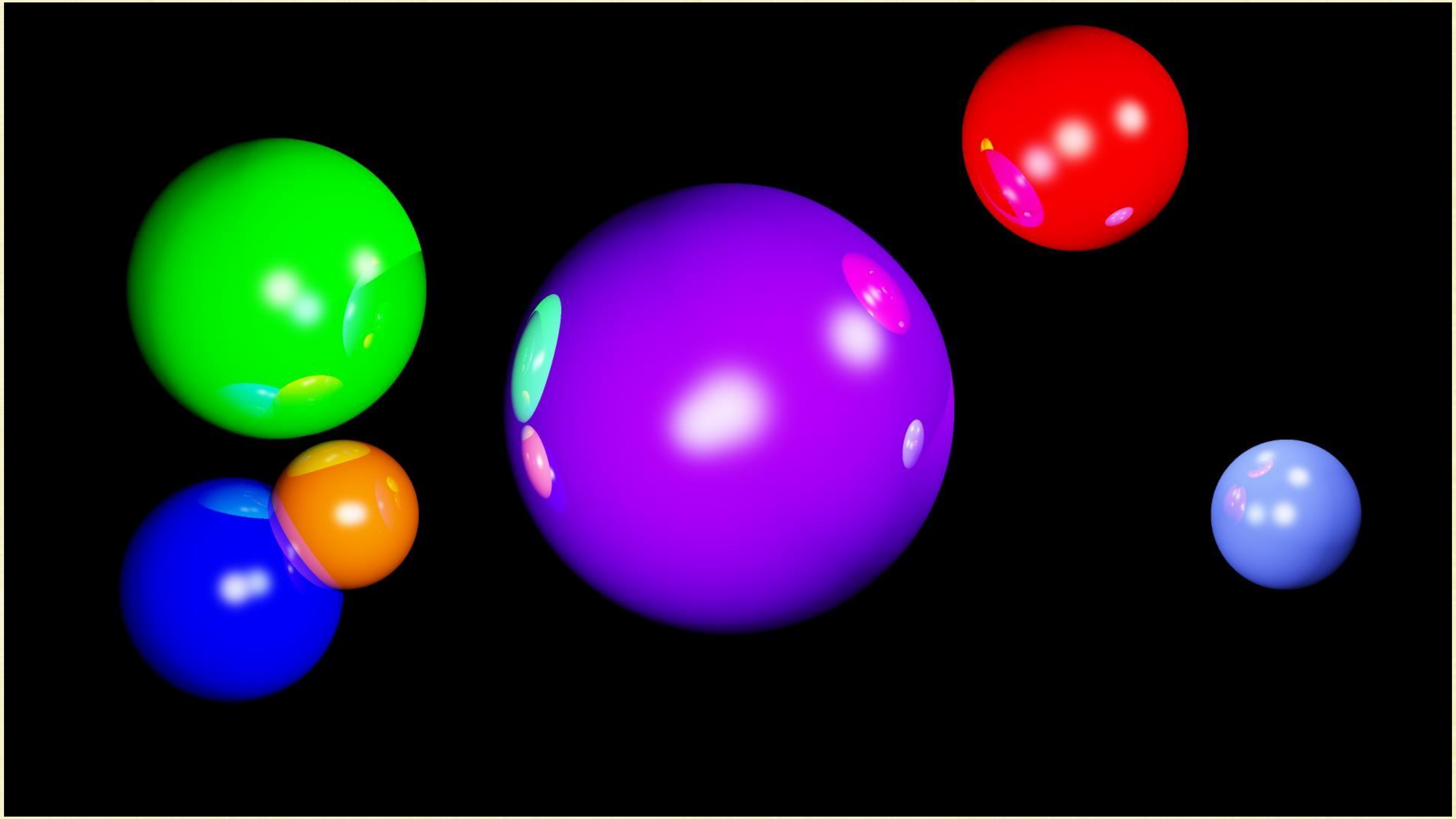
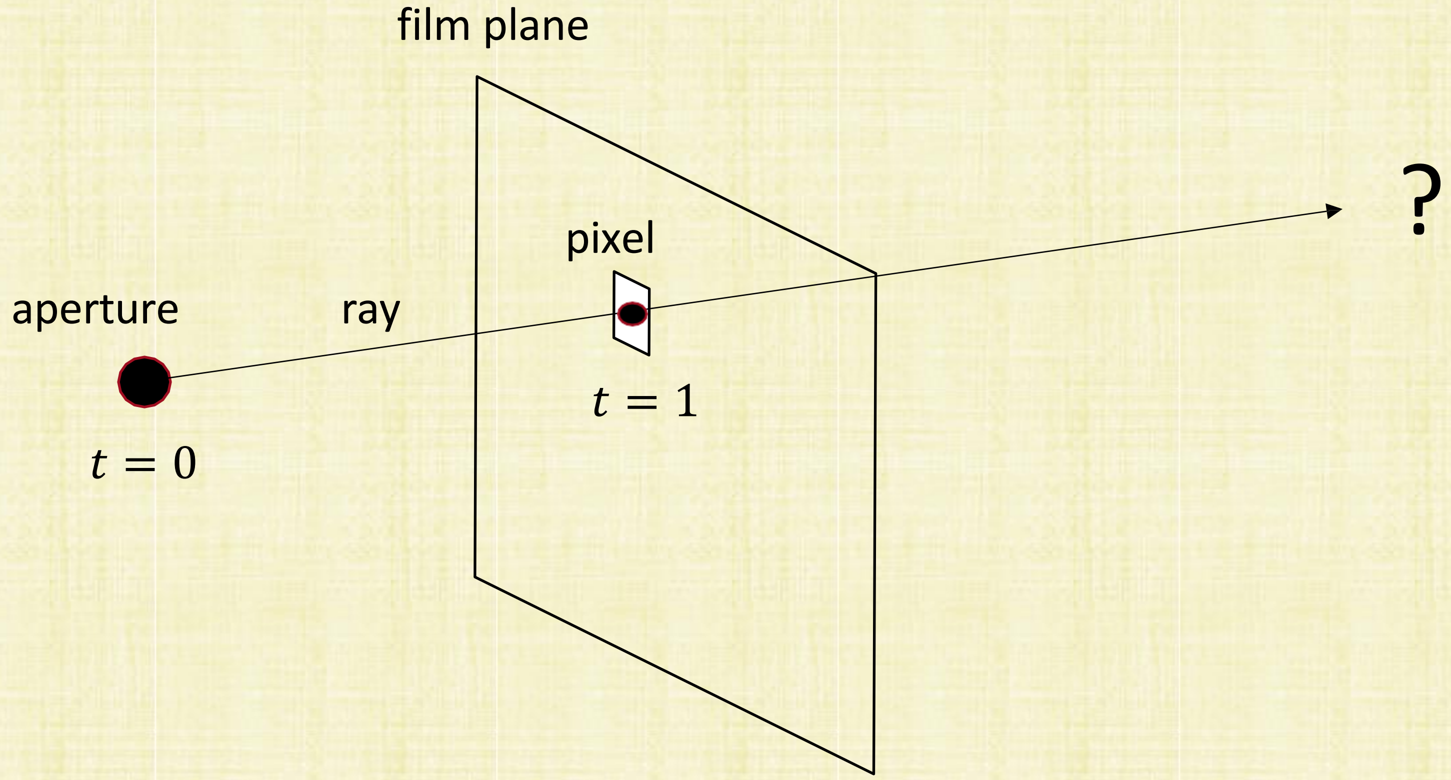
Ray Tracing



Constructing Rays

- For each pixel, create a ray and intersect it with objects in the scene
  - The first intersection is used to determine a color for the pixel
- The ray is $R(t) = A + (P - A)t$ where A is the aperture and P is the pixel location
- The ray is defined by $t \in [0, \infty)$, although only $t \in [1, t_{far}]$ is inside the viewing frustum
  - We only care about the first intersection with $t \geq 1$



Ray-Triangle Intersection

- Given the enormous number of triangles, many approaches have been implemented and tested in various software/hardware settings:

- Option 1: Triangles are contained in planes, so consider the Ray-Plane intersection first
  - A Ray-Plane intersection yields a point, and a subsequent test determines if that point lies inside the triangle
  - Option 1A: Both the triangle and the point can be projected into 2D; then, the 2D triangle rasterization test can be used to determine "inside"
    - Can project into the xy, xz, yz plane by just dropping the z, y, x coordinate (respectively) from the triangle vertices and the point
    - Most robust to drop the coordinate with the largest component in the triangle's normal (so that the projected triangle has maximal area)
  - Option 1B: There is a fully 3D version of the 2D rasterization

- Option 2: Skip the Ray-Plane intersection and consider the Ray-Triangle intersection directly
  - This is similar to how ray tracing works for non-triangle geometry (ray tracers handle non-triangle geometry better than scanline rendering does)

Ray-Plane Intersection

- A plane is defined by a point p_o (that lies on it) and a normal direction N
- A point p is on the plane if $(p - p_o) \cdot N = 0$
- A ray $R(t) = A + (P - A)t$ intersects the plane when $(R(t) - p_o) \cdot N = 0$ for some $t \geq 0$
- That is, $(A + (P - A)t - p_o) \cdot N = 0$ or $(A - p_o) \cdot N + (P - A) \cdot Nt = 0$
- So, $t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$

- As always, if $t \notin \left[1, t_{far}\right]$ or another intersection has a smaller t value, then this intersection is ignored

Notes:

- The length of N cancels (so it need not be unit length)
- A (non-unit length) triangle normal can be computed by taking the cross product of any two edges (as long as the triangle does not have zero area)
- Any triangle vertex can be used as a point on the plane

Direct Ray-Triangle Intersection (Option 2)

- Triangle Basis Vectors: $p = p_2 + \beta_0 u + \beta_1 v$ with $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$
- Points on the ray have $R(t) = A + (P - A)t$
- An intersection point has $A + (P - A)t = p_2 + \beta_0 u + \beta_1 v$
- Or $\begin{pmatrix} u & v & A - P \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ t \end{pmatrix} = A - p_2$ where $\begin{pmatrix} u & v & A - P \end{pmatrix}$ is a 3x3 matrix and $A - p_2$ is a 3x1 vector (3 equations with 3 unknowns)
- The 3x3 coefficient is degenerate when its columns are not full rank
  - That happens when the triangle has zero area or the ray direction, $P - A$, is perpendicular to the plane's normal
- Otherwise, there is a unique solution

- $R(t_{int})$ is inside the triangle, when that unique solution has: $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$

- As always, if $t \notin [1, t_{far}]$ or another intersection has a smaller t value, then this intersection is ignored

Ray-Object Intersections

- As long as a Ray-Geometry intersection routine can be written, ray tracing can be applied to any representation of object geometry
  - In contrast to scanline rendering, where objects need to be turned into triangles
- Besides triangle meshes, ray tracers use: analytic descriptions of geometry, implicitly defined surfaces, parametric surfaces, etc.

- The surfaces of many objects can be written as <u>implicit functions</u>
- That is, $f(p) = 0$ if and only if p is on the surface (e.g. the equation for a plane)
- Sometimes there are additional constraints (such as on the barycentric weights for triangles)

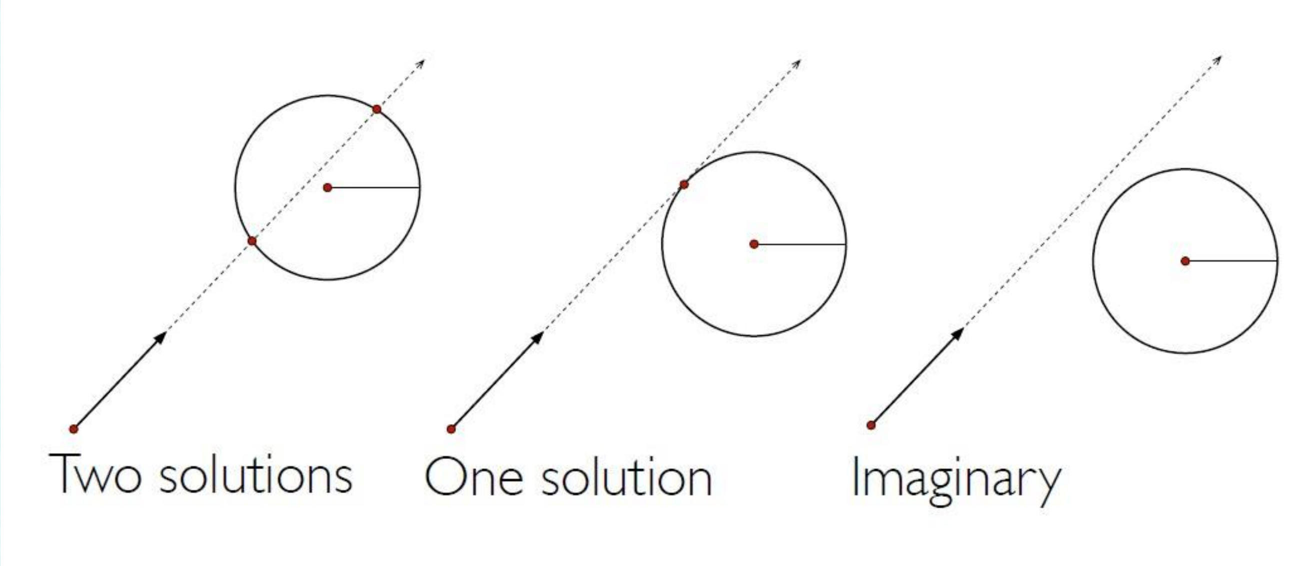
- Ray-Object intersection routines often proceed as follows:
  - substitute the ray equation in for the point, i.e. $f(R(t)) = 0$
  - solve for t
  - check the solution against any additional constraints

Ray-Sphere Intersections

- A point p is on a sphere with center C and radius r when $\|p - C\|_2 = r$
  - Squaring both sides: $(p - C) \cdot (p - C) = r^2$
- Substitute $R(t) = A + (P - A)t$ in for p to get a quadratic equation in t:

$$(P - A) \cdot (P - A)t^2 + 2(P - A) \cdot (A - C)t + (A - C) \cdot (A - C) - r^2 = 0$$

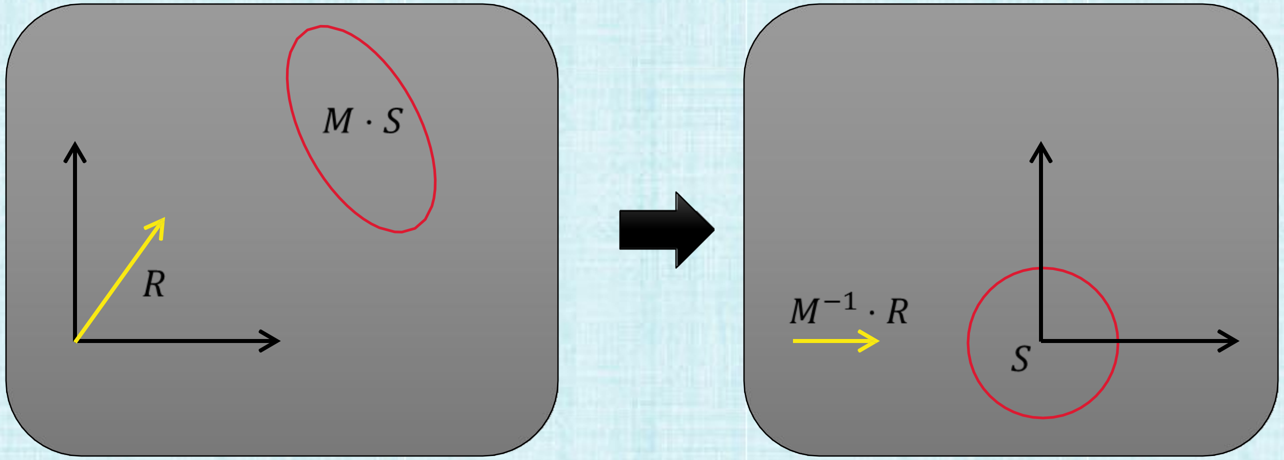
- When the discriminant of this quadratic equation is positive, there are two solutions (choose the one the ray hits first)
- When the discriminant is zero, there is one solution (the ray tangentially grazes the sphere)
- When the discriminant is negative, there are no solutions



Ray Tracing Transformed Objects

- Geometry is typically kept in a convenient object space
- The object space representation is often simpler to deal with
  - E.g., spheres can be centered at the origin, objects are not sheared, coordinates may be non-dimensionalized for numerical robustness, there may be (auxiliary) geometric acceleration structures, more convenient color and texture information, etc.

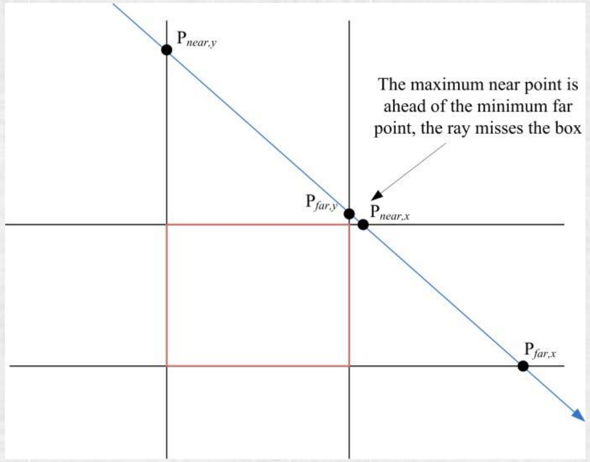
- It is typically preferable to ray trace in object space, rather than world space
- Transform the ray into object space and find the ray-object intersection, then transform the relevant information back to world space

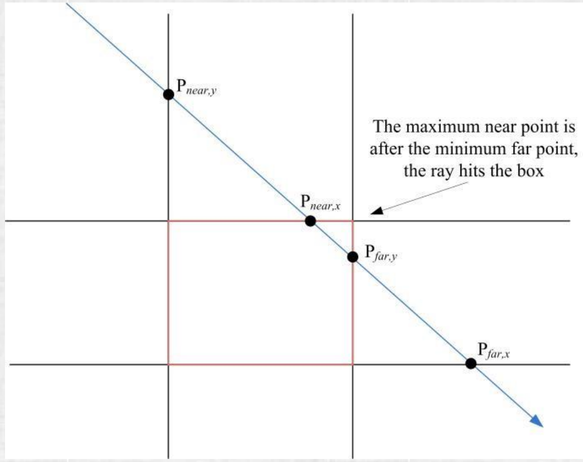


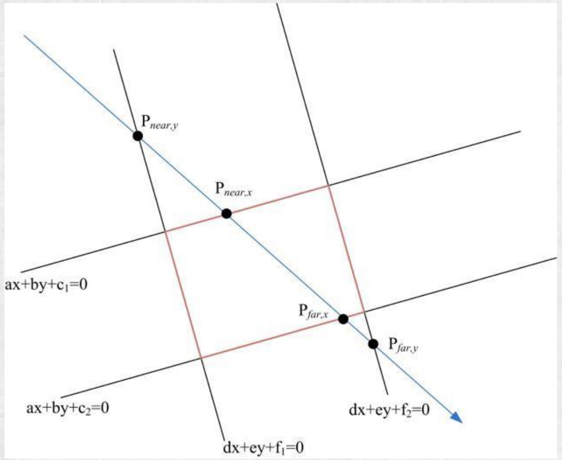
$$\boldsymbol{M \cdot R_{object}(t_{int})}$$

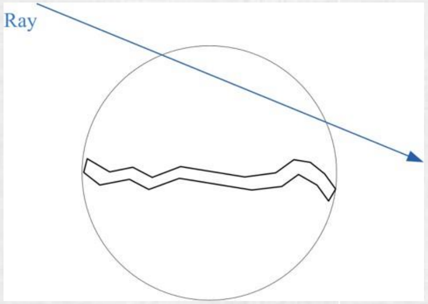
Aside: Code Acceleration

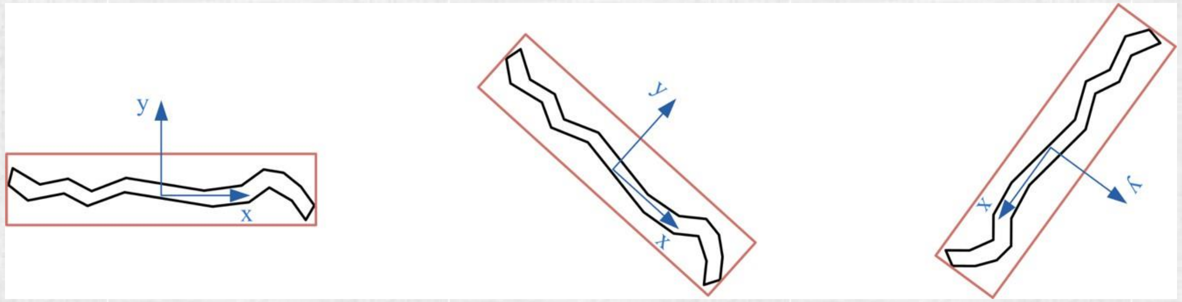
- Ray-Object intersections can be expensive
- So, put complex objects inside simpler objects, and first test for intersections against the simpler object (potentially skipping tests against the complex object)
- Simple <u>bounding volumes</u>: spheres, axis-aligned bounding boxes (AABB), or oriented bounding boxes (OBB)







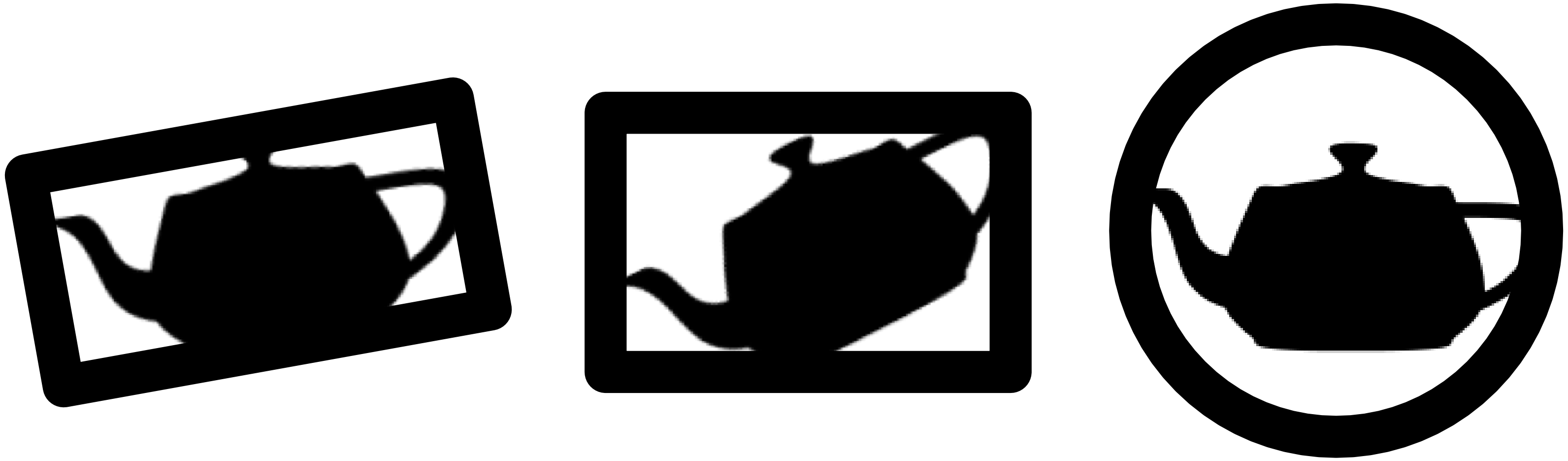




Bounding Volumes

Quick way to avoid intersections: bound complex object with a simple volume

- Object is fully contained in the volume
- If it doesn't hit the volume, it doesn't hit the object
- So test bvol first, then test object if it hits

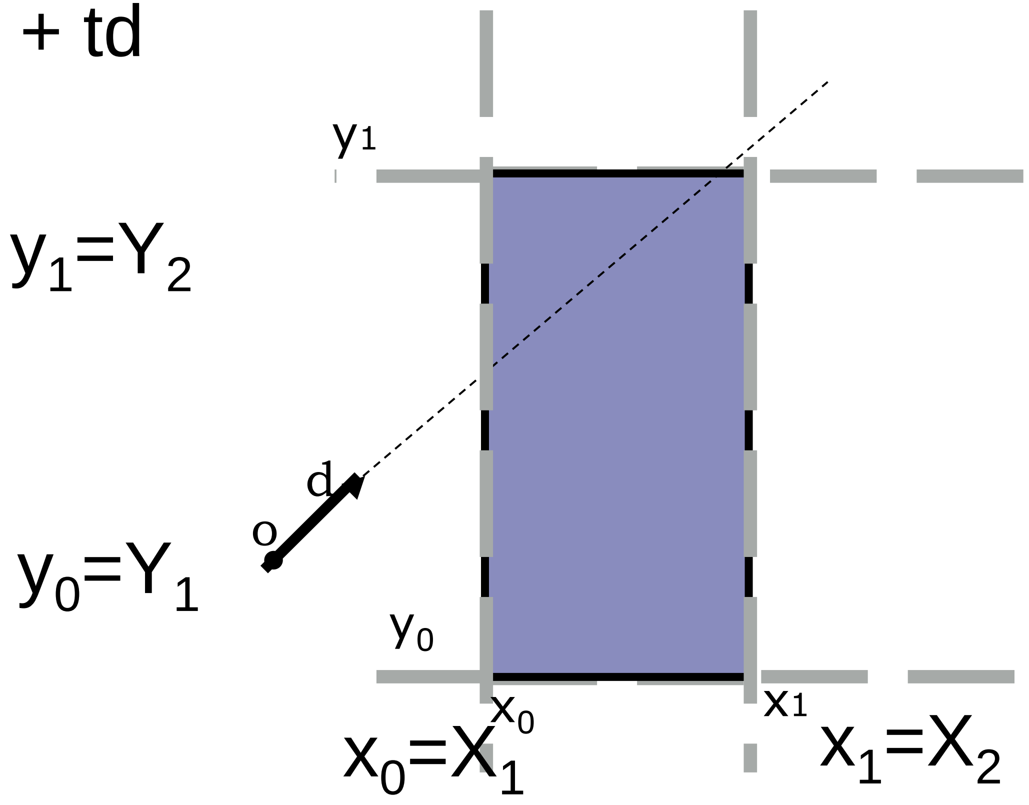


Axis aligned Box

Axis- aligned box

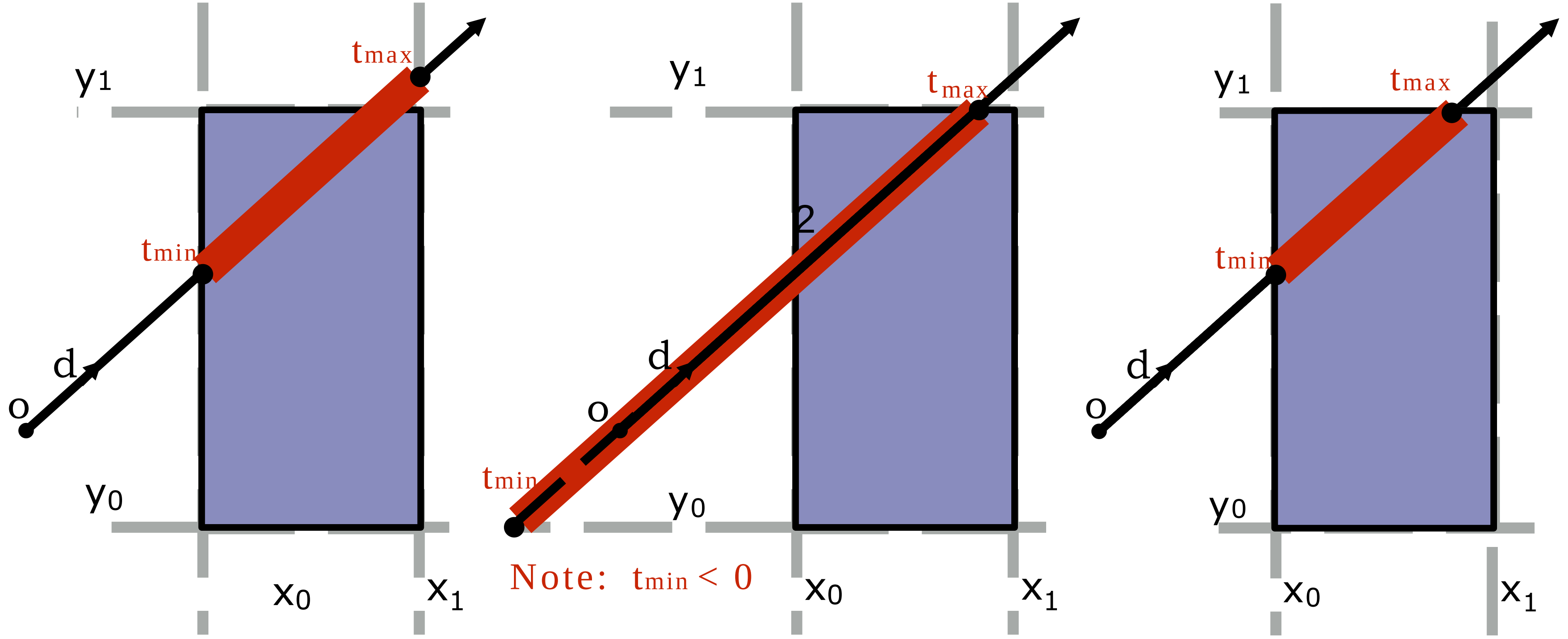
Box: $(X_1,Y_1,Z_1)\text{->}(X_2,Y_2,Z_2)$

Ray: $P(t)=o + td$



Ray Intersection with Axis-Aligned Box

2D example; 3D is the same! Compute intersections with slabs and take intersection of t_{min}/t_{max} intervals

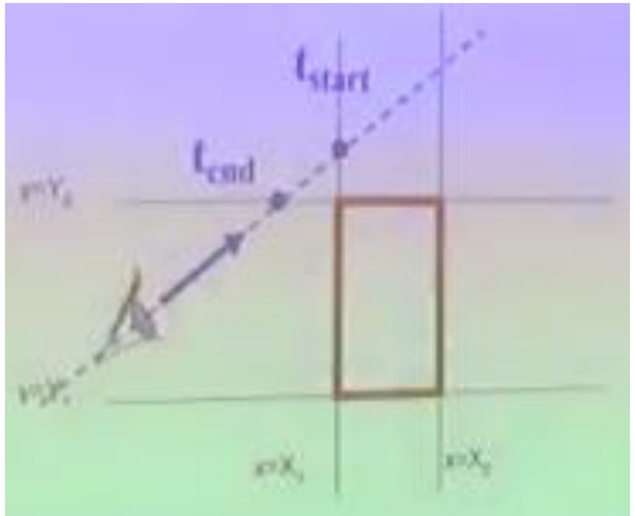


Intersections with *x* planes **Intersections with *y* planes** **intersection result**

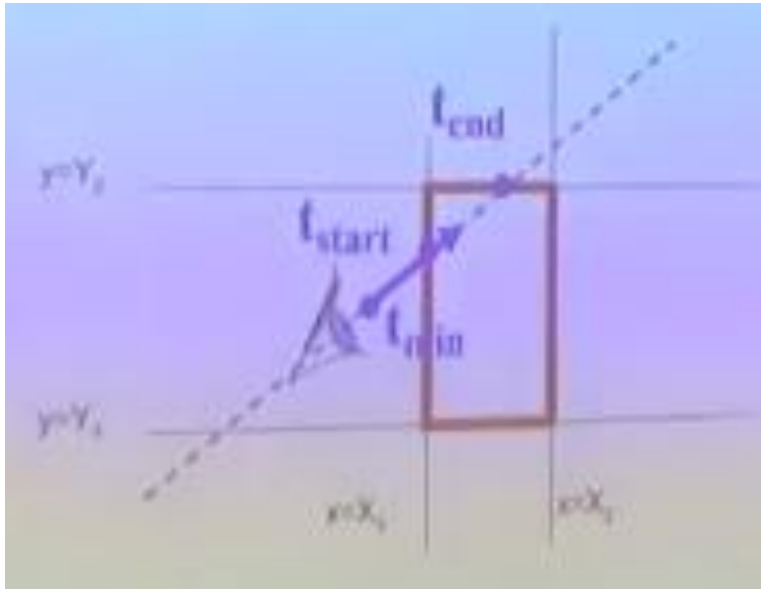
Is there an intersection

1) If $t_{start} > t_{end}$ -> box is missed.
2) If $t_{end} < t_{min}$ -> box is behind.
3) If $t_{start} > t_{min}$ -> closest intersection at t_{start}

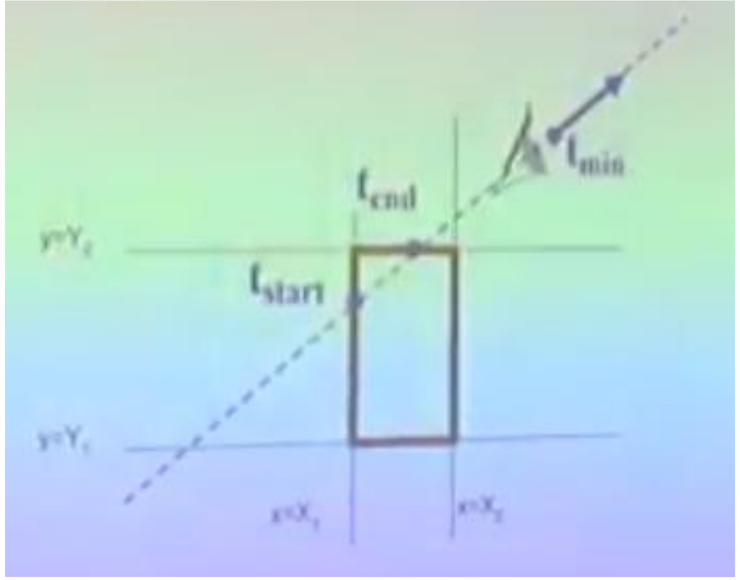
   Else -> closest intersection at t_{end}



1



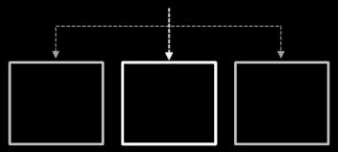
3

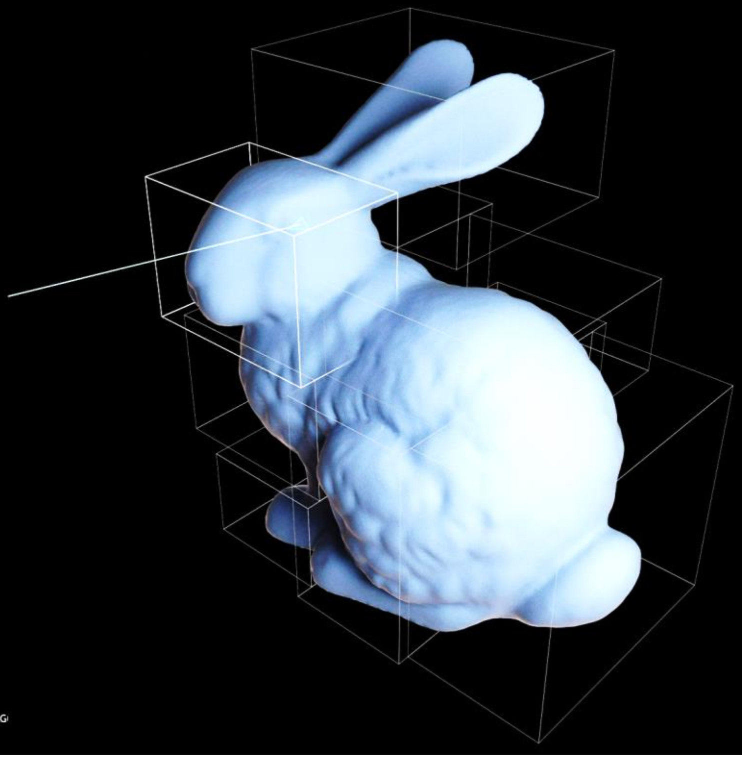


2

BVH ALGORITHM

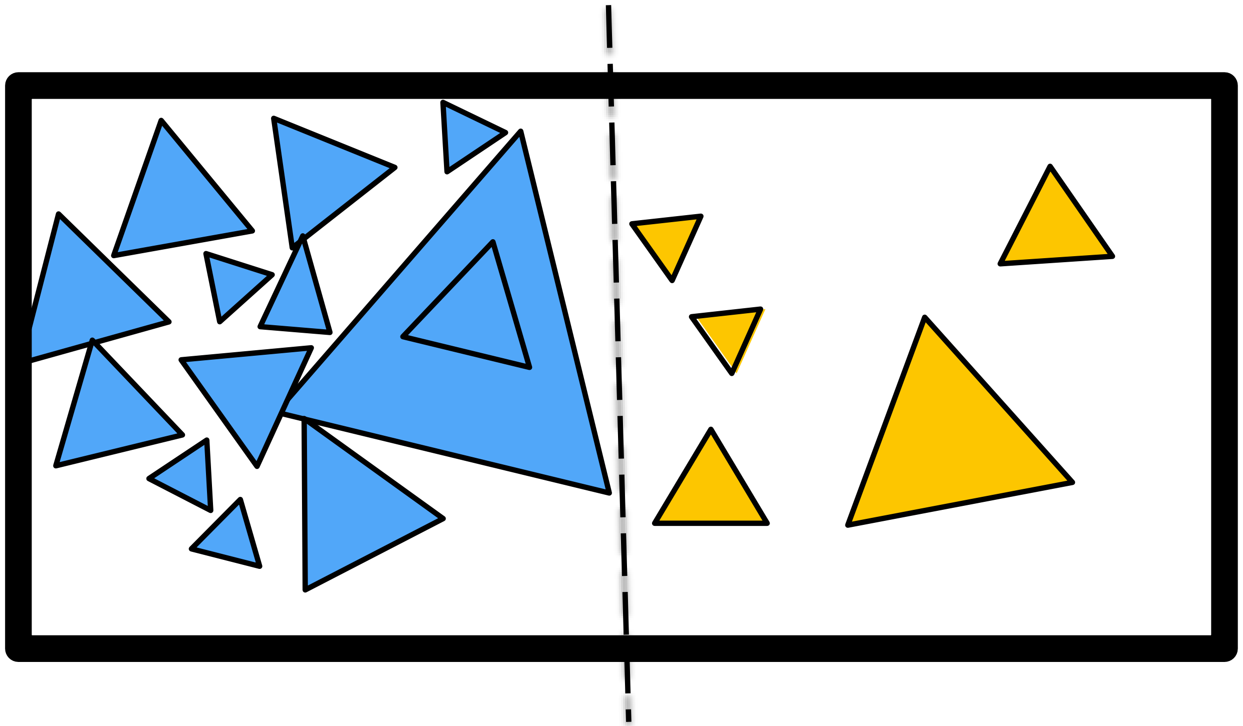
BVH = “Bounding Volume Hierarchy”



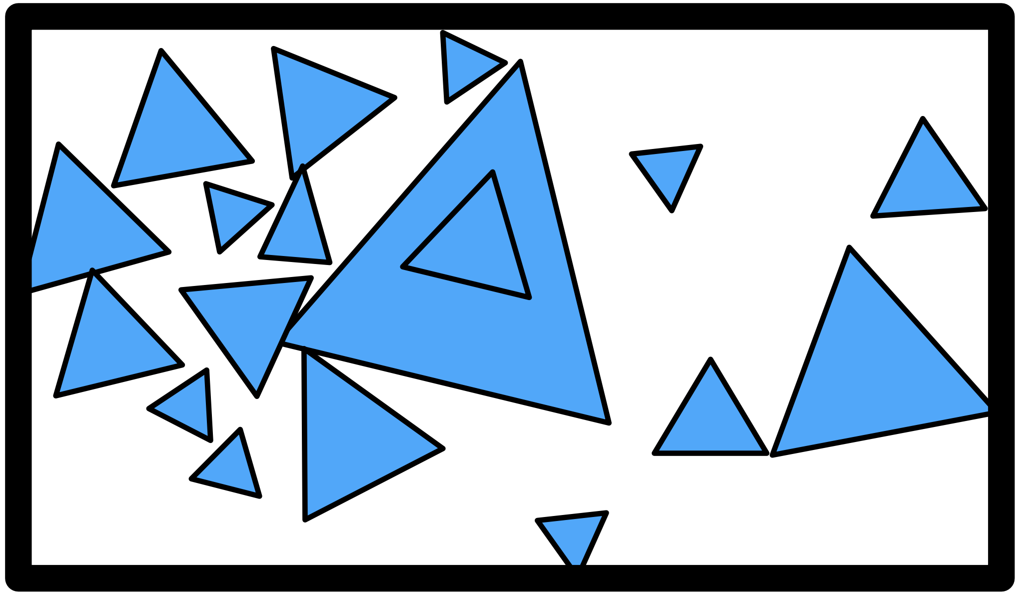


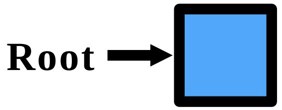
Bounding Volume Hierarchy (BVH)

- Find the bounding box of objects primitives.
- Split objects/primitives into two child BV's
- Recurse, build a binary tree.

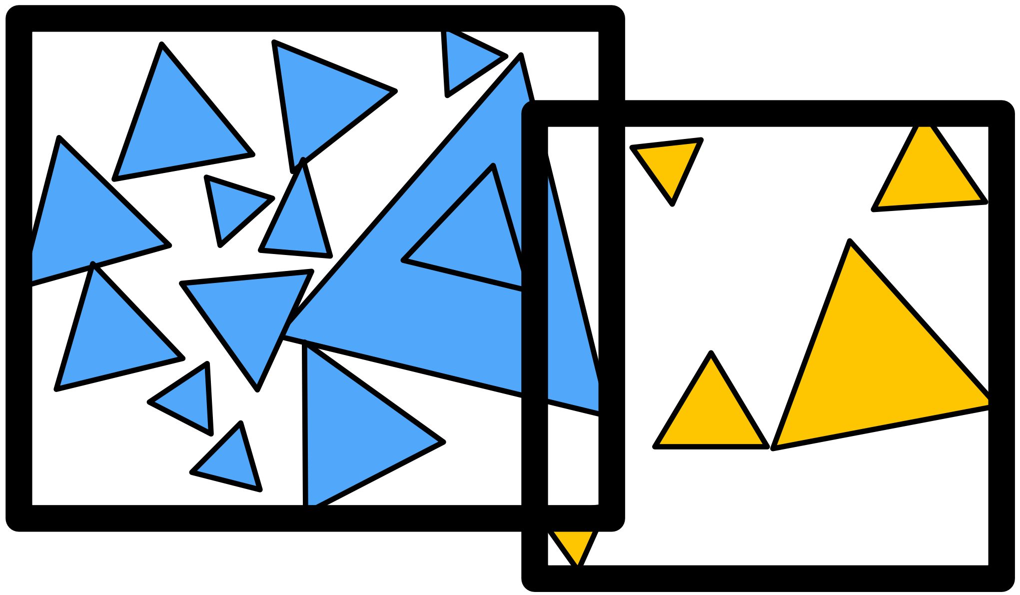


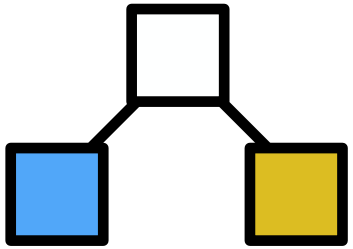
Bounding Volume Hierarchy (BVH)



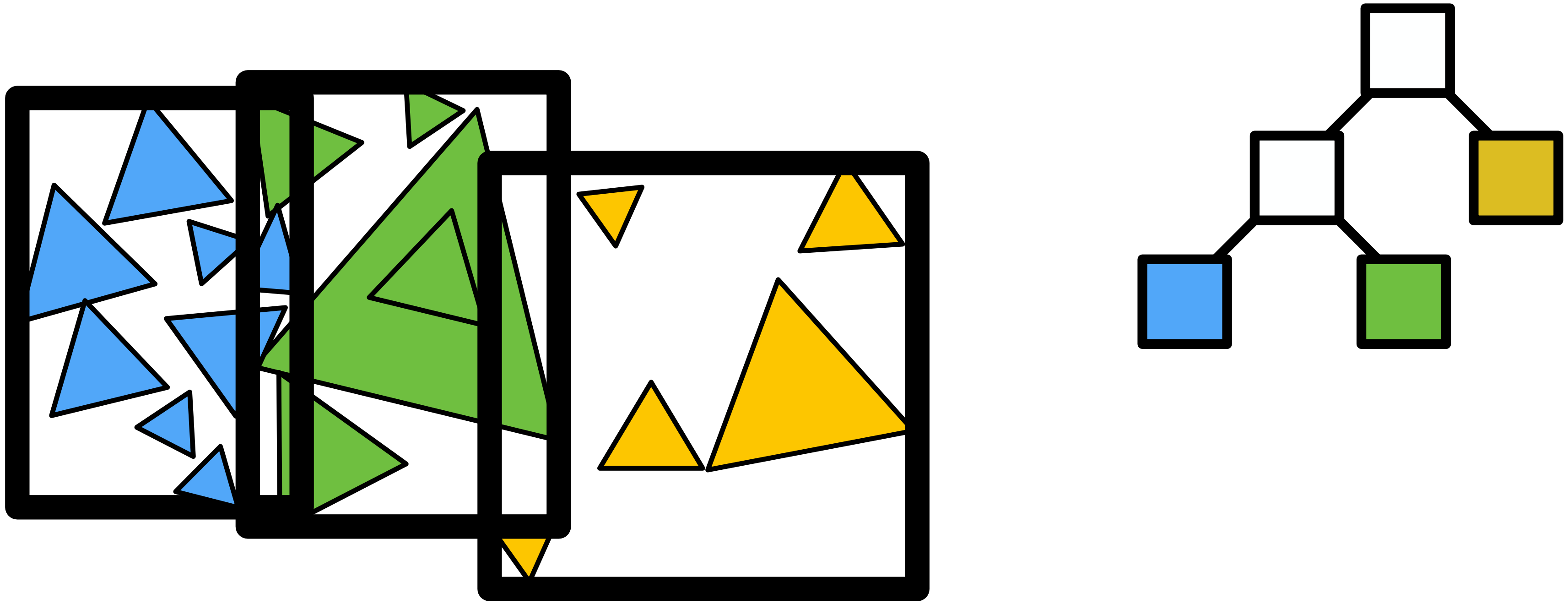


Bounding Volume Hierarchy (BVH)

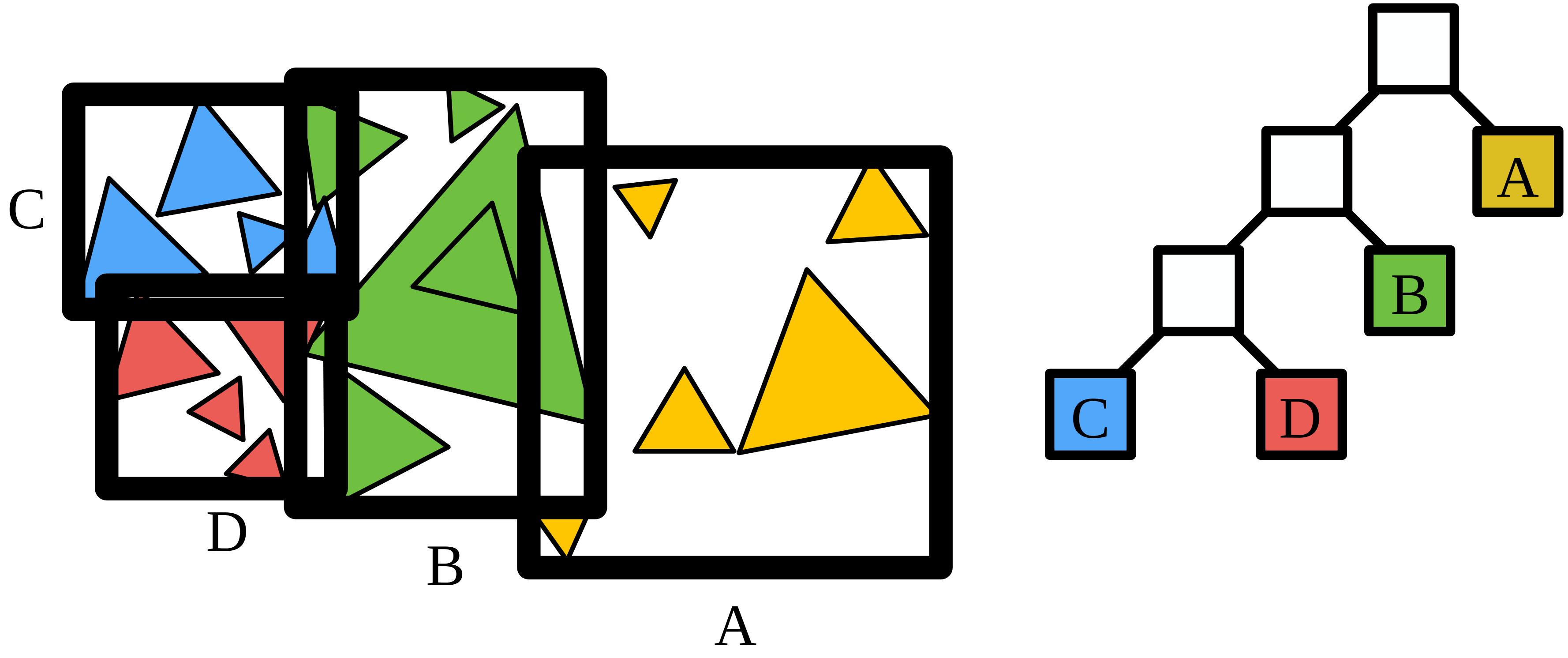


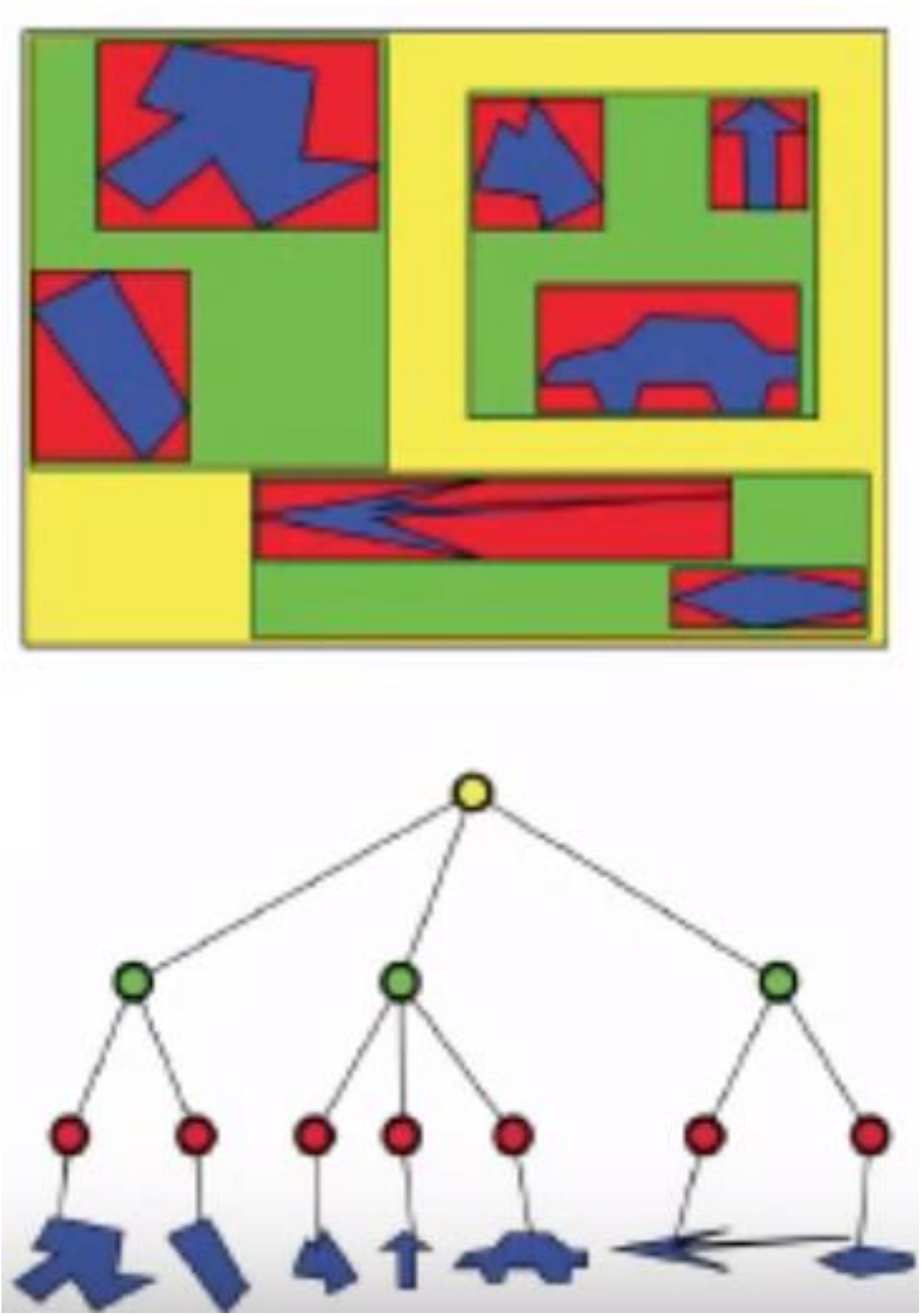


Bounding Volume Hierarchy (BVH)



Bounding Volume Hierarchy (BVH)





Bounding Volume Hierarchy (BVH)

Internal nodes store

- Bounding box
- Children: reference to child nodes

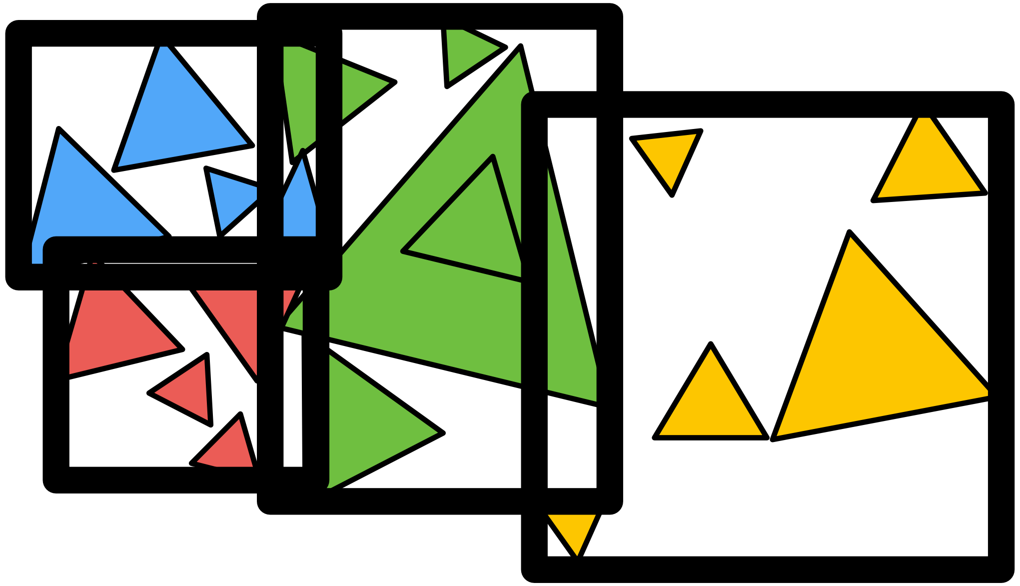
Leaf nodes store

- Bounding box
- List of objects

Nodes represent subset of primitives in scene

- All objects in subtree

BVH Pre-Processing



- Find bounding box
- Recursively split set of objects in two subsets
- Stop when there are just a few objects in each set
- Store obj reference(s) in each leaf node

BVH Pre-Processing

Choosing the set partition

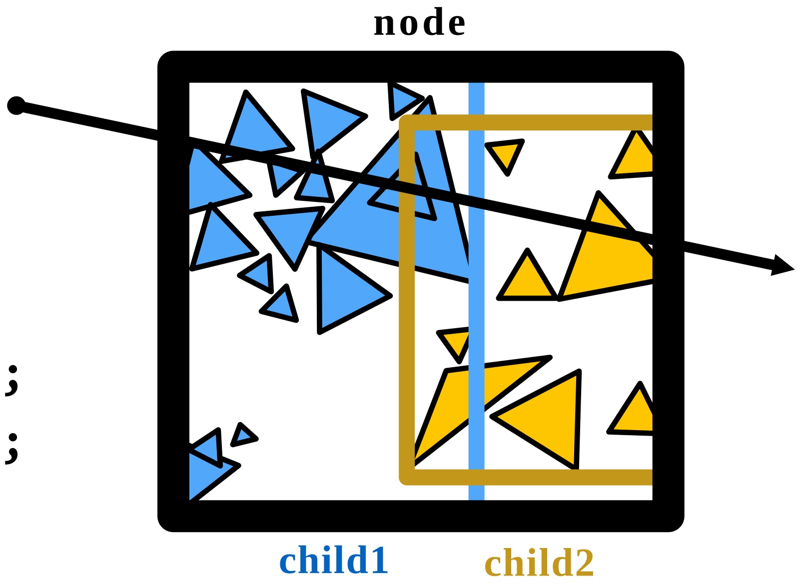
- Choose a dimension to split or optimize over x,y,z
- Simple #1: Split objects around midpoint
- Simple #2: Split at location of median object
- Ideal: split to minimize expected cost of ray intersection

Termination criteria?

- Simple: stop when node contains few elements (e.g. 5)
- Ideal: stop when splitting does not reduce expected cost of ray intersection

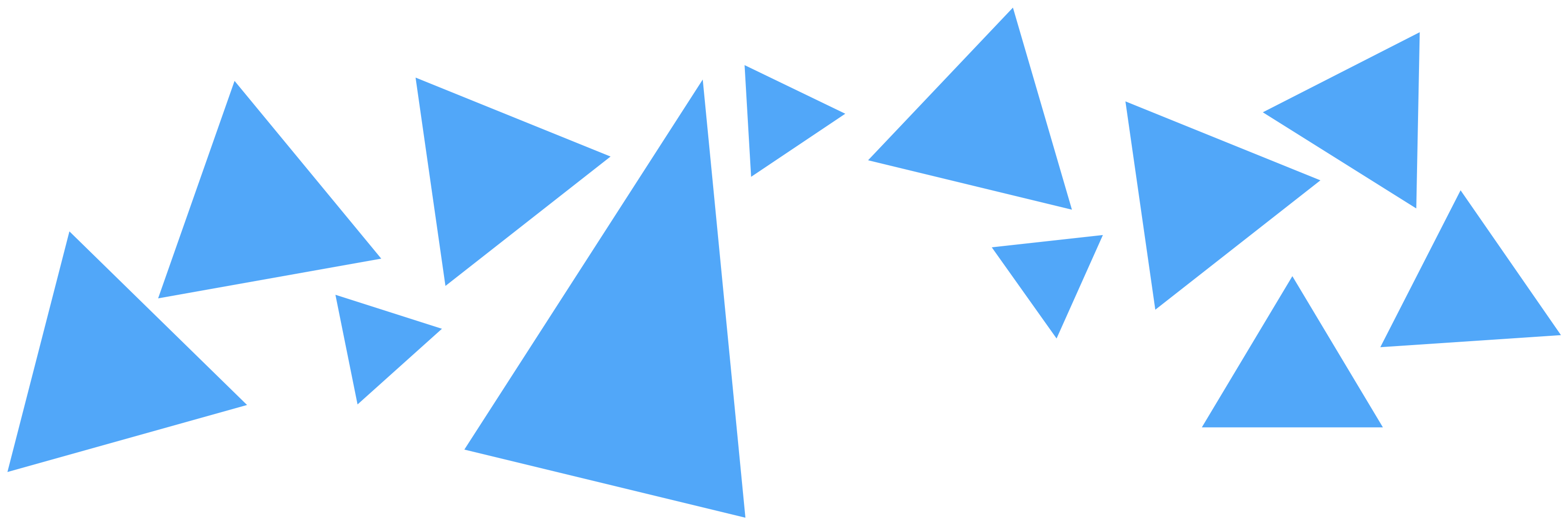
BVH Recursive Traversal

```
Intersect (Ray ray, BVH node)
  if (ray misses node.bbox) return;
  if (node is a leaf node)
     test intersection with all objs;
     return closest intersection;
  hit1 = Intersect (ray, node.child1);
  hit2 = Intersect (ray, node.child2);
  return closer of hit1, hit2;
```

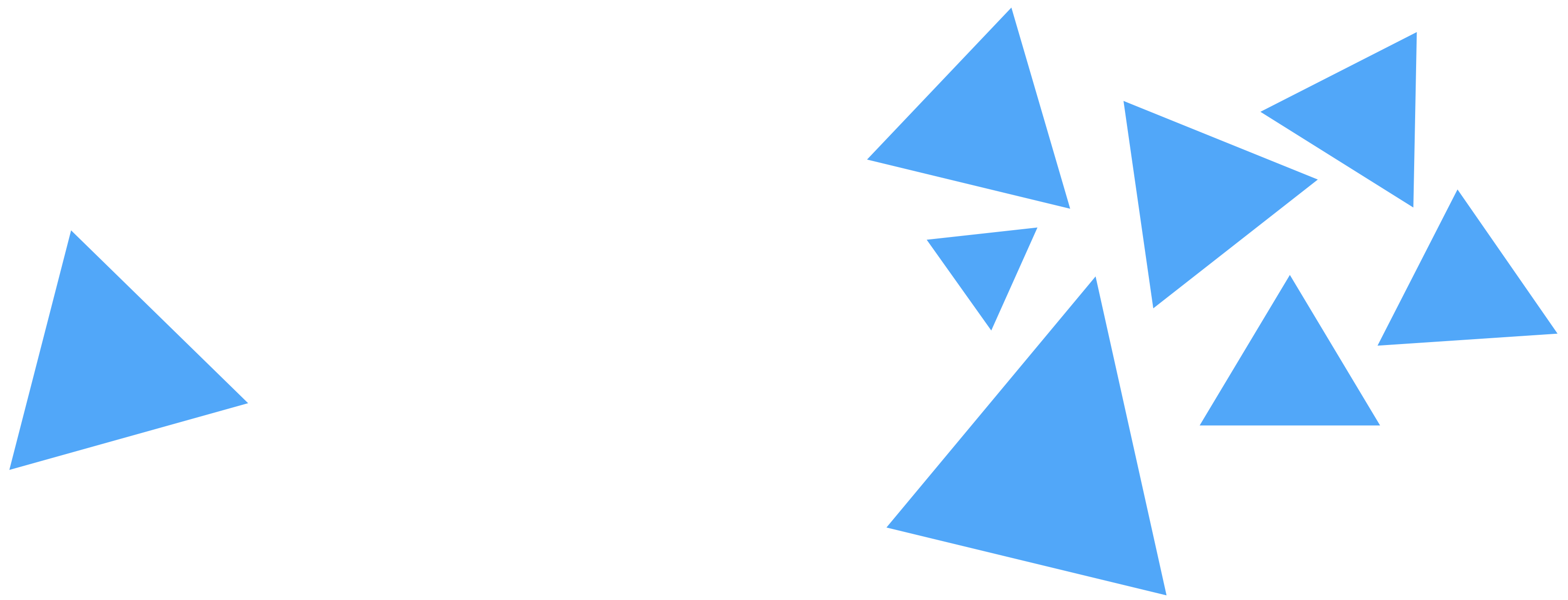


Optimizing Hierarchical Partitions
(How to Split?)

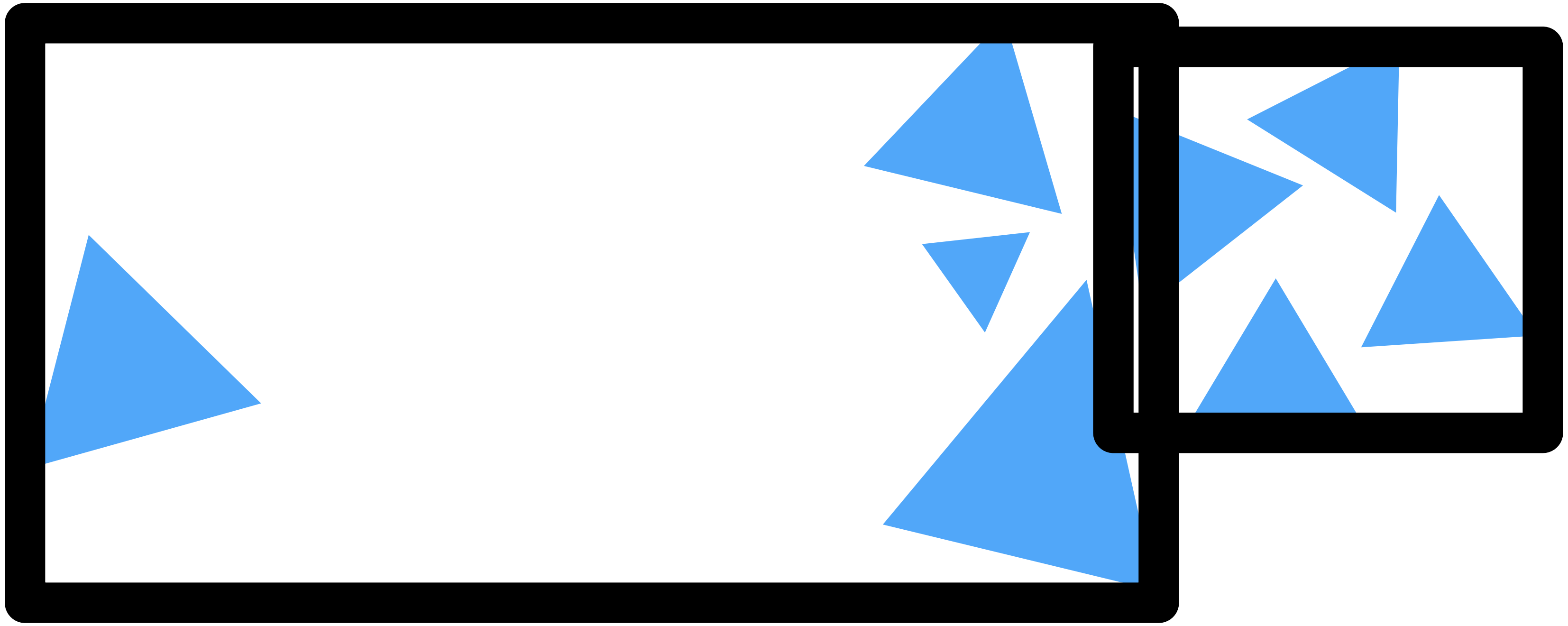
How to Split into Two Sets? (BVH)



How to Split into Two Sets? (BVH)



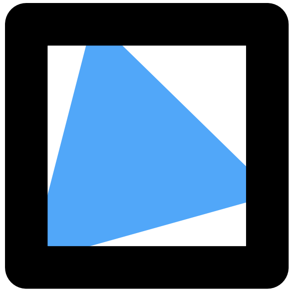
How to Split into Two Sets? (BVH)

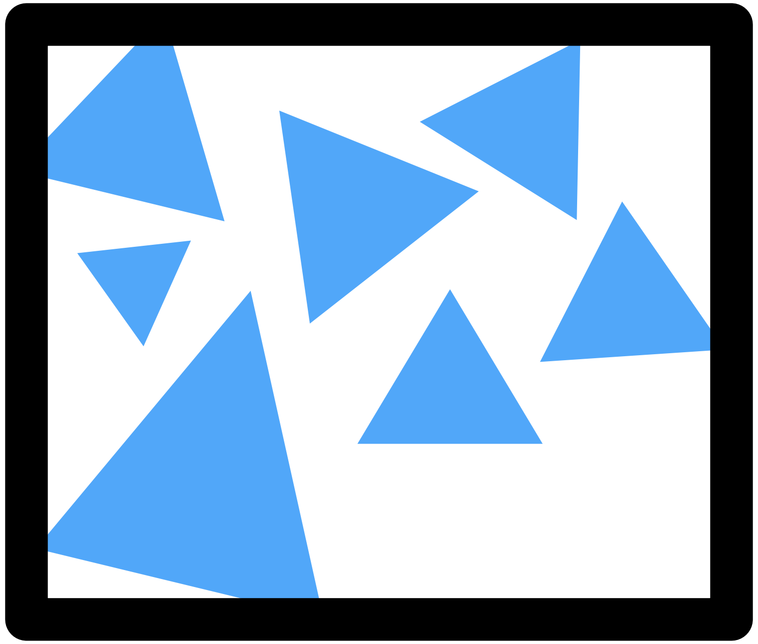


Split at median element

Child nodes have equal numbers of elements

How to Split into Two Sets? (BVH)



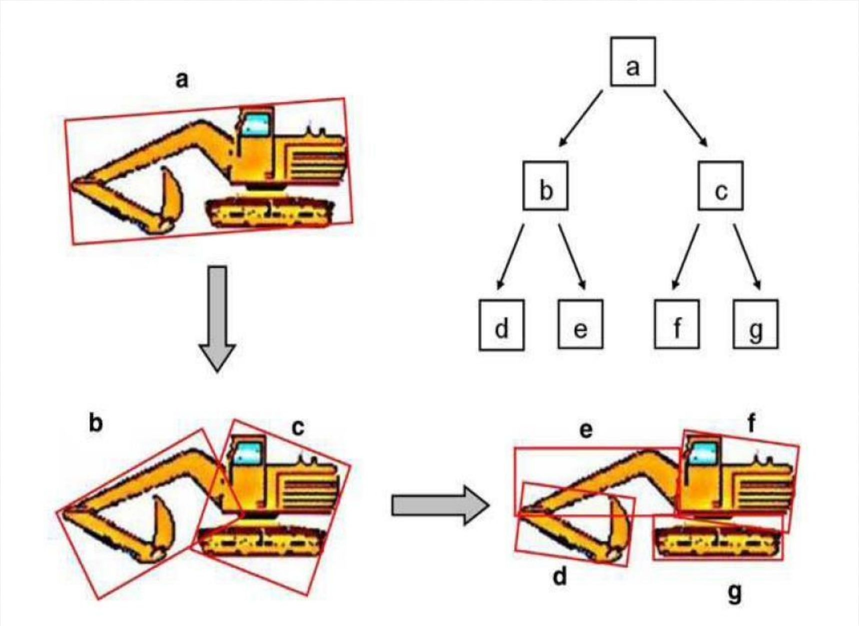


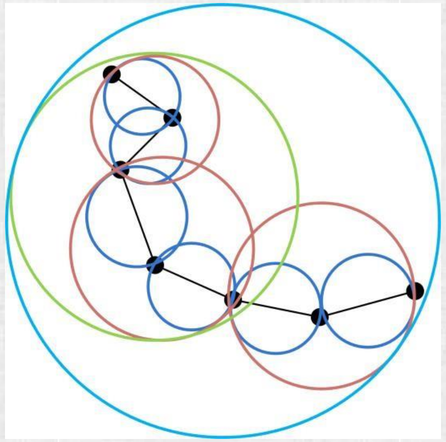
A better split?

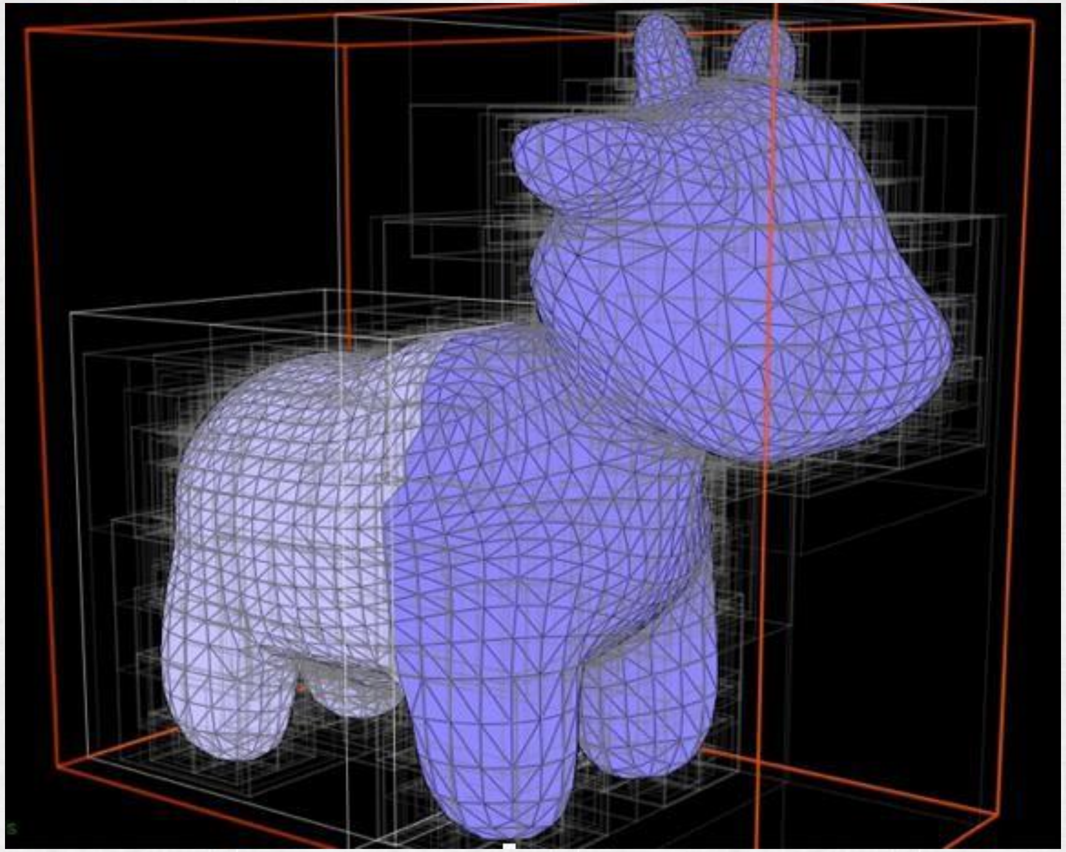
Smaller bounding boxes, avoid overlap and empty space

Aside: Code Acceleration

- For complex objects, build a <u>hierarchical tree structure</u> in object space
- The lowest levels of the tree contain the primitives used for intersections (and have simple geometry bounding them); then, these are combined hierarchically into a $\log n$ height tree
- Starting at the top of a Bounding Volume Hierarchy (BVH), one can prune out many nonessential (missed) ray-object collision checks

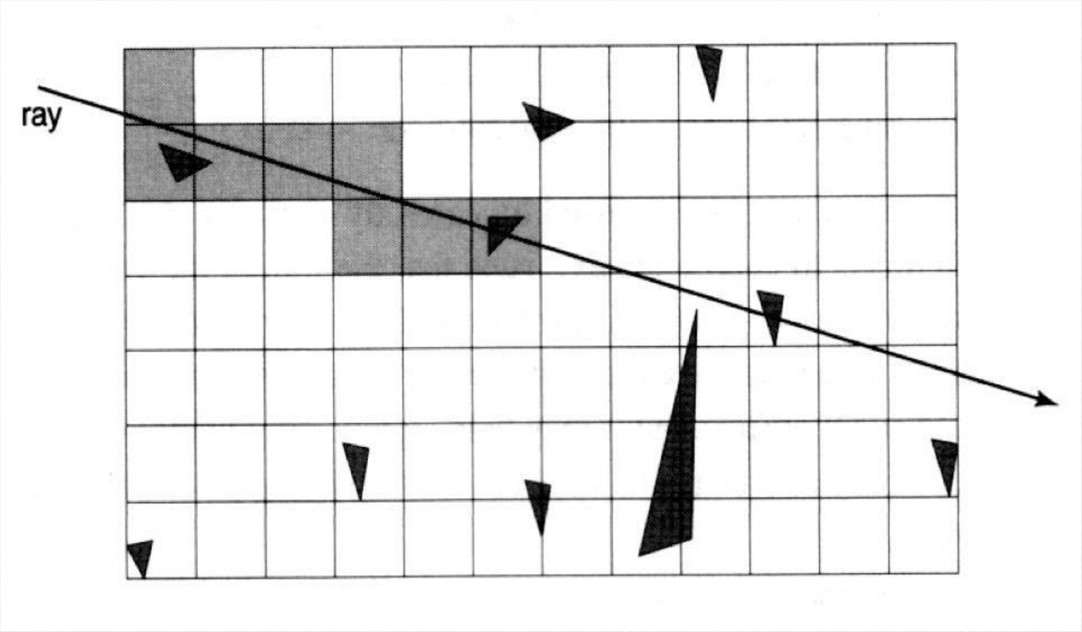


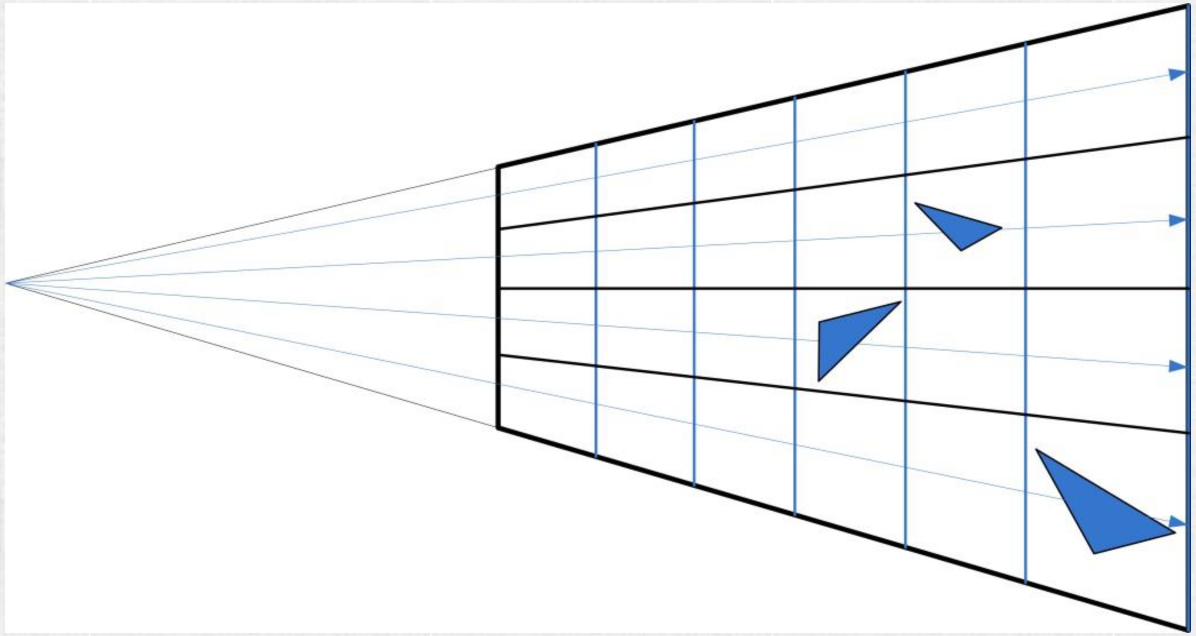




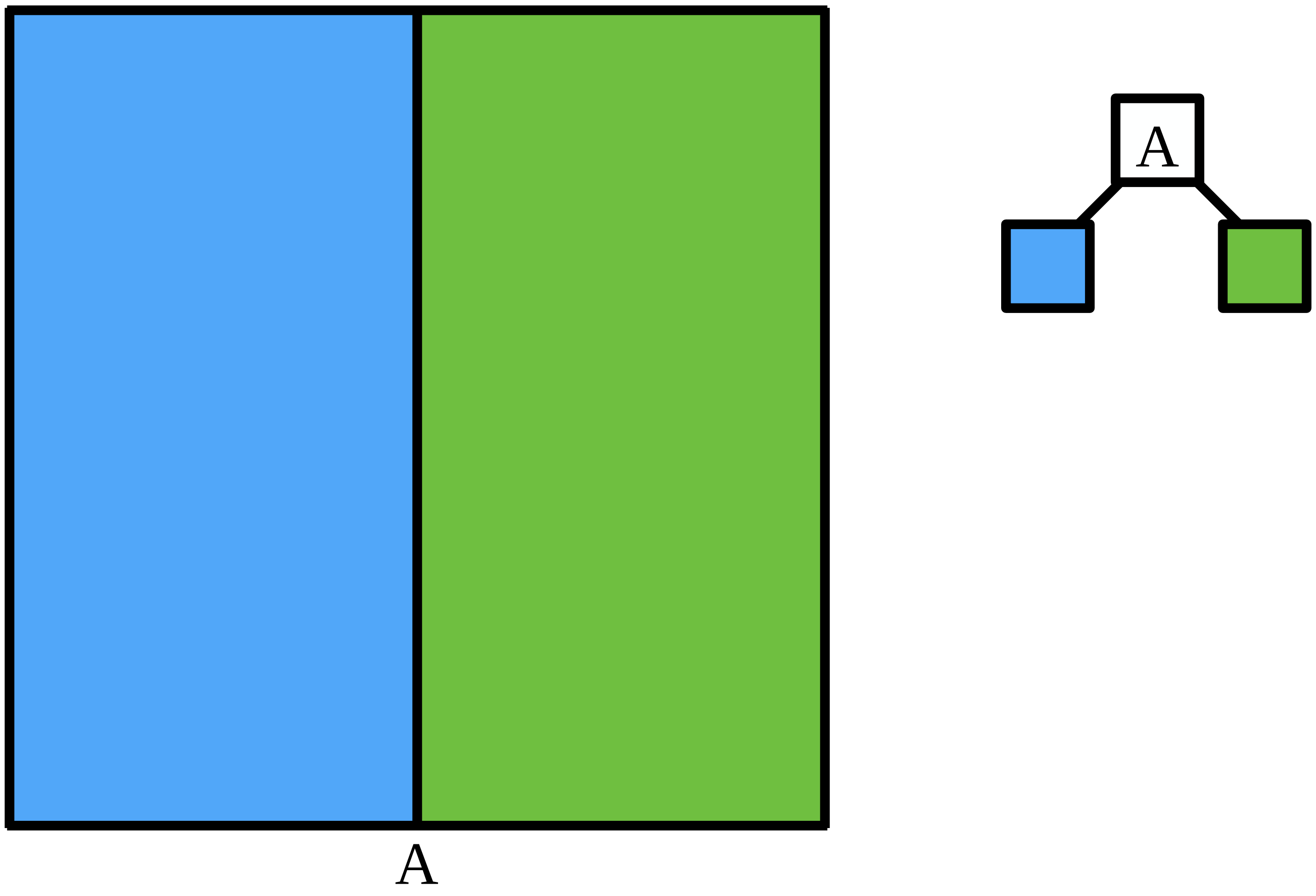
Aside: Code Acceleration

- When there are many objects in the scene, checking rays against all of their top level simple bounding volumes can become expensive
- Thus, world space bounding volume hierarchies, octrees, and K-D trees are used
- Also useful (but flat instead of hierarchical) are uniform spatial partitions (uniform grids) and viewing frustum partitions

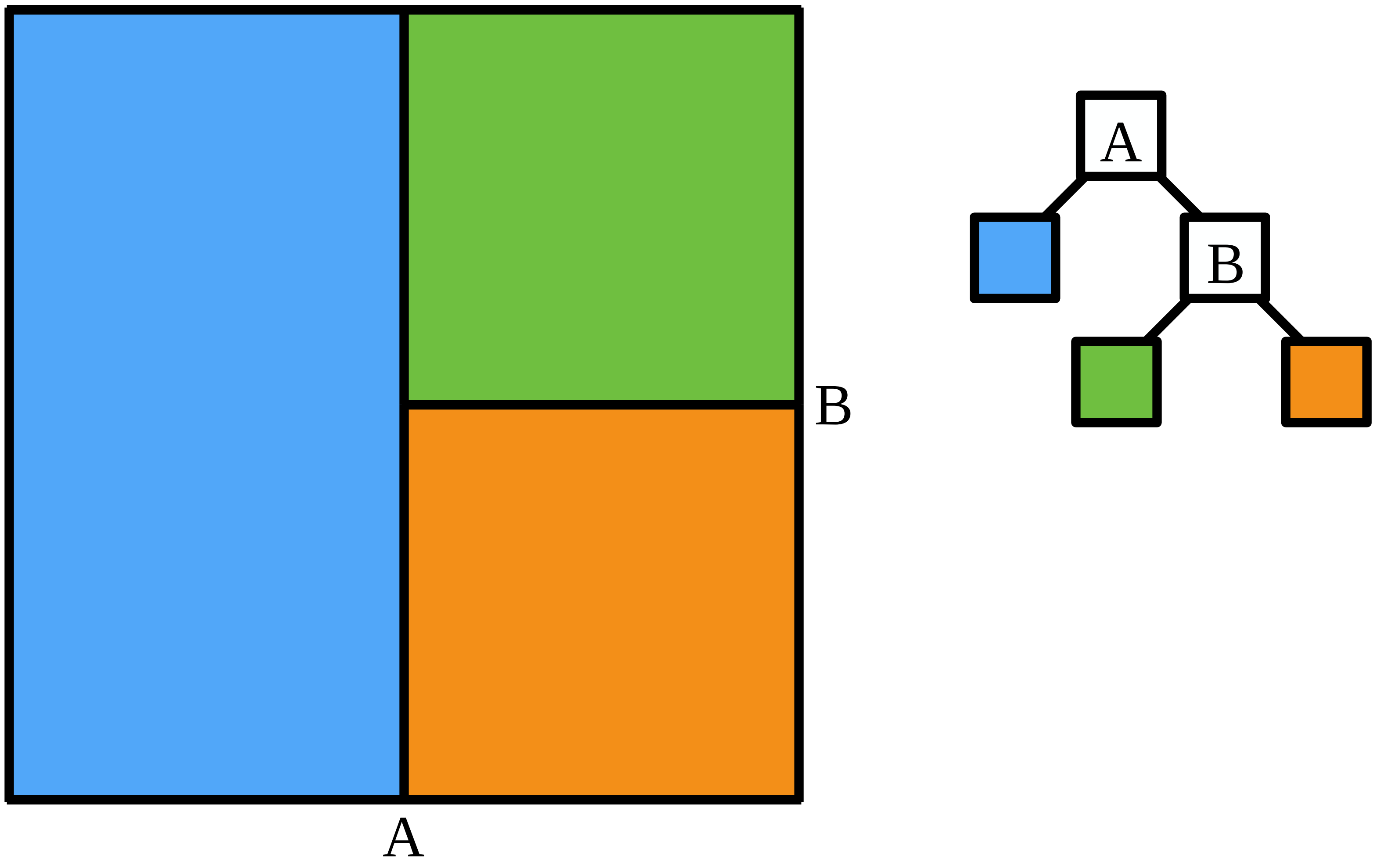




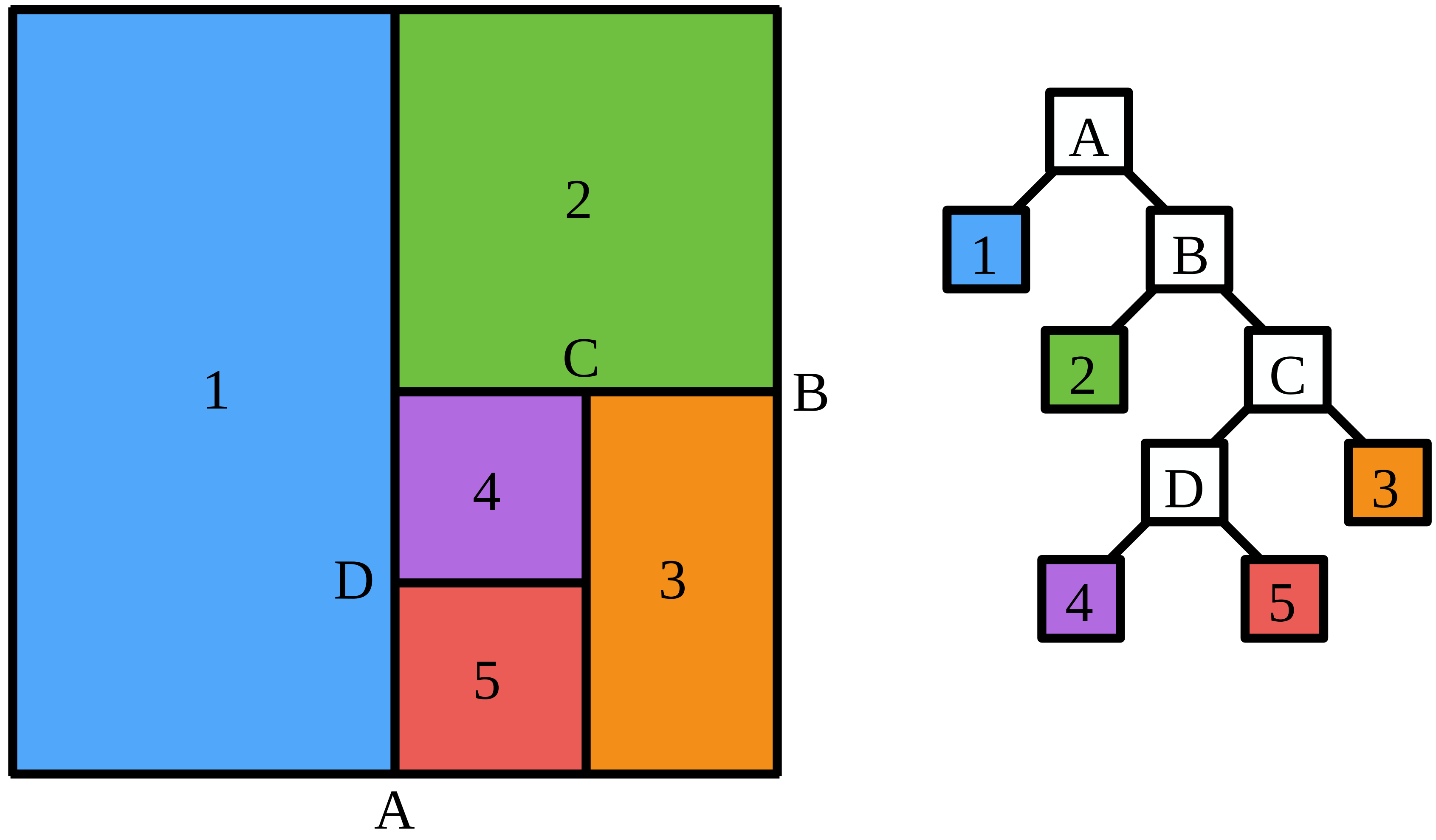
Spatial Hierarchies



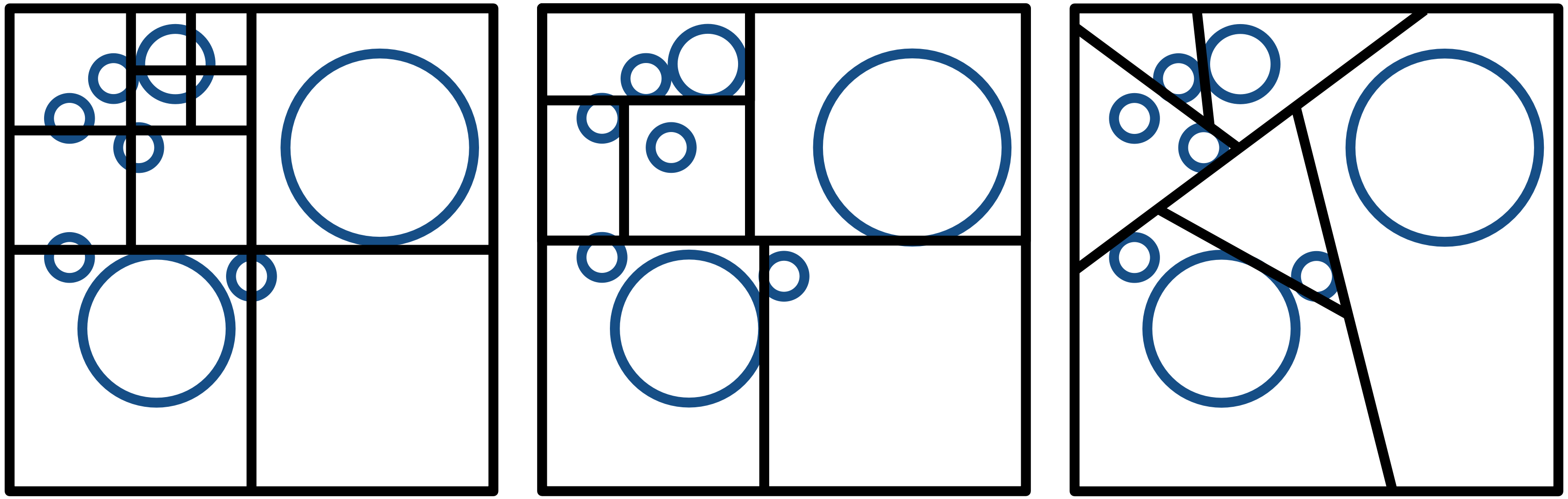
Spatial Hierarchies



Spatial Hierarchies



Spatial Partitioning Variants



Oct-Tree KD-Tree BSP-Tree

KD-Trees

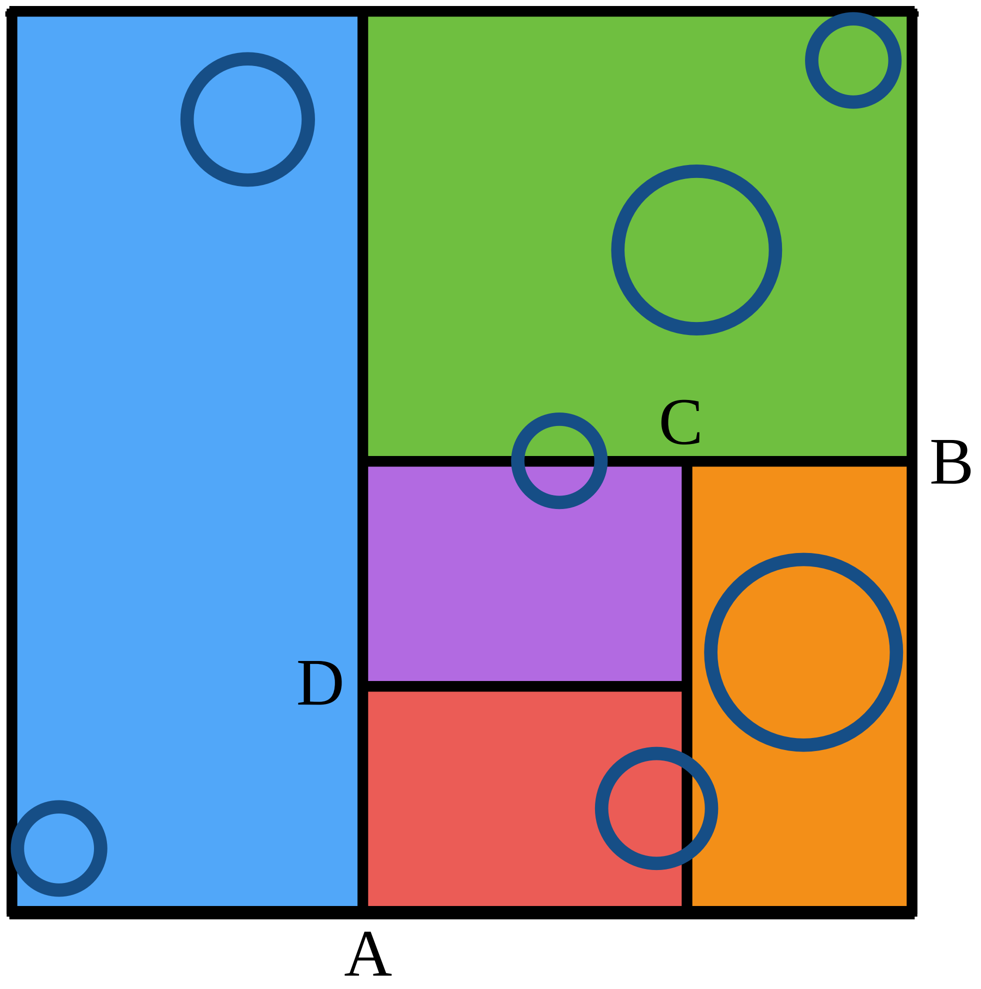
Internal nodes store

- split axis: x-, y-, or z-axis
- split position: coordinate of split plane along axis
- children: reference to child nodes

Leaf nodes store

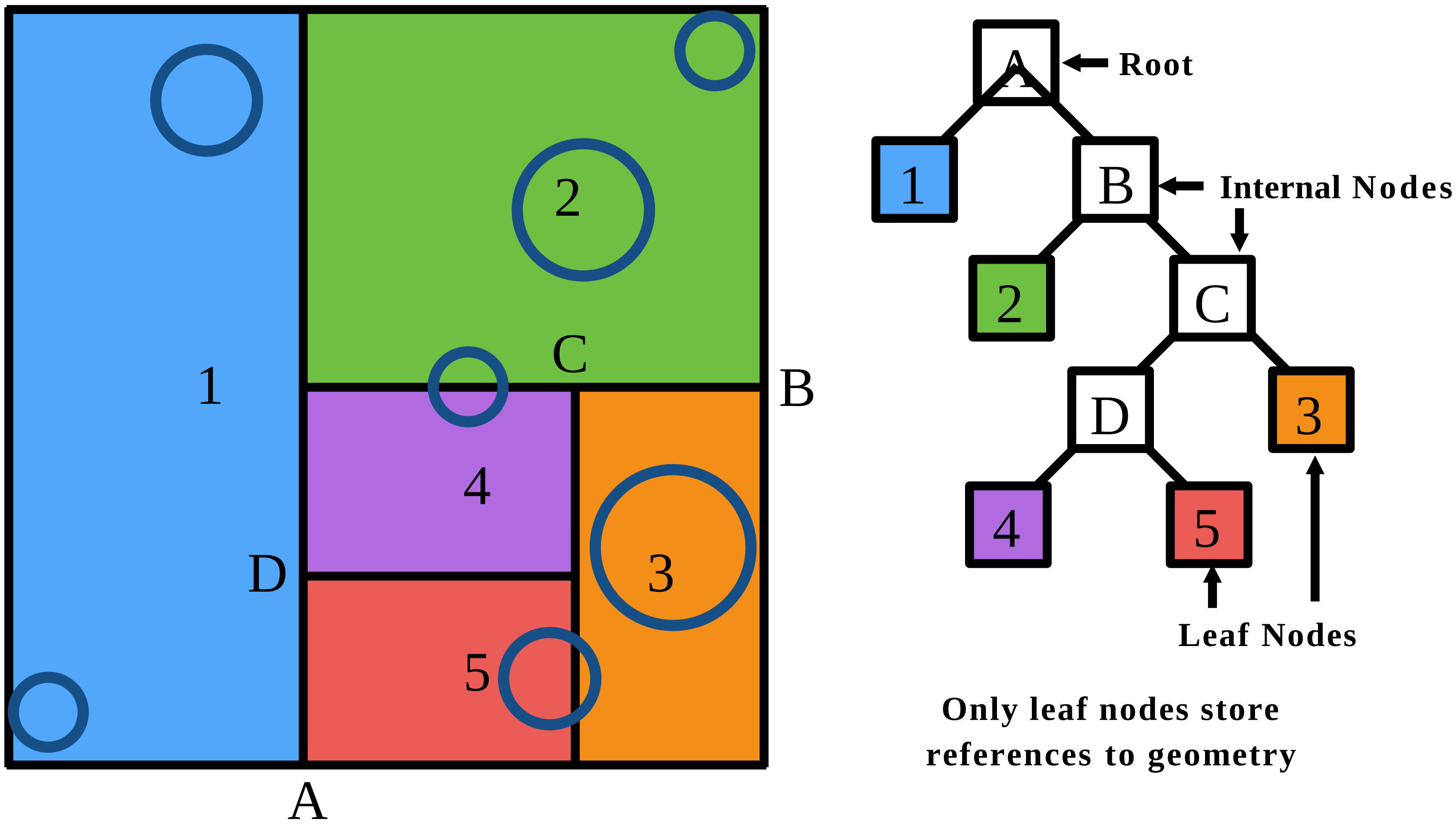
- list of objects
- mailbox information

KD-Tree Pre-Processing



- Find bounding box
- Recursively split cells, axis-aligned planes
- Until termination criteria met (e.g. max #splits or min #objs)
- Store obj references with each leaf node

KD-Tree Pre-Processing



KD-Tree Pre-Processing

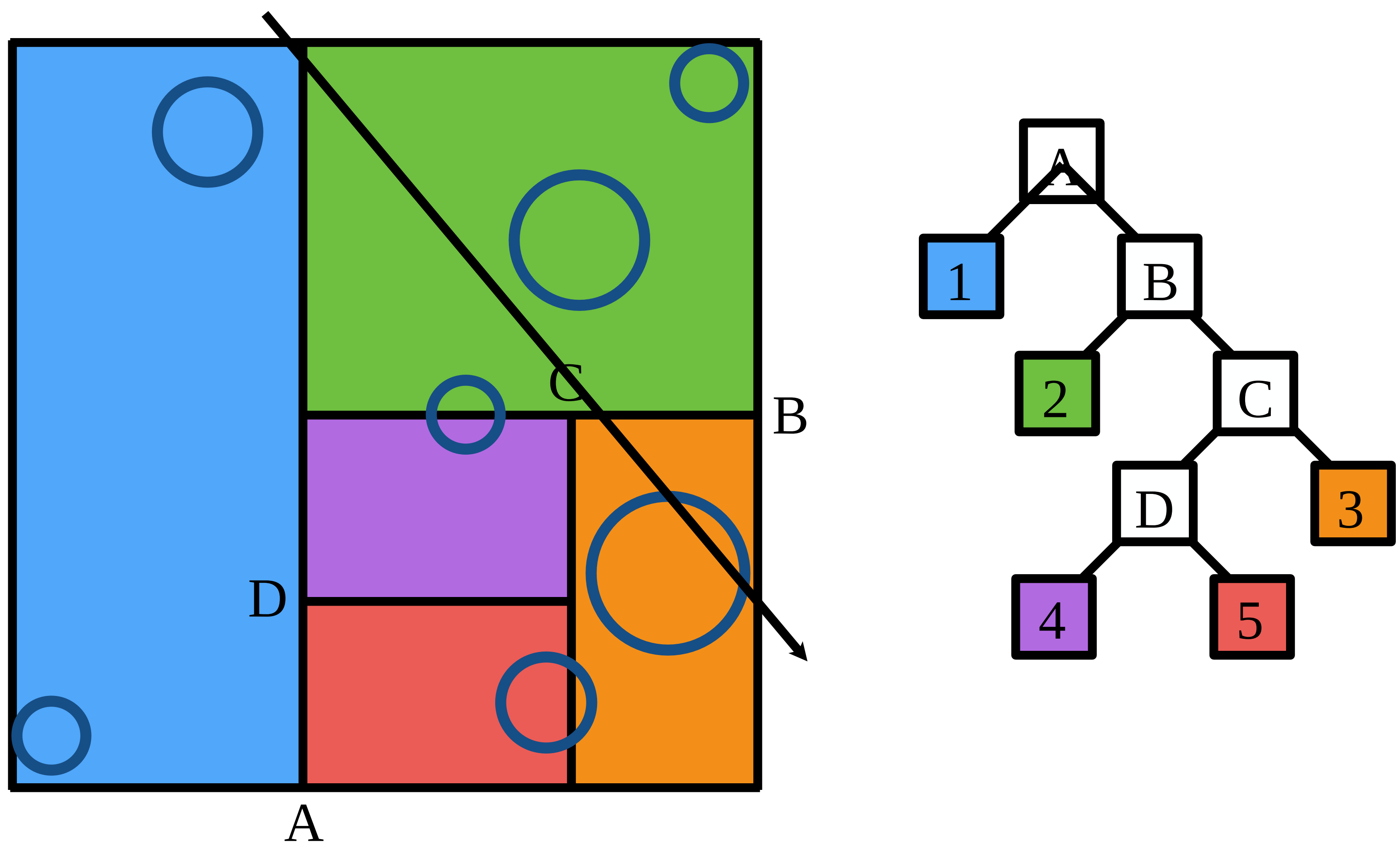
Choosing the split plane

- Simple: midpoint, median split
- Ideal: split to minimize expected cost of ray intersection

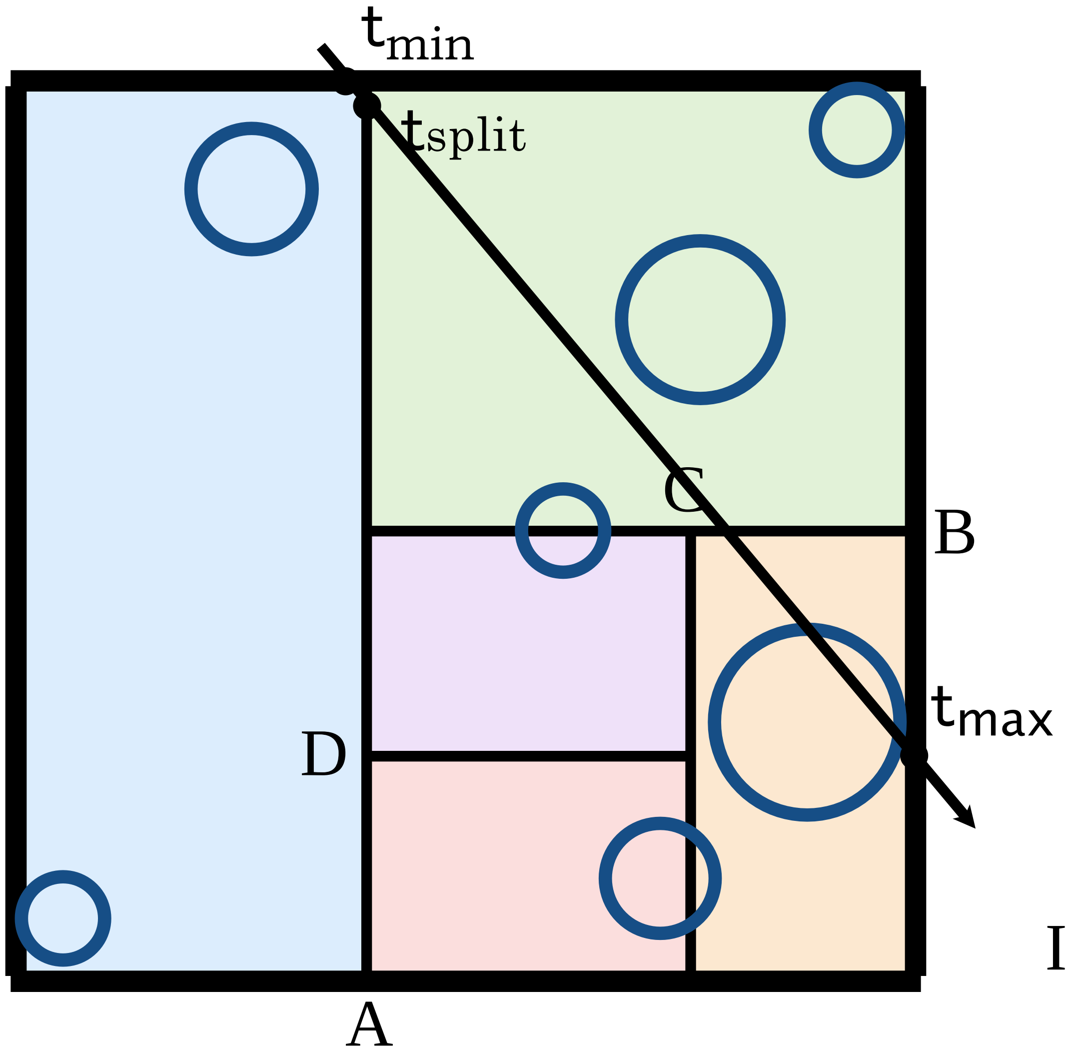
Termination criteria?

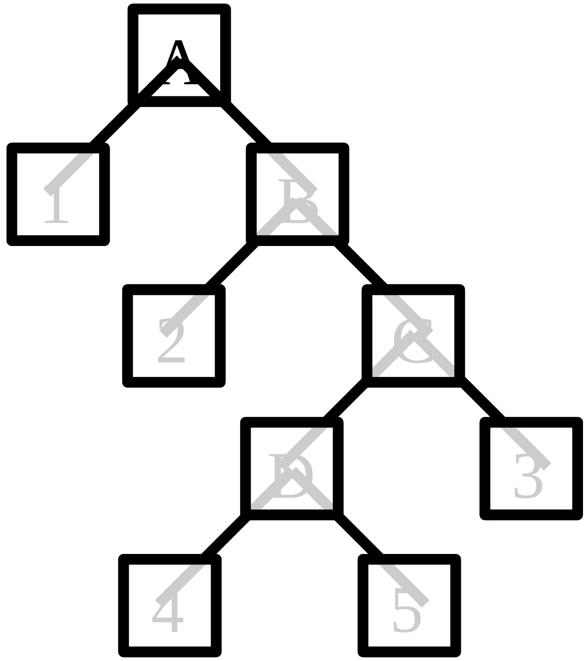
- Simple: common to prescribe maximum tree depth
- Ideal: stop when splitting does not reduce expected cost of ray intersection

Top-Down Recursive In-Order Traversal



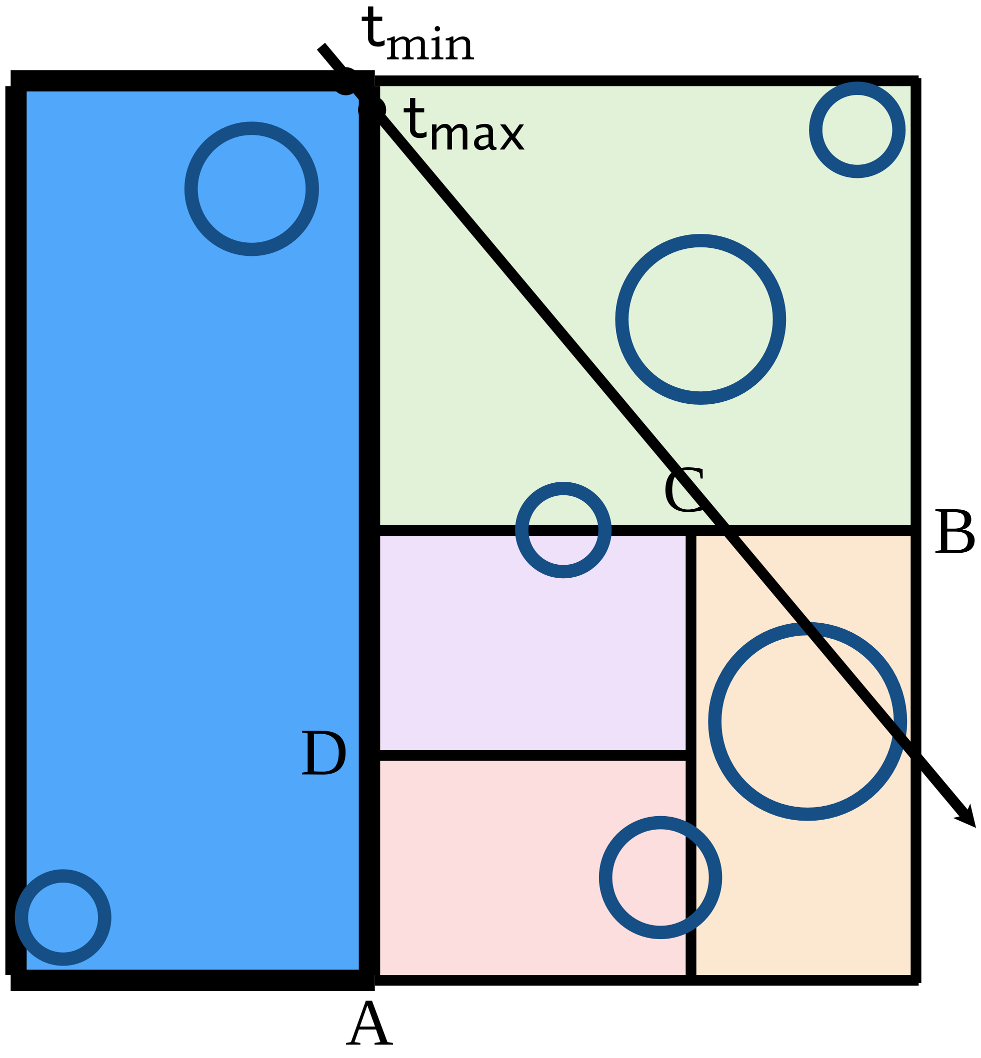
Top-Down Recursive In-Order Traversal

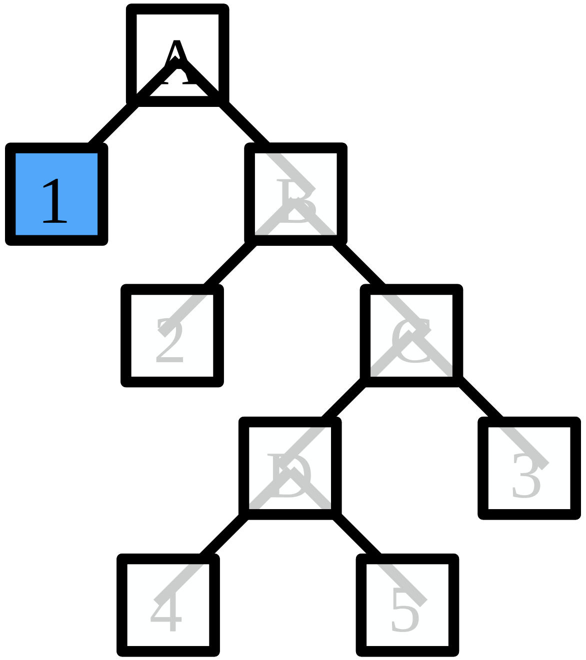




Internal node: split

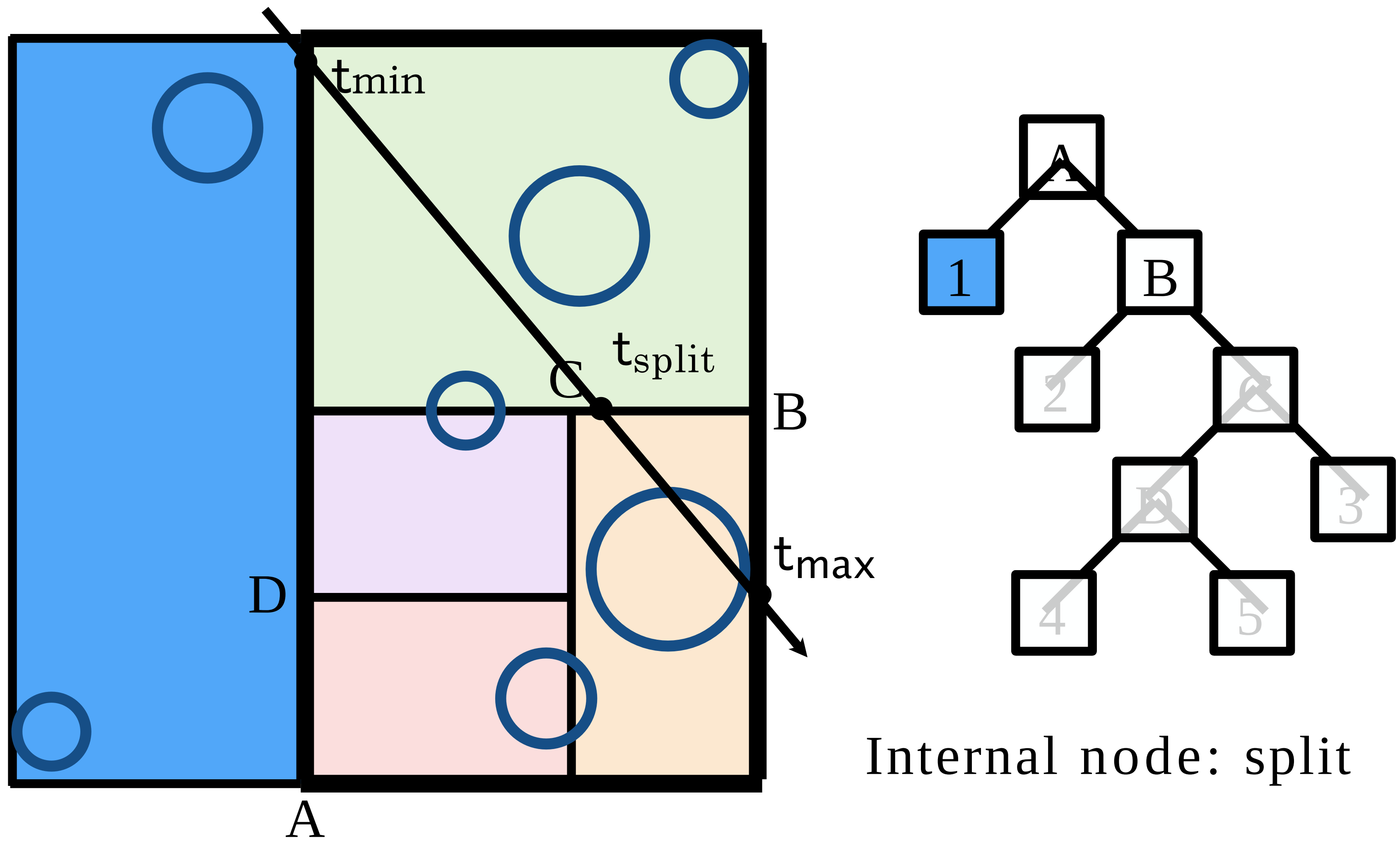
Top-Down Recursive In-Order Traversal





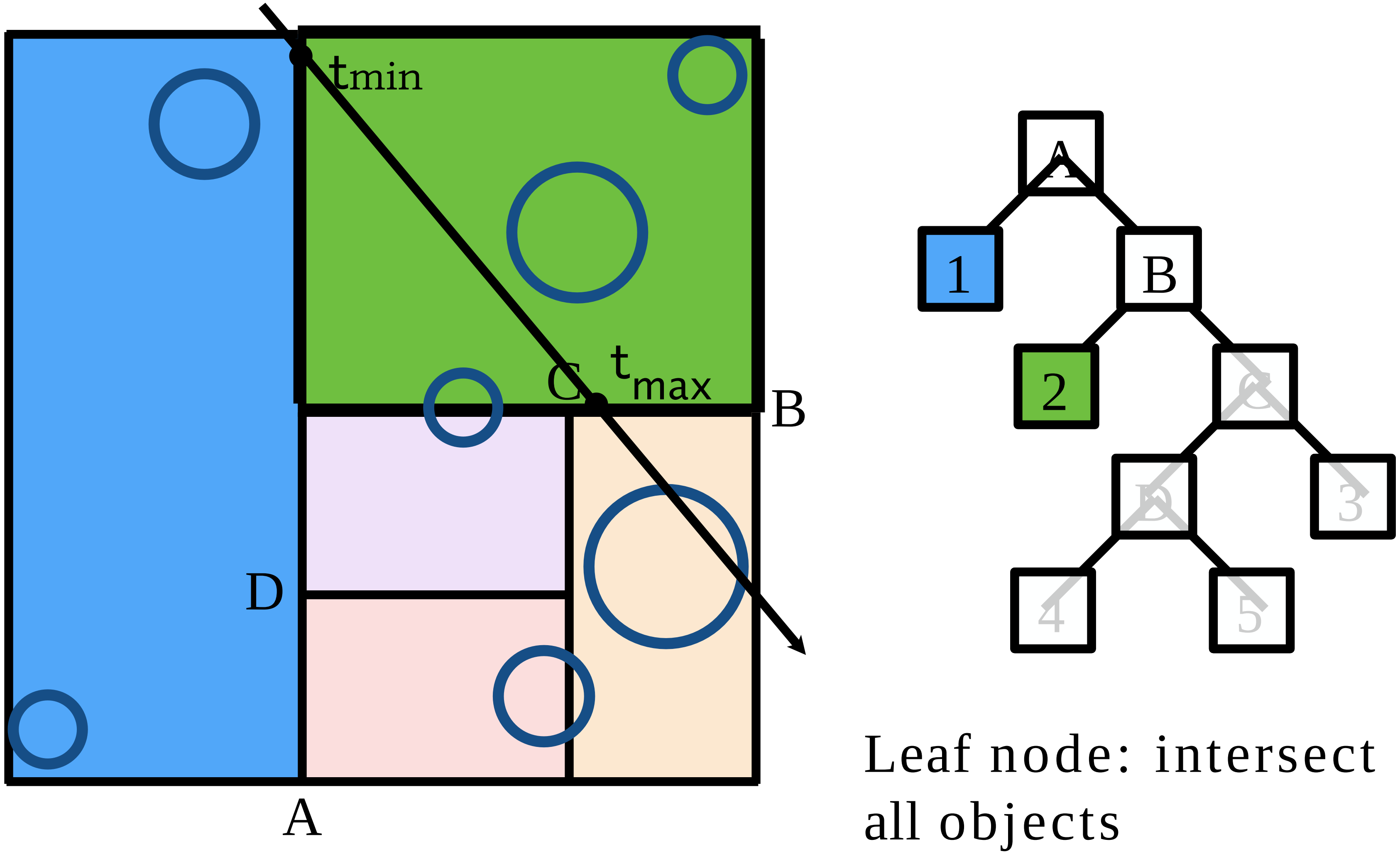
Leaf node: intersect all objects

Top-Down Recursive In-Order Traversal



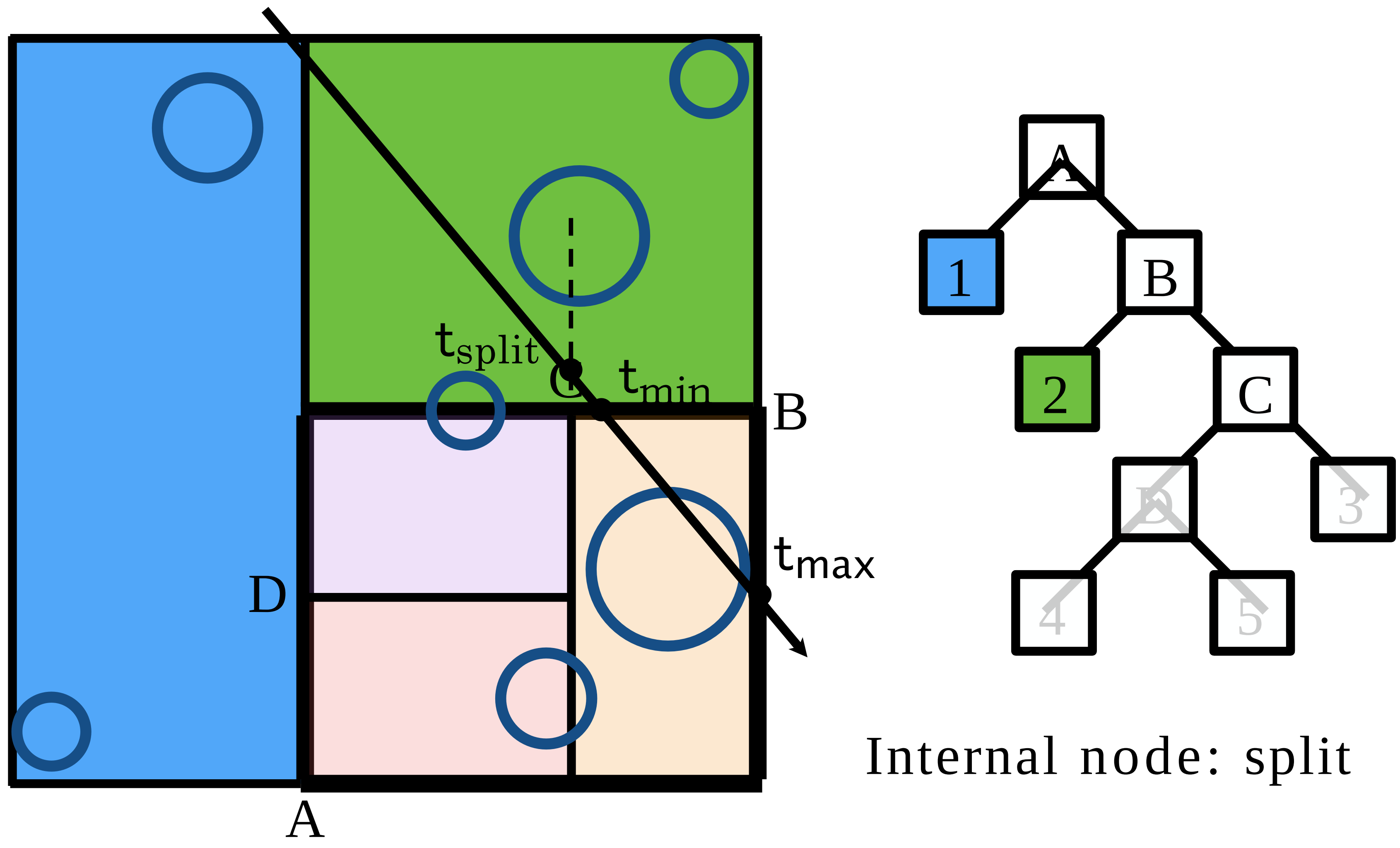
Internal node: split

Top-Down Recursive In-Order Traversal



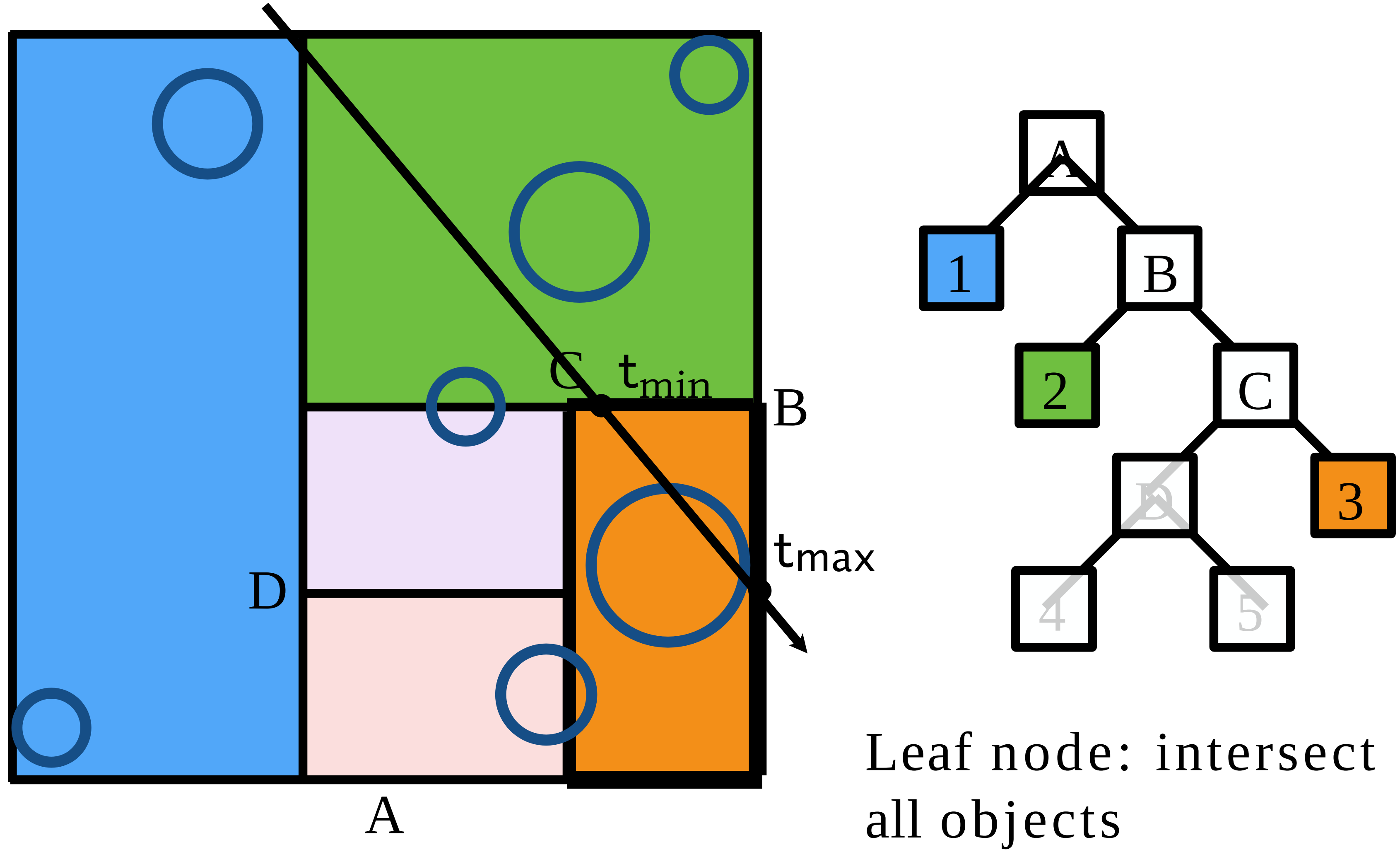
Leaf node: intersect all objects

Top-Down Recursive In-Order Traversal



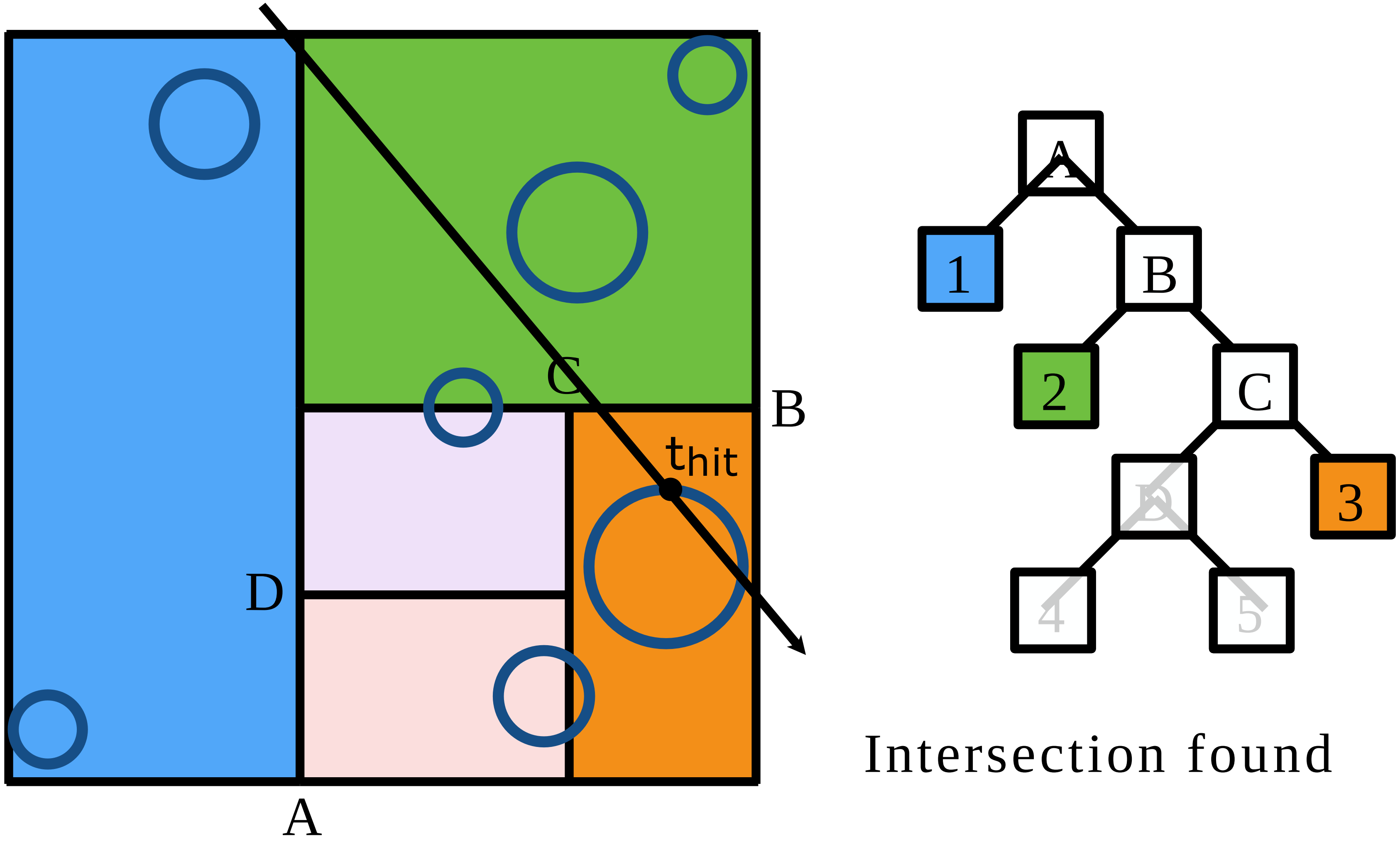
Internal node: split

Top-Down Recursive In-Order Traversal

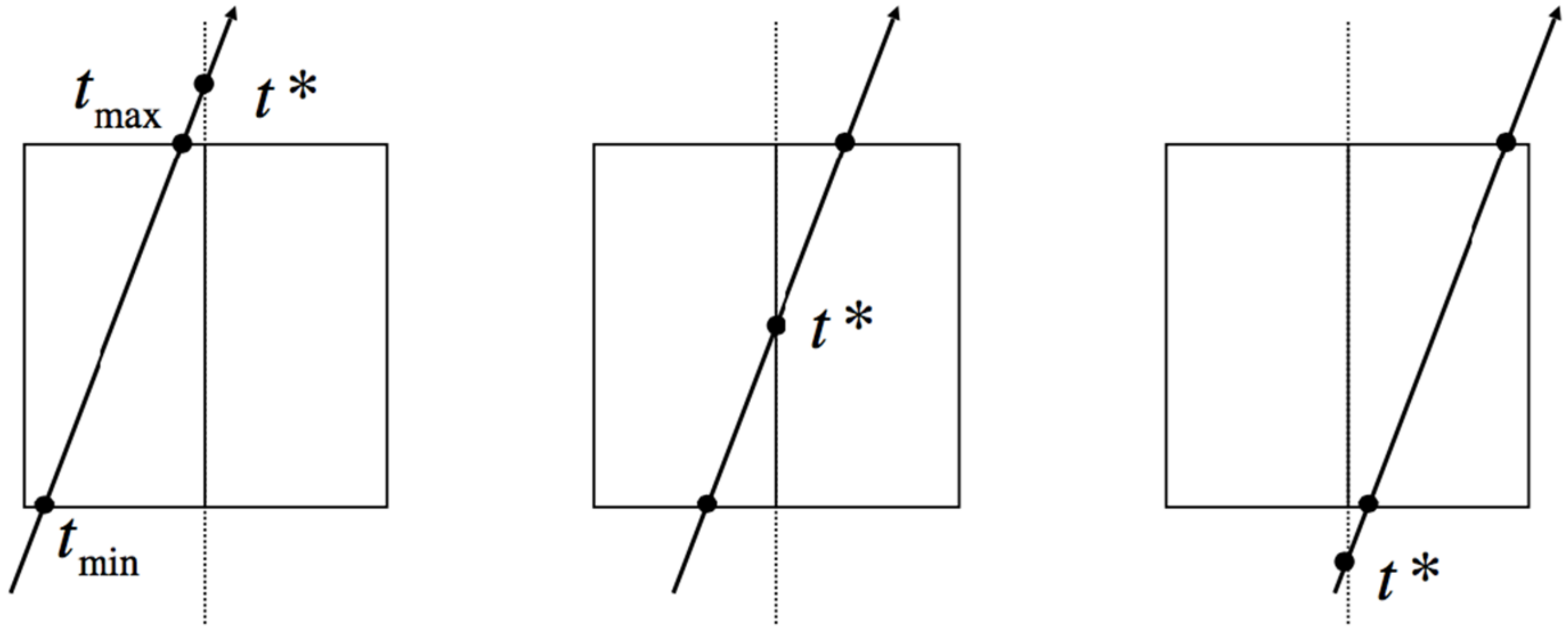


Leaf node: intersect all objects

Top-Down Recursive In-Order Traversal



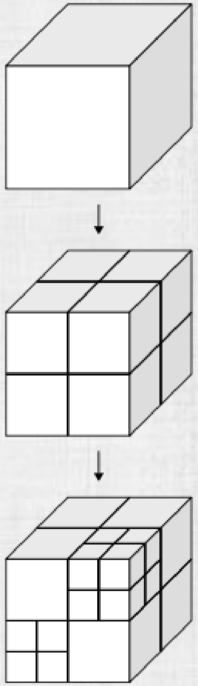
KD-Trees Traversal – Recursive Step

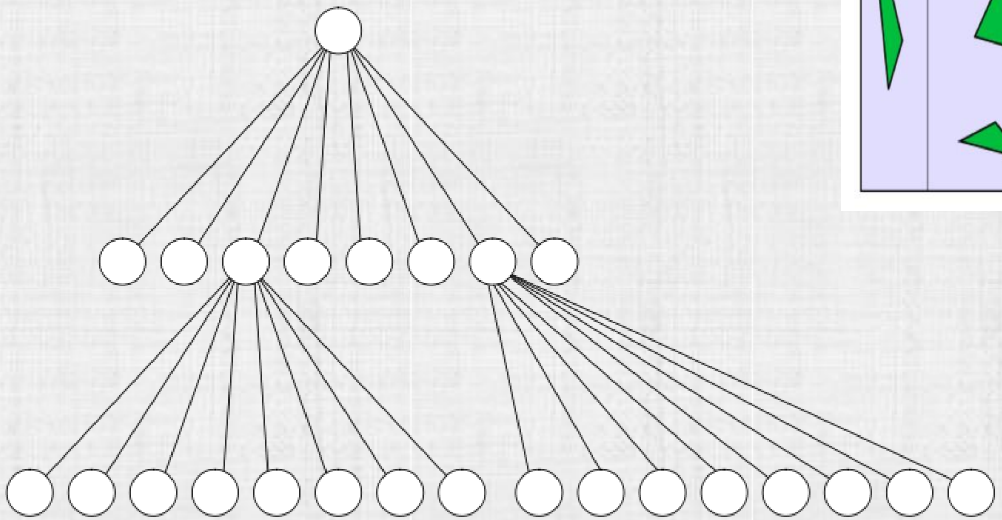


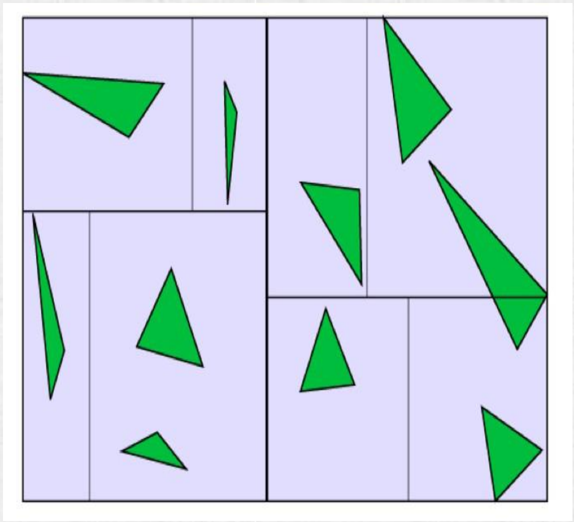
$t_{max} < t^*$	$t_{min} < t^* < t_{max}$	$t^* < t_{min}$
`Intersect(L,tmin,tmax)`	`Intersect(L,tmin,t*)` `Intersect(R,t*,tmax)`	`Intersect(R,tmin,tmax)`

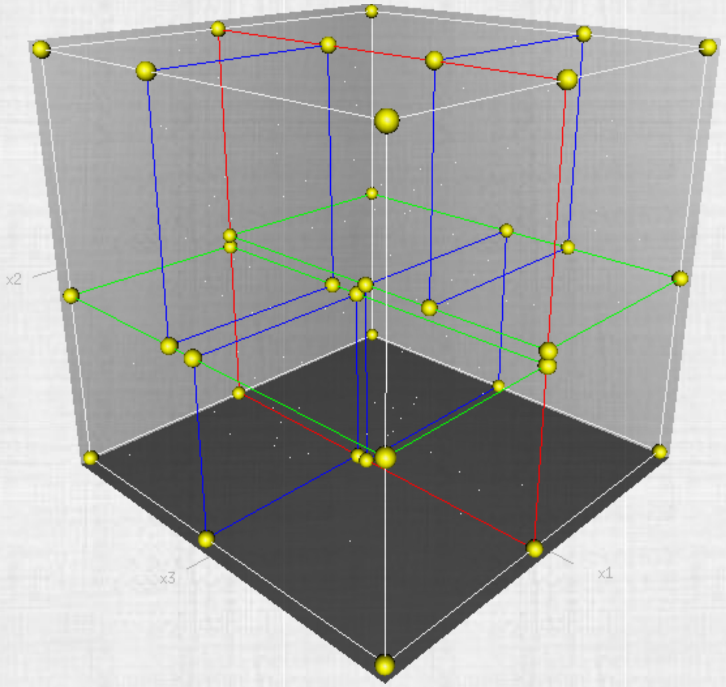
Aside: Code Acceleration

- Instead of a bottom-up bounding volume hierarchy approach, <u>octrees</u> and <u>K-D trees</u> take a top-down approach to hierarchically partitioning objects (and space)





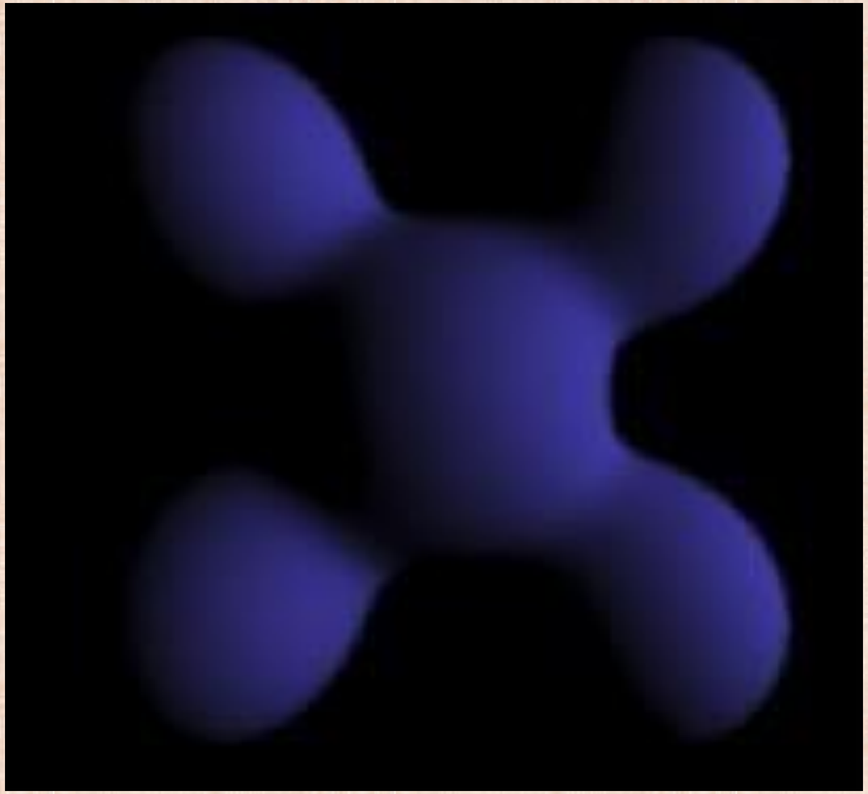




Ambient vs. Diffuse Shading

- Ambient shading colors a pixel when its ray intersects the object
- Diffuse shading attenuates object color based on how far the unit normal is tilted away from the incoming light (note how your eyes/brain imagine a 3D shape)

Ambient



Diffuse