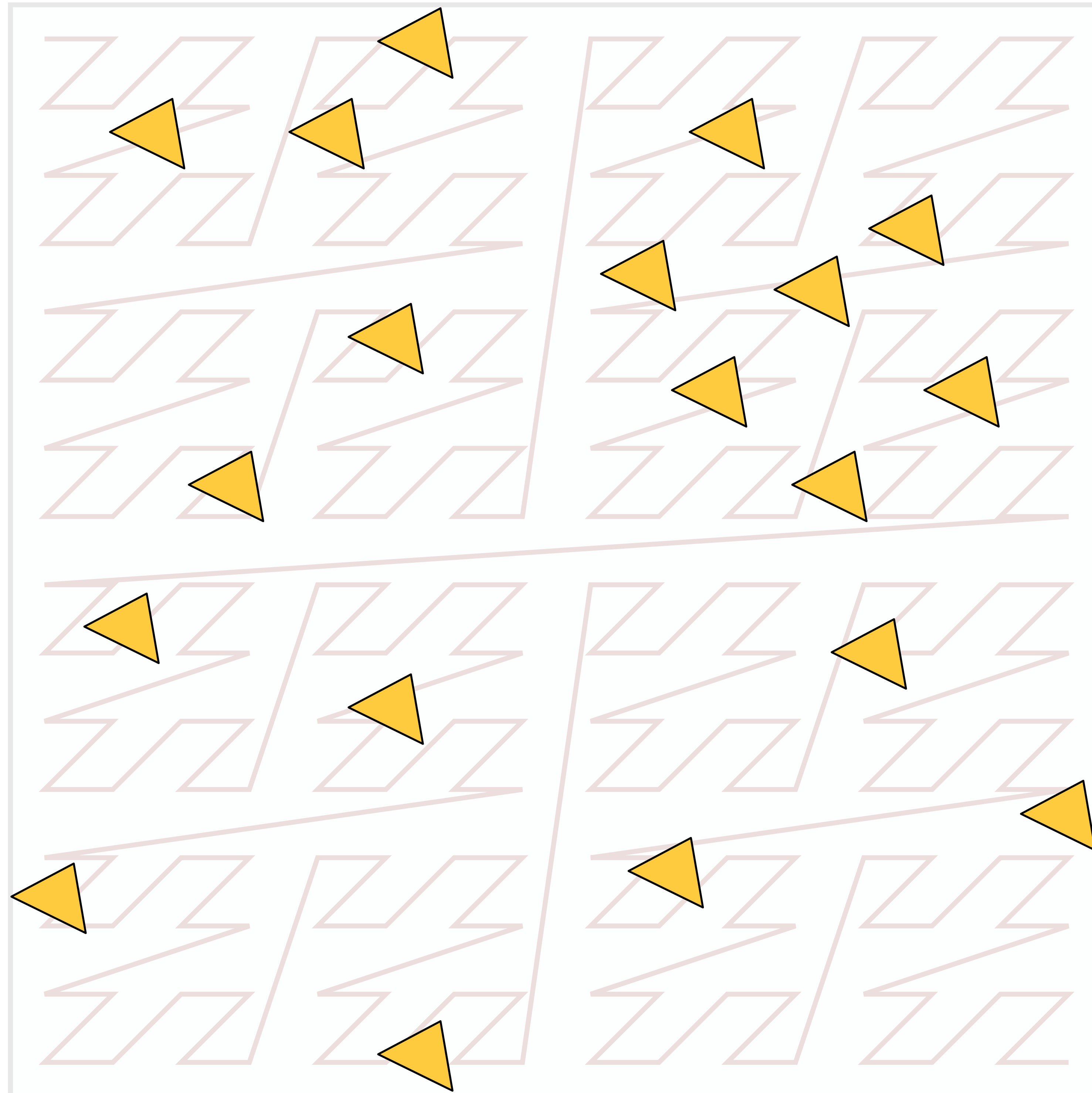
Upper bound is exclusive, e.g. 0…255 = 2^8-1



# Quantize centroids of all triangles like that

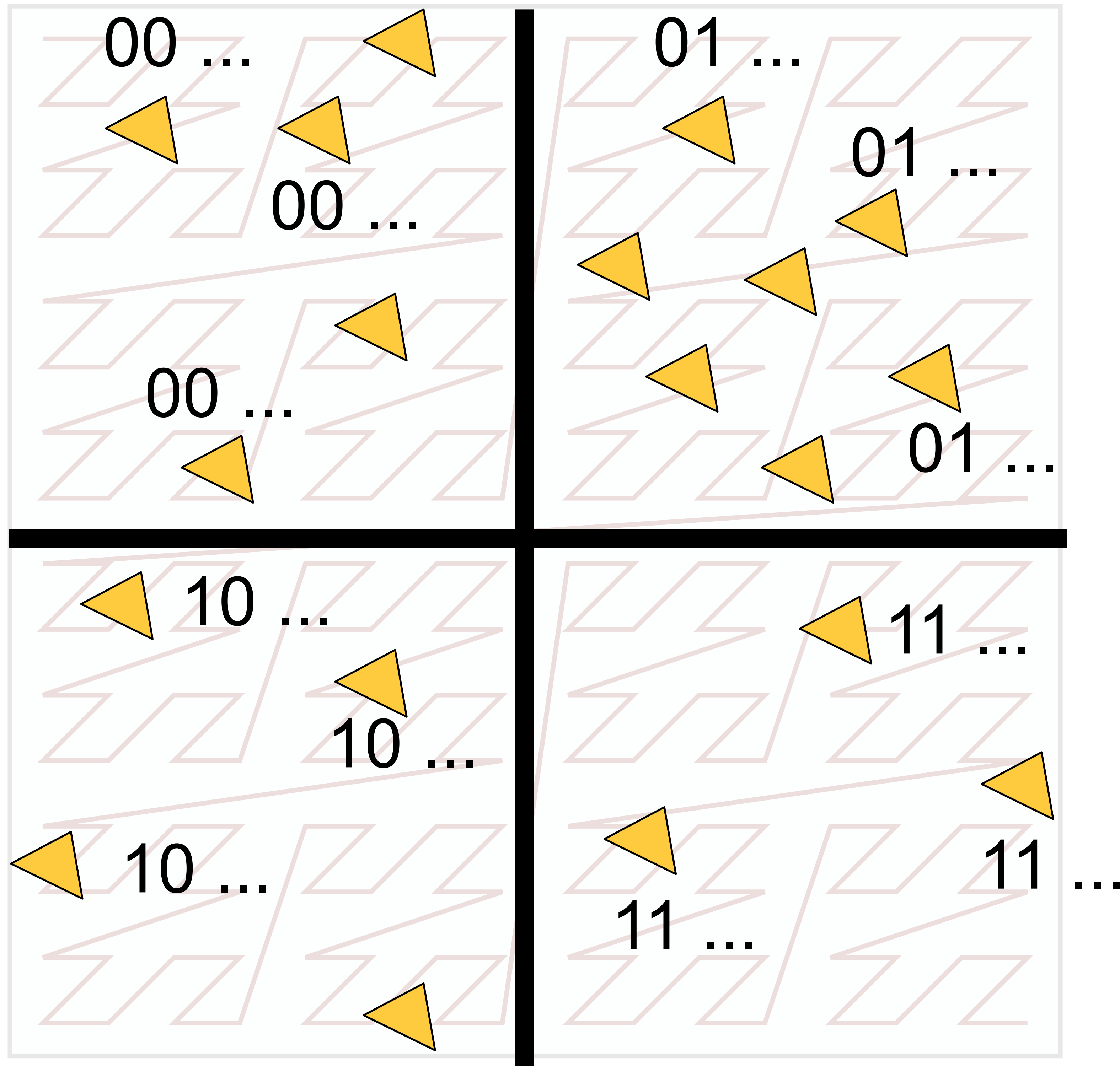


# The Hierarchy is Implicit in the Quantized Coordinates



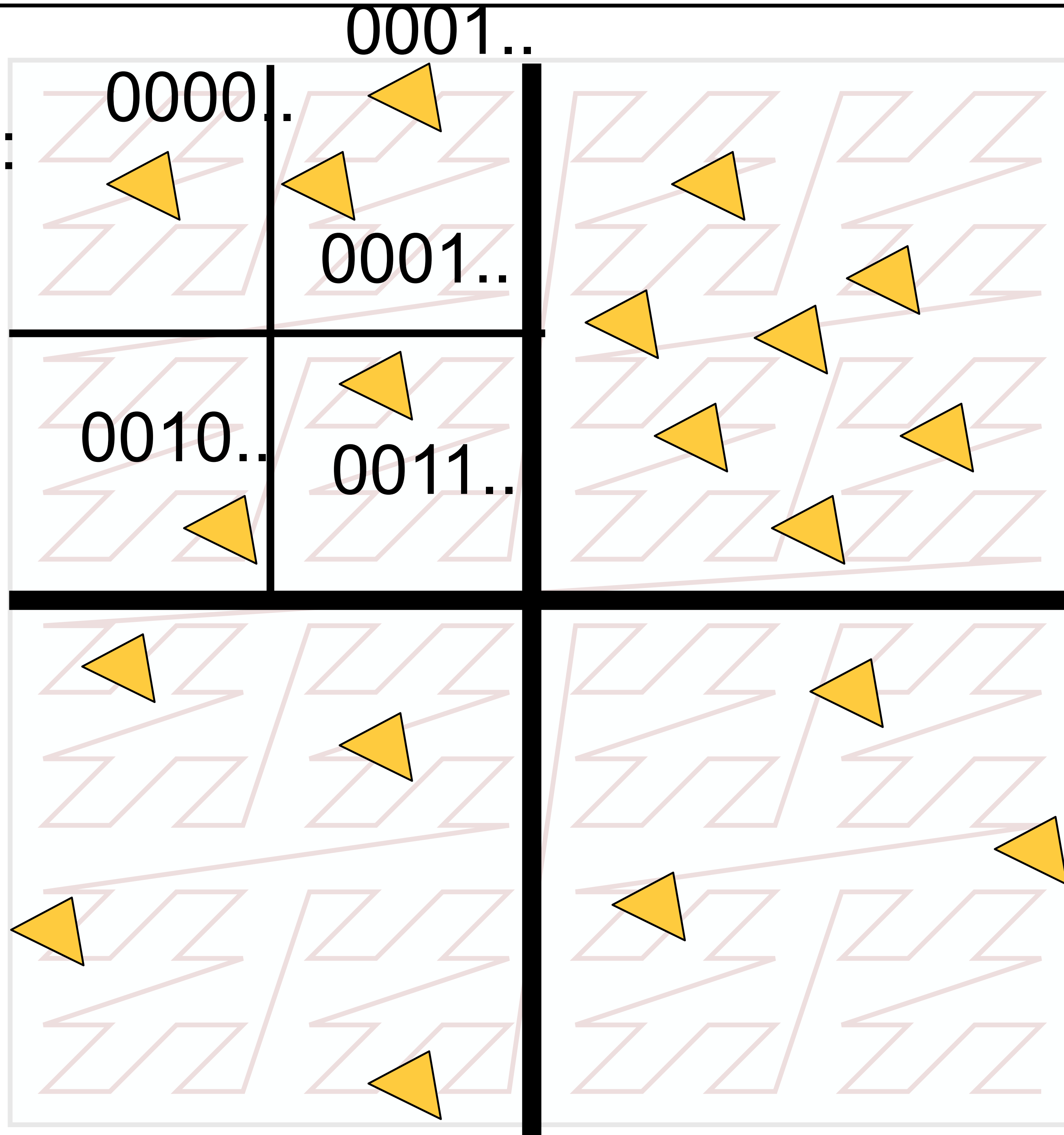
# The Implicit Quadtree Hierarchy

First level:
first two bits



# The Implicit Quadtree Hierarchy

Second level: next two bits



# Notes

- In 2D, Morton order provides implicit quadtree subdivision
  - Every node splits in 4 quadrants
- In 3D, it's an *octree*
  - Every node splits into 8

- Trivially easy to convert into a binary tree
  - Each octree node becomes three binary tree nodes
  - First split x, then y, then z
    - Each split becomes a binary tree node. Simple!

# Morton Code BVH builders

- Extremely fast, can build trees from millions of triangles in milliseconds [Karras2012]
- Also highly convenient for out-of-core builders
  - DreamWorks' point-based global illumination solver uses one of these, see Kontkanen, Tabellion, Overbeck 2011, Coherent Out-of-core Point-Based Global Illumination. Proc. EGSR 2011.

# Assignment 1

- We will publish an leaderboard on the ray tracers you write in the 1st assignment ;)
  - No names shown, of course

# Questions?