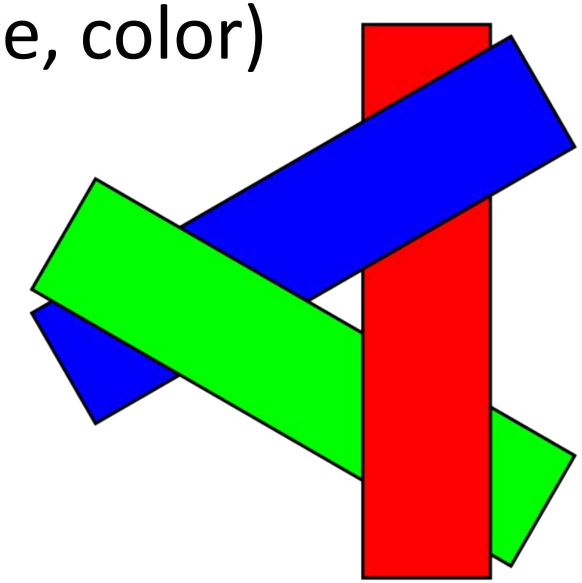
Acceleration RT using BVH

Raghavendra G S

How to produce an image?

- A good image needs realistic *intensity* and *visibility*
  - **Intensity** creates stimulus of optic nerve (black, white, color)
  - **Visibility** makes sure that objects adhere to depth



Source: Wojciech Mula, Wikipedia “Painter's algorithm”

- (Naïve) Ray-Casting Render Loop
  - Shoot a ray through **each** pixel into the scene
  - Iterate over **all** objects and test for intersection
  - Record the **closest** intersection (*visibility*)
  - Compute color and write to pixel (*intensity*)

Render Loop Run Time

- Let‘s look at the basic runtime (single sample per pixel)

```
void render(Camera cam)
{
    for(Pixel& pix : pixels)
    {
        …

        for (Triangle& tri : triangles)
        {
            …
        }

        …
    }
}
```

Render Loop Run Time

- Let‘s look at the basic runtime (single sample per pixel)

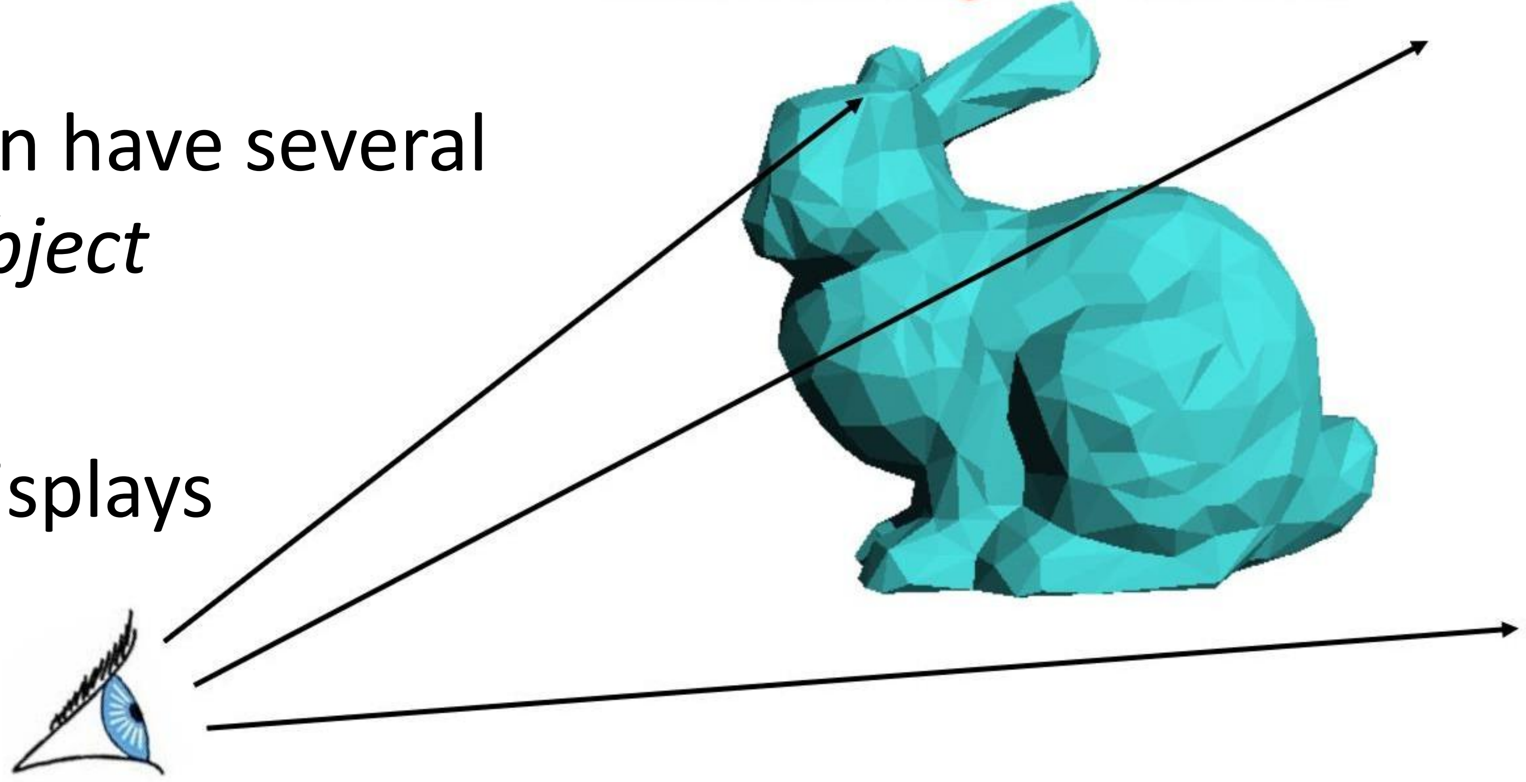
```
void render(Camera cam)
{
    for(Pixel& pix : pixels) # N
    {
        …

        for (Triangle& tri : triangles) # M
        {
            …
        }

        …
    }
}
```

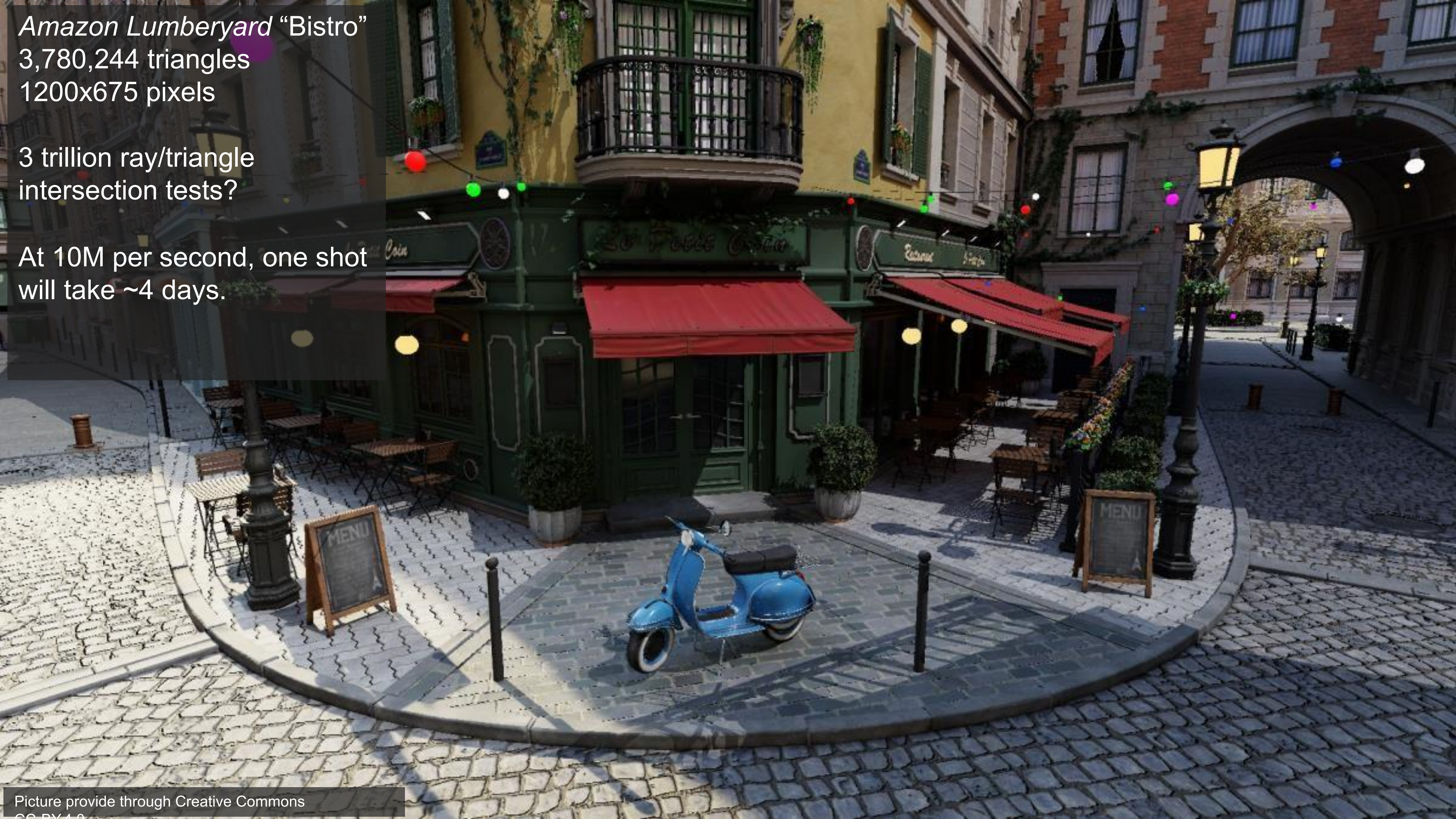
This is $\mathcal{O}(N \cdot M)$, but even worse, it’s $\Omega(N \cdot M)$

Is That Actually a Problem?

- Run time complexity quickly becomes a limiting factor
- High-quality scenes can have several million triangles *per object*
- Current screens and displays are moving towards 4k resolution

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





Amazon Lumberyard “Bistro”
3,780,244 triangles
1200x675 pixels

3 trillion ray/triangle intersection tests?

At 10M per second, one shot will take ~4 days.

Picture provide through Creative Commons
CC BY 4.0

What can we do about it?

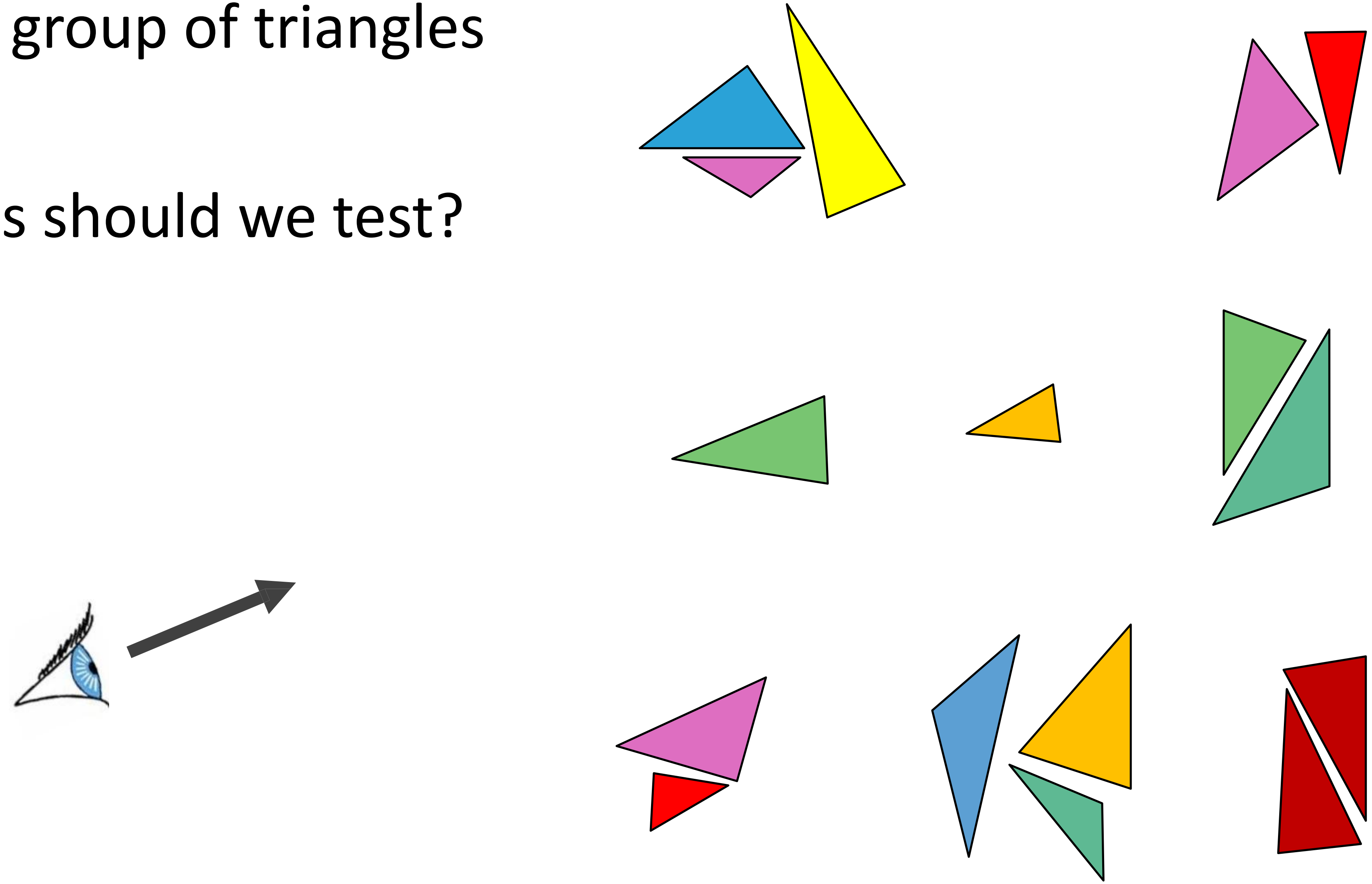
- Find ways to speed up the basic loop for visibility resolution
- Enter “spatial acceleration structures”
- Essentially, pre-process the scene geometry into a structure that reduces expected traversal time to something more reasonable

Spatial Acceleration Structures

Structure	Additional Memory	Building Time	Traversal Time
none	none	none	abysmal

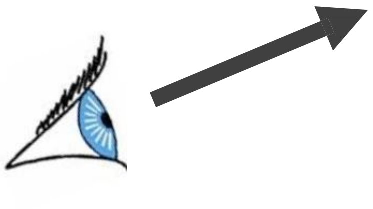
Speeding Up Intersection Tests

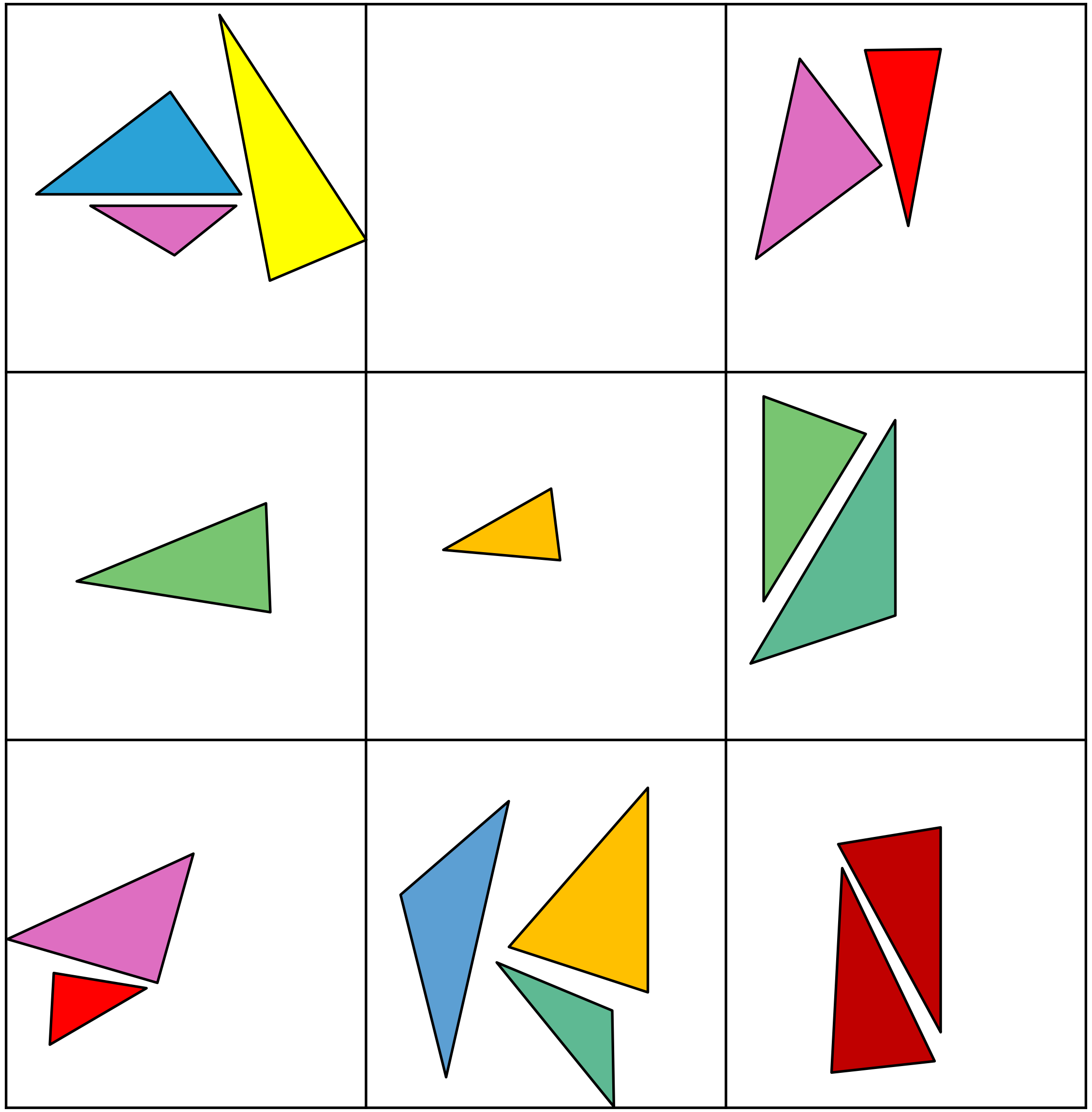
- Consider a group of triangles
- Which ones should we test?



Regular Grids

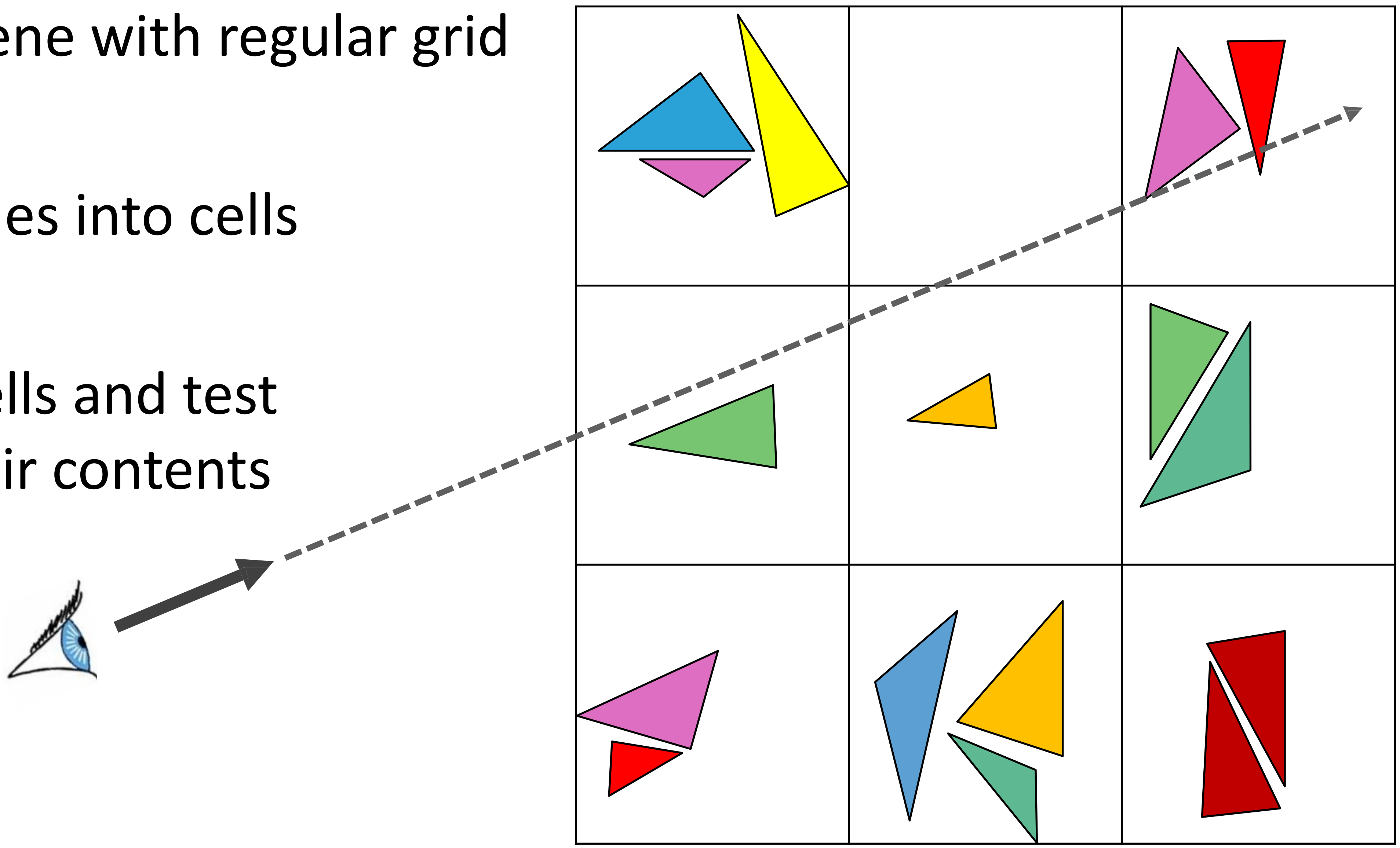
- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents





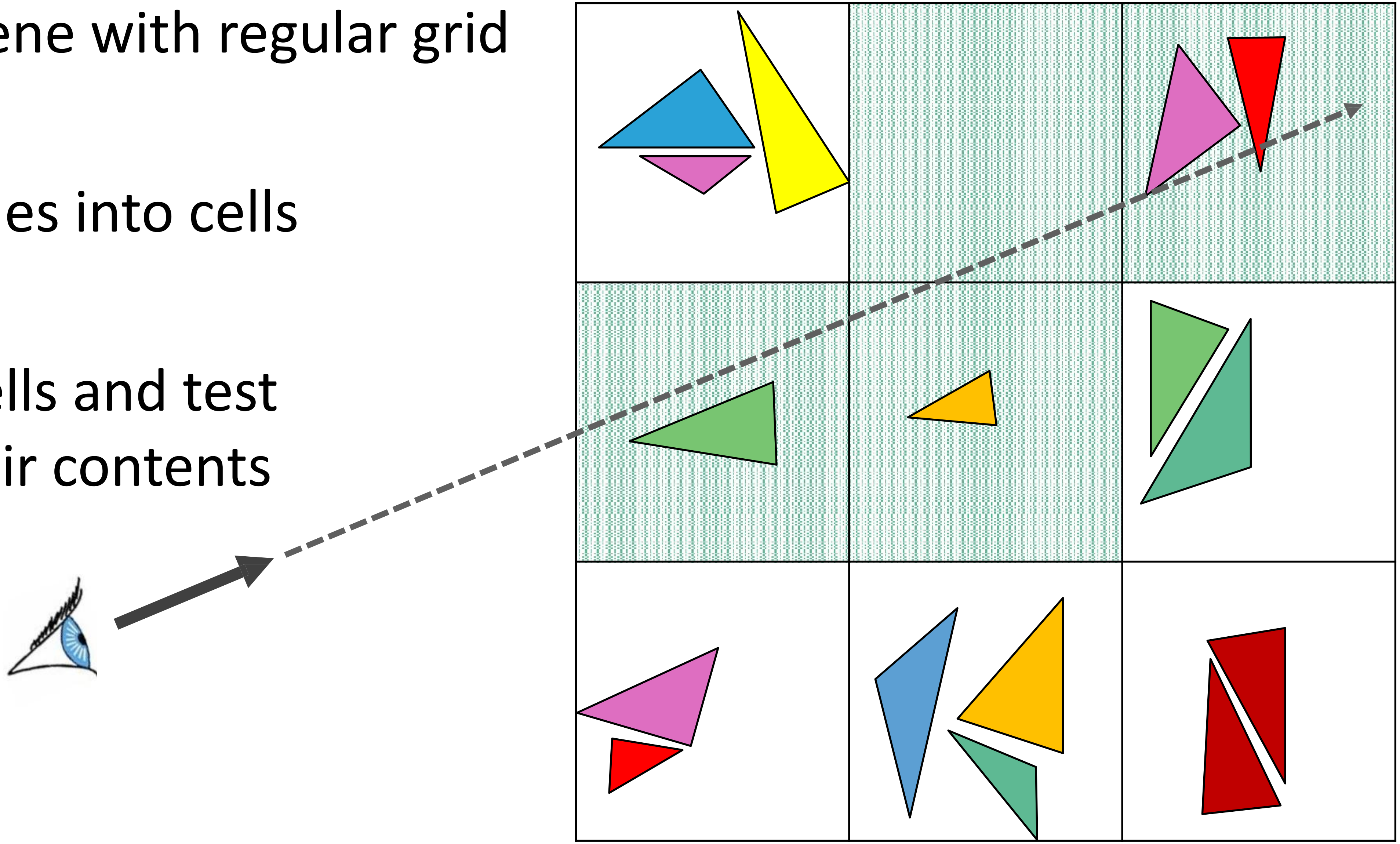
Regular Grids

- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents



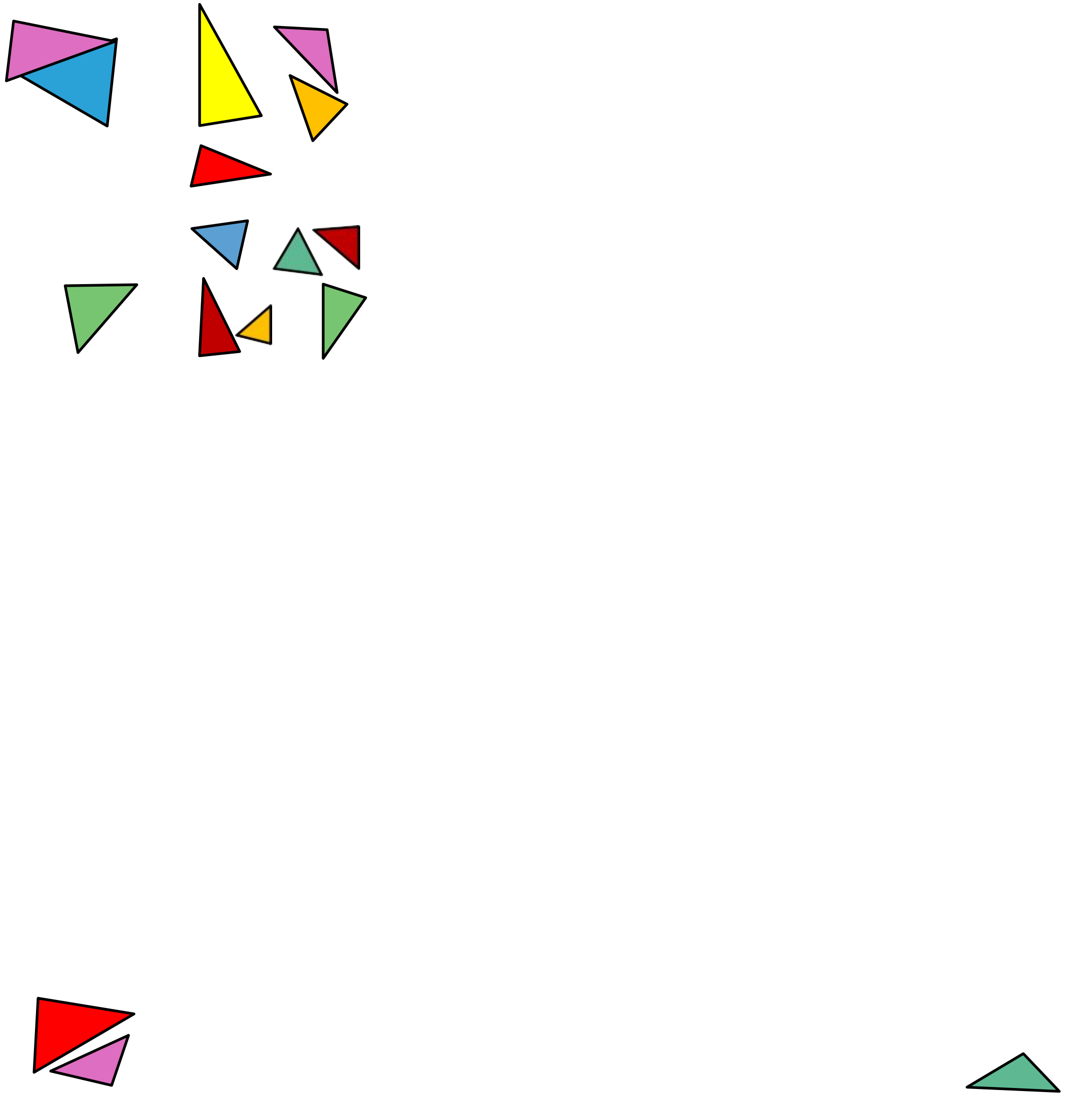
Regular Grids

- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents



Regular Grids

- Geometry is usually not uniform
- Comes in clusters (buildings, characters, vegetation…)

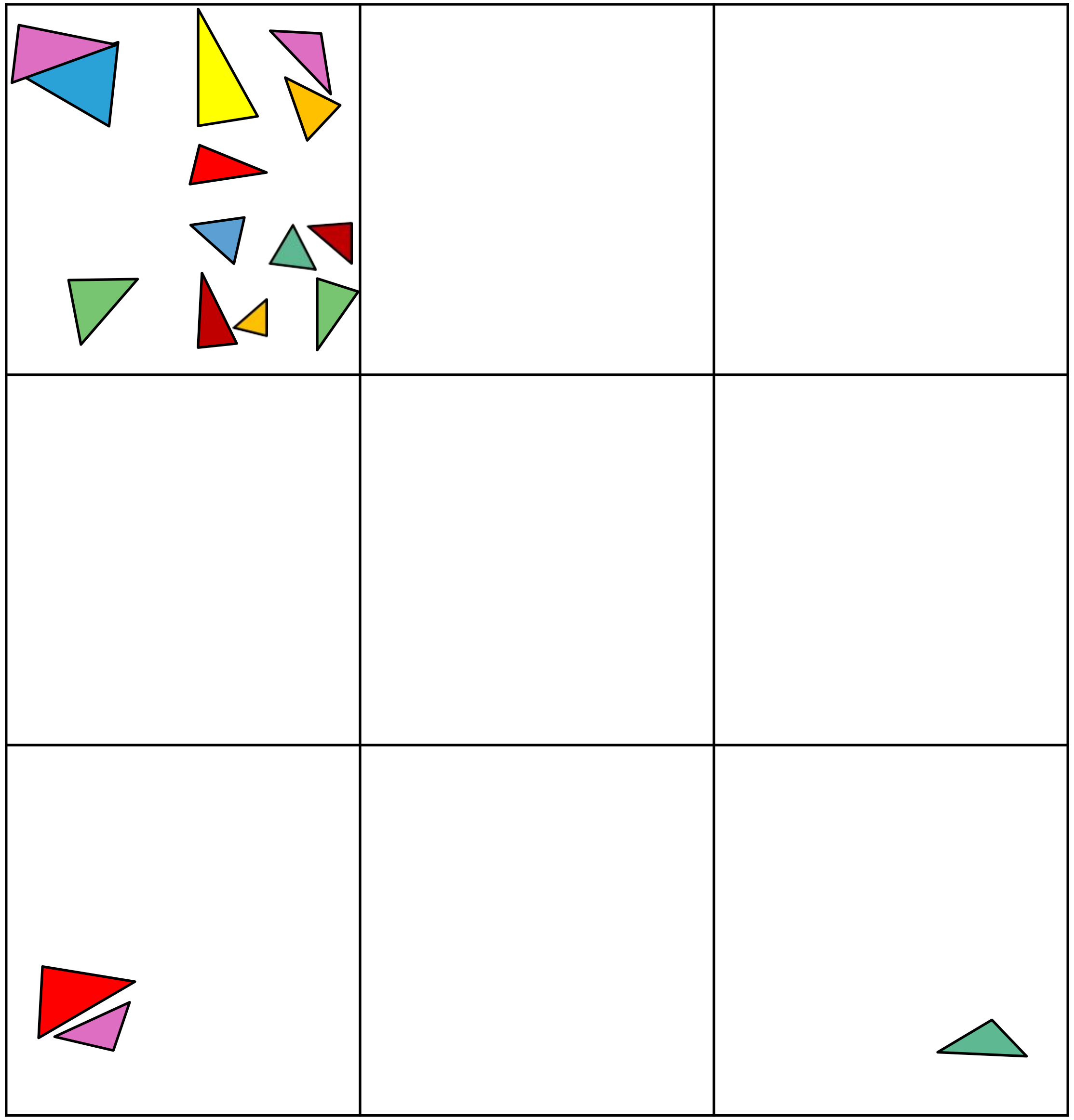


Regular Grids

- Geometry is usually not uniform

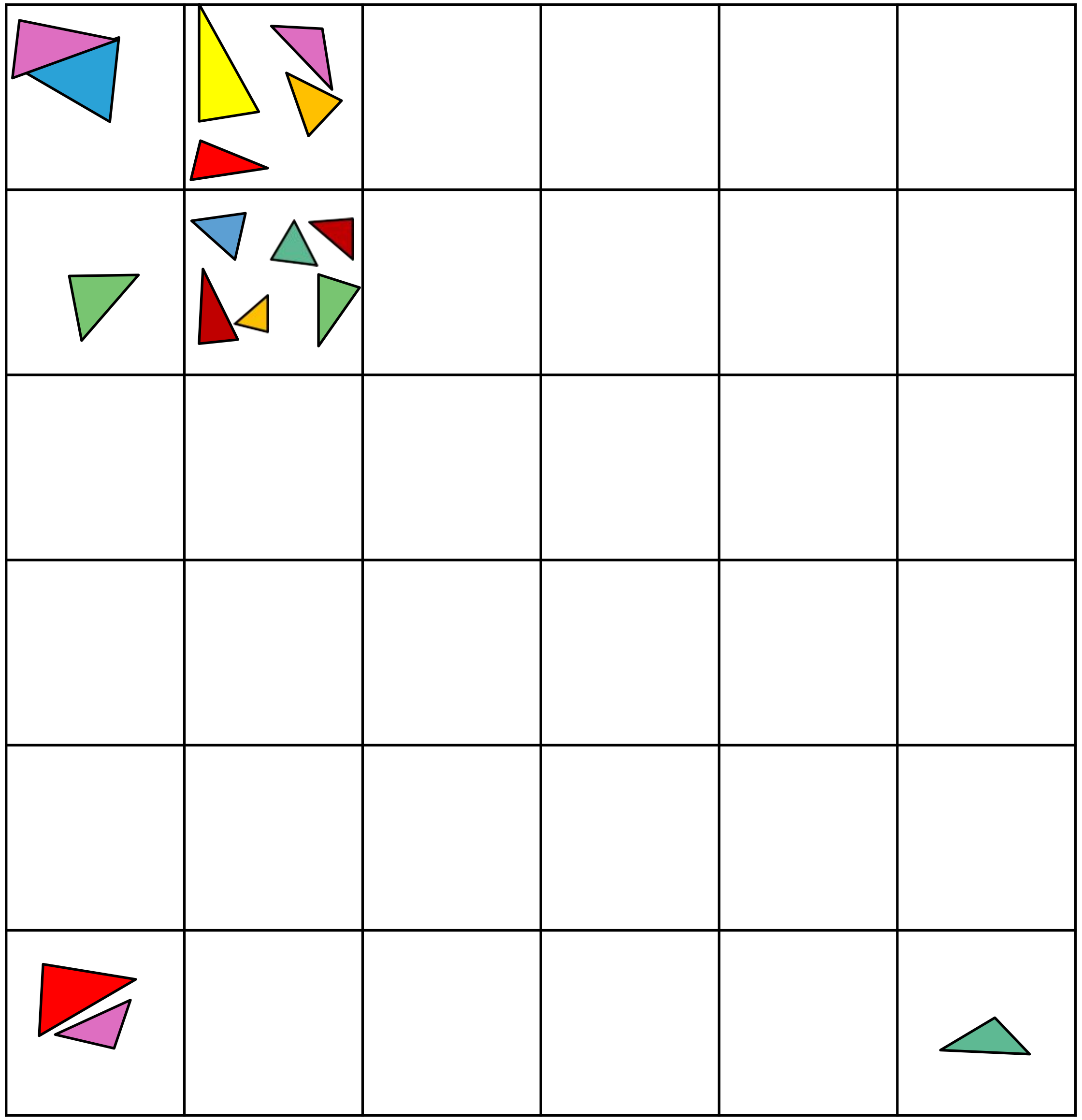
- Comes in clusters (buildings, characters, vegetation…)

- Almost all triangles in one cell! Hitting this cell will be costly!



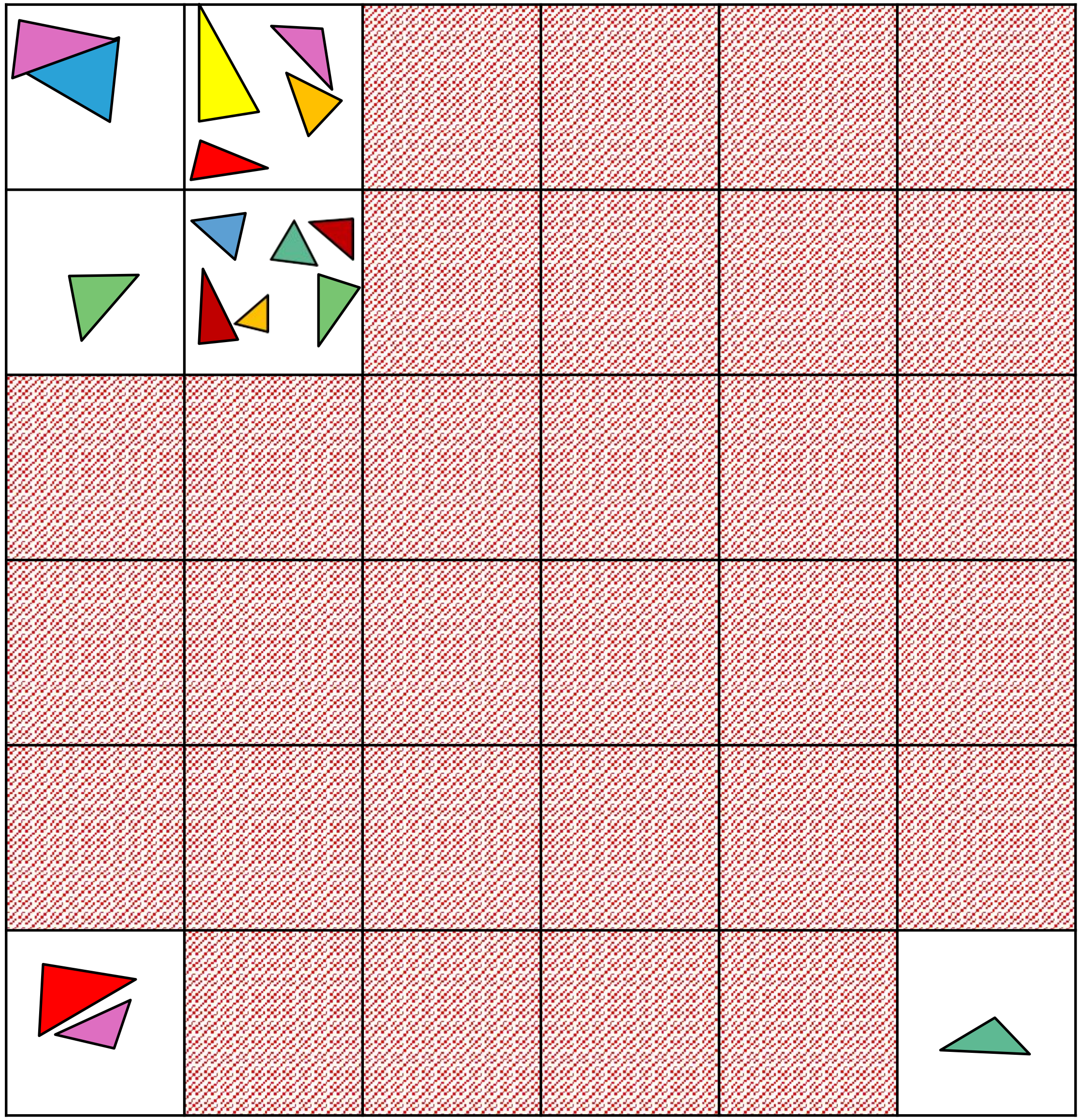
Regular Grids

- Geometry is usually not uniform
- Comes in clusters (buildings, characters, vegetation…)
- ~~Almost all triangles in one cell! Hitting this cell will be costly!~~
- Using a finer grid works



Regular Grids

- Geometry is usually not uniform
- Comes in clusters (buildings, characters, vegetation…)
- ~~Almost all triangles in one cell! Hitting this cell will be costly!~~
- Using a finer grid works, but most of its cells are unused!

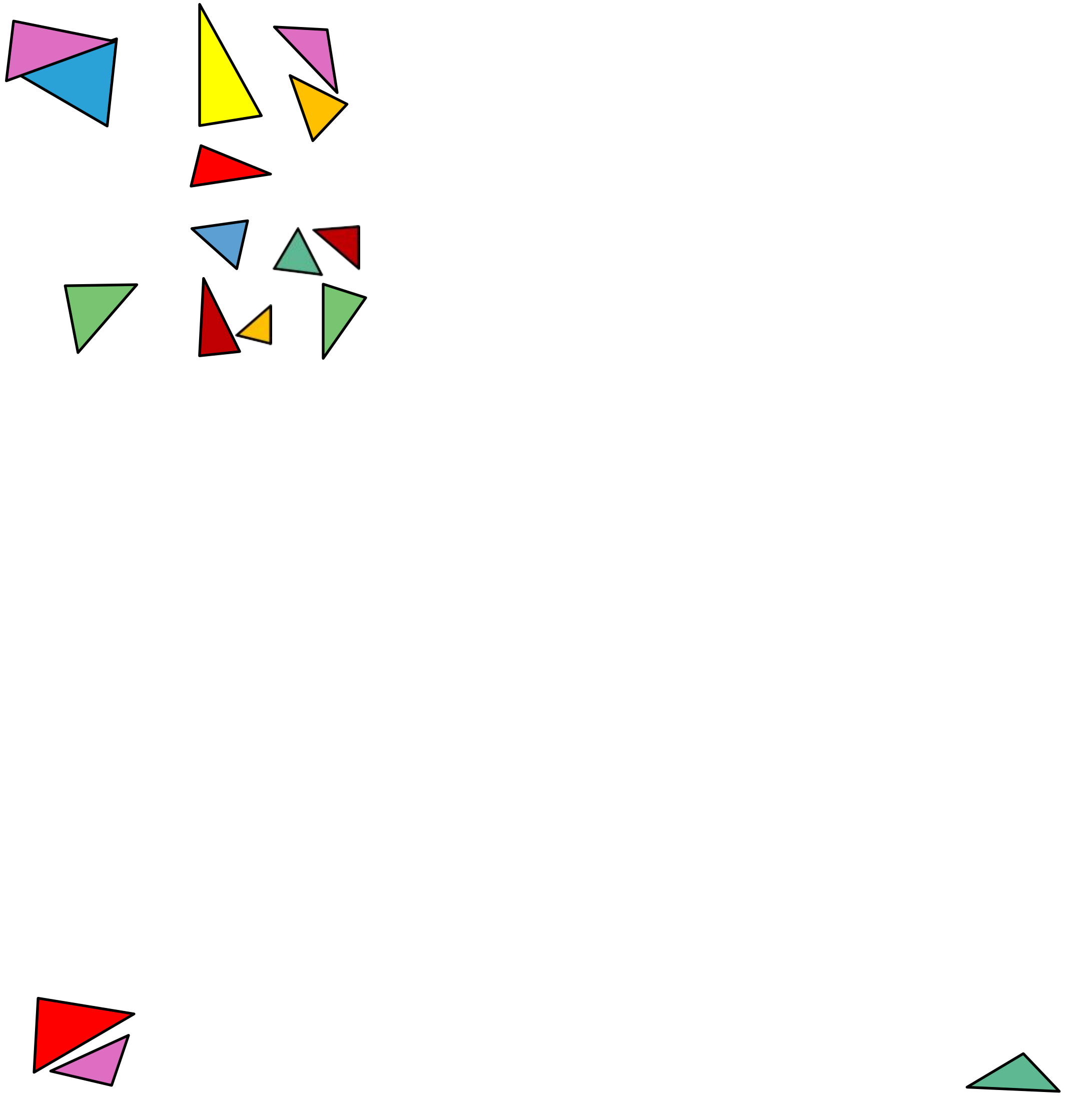


Spatial Acceleration Structures

Structure	Memory Consumption	Building Time	(Expected) Traversal Time
none	none	none	abysmal
Regular Grid	low – high (resolution)	low	uniform scene: ok otherwise: bad

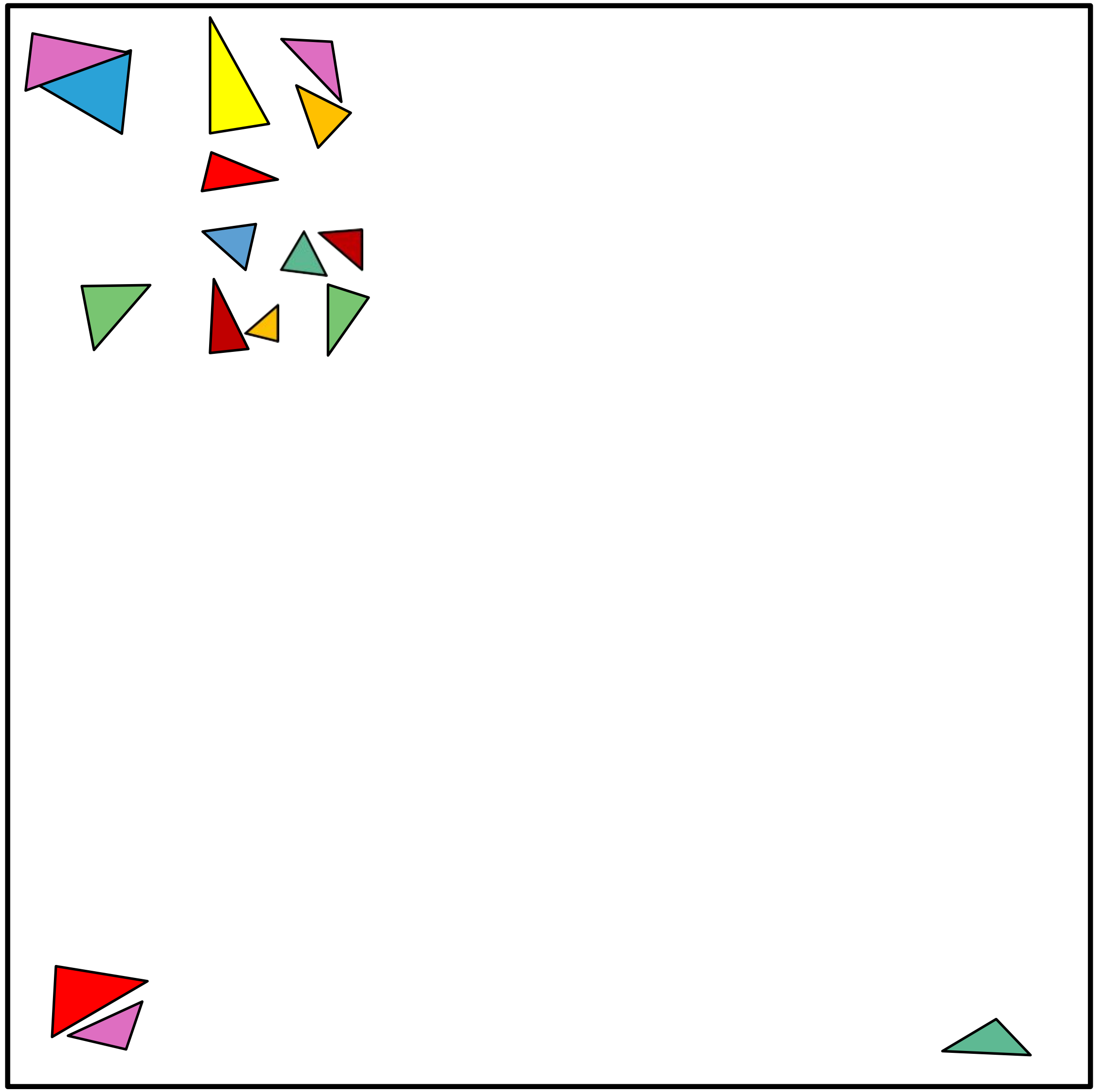
Quadtrees and Octrees

- Start with scene bounds, do finer subdivisions only if needed
- Define parameters S_{max}, N_{leaf}
- Recursively split bounds into *quadrants* (2D) or *octants* (3D)
- Stop after S_{max} subdivisions or if no cell has $> N_{leaf}$ triangles



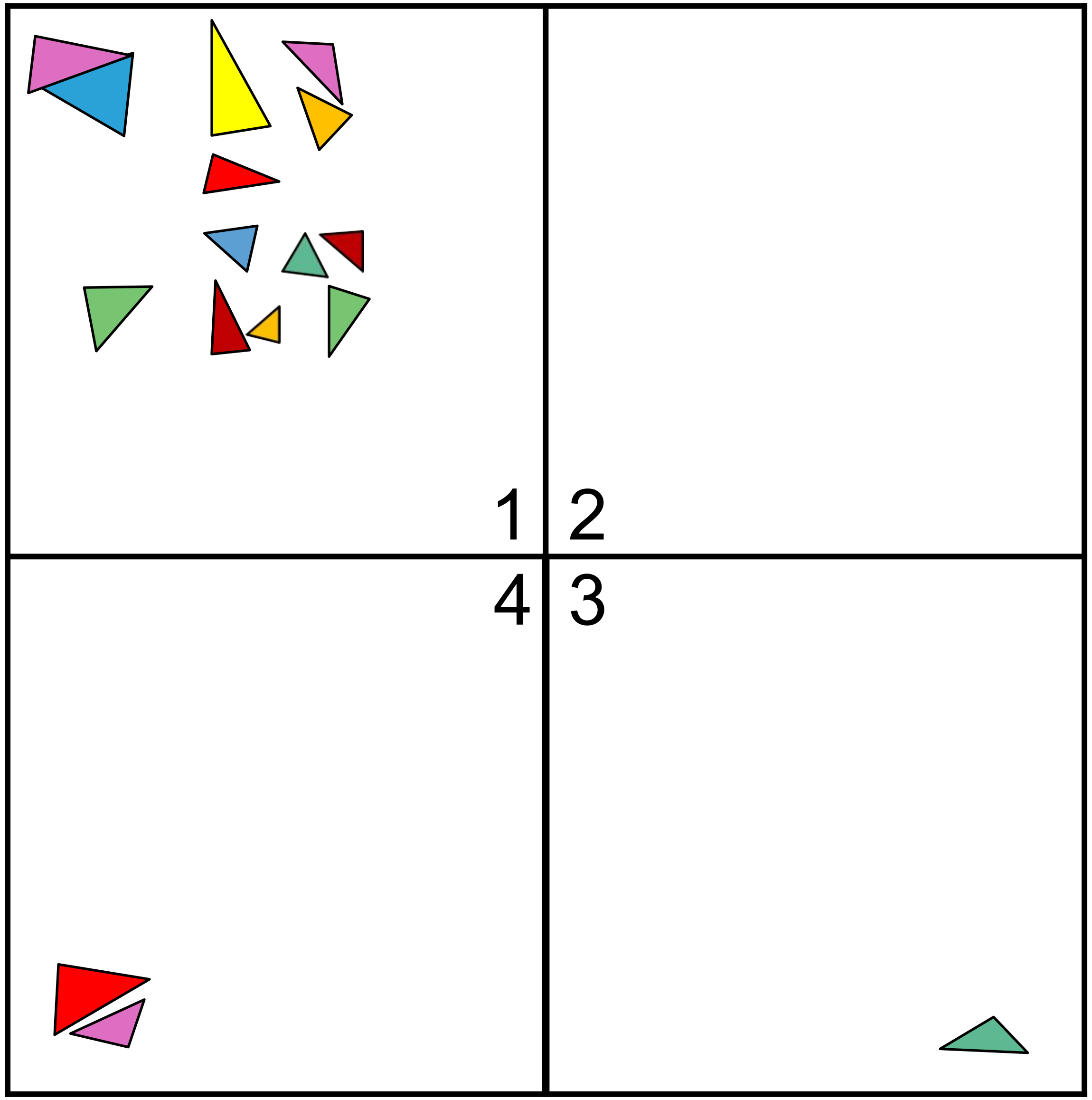
Quad and Octrees: $N_{leaf} = 4$

- Start with scene bounds, do finer subdivisions only if needed
- Define parameters S_{max}, N_{leaf}
- Recursively split bounds into *quadrants* (2D) or *octants* (3D)
- Stop after S_{max} subdivisions or if no cell has $> N_{leaf}$ triangles

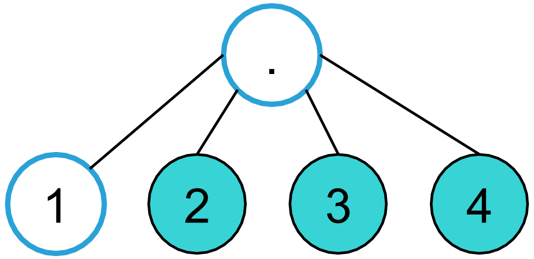


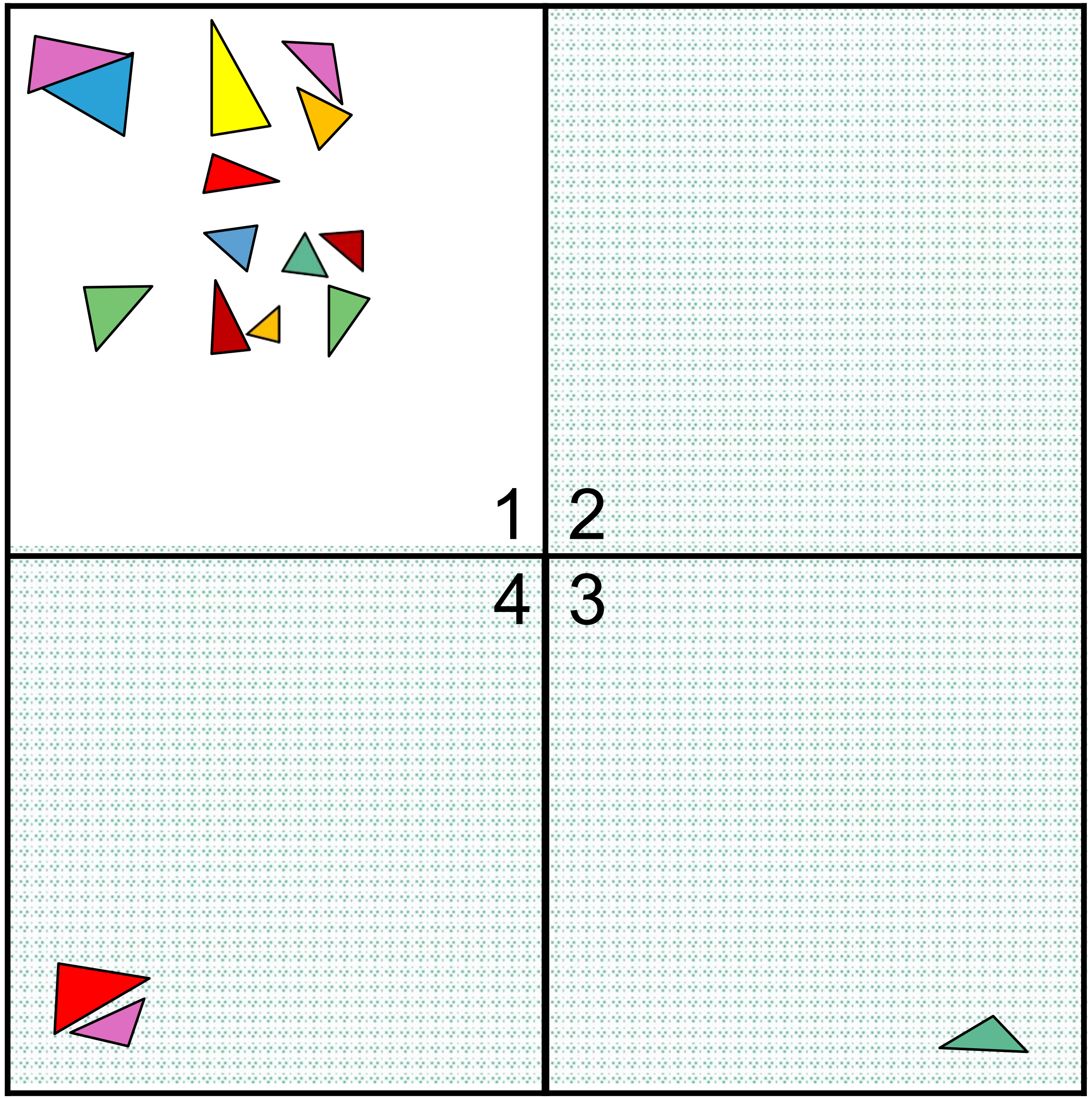
Quad and Octrees: $N_{leaf} = 4$

- Start with scene bounds, do finer subdivisions only if needed
- Define parameters S_{max}, N_{leaf}
- Recursively split bounds into *quadrants* (2D) or *octants* (3D)
- Stop after S_{max} subdivisions or if no cell has $> N_{leaf}$ triangles

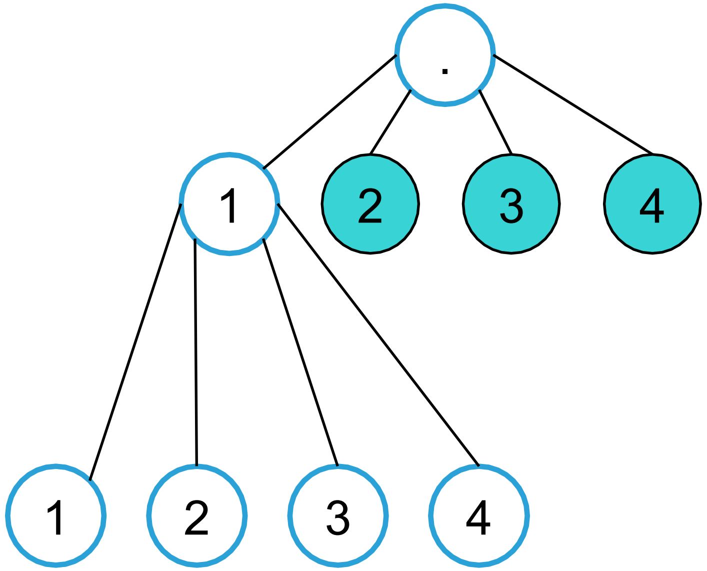


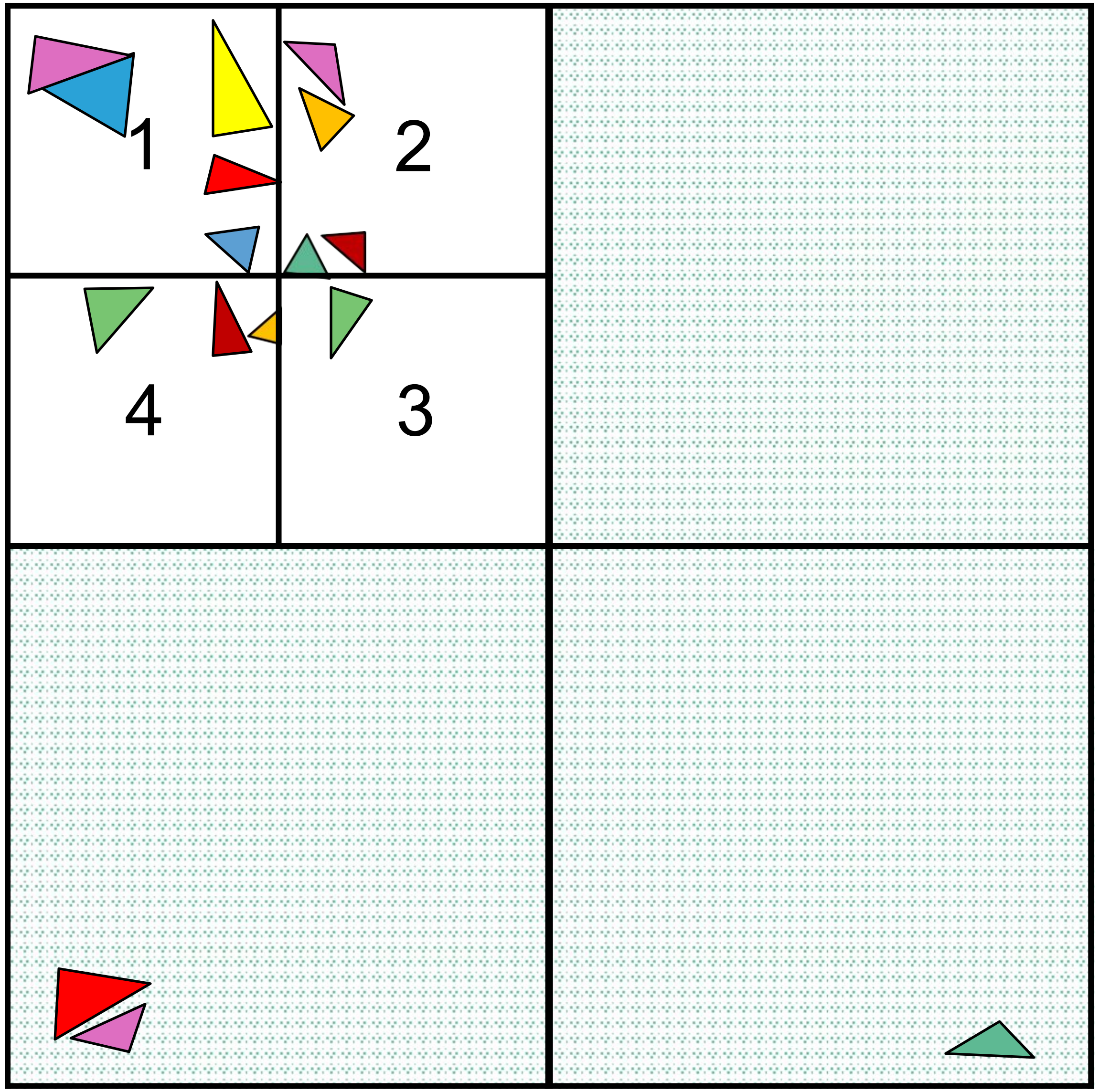
Quad and Octrees: $N_{leaf} = 4$



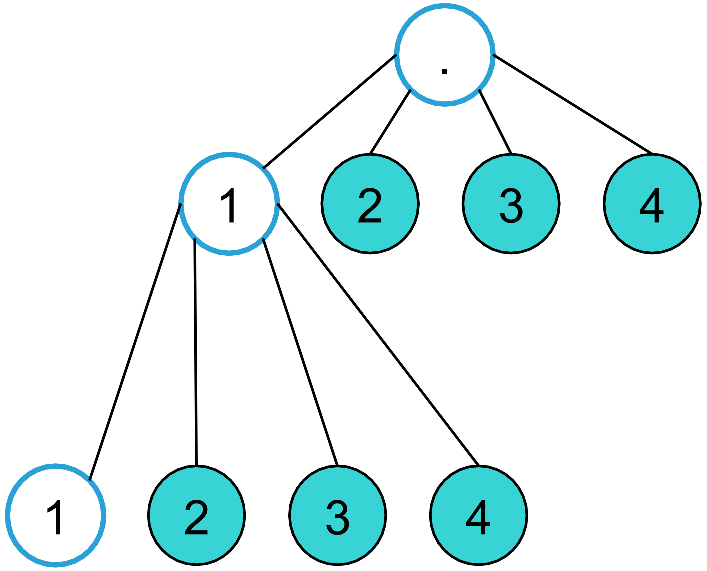


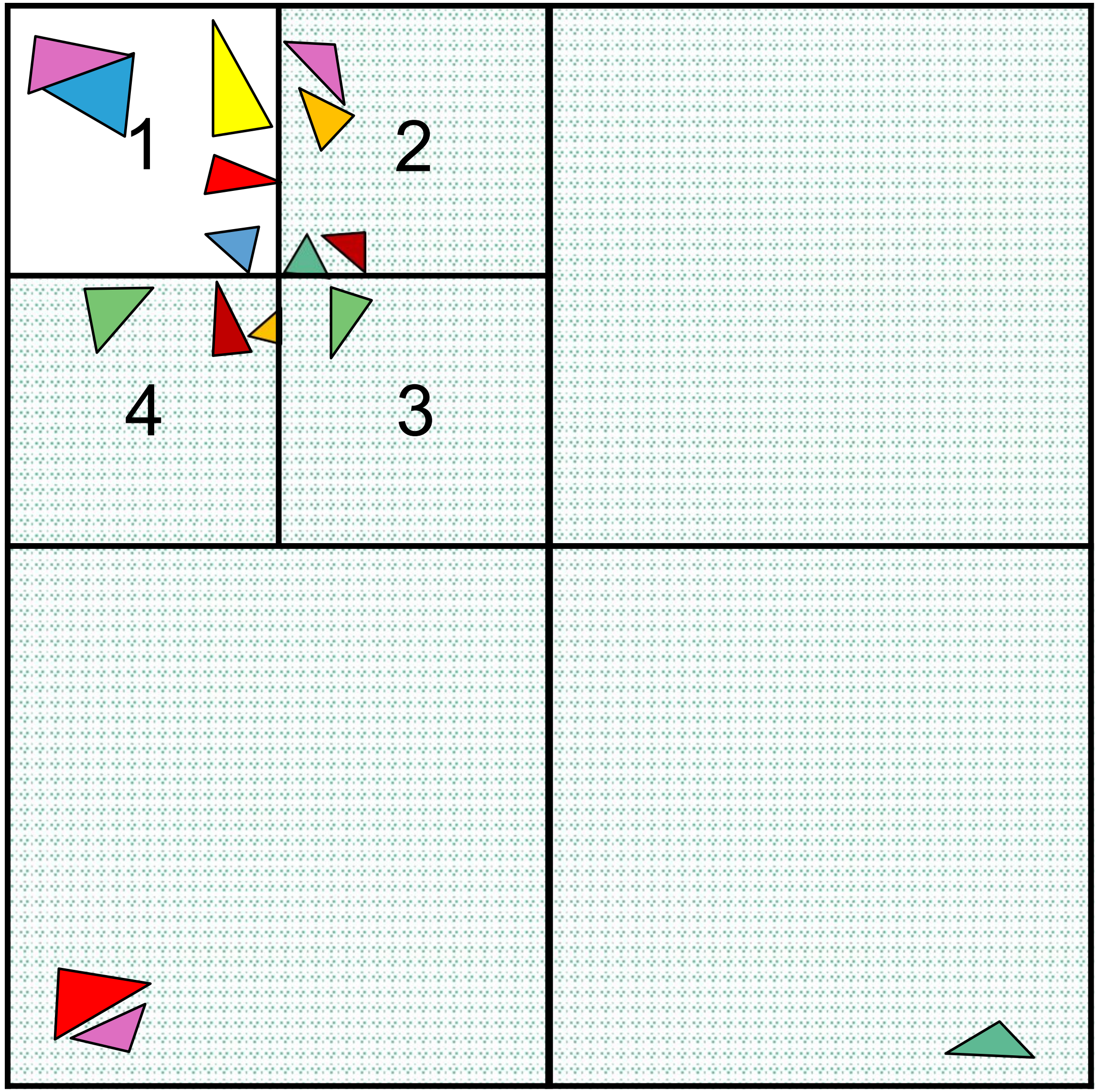
Quad and Octrees: $N_{leaf} = 4$



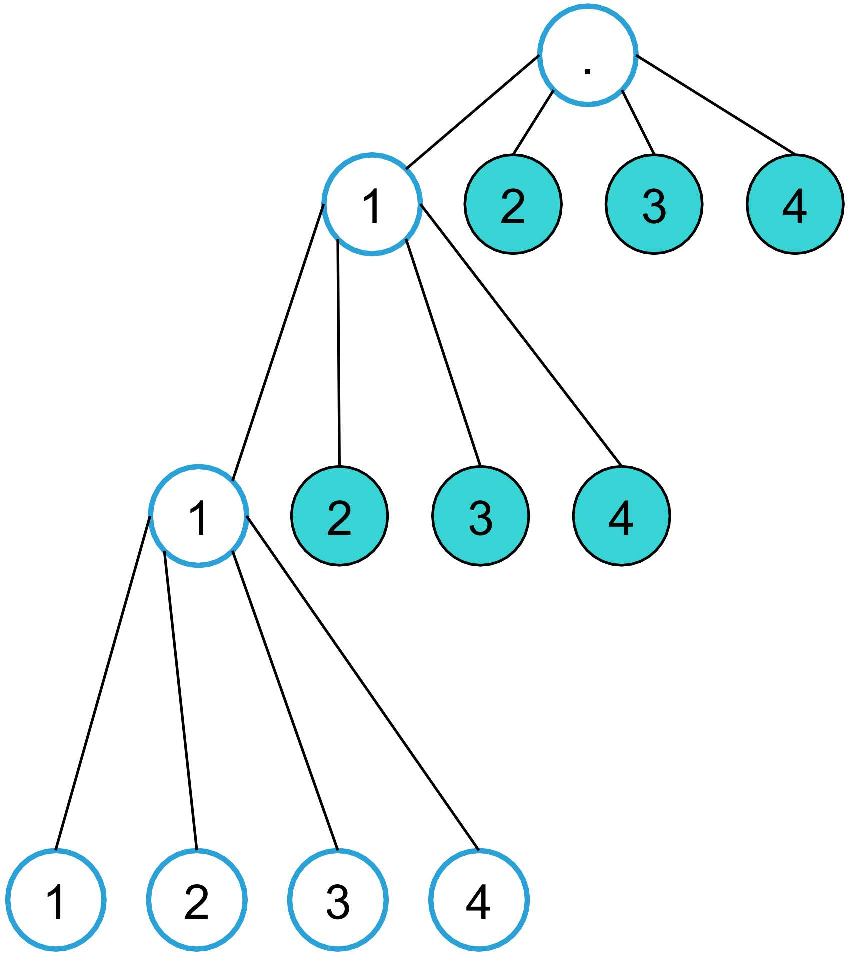


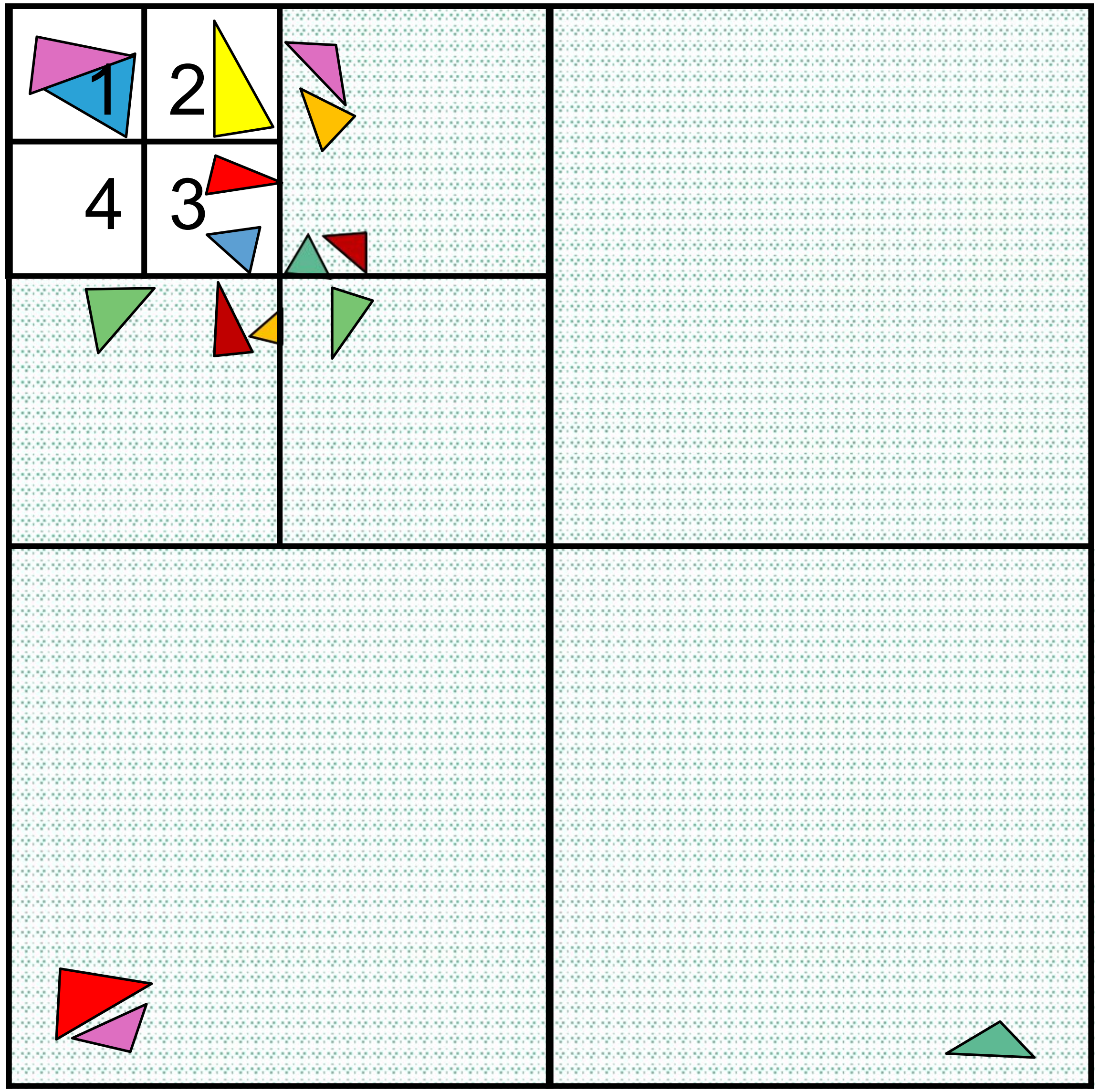
Quad and Octrees: $N_{leaf} = 4$



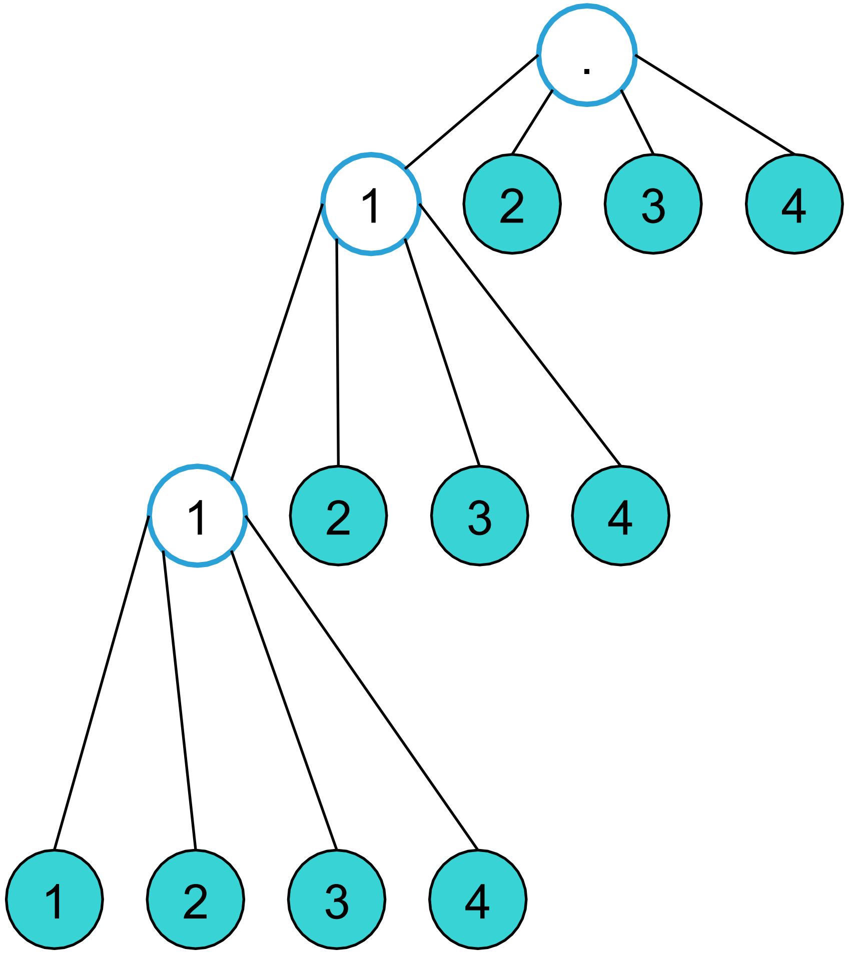


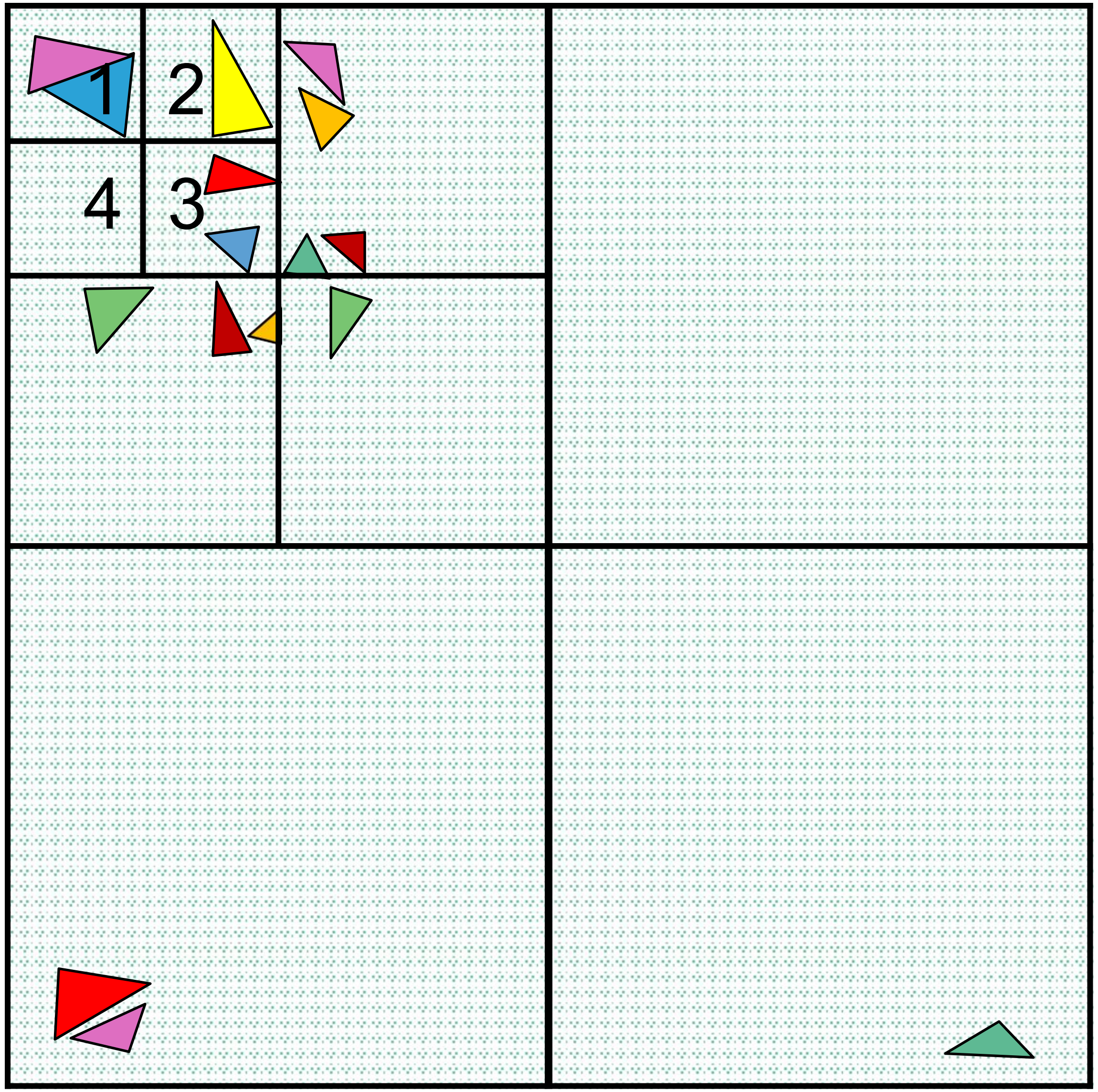
Quad and Octrees: $N_{leaf} = 4$





Quad and Octrees: $N_{leaf} = 4$

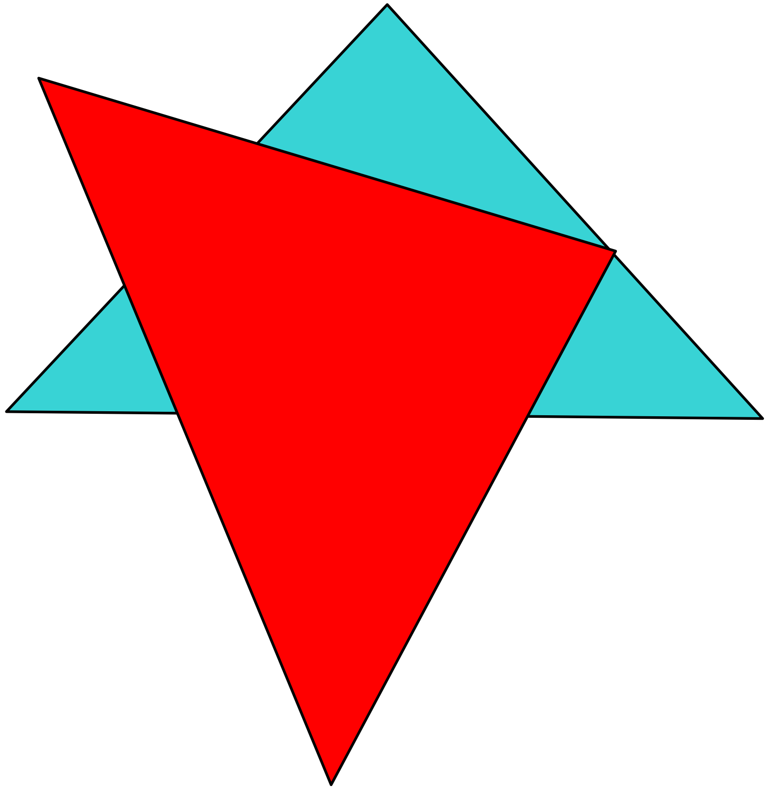




Quad and Octrees

- Triangles may not be contained within a quadrant or octant

- Triangles must be referenced in all overlapping cells or *split* at the border into smaller ones

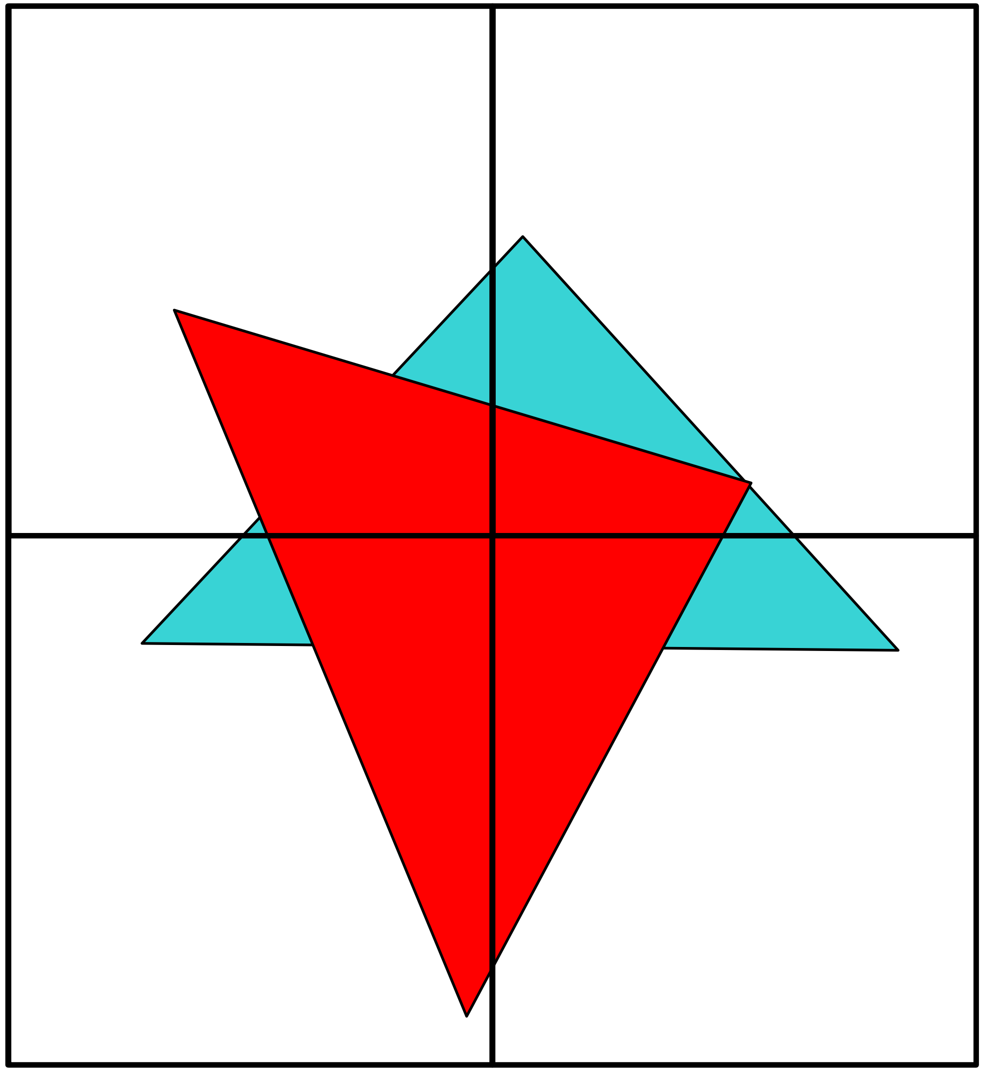


Quad and Octrees

- Triangles may not be contained within a quadrant or octant

- Triangles must be referenced in all overlapping cells or *split* at the border into smaller ones

- Can drastically increase memory consumption!

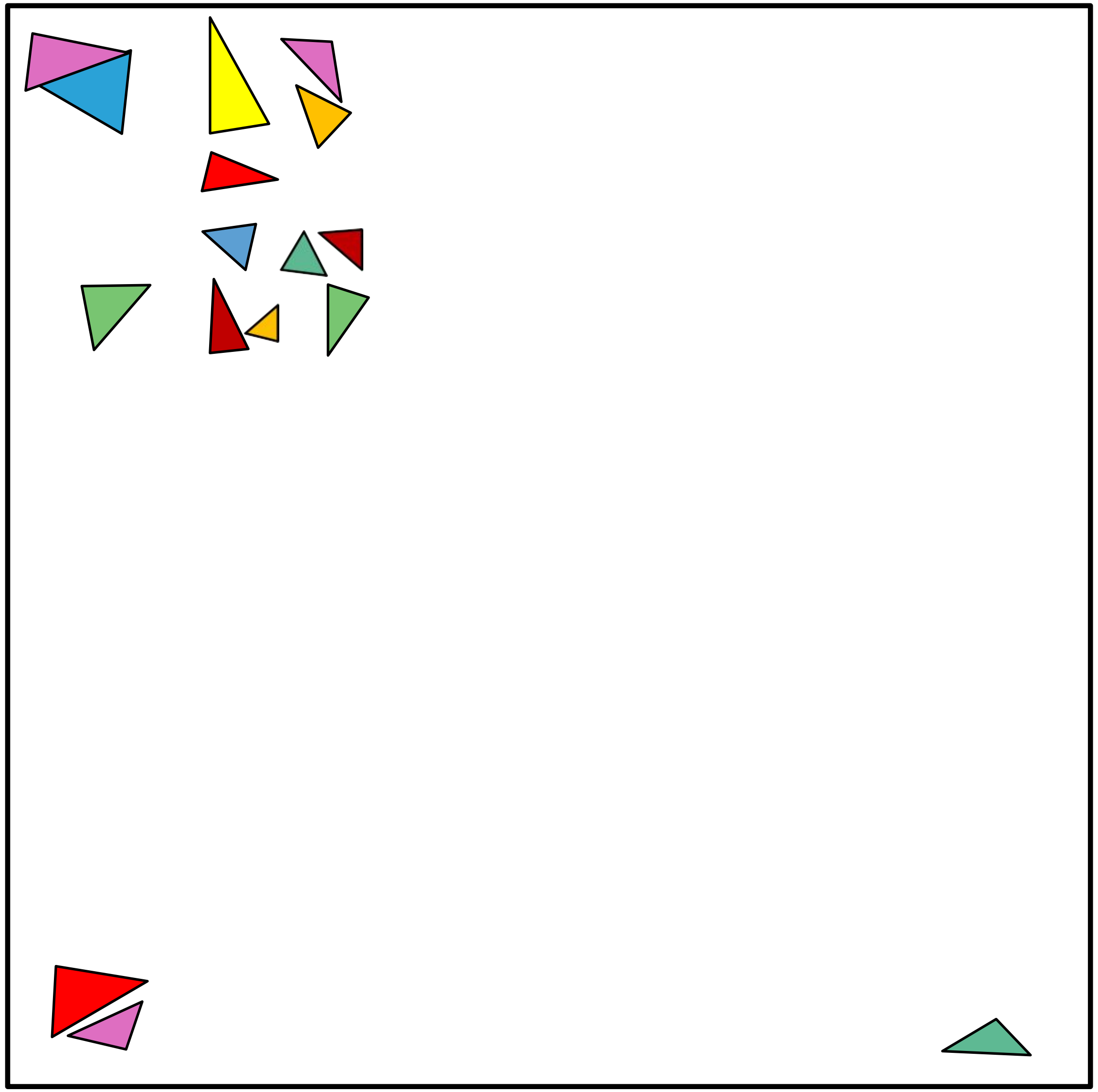


Spatial Acceleration Structures

Structure	Memory Consumption	Building Time	(Expected) Traversal Time
none	none	none	abysmal
Regular Grid	low – high (resolution)	low	uniform scene: ok otherwise: bad
Quadtree/Octree	low – high (overlap/uniformity)	low	good

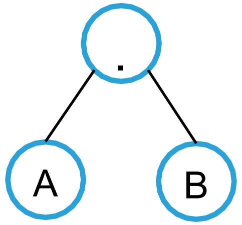
BSP Trees & K-d Trees

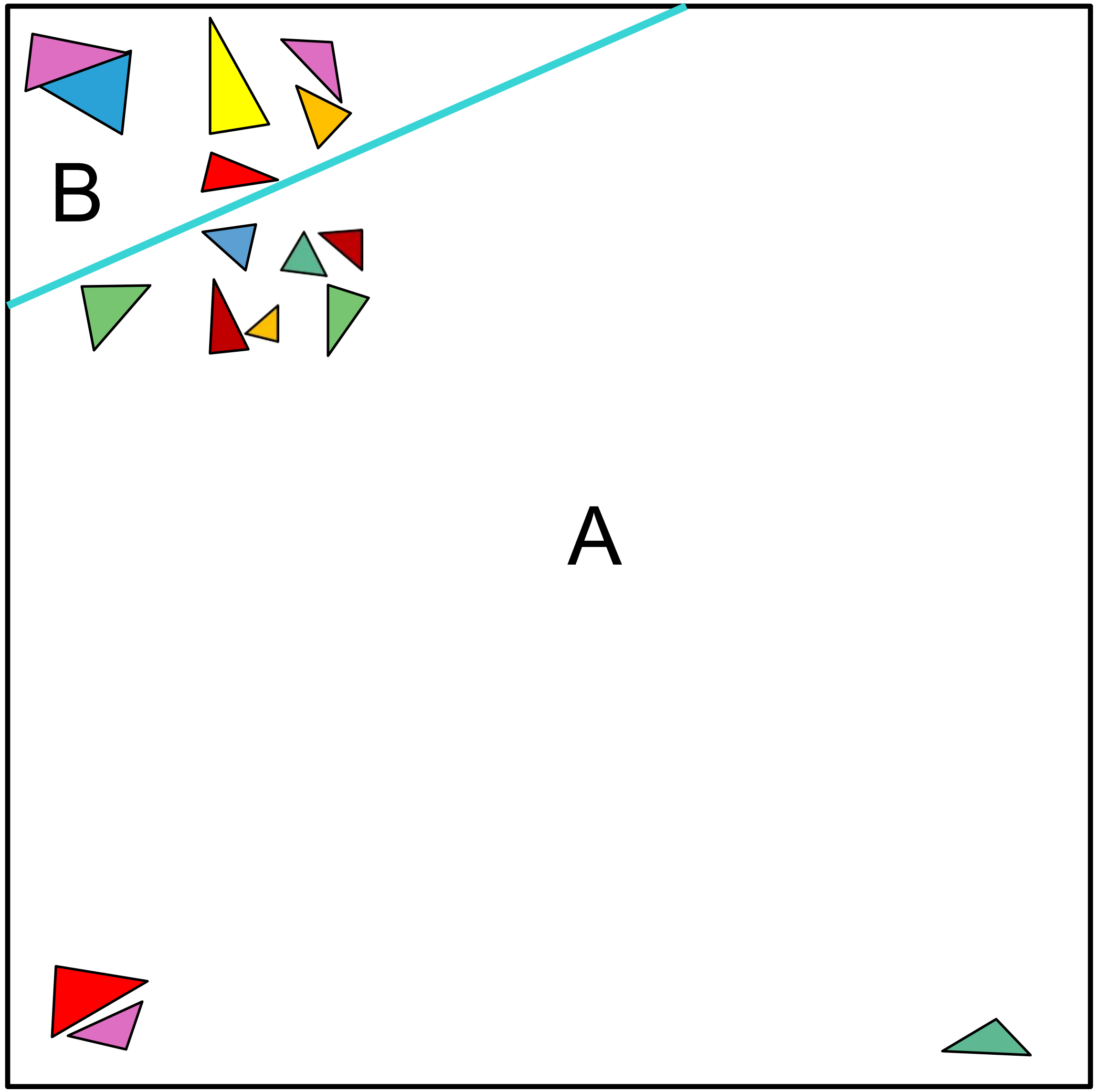
- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!



BSP Trees & K-d Trees, $N_{leaf} = 4$

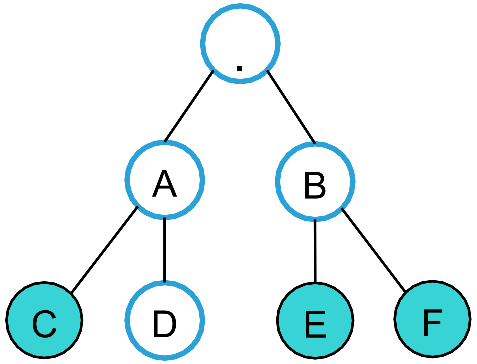
- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

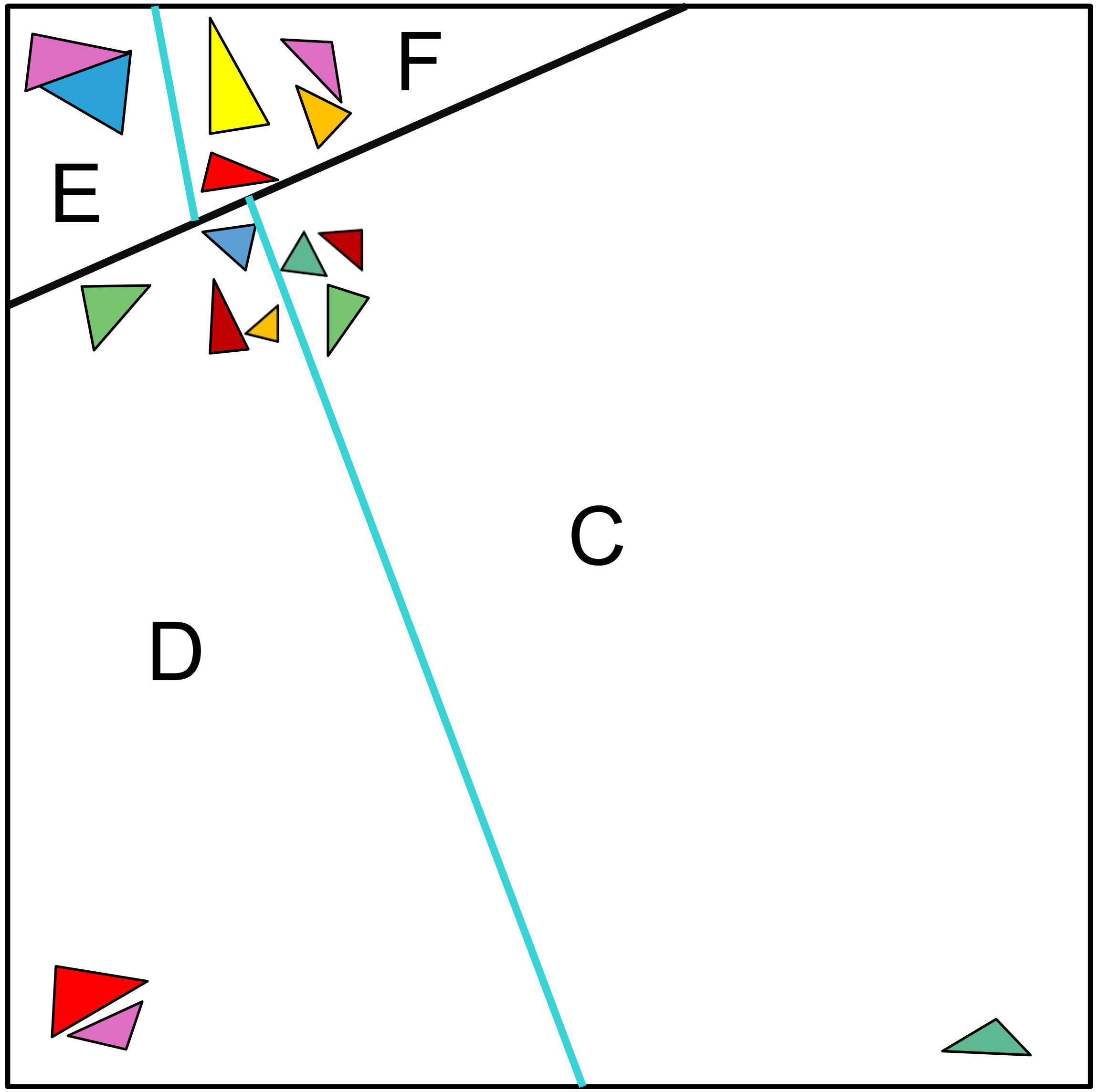




BSP Trees & K-d Trees, $N_{leaf} = 4$

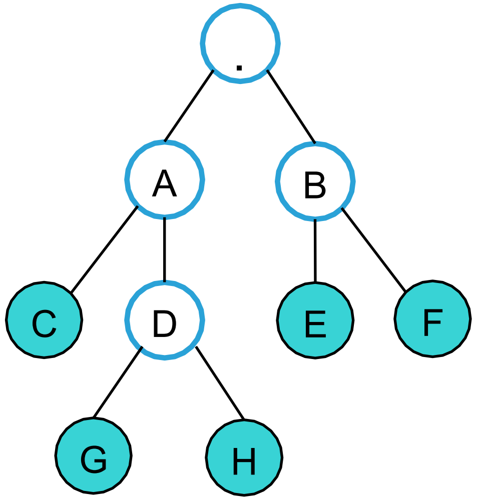
- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

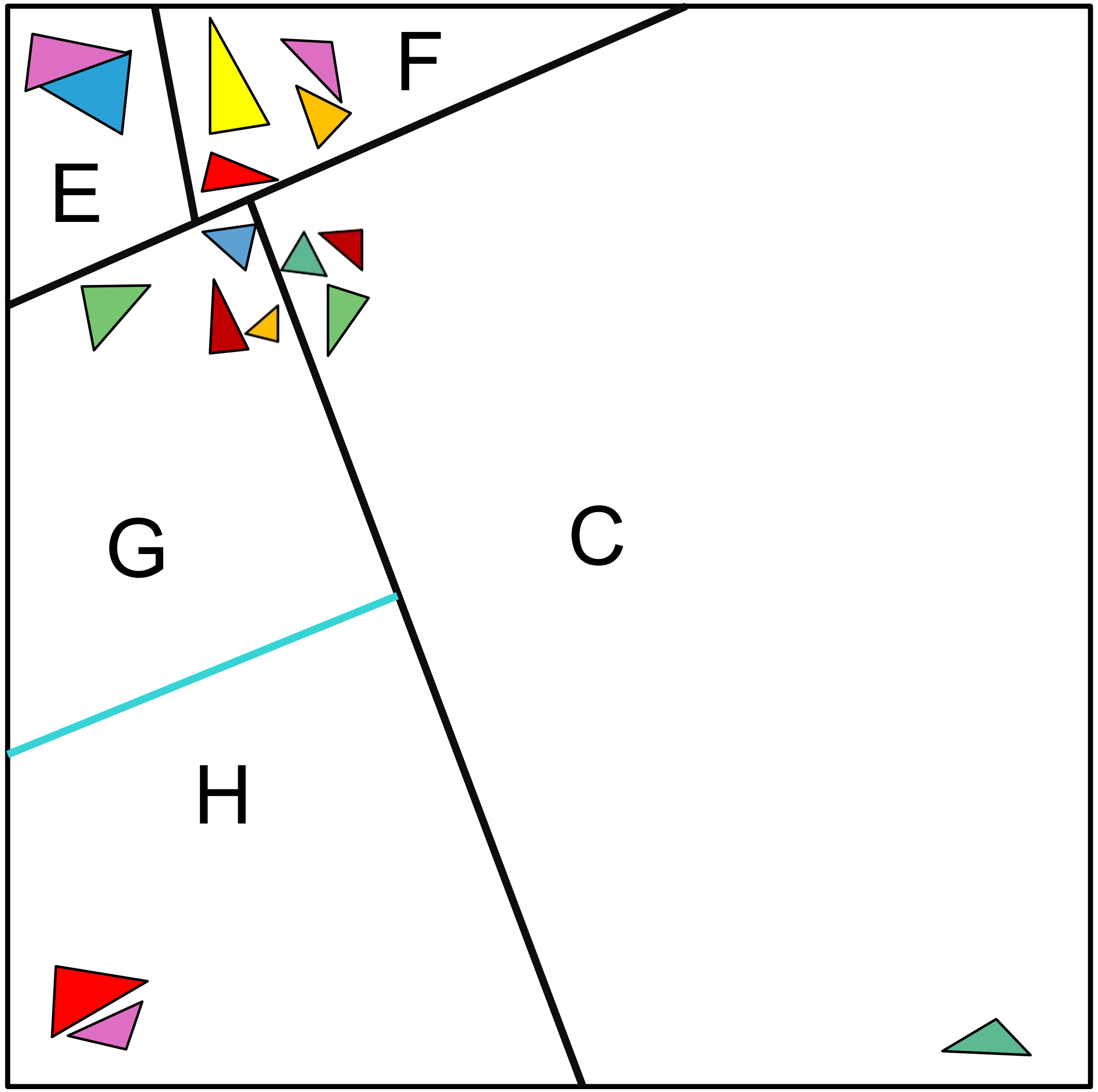




BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

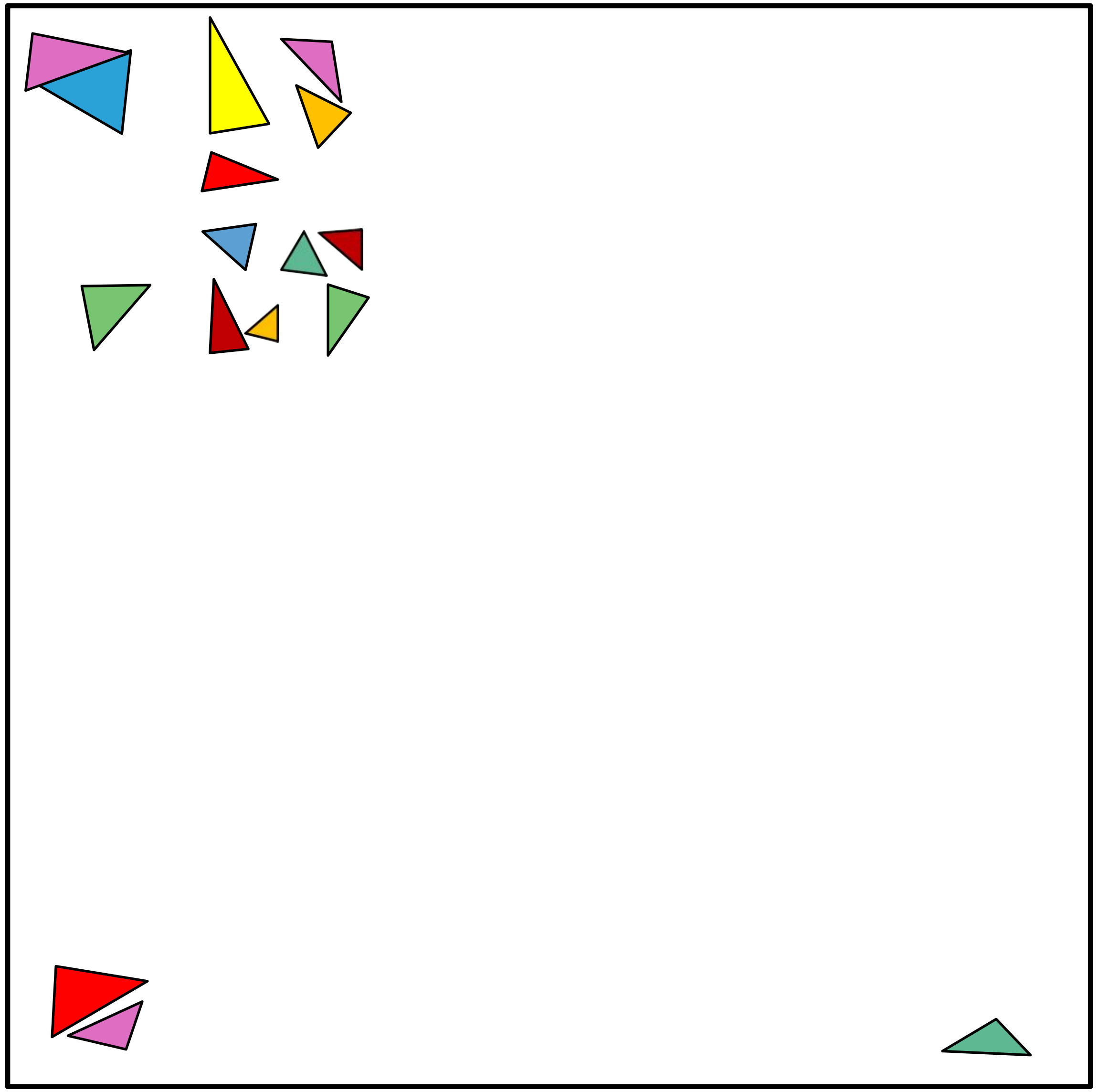




BSP Trees & K-d Trees

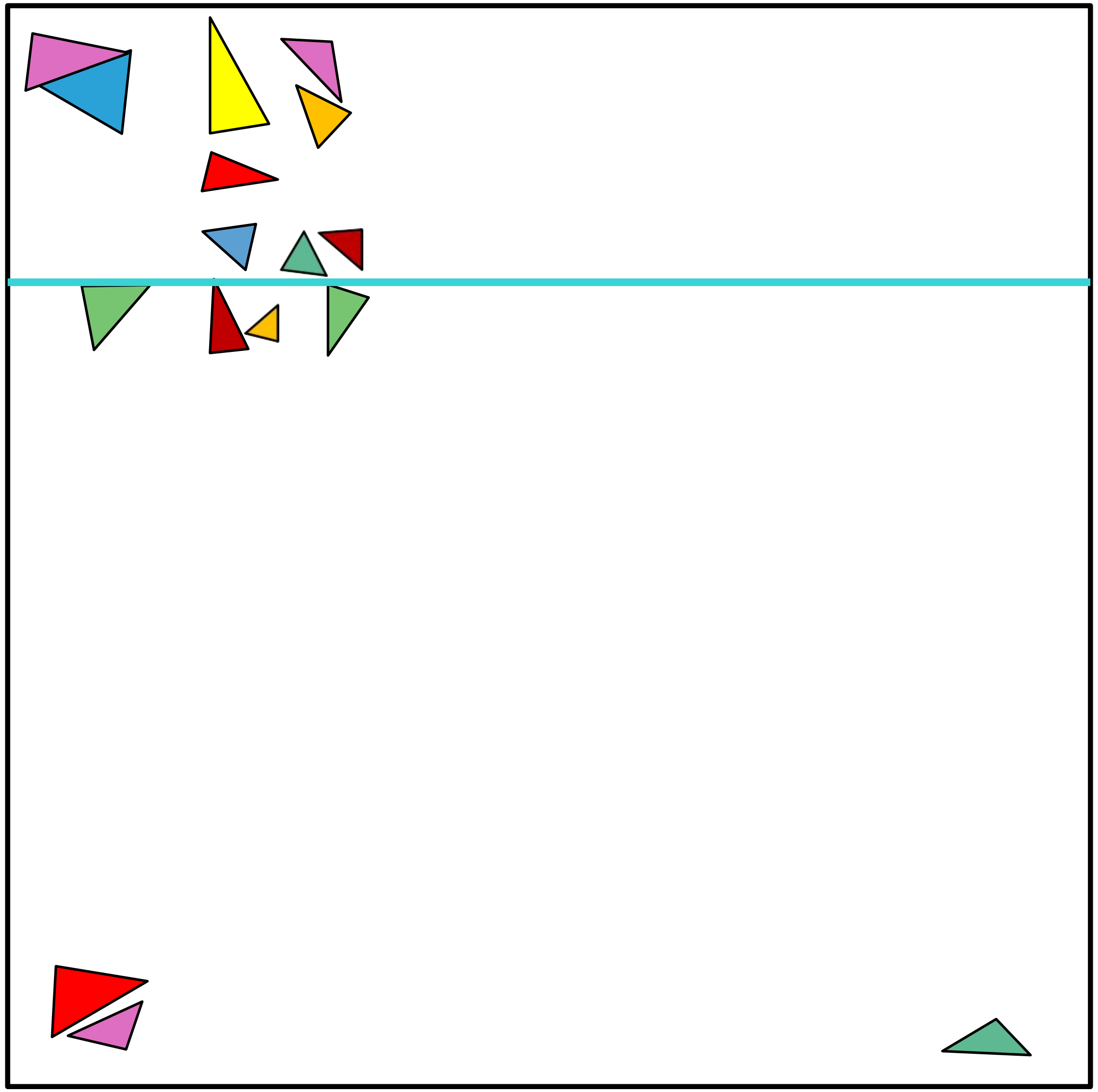
- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits



BSP Trees & K-d Trees, $N_{leaf} = 4$

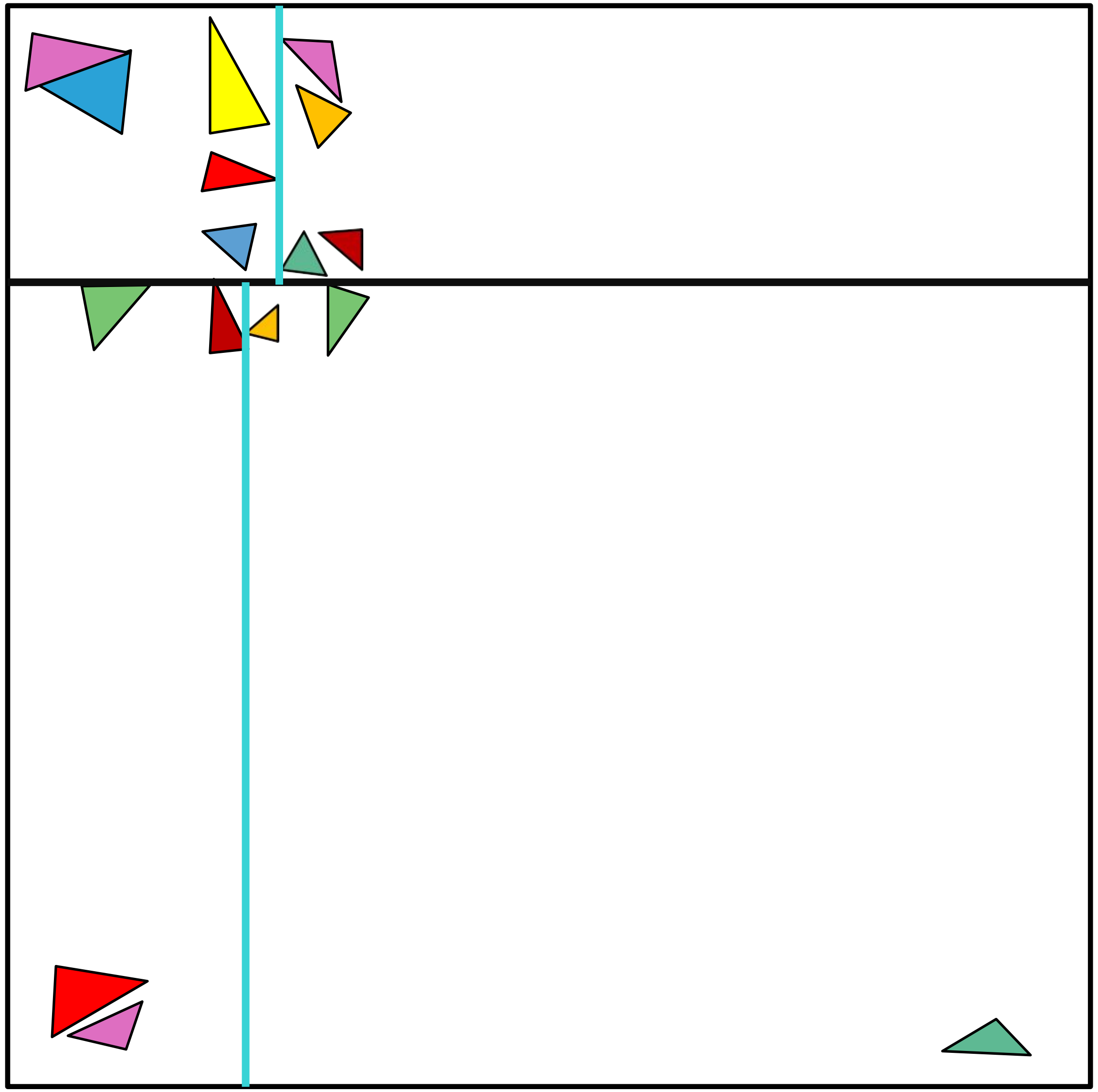
- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!
- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits



BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

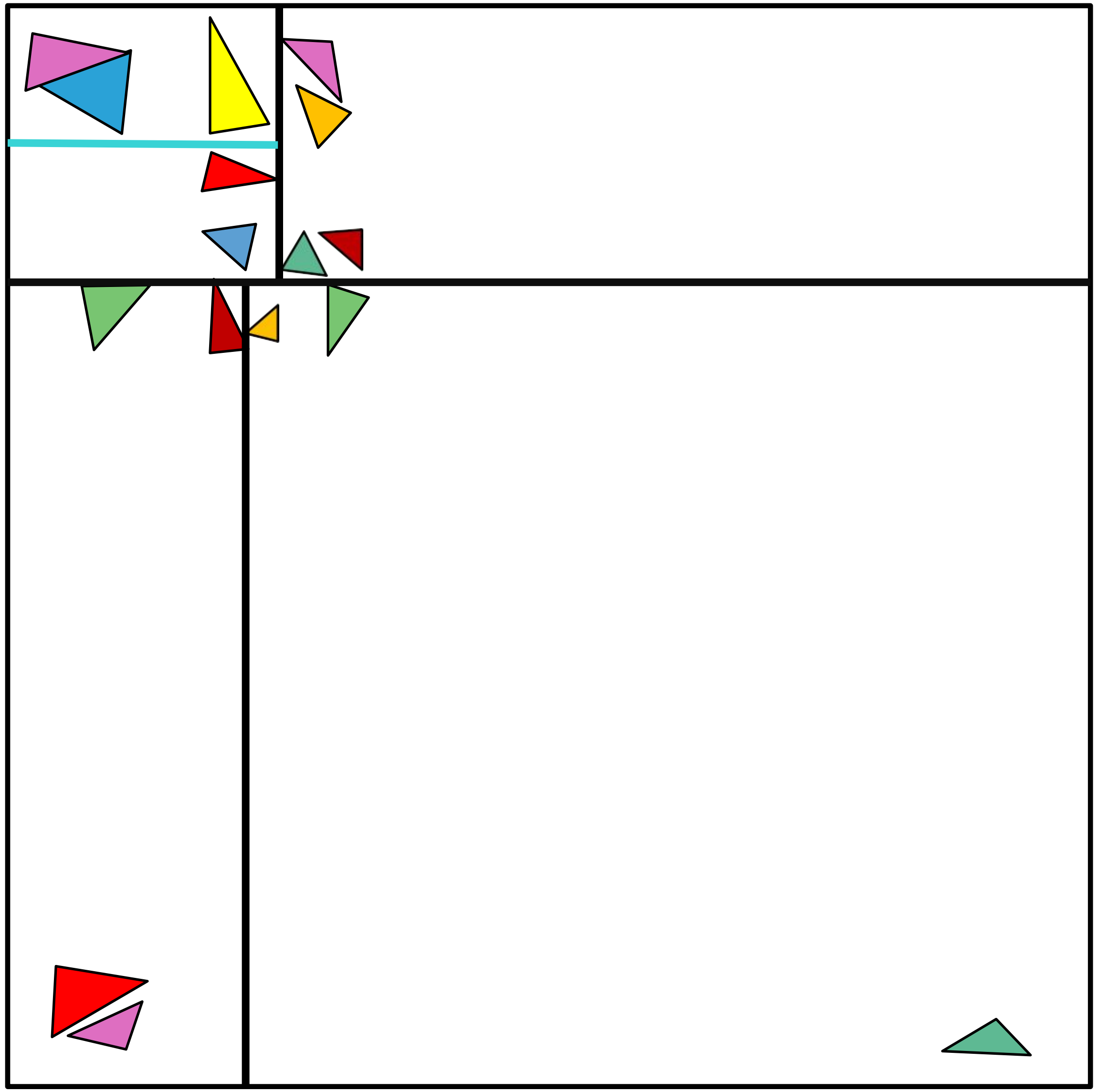
- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits



BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits



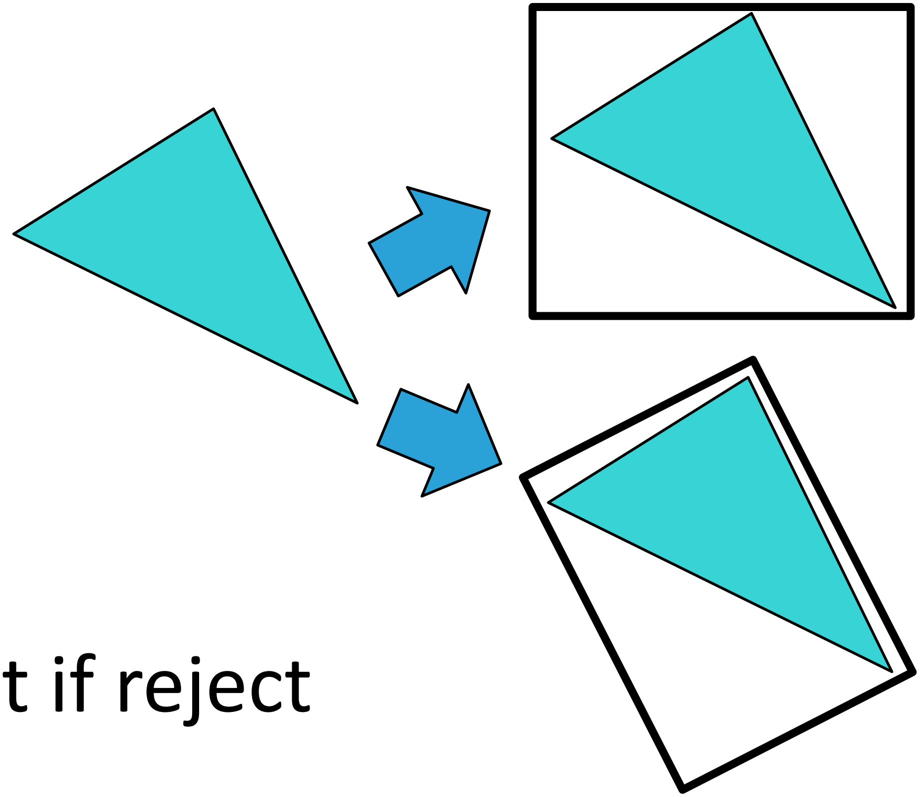
Spatial Acceleration Structures

Structure	Memory Consumption	Building Time	(Expected) Traversal Time
none	none	none	abysmal
Regular Grid	low – high (resolution)	low	uniform scene: ok otherwise: bad
Quadtree/Octree	low – high (overlap/uniformity)	low	good
K-d Tree	low – high (overlap)	low – high	good – excellent

Bounding Volumes

- Find enclosing ("conservative") volumes that are easier to test Ideally: tight, but easy to check for intersection with ray
-

  Common choices:
-
  - Bounding Spheres
  - Bounding Boxes
    - Axis-aligned (AABB)
    - Oriented (OBB)

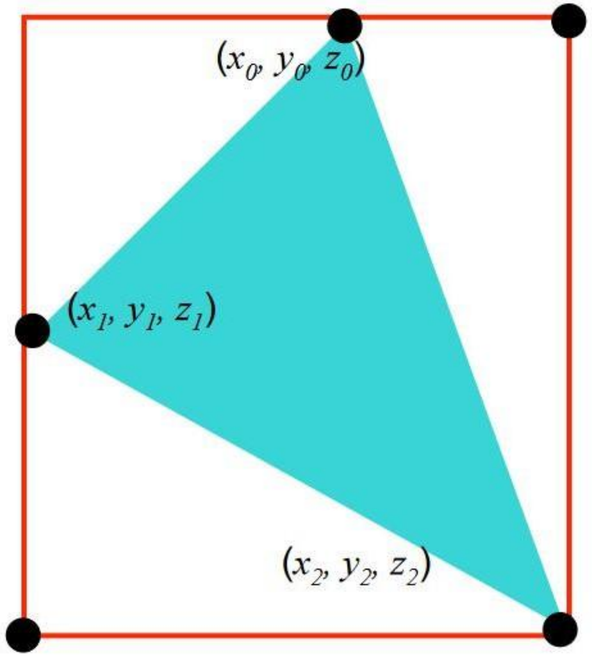


- Saves on computational effort if reject

Axis-Aligned Bounding Boxes (AABBs)

- AABBs are defined by their two extrema (min/max)

- Linear run time to compute
  - Iterate over all vertices
  - Keep min/max values for each dimension
  - Done!



$$(x_{max}, y_{max}, z_{max}) = (\max(x_0,x_1,x_2), \max(y_0,y_1,y_2), \max(z_0,z_1,z_2))$$

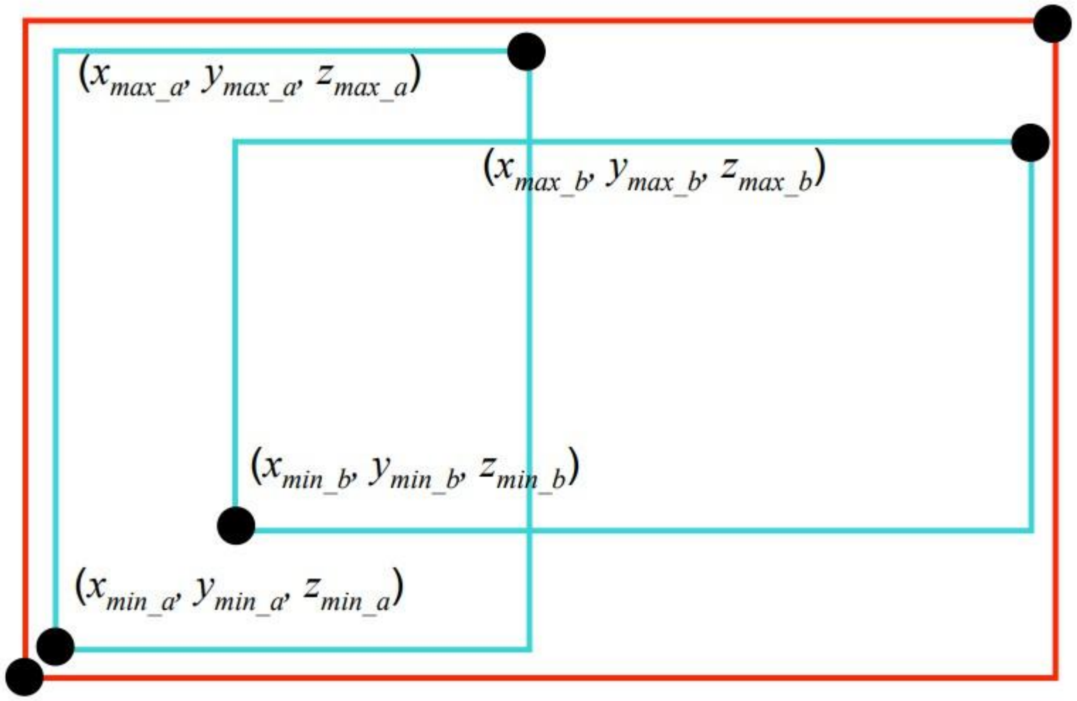
$$(x_{min}, y_{min}, z_{min}) = (\min(x_0,x_1,x_2), \min(y_0,y_1,y_2), \min(z_0,z_1,z_2))$$

Merging AABBs

- Find the AABB that encloses multiple, smaller AABBs

- Operates only on extrema of each smaller AABB

- Merging process is commutative


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

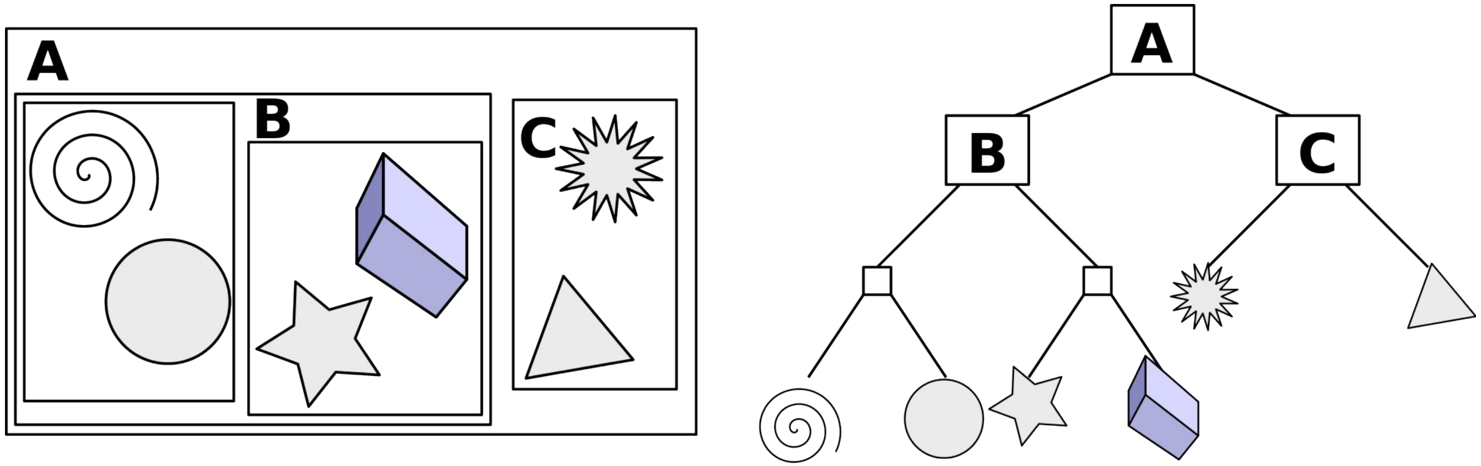
$$(x_{max}, y_{max}, z_{max}) = (max(x_{max\_a}, x_{max\_b}), max(y_{max\_a}, y_{max\_b}), max(z_{max\_a}, z_{max\_b}))$$

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

Bounding Volume Hierarchy (BVH)

- The final hierarchy is (again) a tree structure with N leaf nodes

- Leaf nodes can be
  - Individual triangles
  - Clusters (e.g., $\leq 10\Delta$)

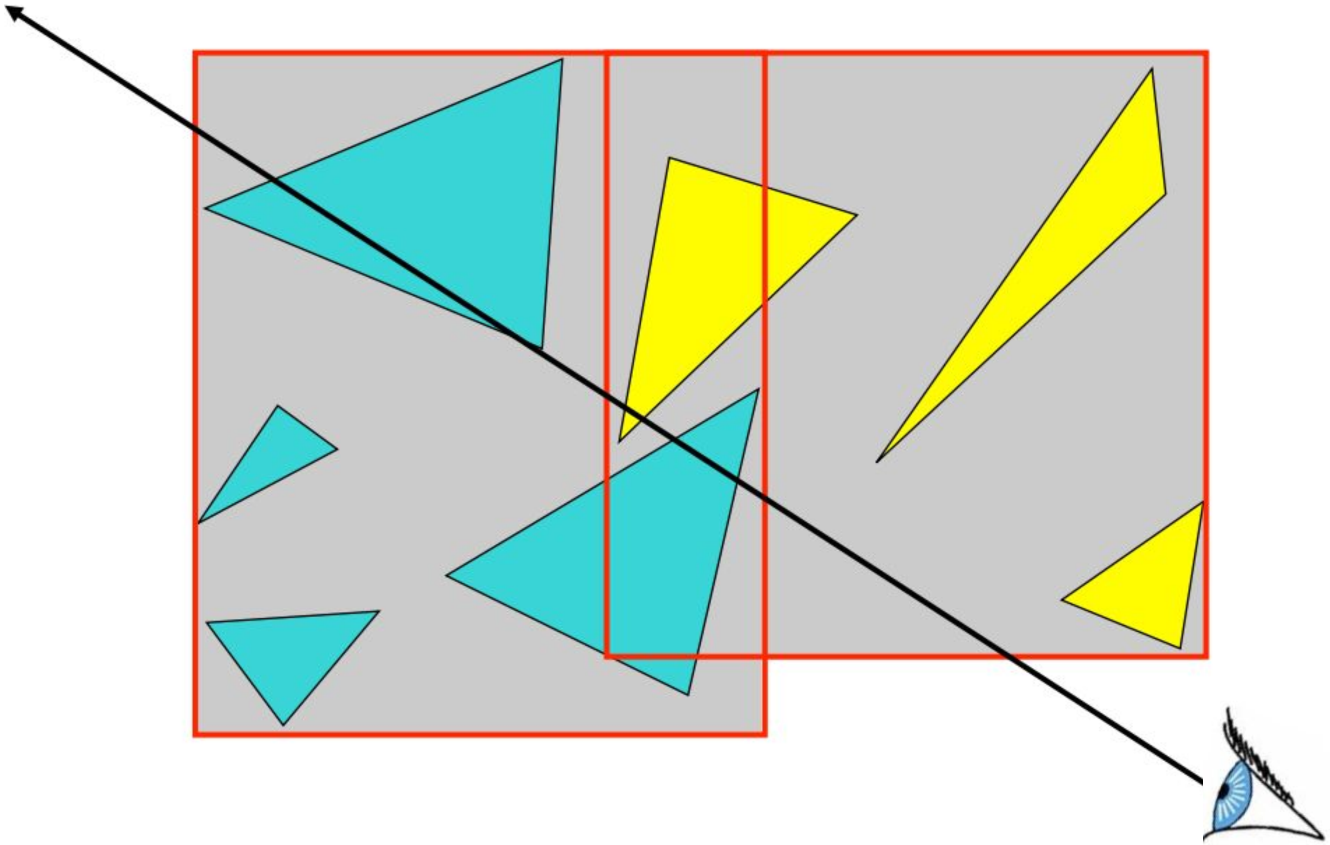


Source: Schreiberx, Wikipedia “Bounding Volume Hierarchy”

- Total number of nodes for a binary tree: $2N - 1$
  - If balanced, it takes $\sim \log N$ steps to reach a leaf from the root
  - If trees have more than 2 branches, they require fewer nodes

What makes BVHs special?

- Important feature: bounding volumes can ***overlap!***
- No duplicate references or split triangles necessary!
- Implicitly limits the amount of memory required

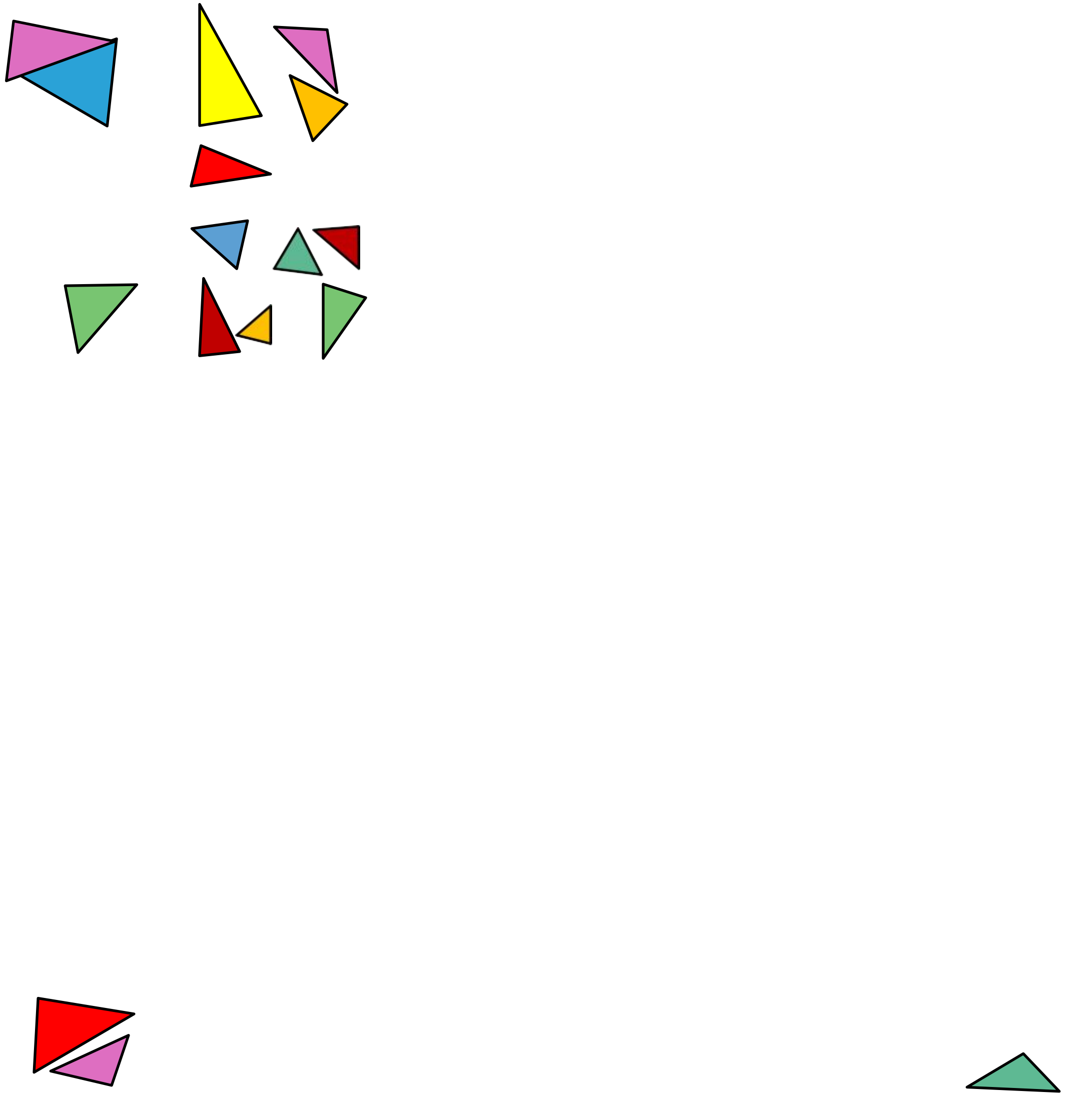


BVH Building

- Generating BVH and tree for input triangle geometry

- CPU: usually top-down
  GPU: usually bottom-up

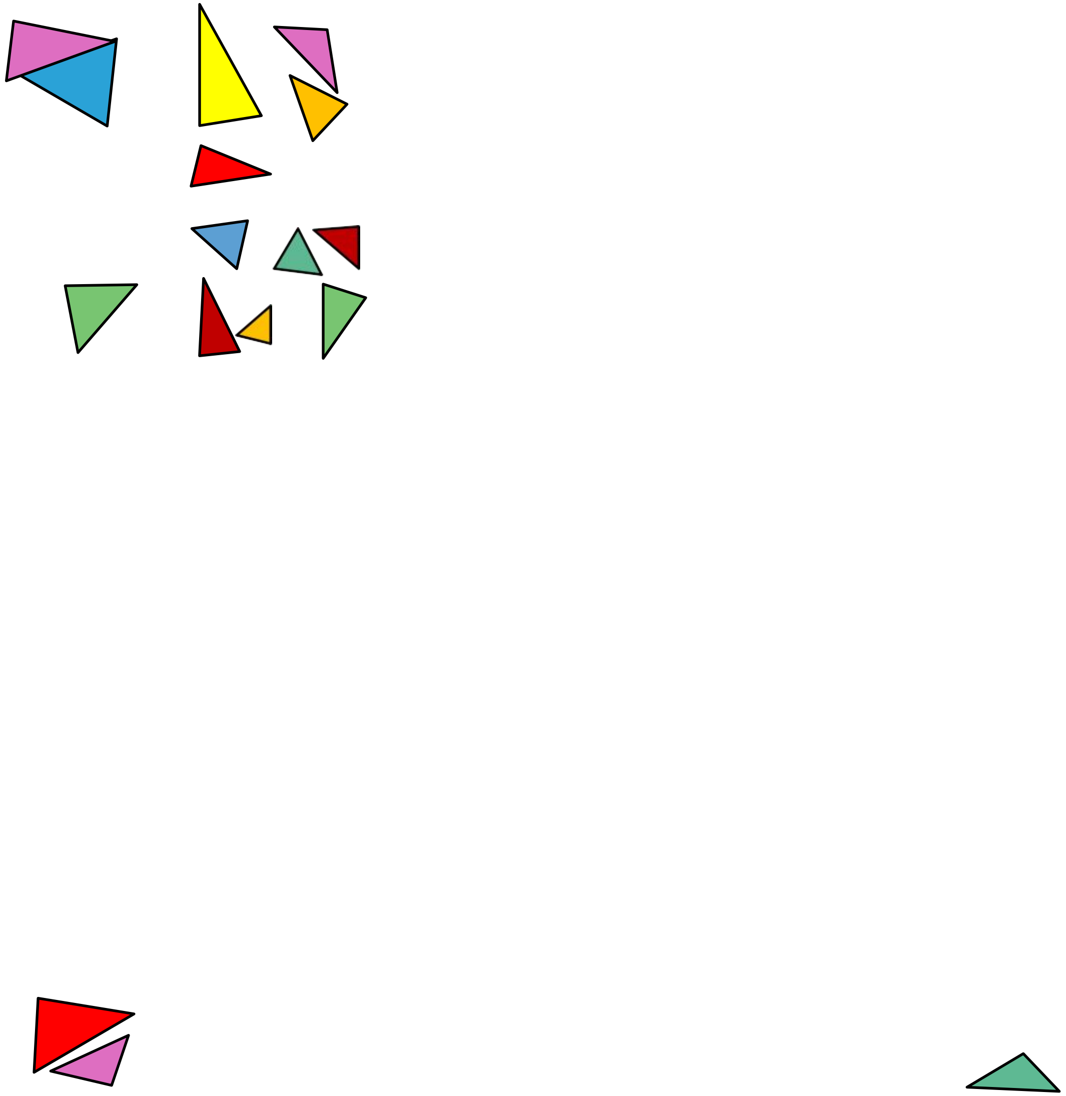
- From here on out, we will consider box BVHs only



BVH Building, Top-Down

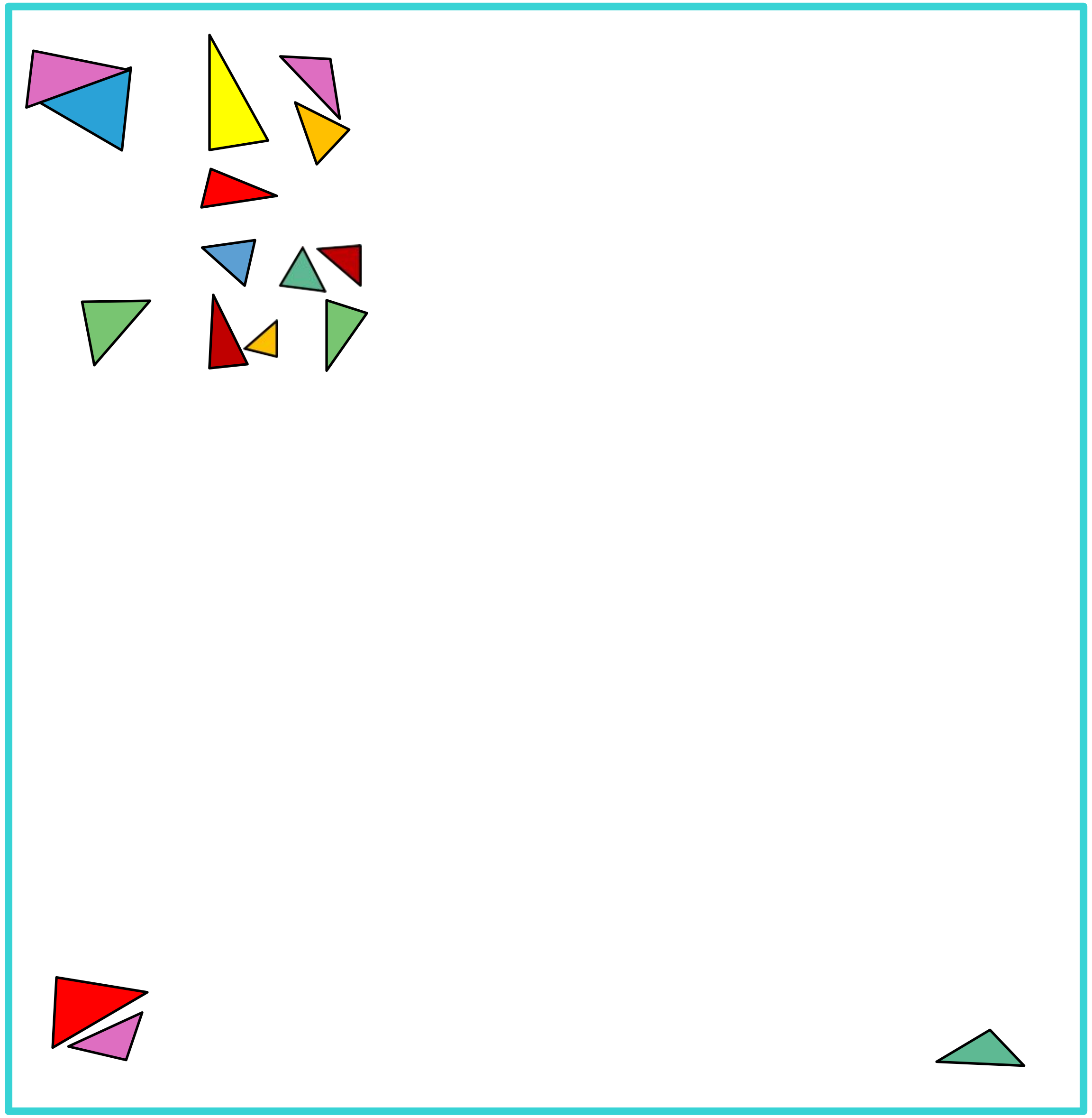
- Define N_{leaf} for leaves

- For each node, do the following:
  - Compute bounding box that fully encloses triangles & store
  - Holds $\leq N_{leaf}$ triangles? Stop.
  - Else, split into child groups
  - Make one new node per group
  - Set them as children of current
  - Repeat with child nodes



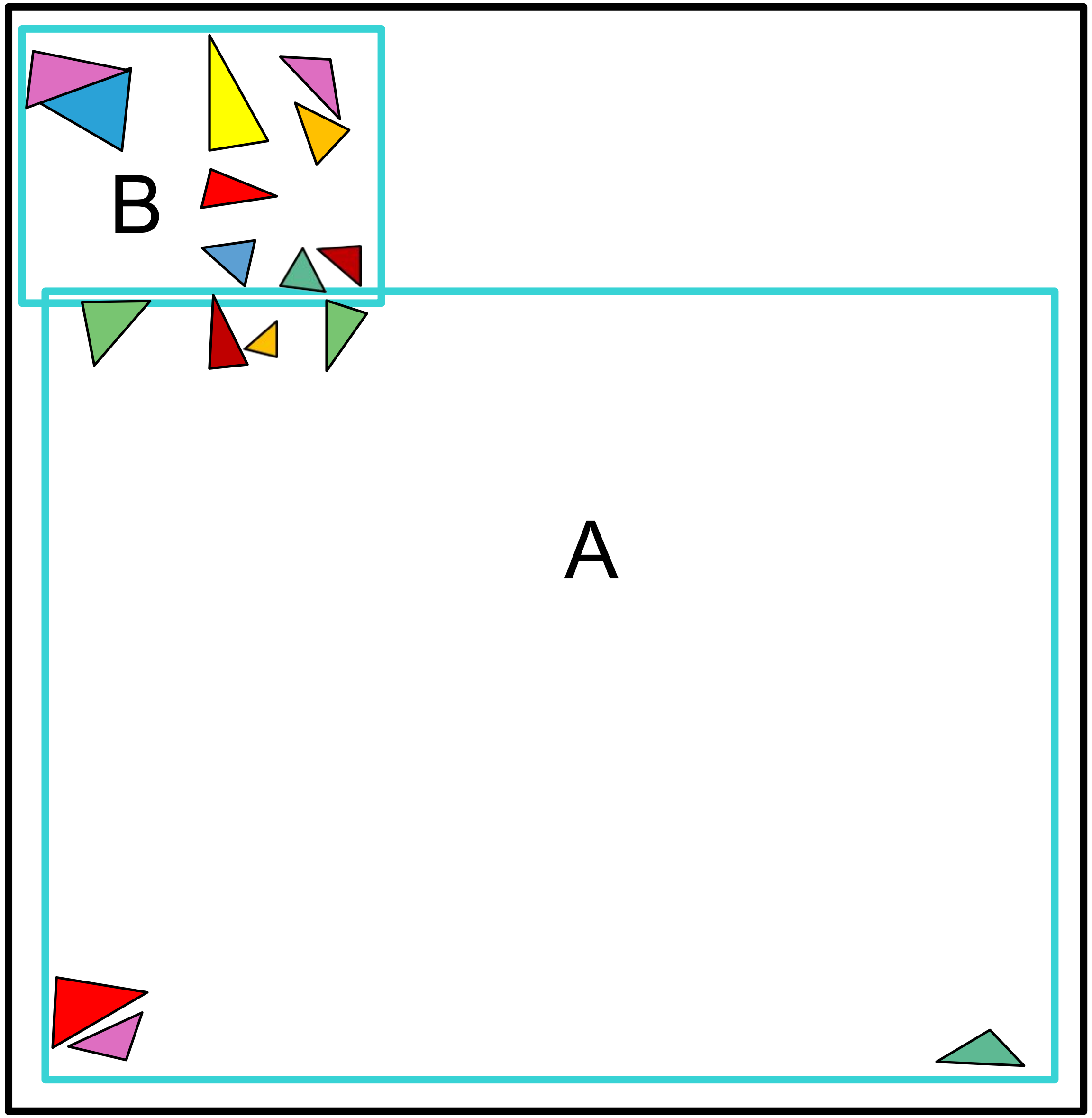
BVH Building, Top-Down, $N_{leaf} = 4$

- Define N_{leaf} for leaves
- For each node, do the following:
  - Compute bounding box that fully encloses triangles & store
  - Holds $\leq N_{leaf}$ triangles? Stop.
  - Else, split into child groups
  - Make one new node per group
  - Set them as children of current
  - Repeat with child nodes

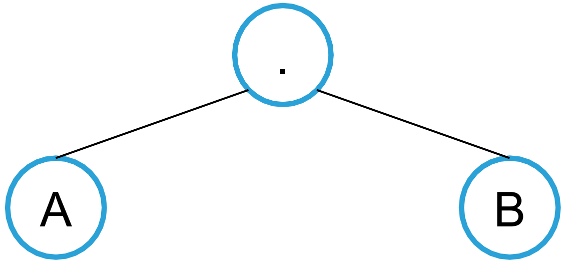


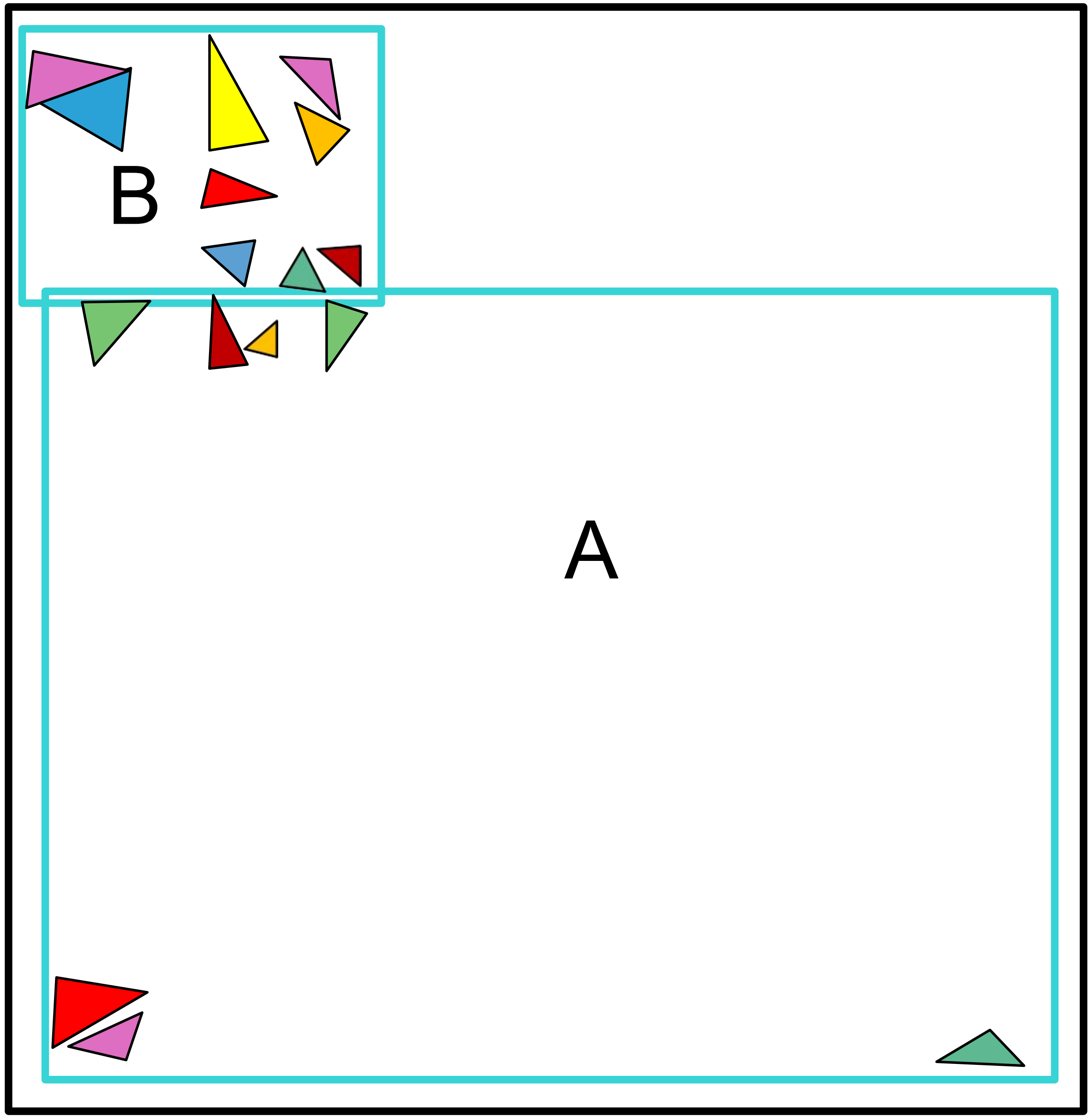
BVH Building, Top-Down, $N_{leaf} = 4$

- Define N_{leaf} for leaves
- For each node, do the following:
  - Compute bounding box that fully encloses triangles & store
  - Holds $\leq N_{leaf}$ triangles? Stop.
  - Else, split into child groups
  - Make one new node per group
  - Set them as children of current
  - Repeat with child nodes

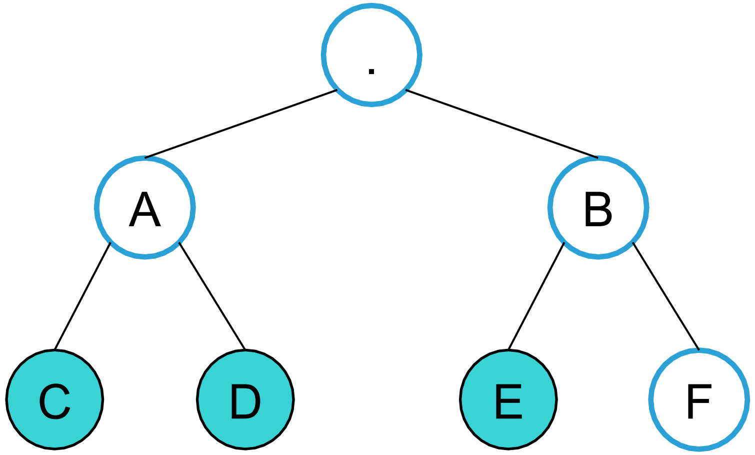


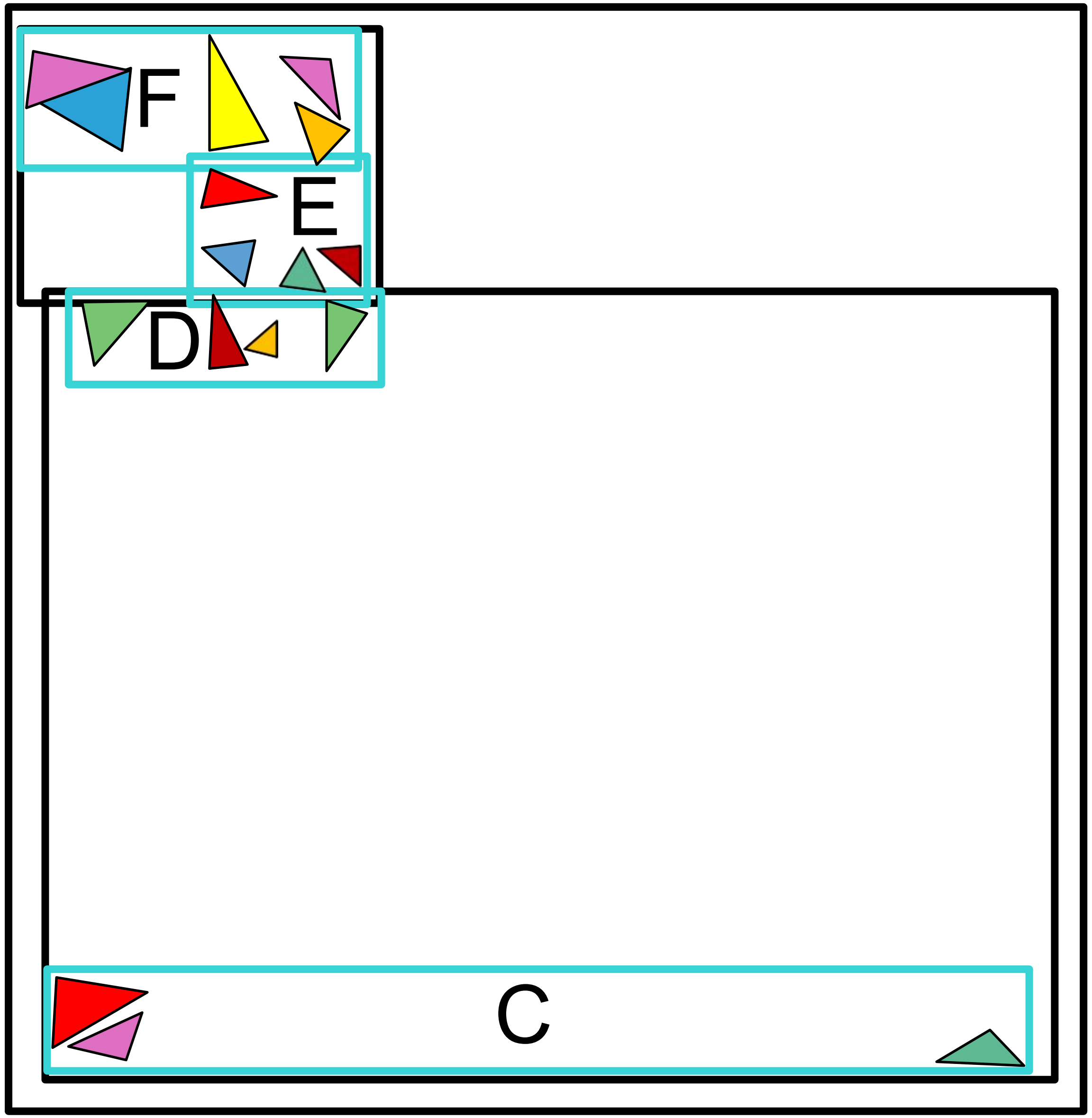
BVH Building, Top-Down, $N_{leaf} = 4$



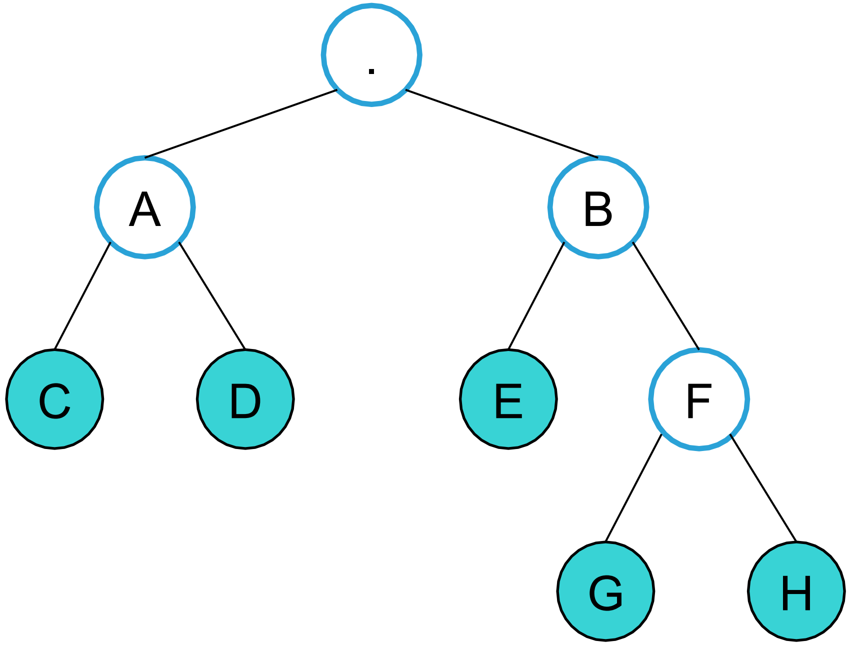


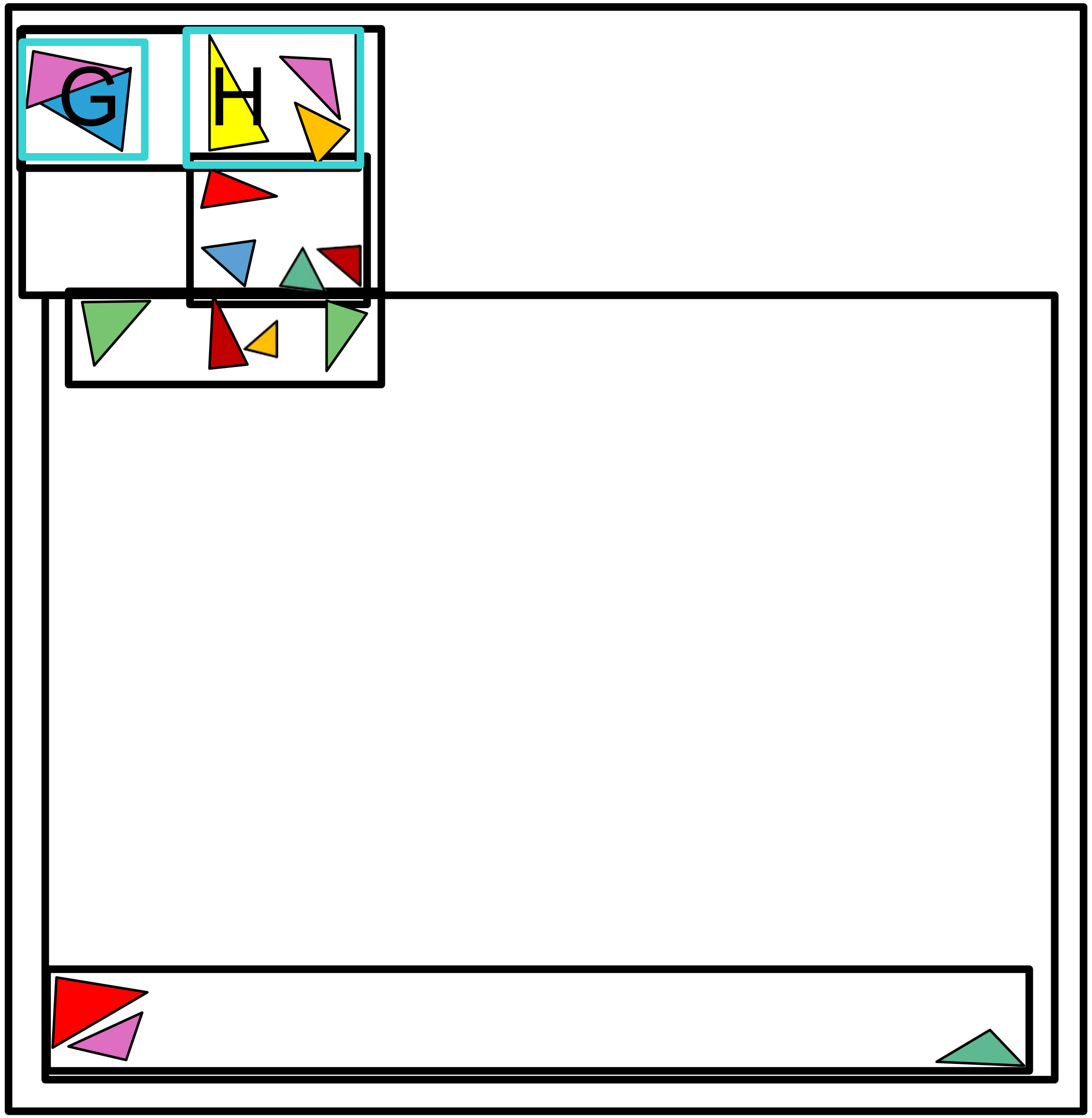
BVH Building, Top-Down, $N_{leaf} = 4$





BVH Building, Top-Down, $N_{leaf} = 4$





How to split a node?

- Which axes to consider for building bounding boxes/splitting?
  - Basis vectors $(1,0,0)$, $(0,1,0)$, $(0,0,1)$
  - only Oriented basis vectors only
  - Arbitrary
- Where to split?
  - Spatial median
  - Object median
  - Something more elaborate...

How to split a node?

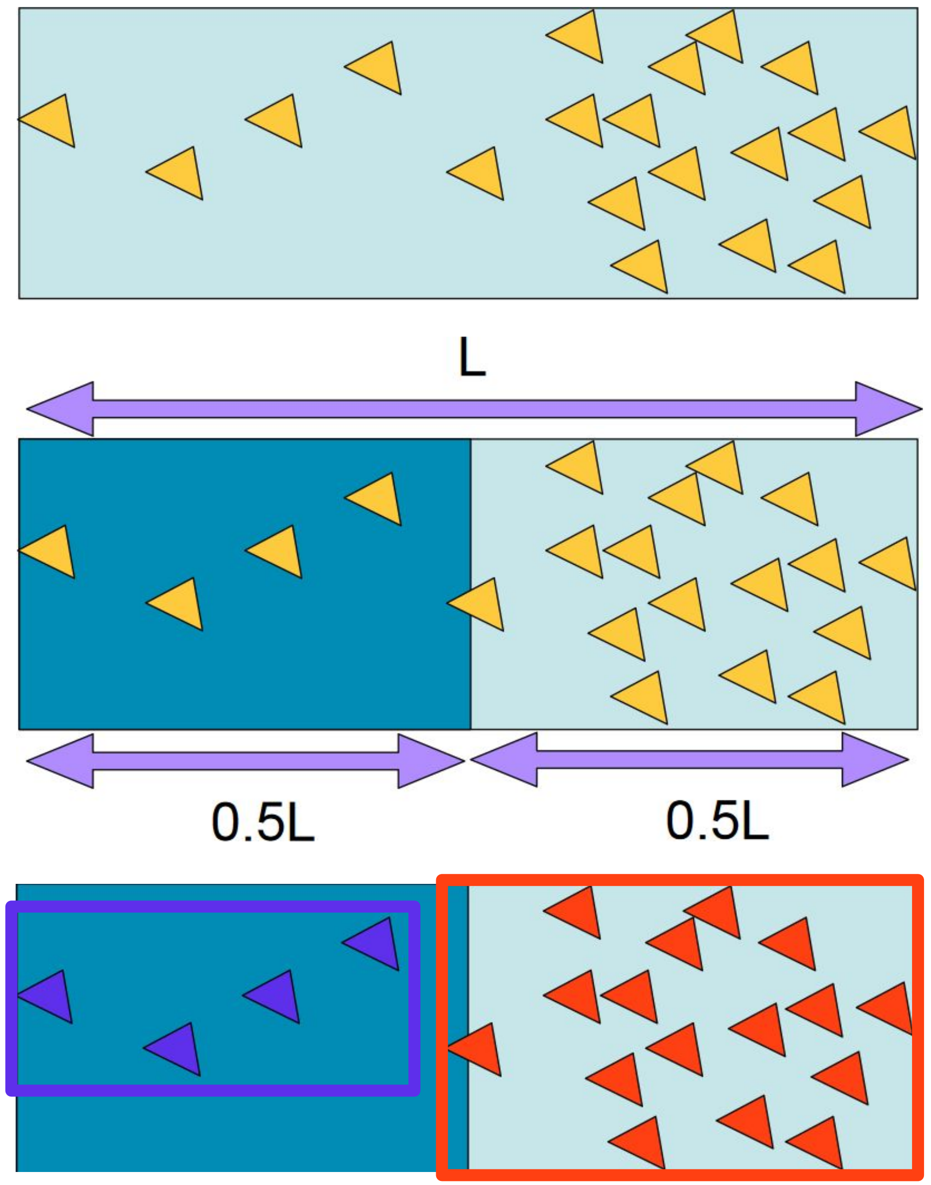
- Which axes to consider for building bounding boxes/splitting?
  - Basis vectors $(1,0,0)$, $(0,1,0)$, $(0,0,1)$
  - only Oriented basis vectors only
  - Arbitrary

Algorithms exist (e.g. “separating axis theorem”), but usually very slow!

- Where to split?
  - Spatial median
  - Object median
  - Something more elaborate...

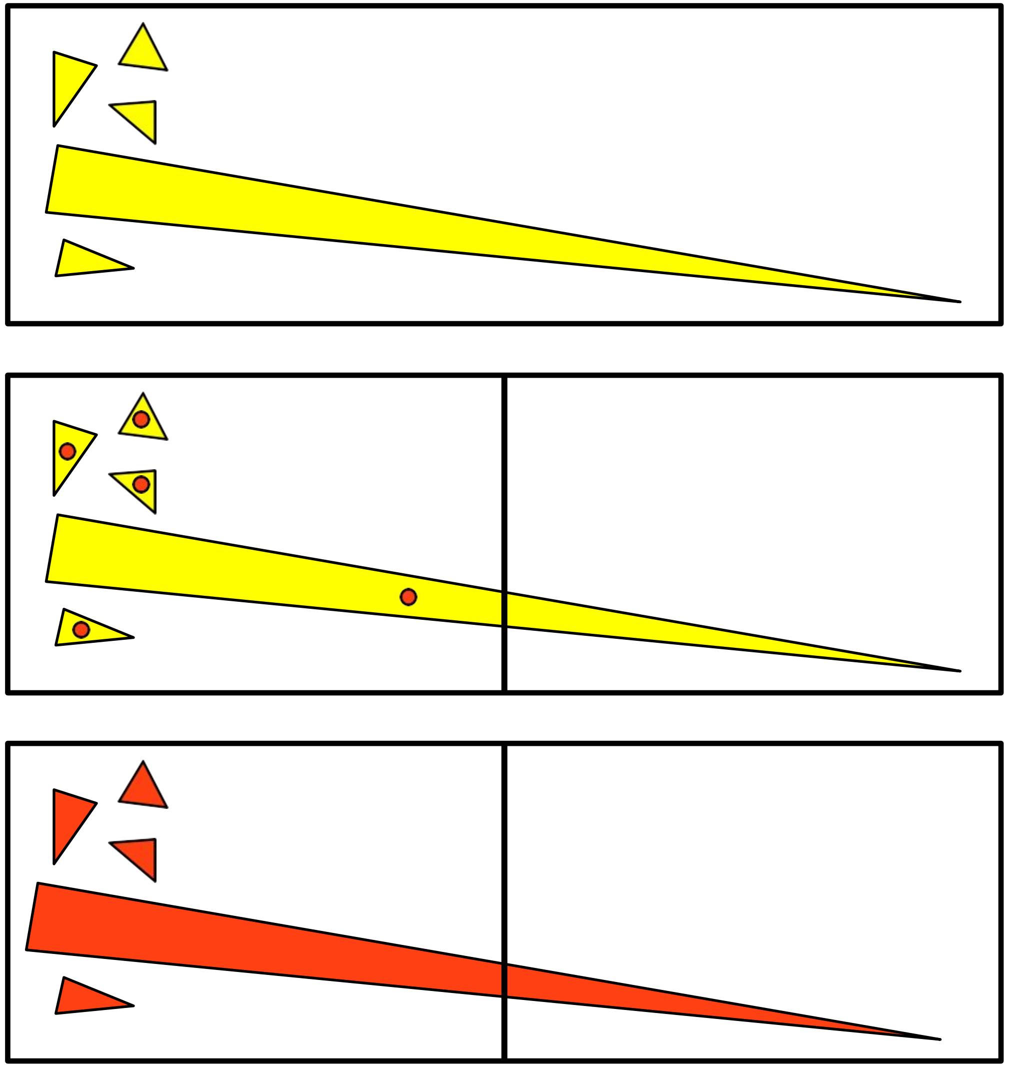
Splitting at spatial median

- Pick the longest axis (X/Y/Z) of current node bounds
- Find the midpoint on that axis
- Assign triangles to A/B based on which side of the midpoint each triangle's *centroid* lies on
- Continue recursion with A/B



Splitting at spatial median

- Careful: can result in infinite recursion!
- All triangles are assigned again to one node, none in the other
- Can guard against it in several ways
  - Limit max. number of split attempts
  - Try other axes if one node is empty
  - Compute box over triangle centroids and split that on longest axis instead



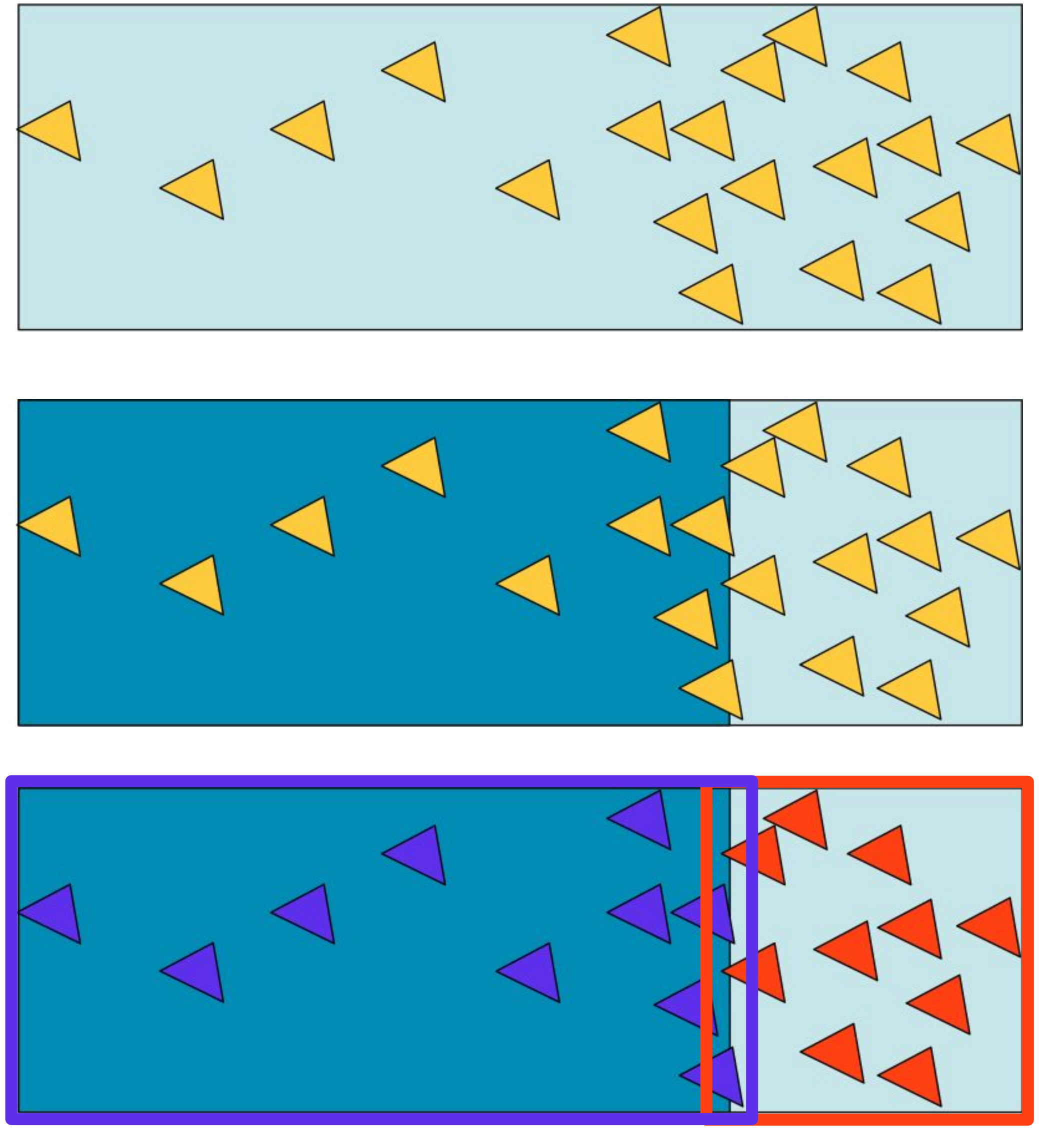
Splitting at object median

- Pick an axis. Can try them all, don’t pick the same every time

- Sort triangles according to their centroid’s position on that axis

- Assign first half of the sorted triangles to A, the second to B

- Continue recursion with A/B



BVH Traversal

1. Set $t_{min} = \infty$. Start at root node, return if it doesn’t intersect ray.

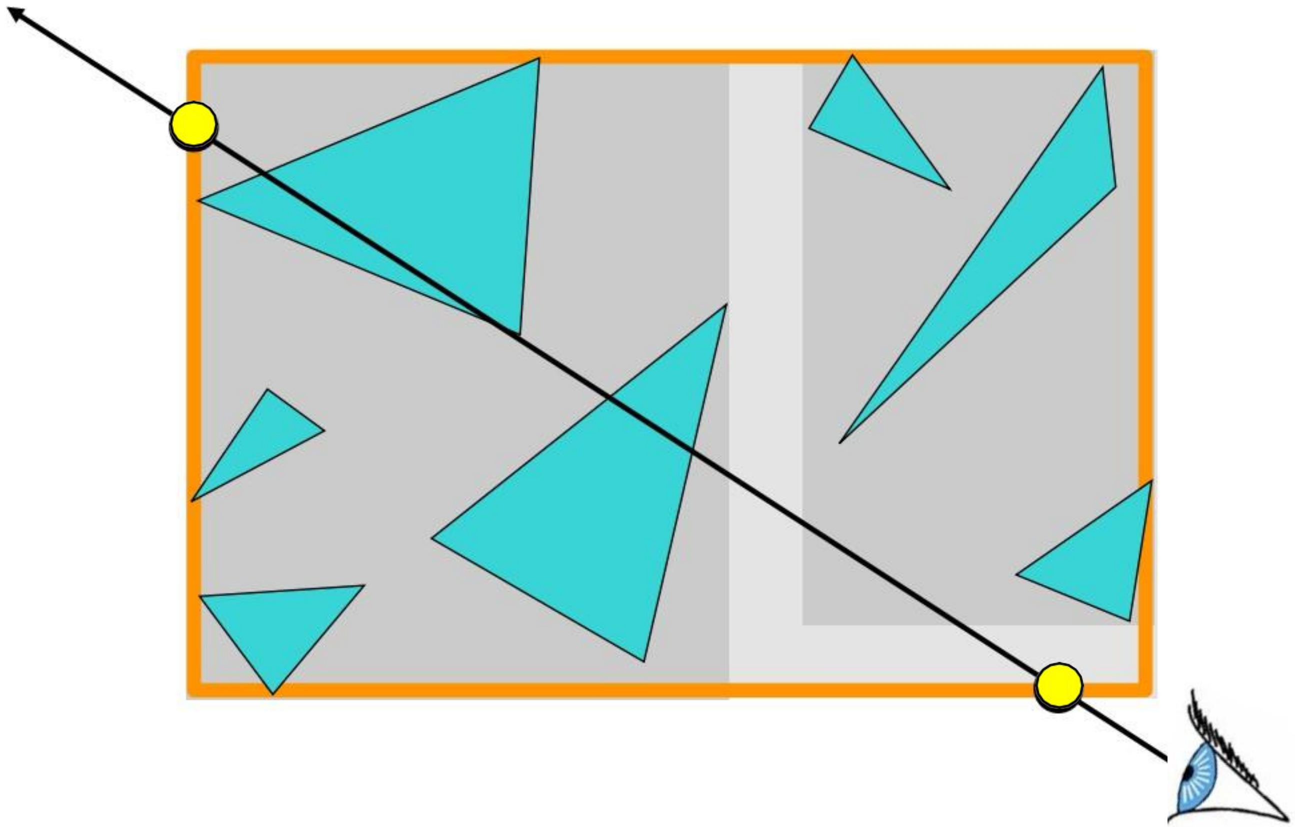
2. Process node if its closest intersection with ray is closer than t_{min}

3. If it’s an inner node, run from 1. for child nodes that intersect ray
   - Process the closest node first

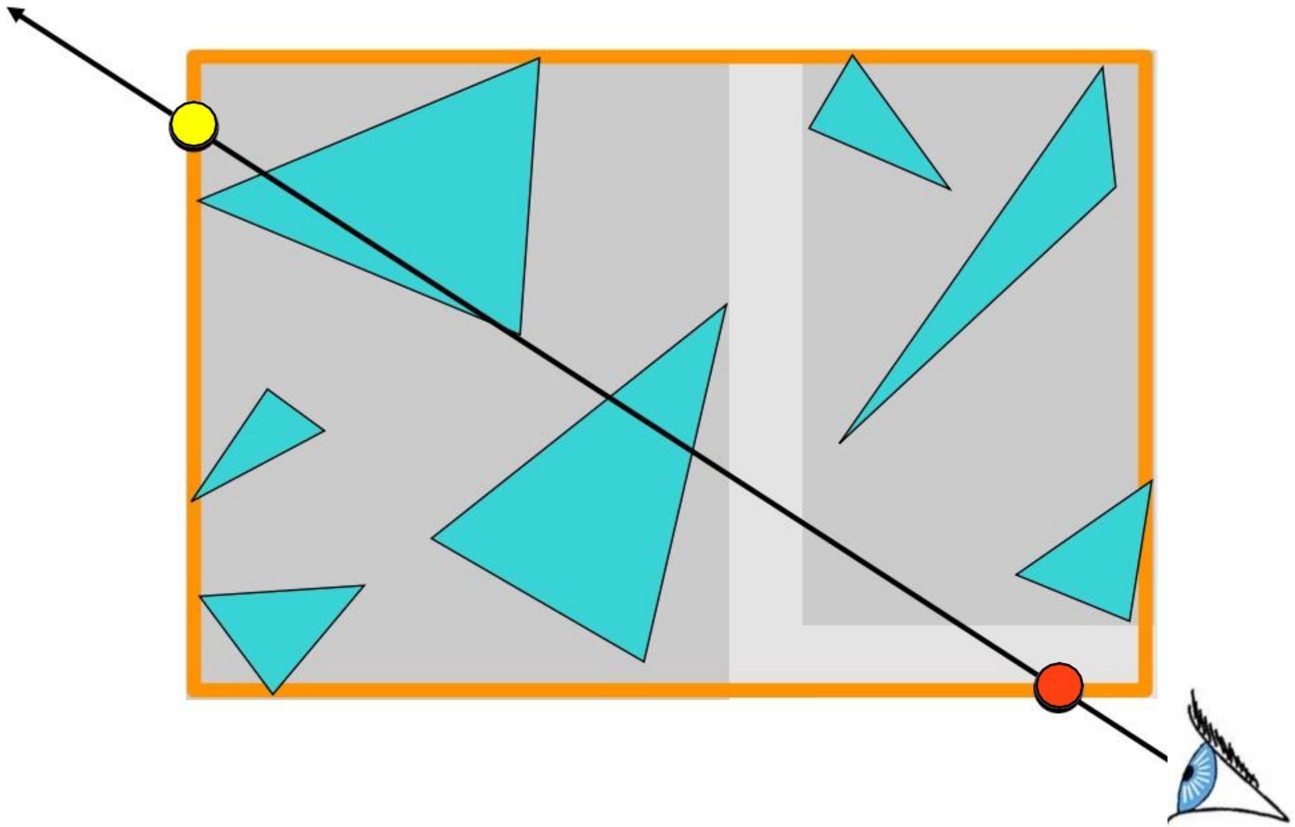
     Keep others on stack to process further ones later (recursion works)

3. If it’s a leaf, check triangles and update t_{min} in case of closer hit

1. Process node if its closest intersection with ray is closer than t_{min}

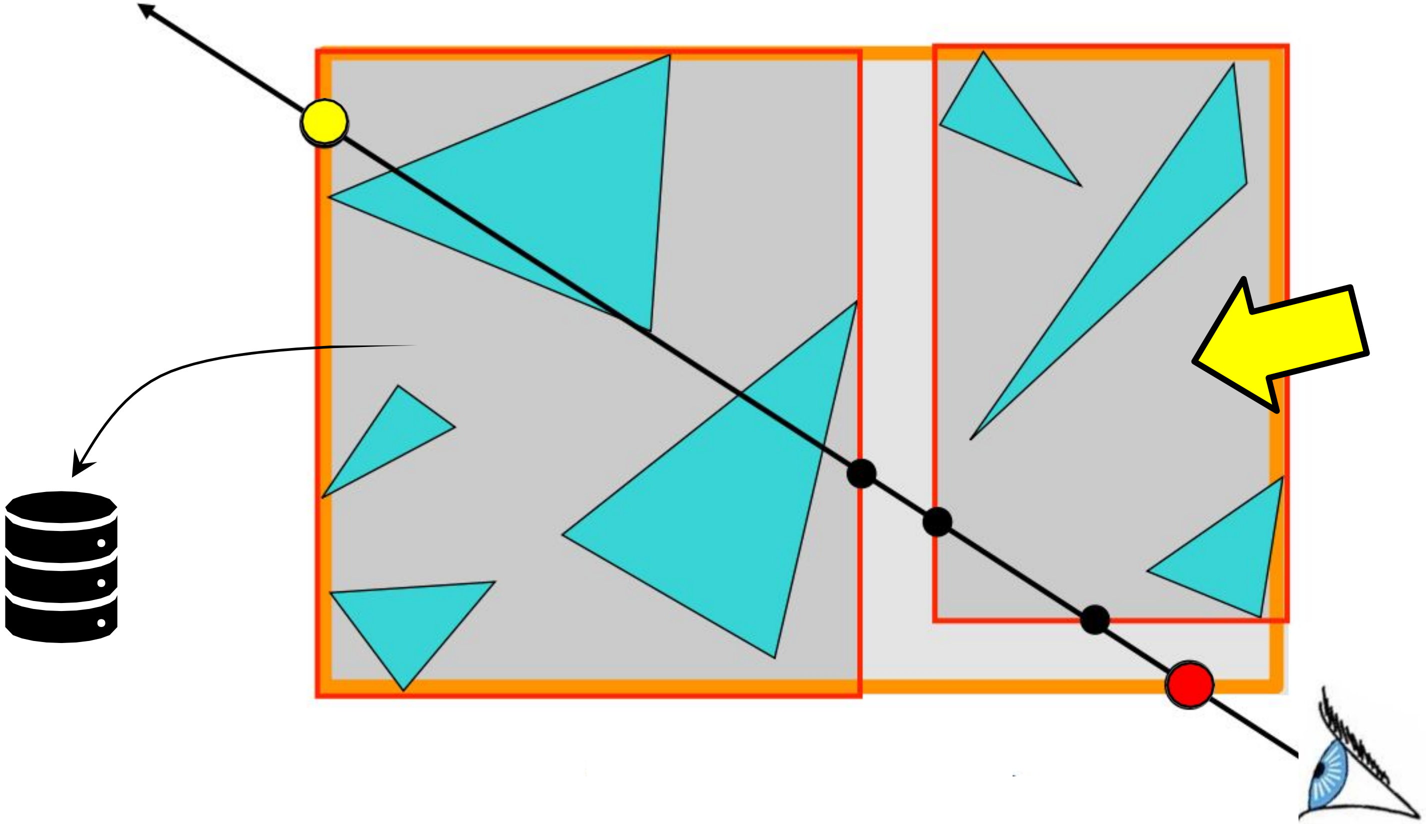


1. Process node if its closest intersection with ray is closer than t_{min}

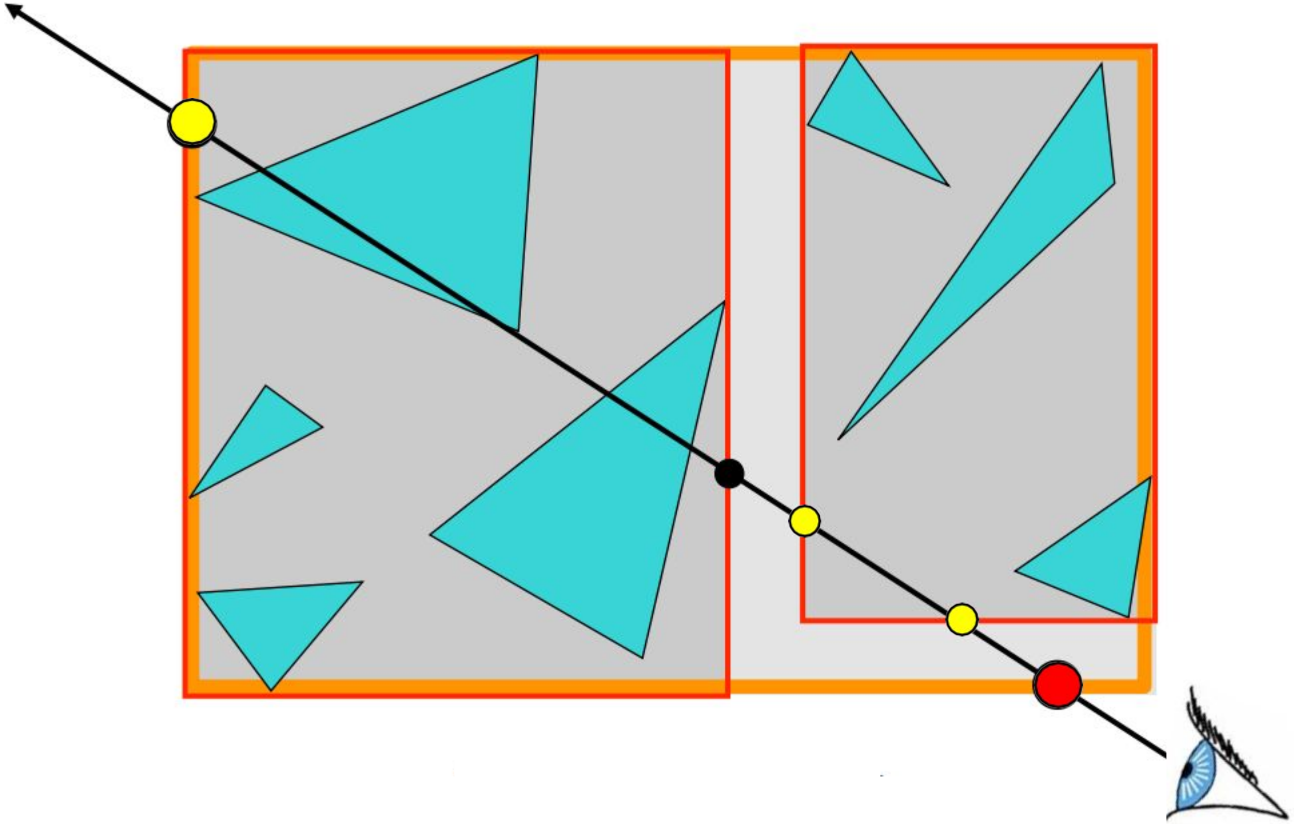


BVH Traversal Example

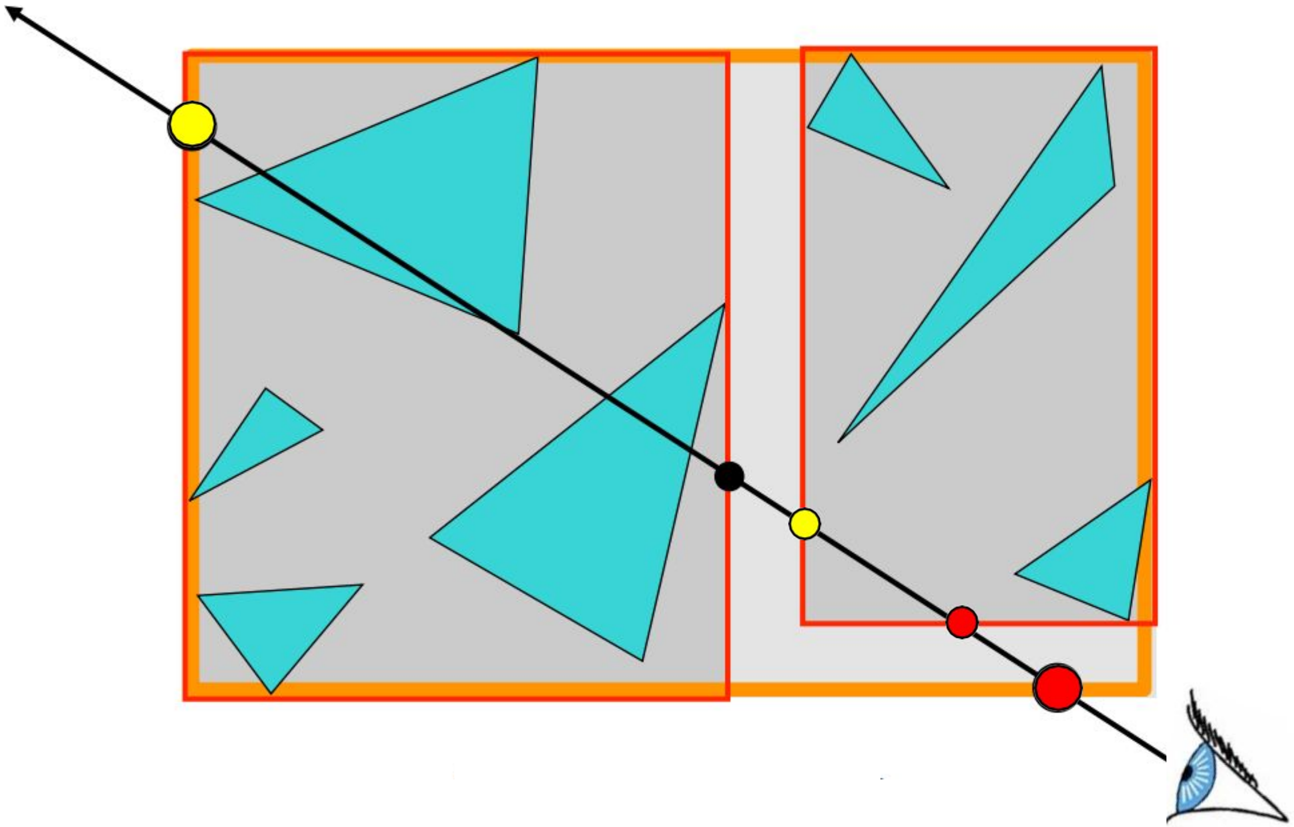
2. If it’s an inner node, run from 1. for child nodes that intersect ray
   - Process the closest node first
   - Keep others on stack to process further ones later



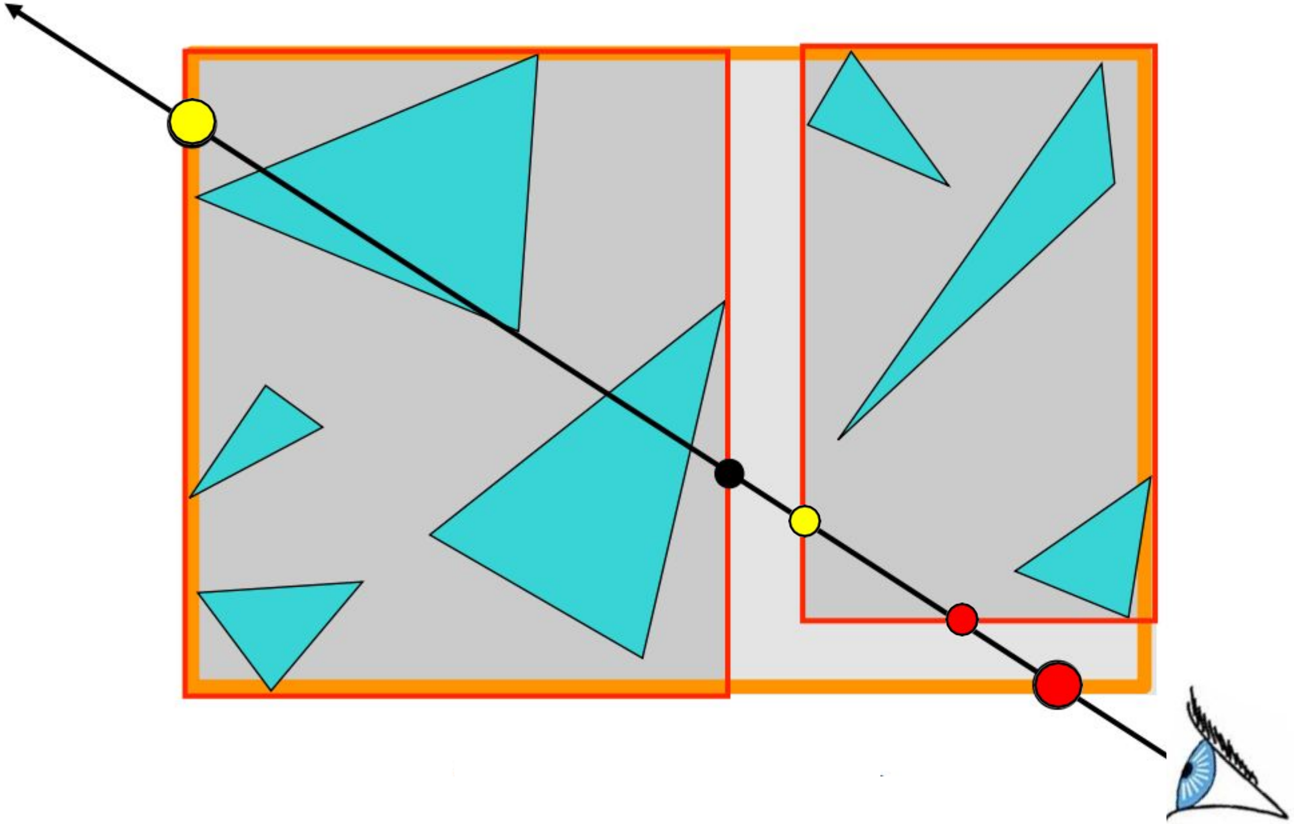
1. Process node if its closest intersection with ray is closer than t_{min}



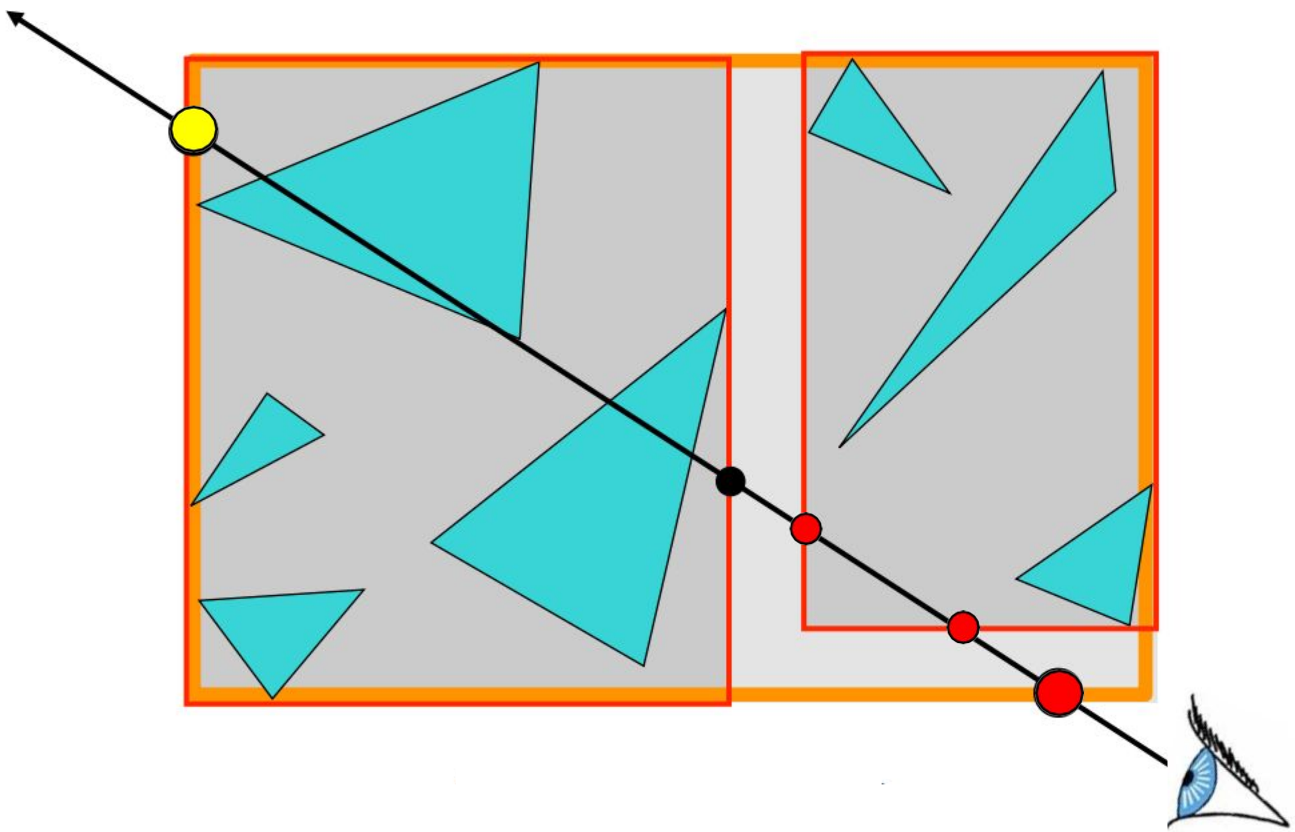
1. Process node if its closest intersection with ray is closer than t_{min}



3. If it's a leaf, check triangles and update t_{min} in case of closer hit

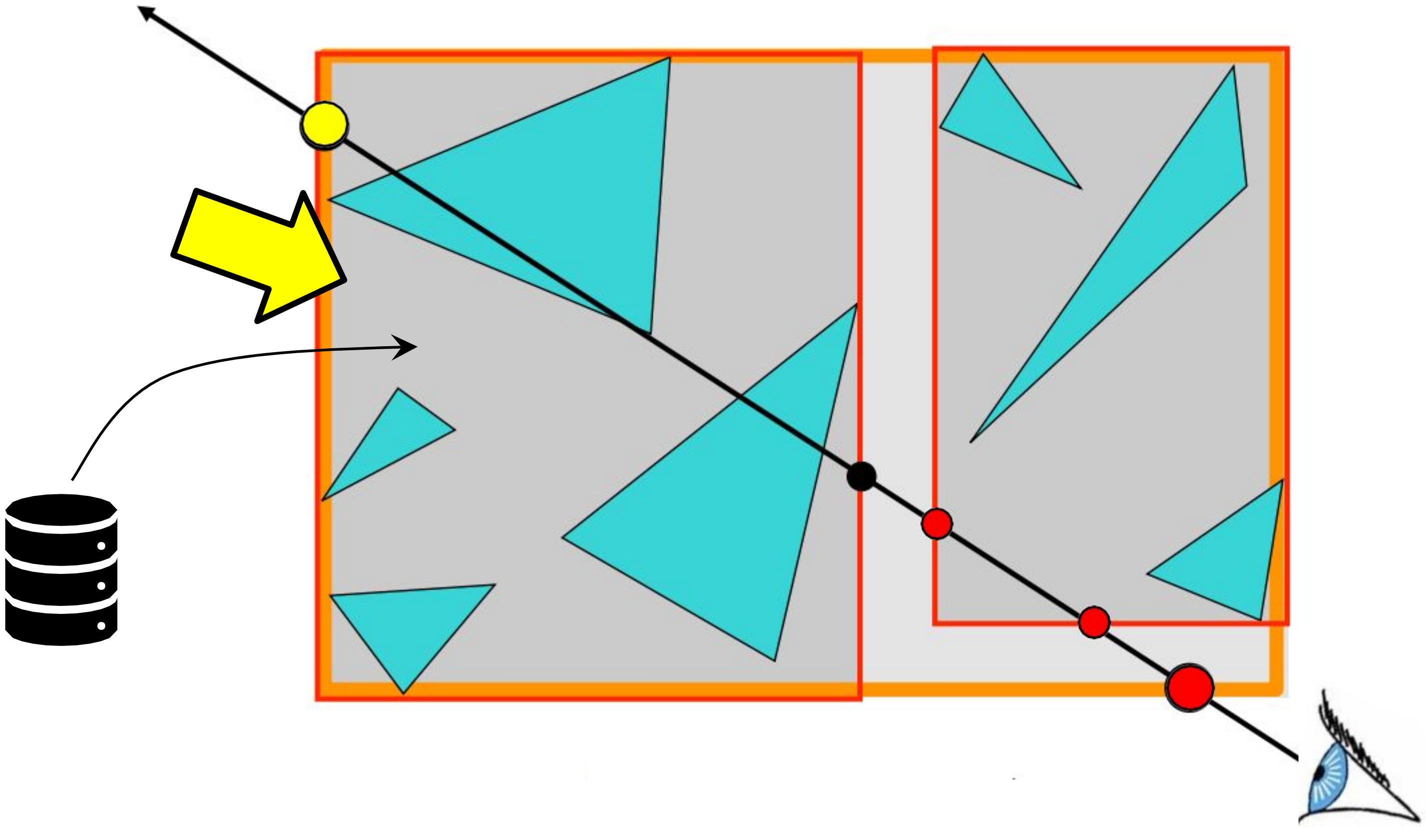


3. If it’s a leaf, check triangles and update t_{min} in case of closer hit

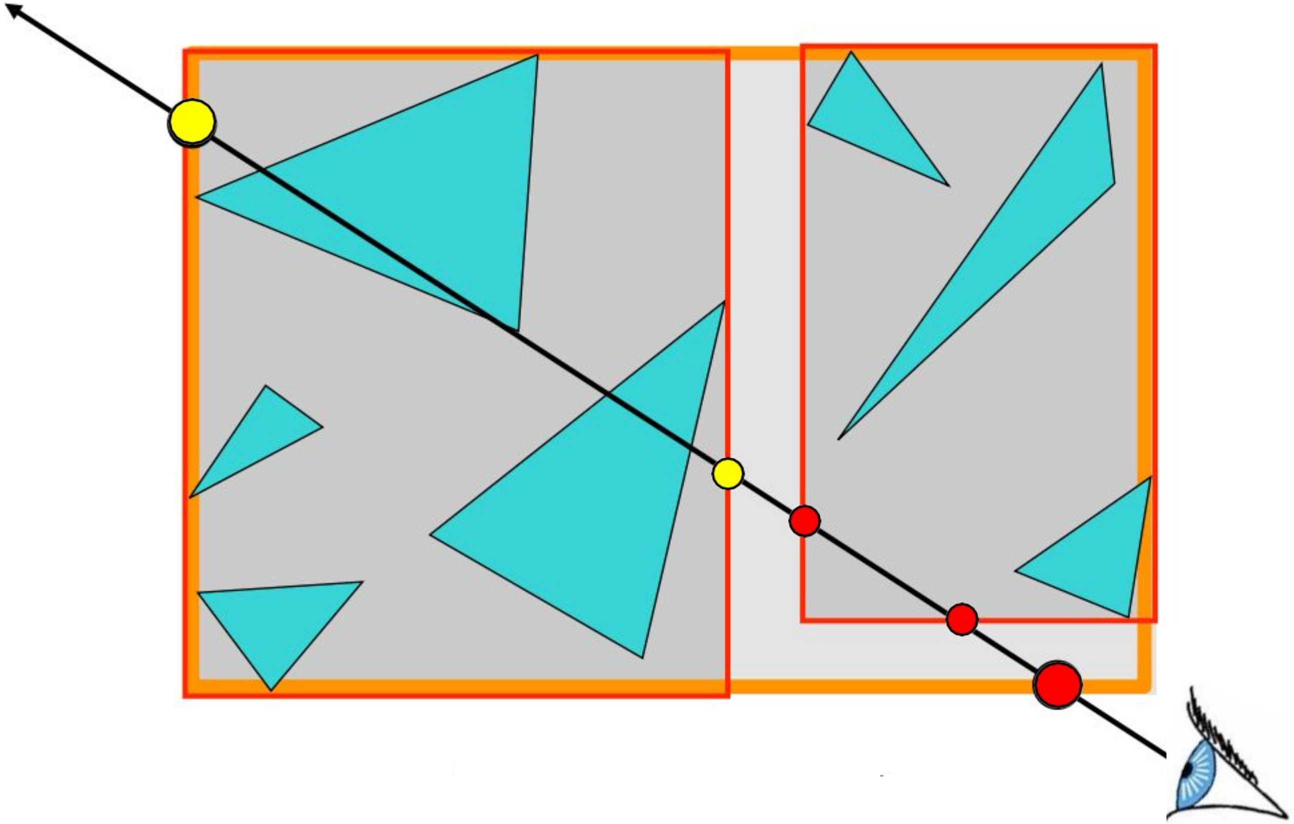


BVH Traversal Example

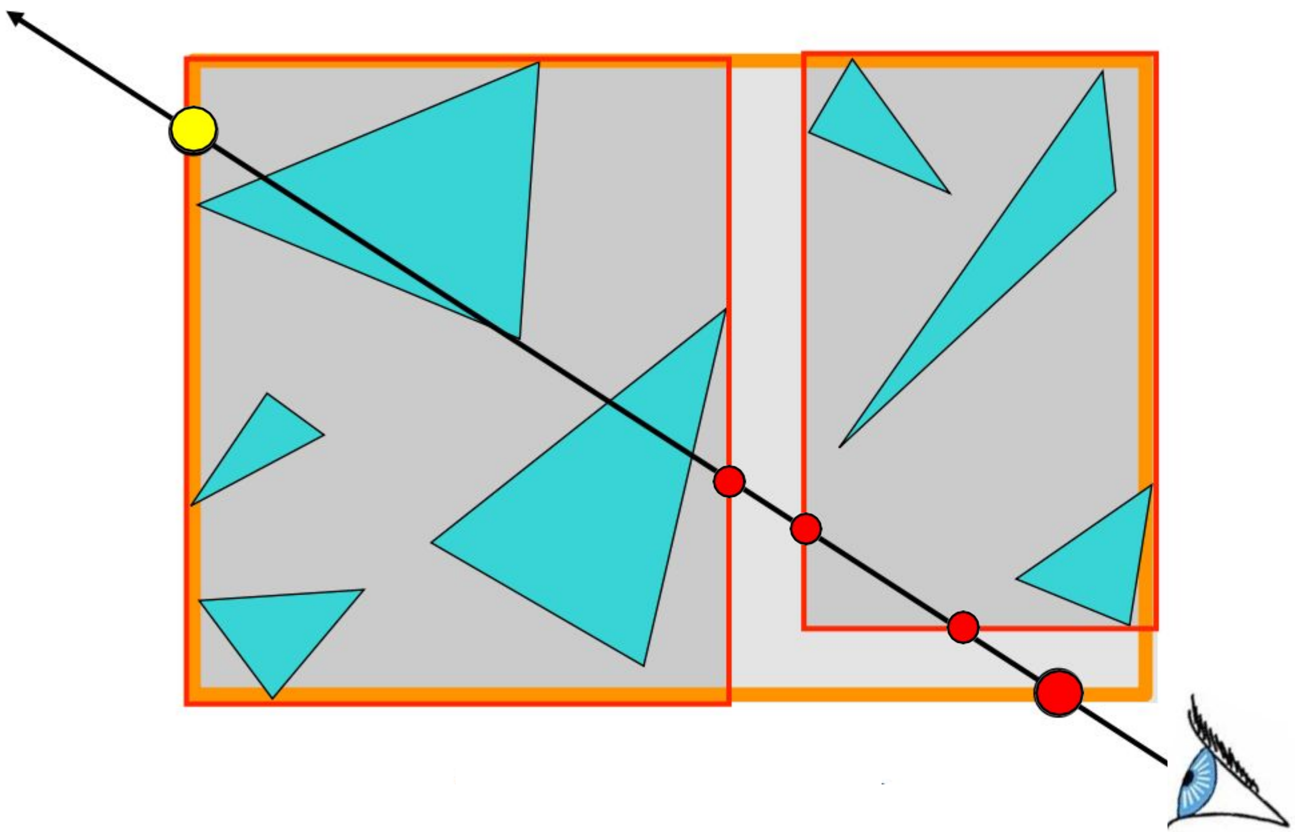
2. If it’s an inner node, run from 1. for child nodes that intersect ray
   - Process the closest node first
   - Keep others on stack to process further ones later



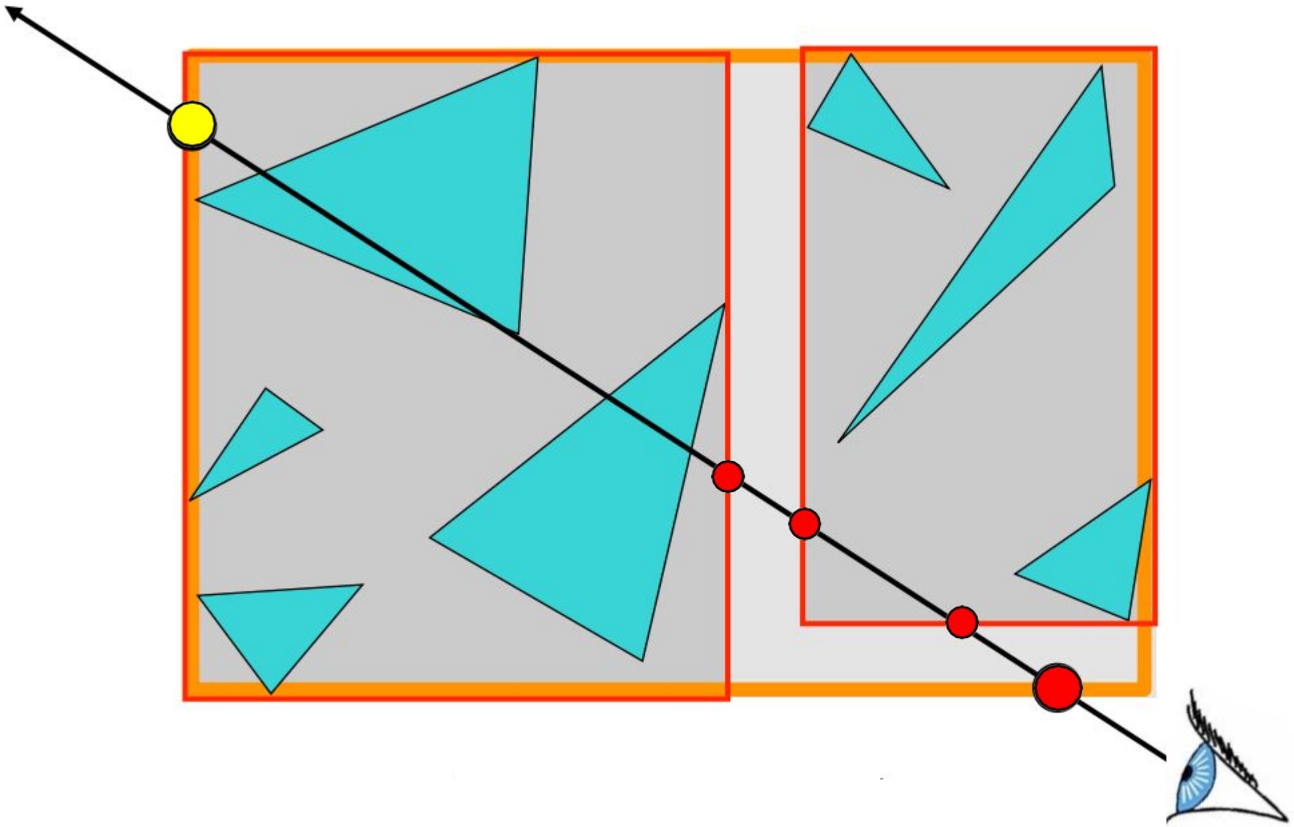
1. Process node if its closest intersection with ray is closer than t_{min}



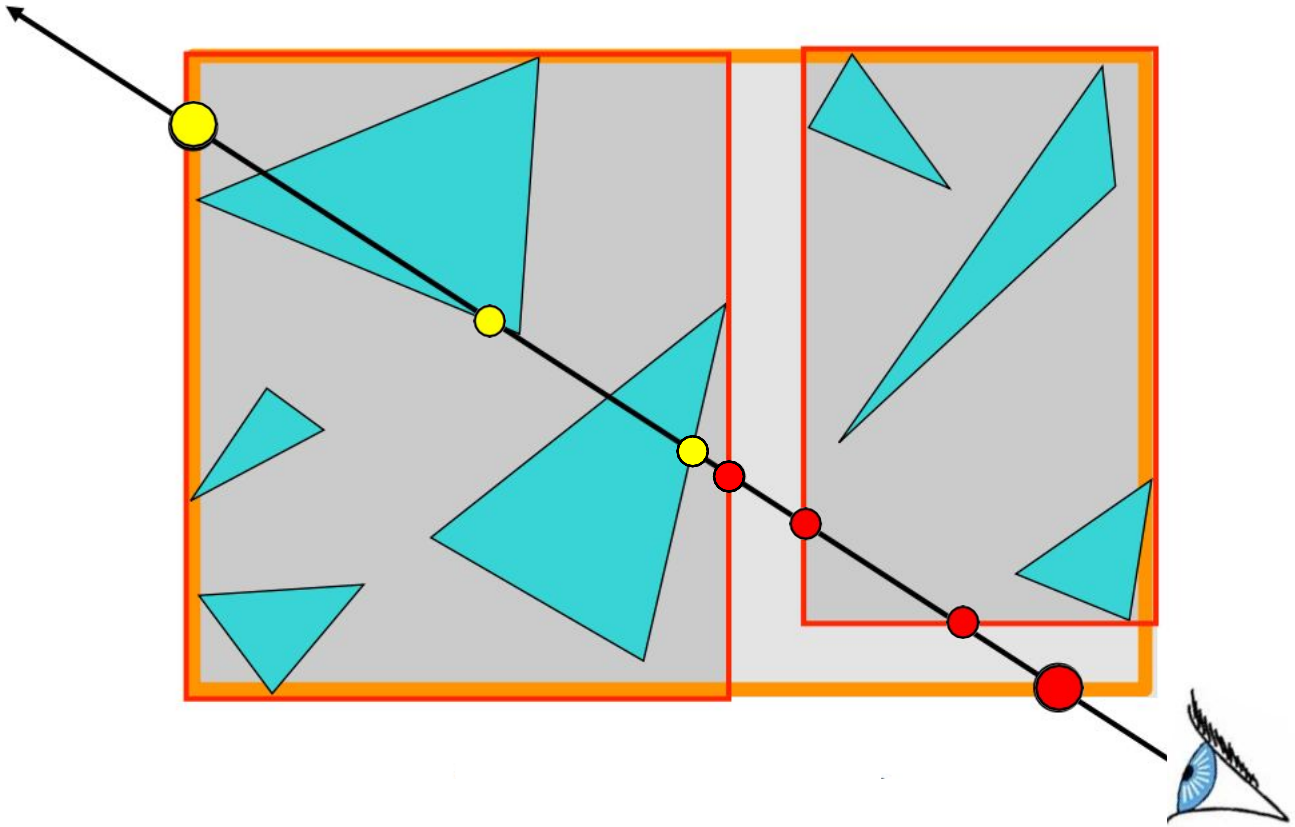
1. Process node if its closest intersection with ray is closer than t_{min}



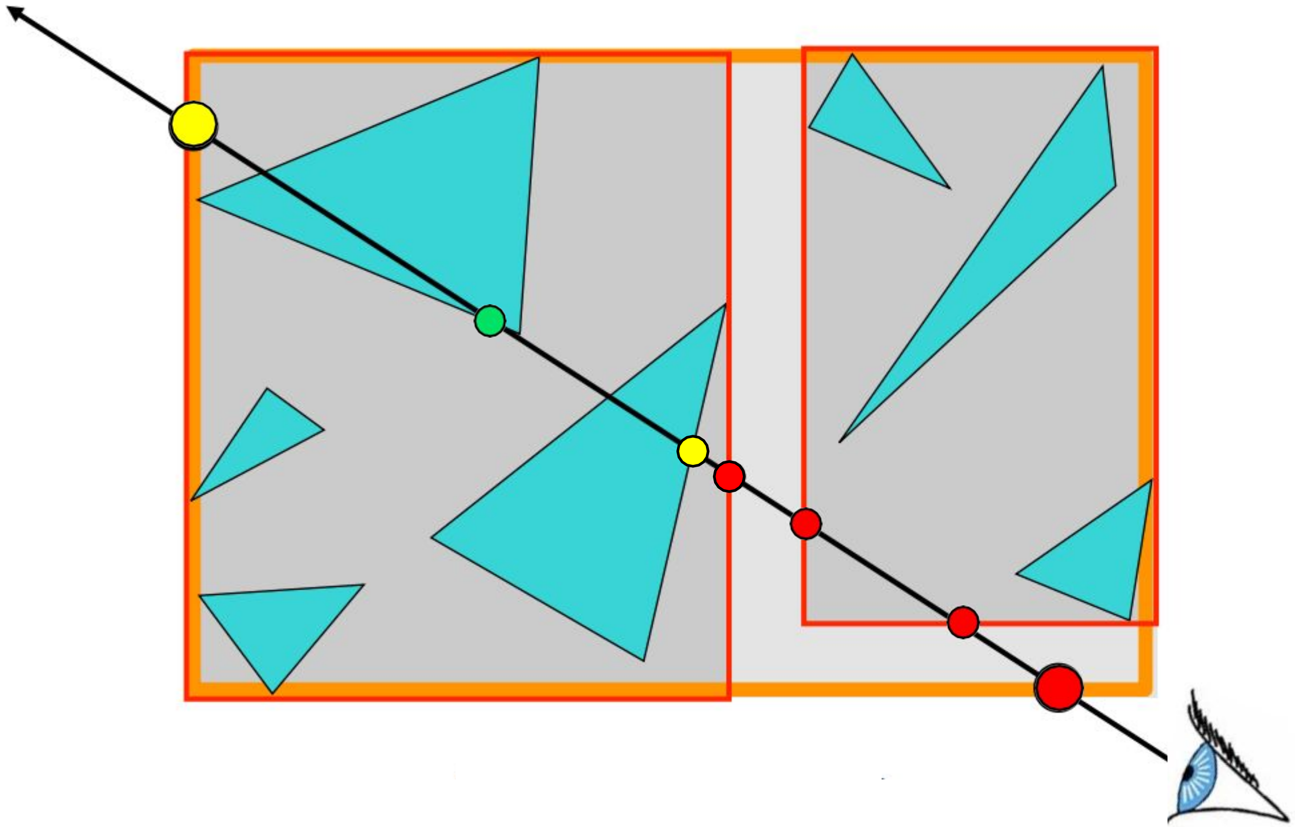
3. If it’s a leaf, check triangles and update t_{min} in case of closer hit (🟢)



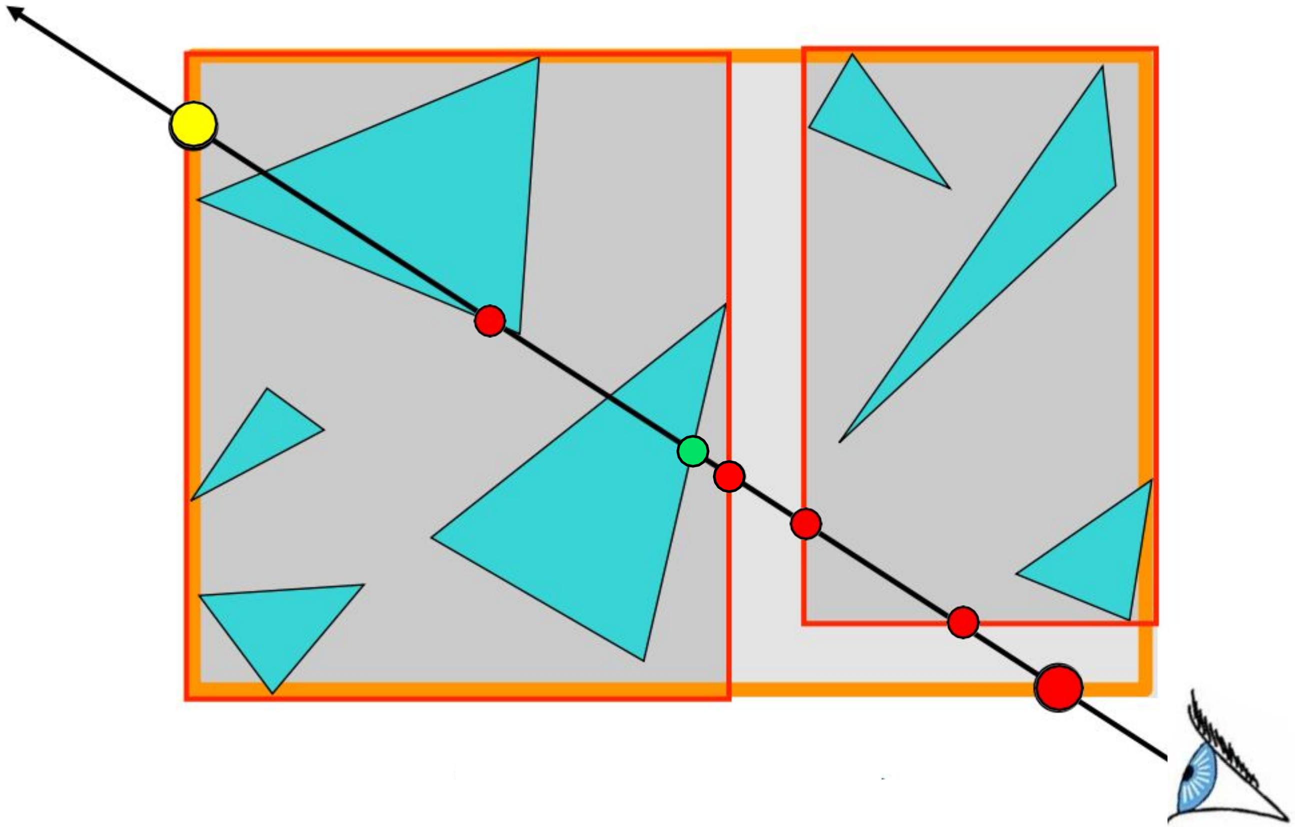
3. If it’s a leaf, check triangles and update t_{min} in case of closer hit (🟢)



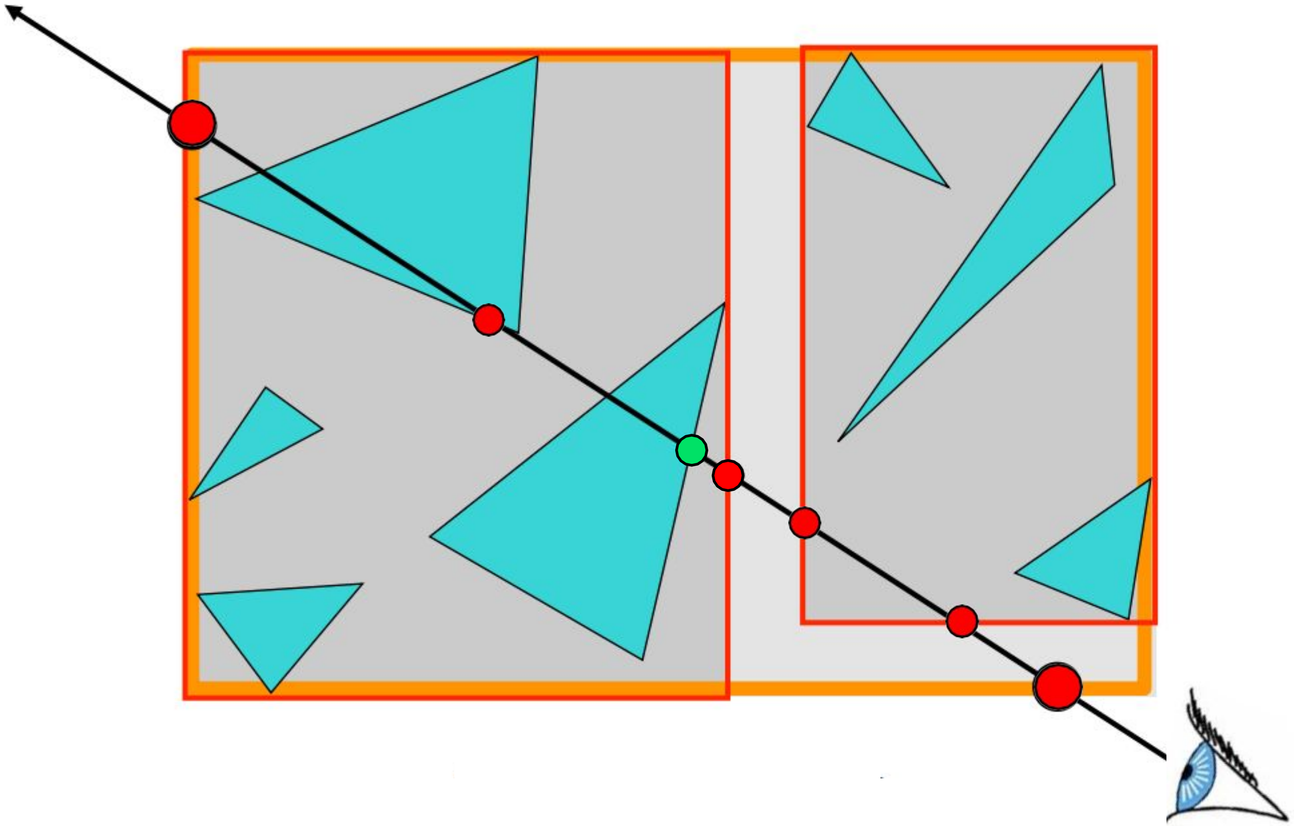
3. If it’s a leaf, check triangles and update t_{min} in case of closer hit (🟢)



3. If it’s a leaf, check triangles and update t_{min} in case of closer hit (🟢)



3. If it’s a leaf, check triangles and update t_{min} in case of closer hit (🟢)



The Surface Area Heuristic [1]

- Simple, but powerful heuristic for choosing splits

- Created with traversal in mind, based on the following ideas:
  - Assume rays are uniformly distributed in space
  - Probability of a ray hitting a node is proportional to its **surface area**
  - Cost of traversing it depends on the **number of triangles** in its leaves
  - Hence, **avoid large nodes with many triangles**, because:
    - They have a tendency to get checked often
    - Getting a definite result (reject or closest hit) is likely to be expensive

Goal: To split a node, find the hyperplane $b$that minimizes

$f(b)=LSA(b)\cdot L(b)+RSA(b)\cdot(N-L(b))$, where

- $LSA(b)/RSA(b)$are the **surface area** of the nodes that enclose the triangles whose centroid is on the “left”/“right” of the split plane b
- $L(b)$is the **number of primitives on the “left”** of b

- N is the **total number of primitives** in the node

We want to constrain the search space for a good split

Pick a set of axes to test (e.g., 3D basis vectors X/Y/Z)

When splitting a node with N triangles, for each axis

Sort all triangles by their centroid's position on that axis

Find the index i that minimizes

$f\ (i) = LSA(\ i) \cdot (\ i) + RSA(i) \cdot (N - i),\ \text{where}$

- $LSA(i)$ is the surface area of the AABB over sorted triangles $[0, i)$
- $RSA(i)$ is the surface area of the AABB over sorted triangles $[i, N)$

Select the axis and index i with the best $f\ (i)$ for the split overall!

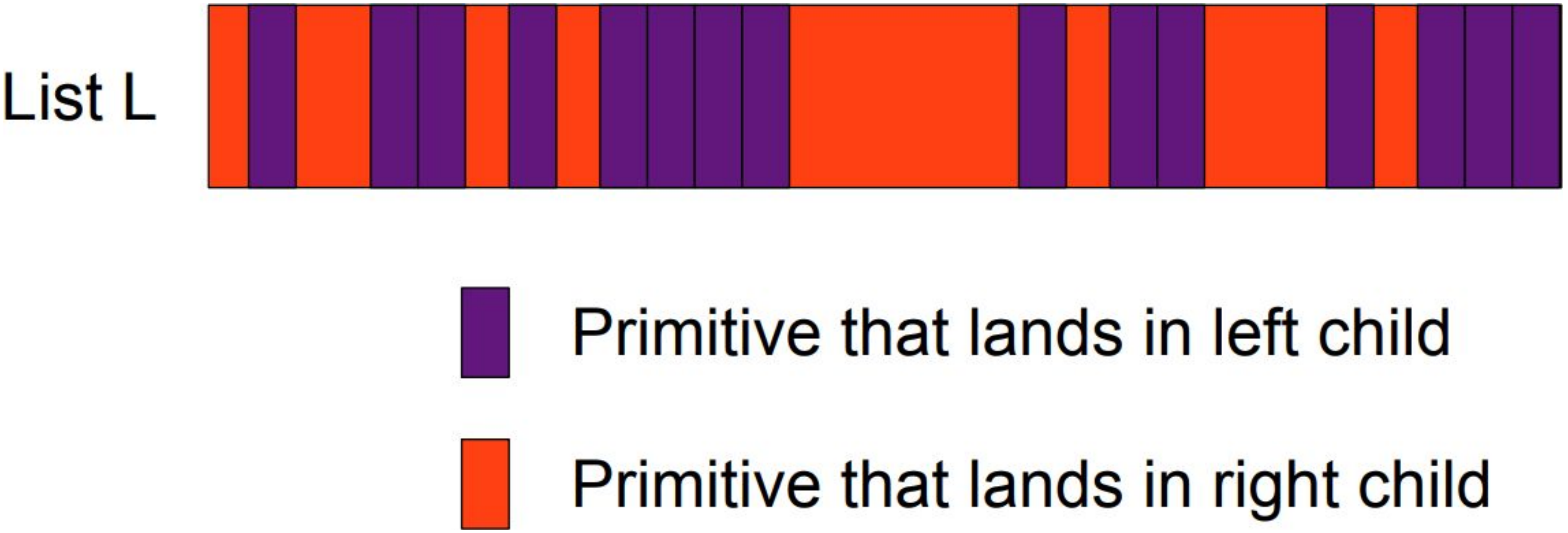
Spatial Acceleration Structures

Structure	Memory Consumption	Building Time	(Expected) Traversal Time
none	none	none	abysmal
Regular Grid	low – high (resolution)	low	uniform scene: ok otherwise: bad
Quadtree/Octree	low – high (overlap/uniformity)	low	good
K-d Tree	low – high (overlap)	low – high	good – excellent
BVH	**low**	**low – high**	**good – excellent**

BVH Building Hints

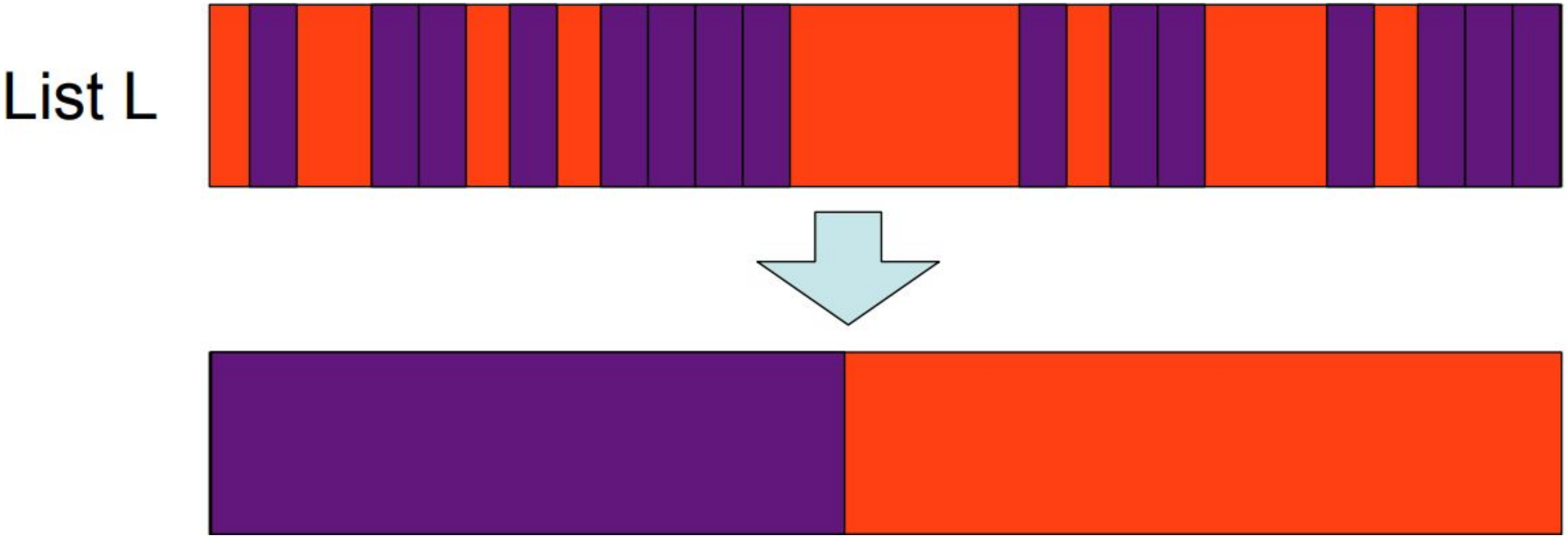
- For each split, sort the node’s portion of the triangle list *L* in-place

- When constructing child nodes, pass them *L* and *start/end* indices


List L
Primitive that lands in left child
Primitive that lands in right child

BVH Building Hints

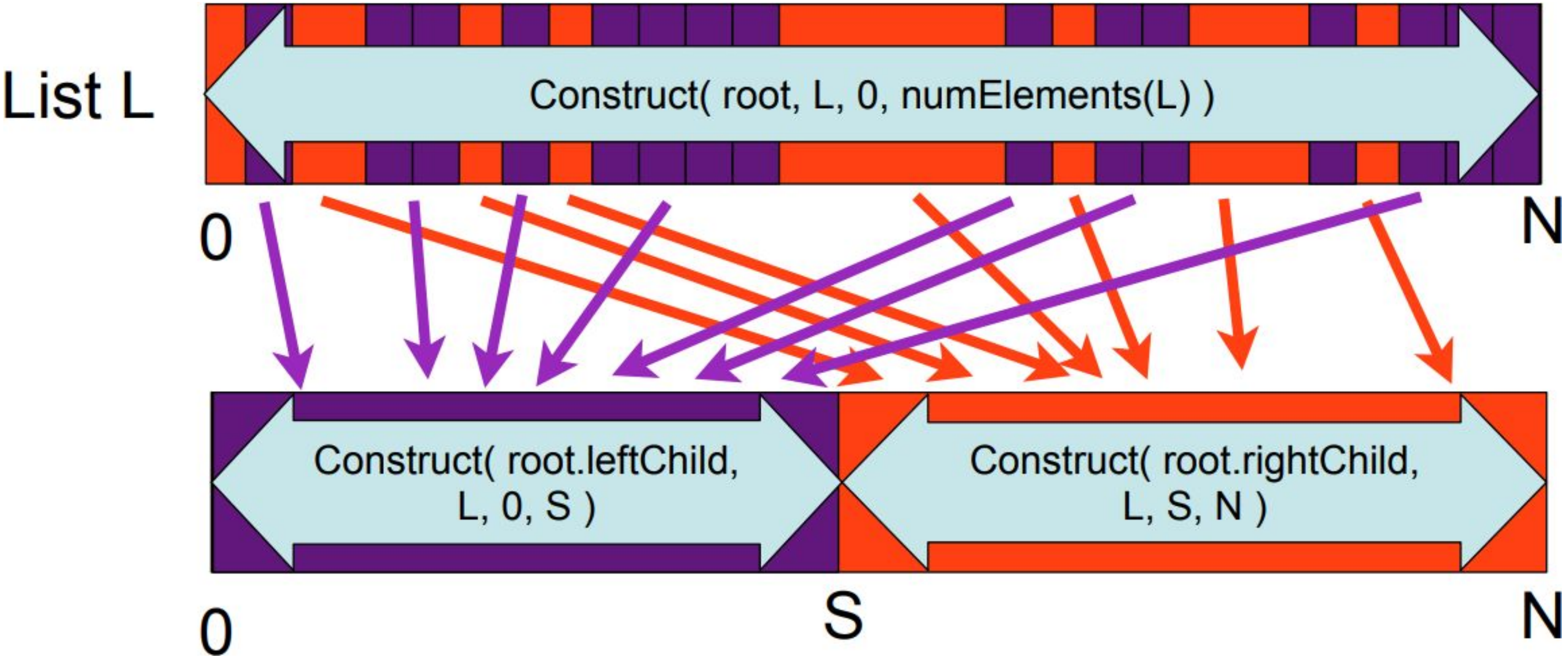
- For each split, sort the node’s portion of the triangle list *L* in-place
- When constructing child nodes, pass them *L* and *start/end* indices



BVH Building Hints

- For each split, sort the node’s portion of the triangle list *L* in-place

- When constructing child nodes, pass them *L* and *start/end* indices


List L
Construct(root, L, 0, numElements(L))
0
N
Construct(root.leftChild, L, 0, S)
Construct(root.rightChild, L, S, N)
0
S
N

SAH Coding Hints

- Don’t loop over triangles at each i to get $LSA(i)$ and $RSA(i)$!

- Precompute them once per node and axis instead
  - Create two 0-volume bounding boxes BB_L, BB_R
  - Allocate N+1 entries for LSA/RSA, set $LSA(0) = RSA(N) = 0$
  - Iterate i over range $[1, N]$, for each i:
    - Merge BB_L with the AABB of sorted triangle with index $(i-1)$
    - Store surface area of BB_L as value for $LSA(i)$
    - Merge BB_R with the AABB of sorted triangle with index $(N-i)$
    - Store surface area of BB_R as value for $RSA(N-i)$

BVH vs K-d Tree vs Others

- Each have their specializations, strengths and weaknesses
  E.g., K-d Trees with ropes do not require a stack for traversal [5]

- Which acceleration structure is the **best** is contentious

- Currently, BVHs are extremely widespread and well-understood

State-of-the-Art Variants and Trends

- Higher child counts (>2) per node, mixed nodes (children + triangles)
  Actually DO split triangles sometimes to get maximal performance

- Build BVHs bottom-up in parallel on the GPU [3]

- In animated scenes, reuse BVHs, update those parts that change

- Actually use built-in traversal logic of GPU hardware (NVIDIA RTX!)