Computer Graphics

- Acceleration Structures -

Philippe Weier

Alexander Rath

Philipp Slusallek

Acceleration Strategies

- **Naïve Ray Tracing is too expensive**
  - Need hundreds of millions rays per second
  - Scenes consist of millions of triangles
- **Faster ray-primitive intersection algorithms**
  - Only reduce complexity by a constant factor. (Still relevant!)
- **Reduce complexity by sorting data as a pre-process**
  - Acceleration structures
    - Guide range search process: dictionaries for ray tracing
    - Limit distance of the search along the ray
  - Eliminate intersection candidates
    - Can reduce average complexity from O(n) to $O(\log n)$
    - Worst case still $O(n)$

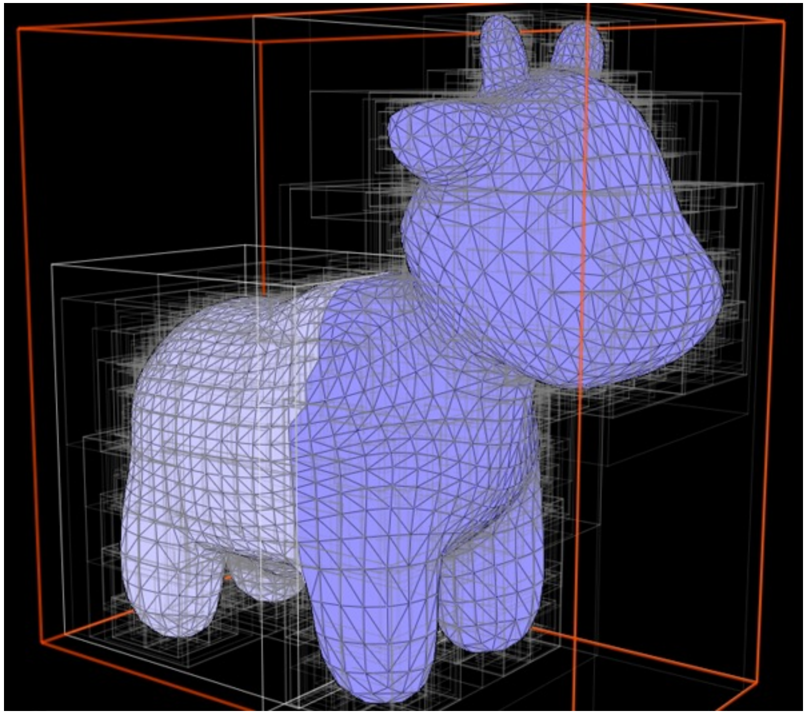
Acceleration Structures

- **Object Partitioning**
  - Partition objects into groups
  - Store in a data structure (tree)
  - Every object appears once in the data structure
  - Possible spatial overlap
- **Spatial Partitioning**
  - Subdivide space into disjoint fully covering regions
  - Store in a data structure (tree or table)
  - Every region appears once in the data structure
  - Possibly multiple references to the same object
- **Directional Partitioning**
  - Subdivide directions into cells
- **5D Partitioning**
  - Subdivide space and direction
  - Close to pre-compute visibility for all points and all directions

OBJECT PARTITIONING

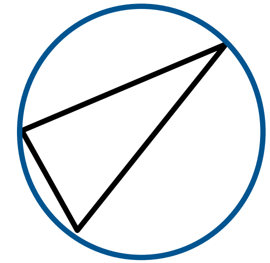
Bounding Volumes

- **Key Idea**
  - Enclose complex geometry in simple bounding volume
  - A ray missing the bounding volume misses the object
  - Only test object intersection if ray hits bounding volume

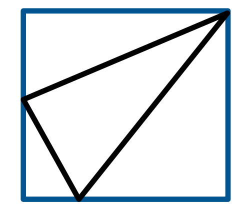


Bounding Volumes Types

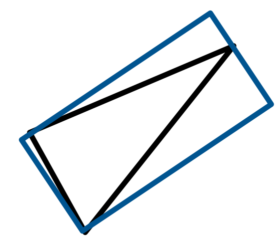
- **Sphere**
  - Very fast intersection computation
  - Often inefficient because too large



- **Axis-aligned bounding box (AABB)**
  - Simple intersection computation
  - Very simple extension of min-max bounds
  - Sometimes too large



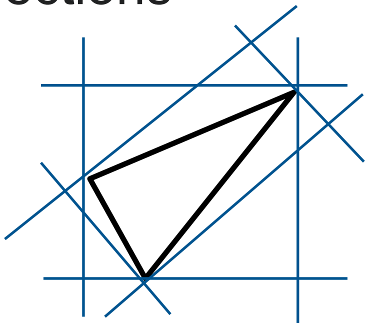
- **Non-axis-aligned box**
  - A.k.a. oriented bounding box (OBB)
  - Often better fit
  - More complex computation

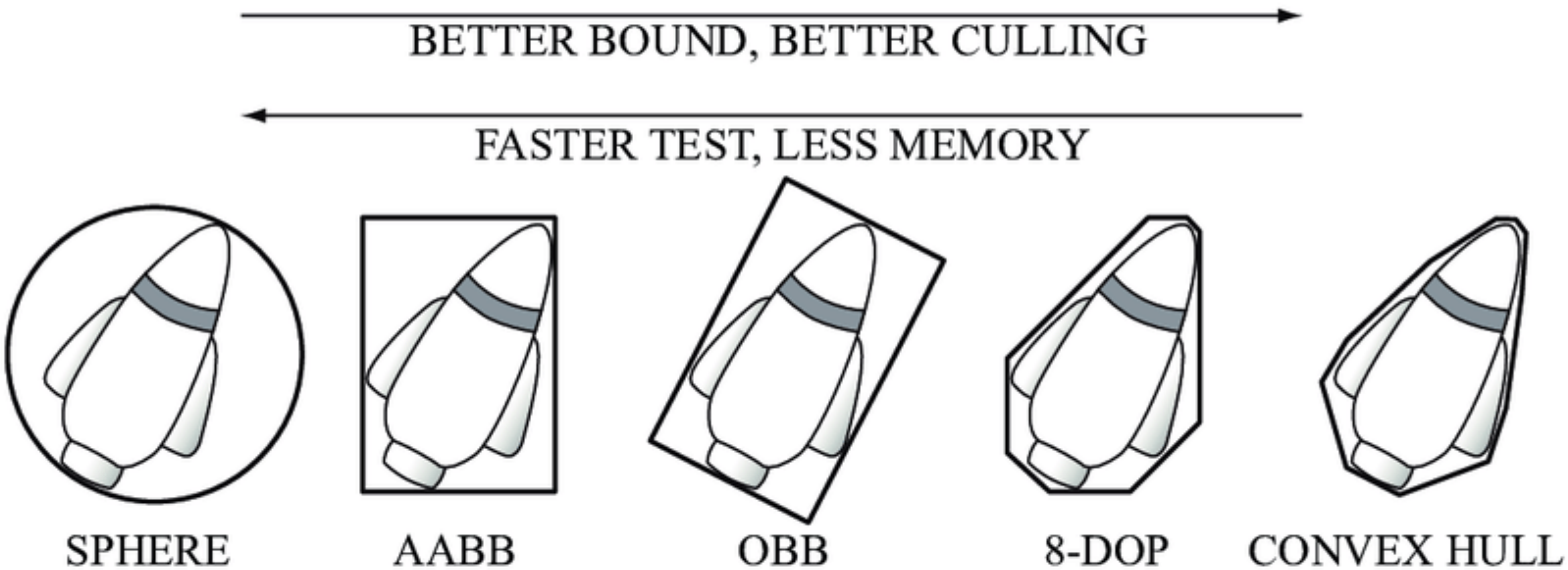


- **(Unbounded) Plane**
  - Infinite bounding box!!!!
  - Best to keep planes outside of acceleration structures
- **Neural bounding? (Liu, et. al 2023)**

Bounding Volumes Types

- **k-DOP** (discrete oriented polytope)
  - Boolean intersection of *k* bounding slabs along k directions
  - Pairs of half spaces





Bounding Volumes Types

- **Neural bounding (Liu, et. al 2023)**

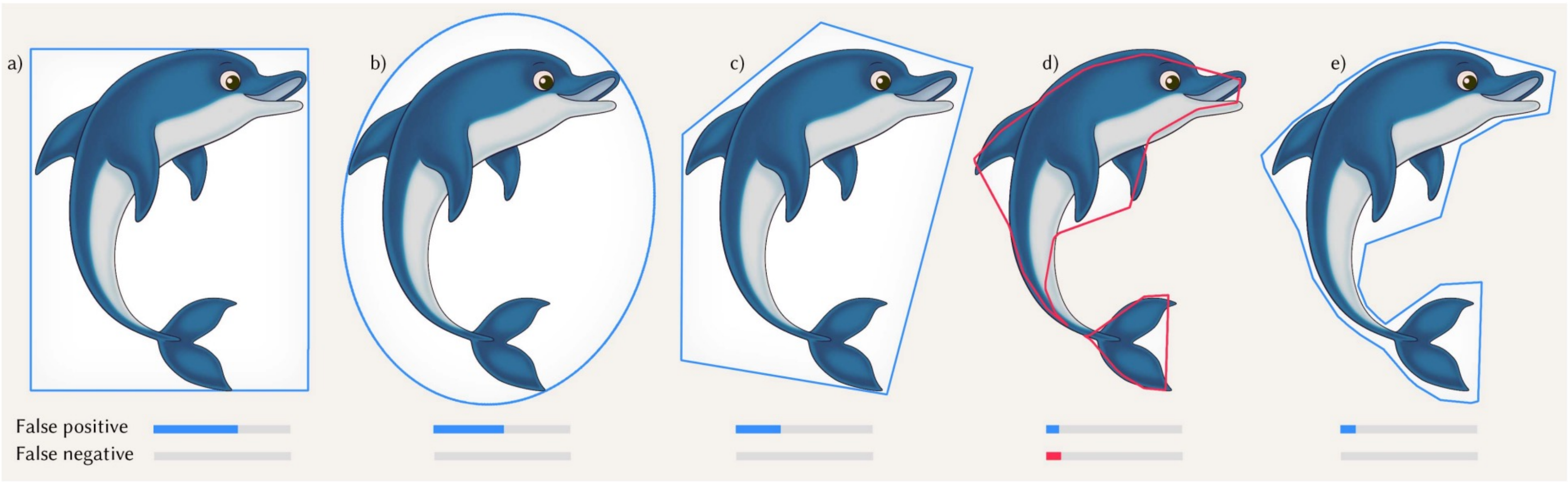
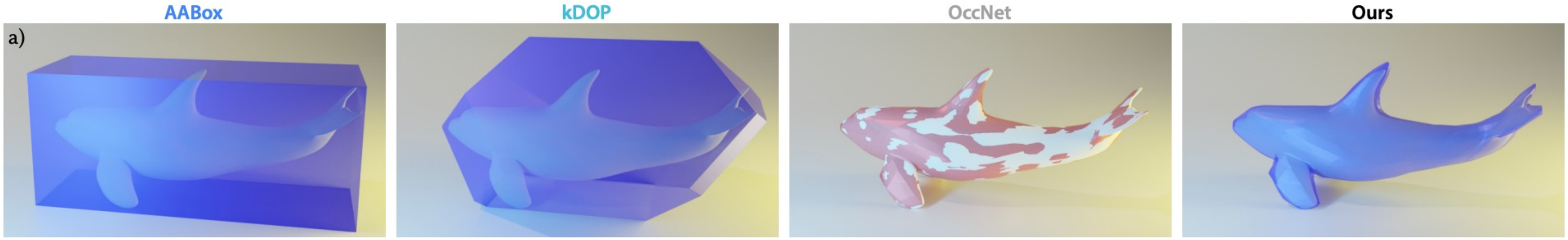
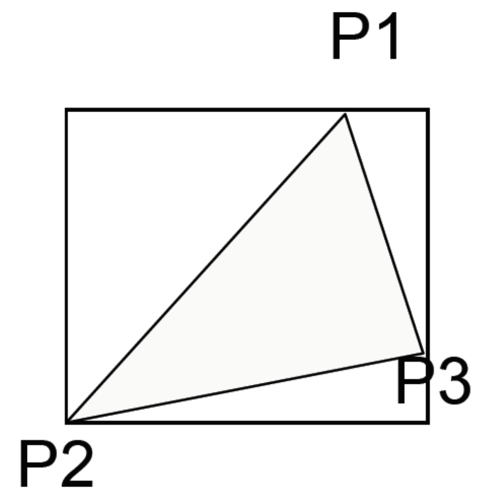


Figure 1: *Different bounding volume types classifying 2D space as maybe-object or certainly-not-object, from left to right: box (**a**), ellipsoid (**b**), k-oriented planes (**c**), common neural networks (**d**) and a neural network trained using our approach (**e**). While common boundings are not tight, common neural networks are not conservative, missing parts of the dolphin, while ours is both tight and has no false negatives.*

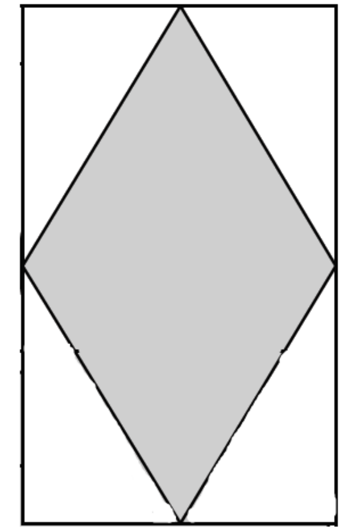


Bounding Volumes Construction

- **Triangle**
  - bmin.x = min(p1.x, p2.x, p3.x)
  - bmax.x = max(p1.x, p2.x, p3.x)
  - Similar for bmin.y, bmax.y, bmin.z, bmax.z

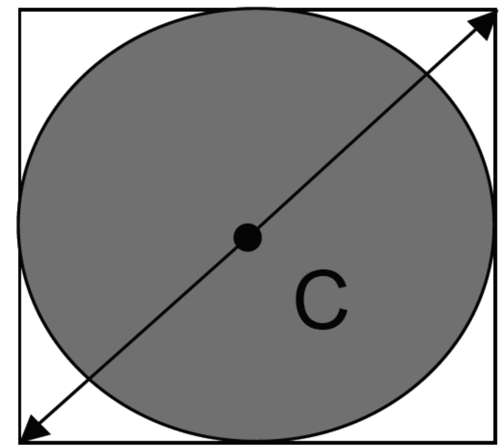


- **Parallelogram**
  - p1 = p0 + e1
  - p2 = p0 + e2
  - p3 = p0 + e1 + e2
  - bmin.x = min(p0.x, p1.x, p2.x, p3.x)
  - bmax.x = max(p0.x, p1.x, p2.x, p3.x)
  - Similar for bmin.y, bmax.y, bmin.z, bmax.z

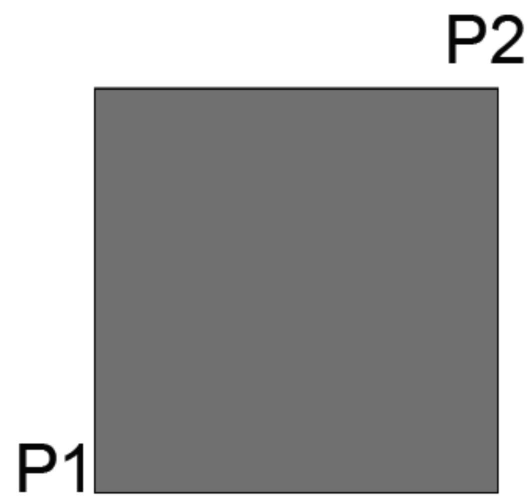


Bounding Volumes Construction

- **Sphere**
  - bmin = center - Vector(radius, radius, radius)
  - bmax = center - Vector(radius, radius, radius)



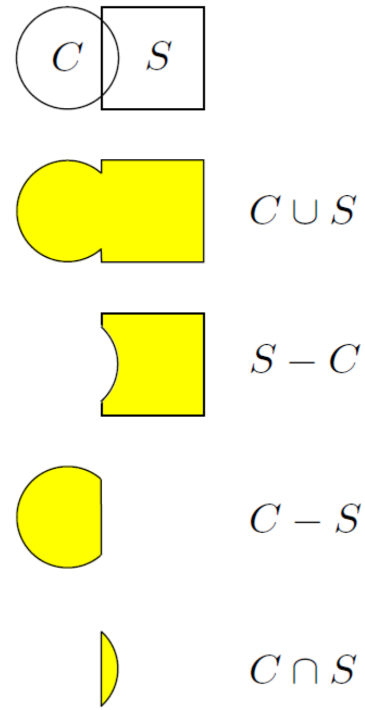
- **Axis-Aligned Box**
  - bmin.x = min(p1.x, p2.x)
  - bmax.x = max(p1.x, p2.x)
  - Similar for bmin.y, bmax.y, bmin.z, bmax.z



- **Oriented Box**
  - bmin.x = min(p1.x, p2.x, … , p8.x)
  - bmax.x = max(p1.x, p2.x, … , p8.x)
  - Similar for bmin.y, bmax.y, bmin.z, bmax.z

Bounding Volumes Construction

- **Constructive Solid Geometry**
  - Union: C U S
    - bmin.x = min(bminC.x, bminS.x)
    - bmax.x = max(bmaxC.x, bmaxS.x)
  - Difference: S - C
    - bmin.x = bminS.x
    - bmax.x = bmaxS.x
  - Intersection: C ∩ S
    - bmin.x = max(bminC.x, bminS.x)
    - bmax.x = min(bmaxC.x, bmaxS.x)



- **Neural Bounding**

$$\mathcal{L}(\theta) = \int c(\mathbf{r})\mathrm{d}\mathbf{r}, \quad c(\mathbf{r}) = \begin{cases} 0 \text{ if } g(\mathbf{r}) = 0 \text{ and } h_\theta(\mathbf{r}) = 0, & \text{TN} \\ \alpha \text{ if } g(\mathbf{r}) = 1 \text{ and } h_\theta(\mathbf{r}) = 0, & \text{FN} \\ \beta \text{ if } g(\mathbf{r}) = 0 \text{ and } h_\theta(\mathbf{r}) = 1, & \text{FP} \\ 0 \text{ if } g(\mathbf{r}) = 1 \text{ and } h_\theta(\mathbf{r}) = 1, & \text{TP} \end{cases}$$

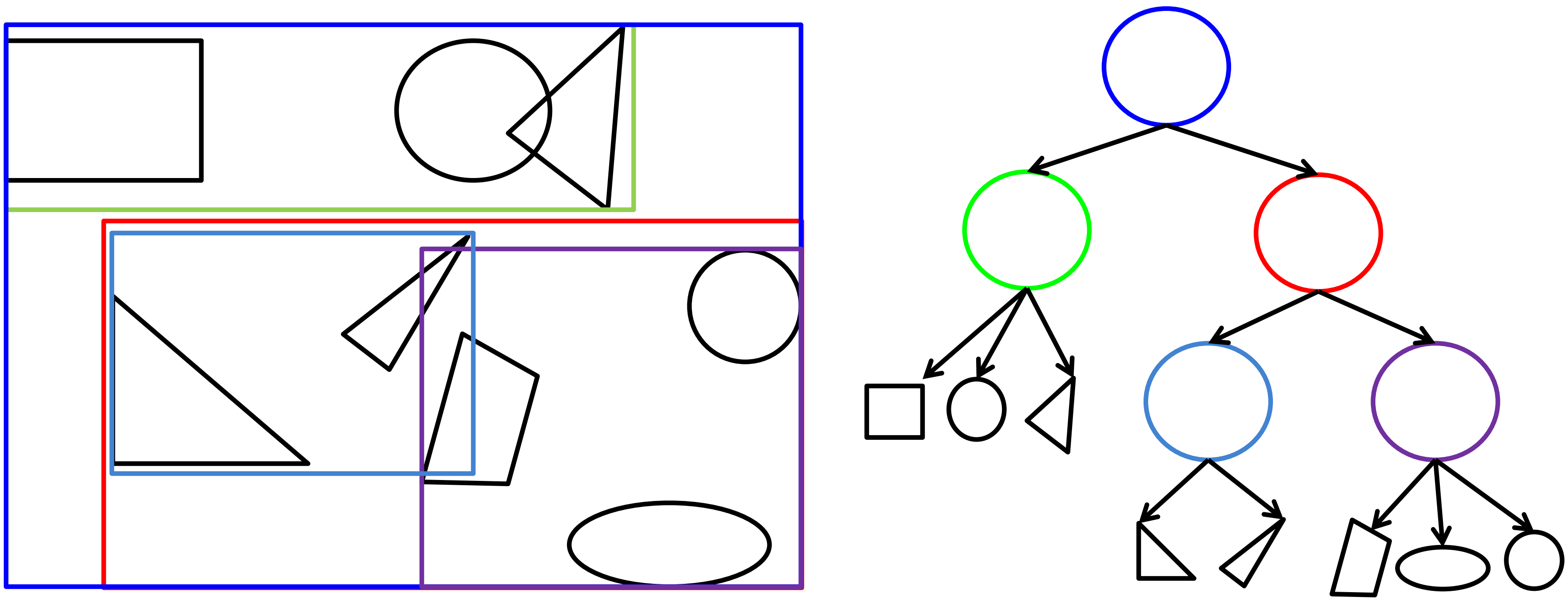
Bounding Volumes Discussion

- **Benefits**
  - Can reduce the overall cost by a constant factor
- **Limitations**
  - Does not change the asymptotic cost
- **Solutions**
  - Build a hierarchy of bounding volumes

Philipp Slusallek, Philippe Weier, Alexander Rath

Bounding Volume Hierarchy

- **Hierarchical partitioning of the set of objects**
- **Form a tree structure**
  - Each inner node stores pointers to child nodes
  - Each leaf node stores pointers to objects
  - All nodes store a volume enclosing all sub-trees

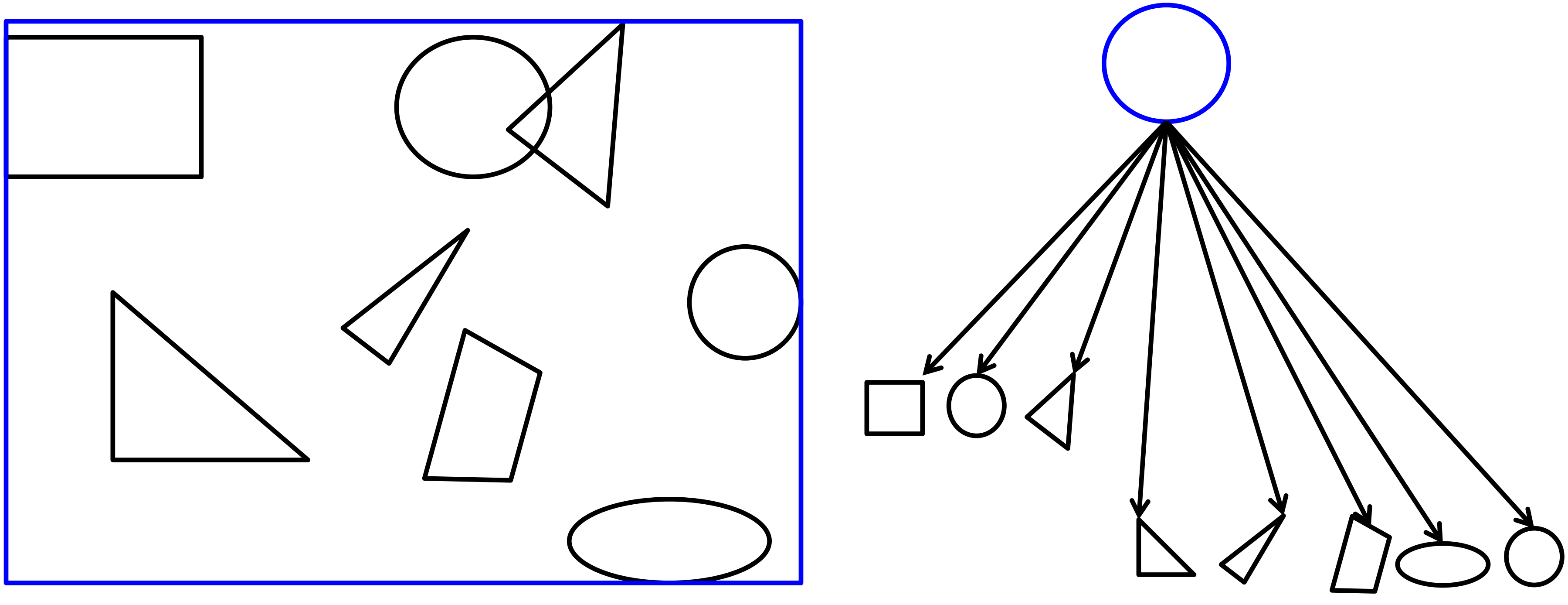


BVH Construction

- **Insertion-Based**
  - Insert each object at the root
  - Trickle down to sub-tree with minimal cost
  - Sensitive to order of insertion
- **Bottom-Up**
  - Group close-by objects together
  - Recursively group close-by groups together
  - Emphasis at the bottom of the tree
- **Top-Down**
  - Subdivide objects into groups
  - Recursively subdivide groups into sub-groups
  - Emphasis at the top of the tree

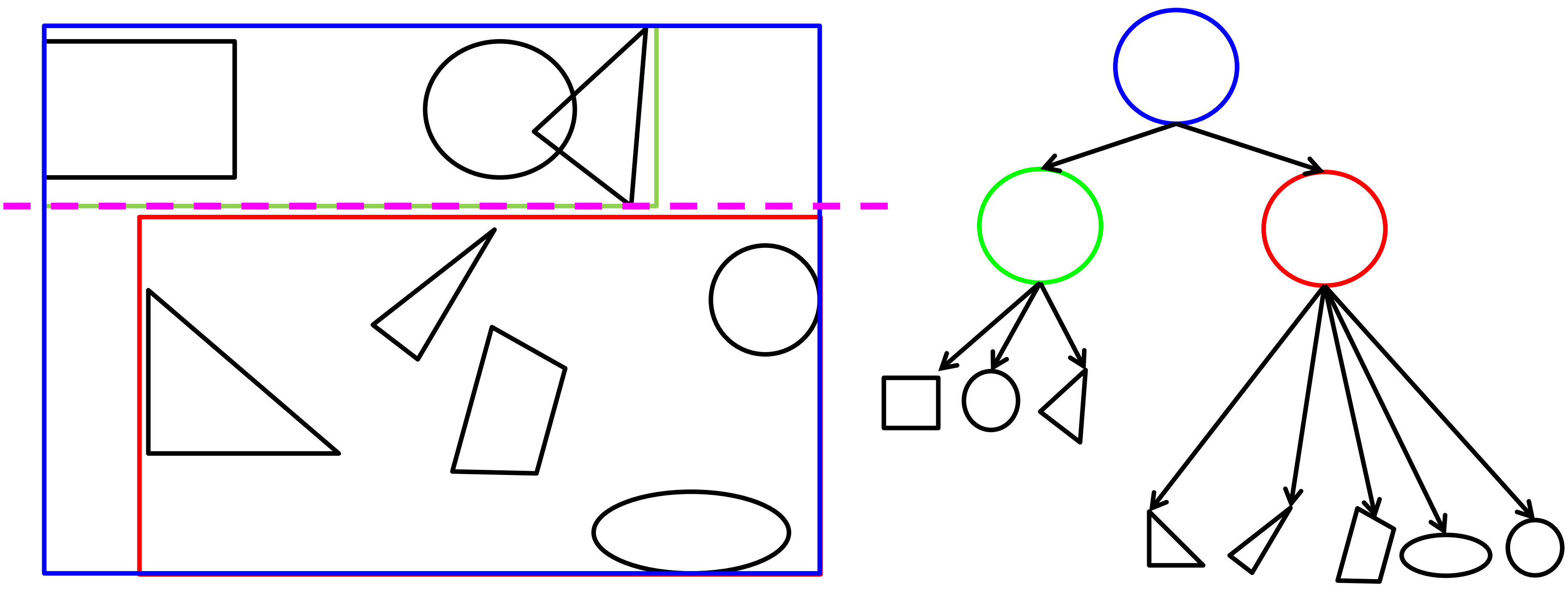
BVH Construction

- **Compute overall bounding box**



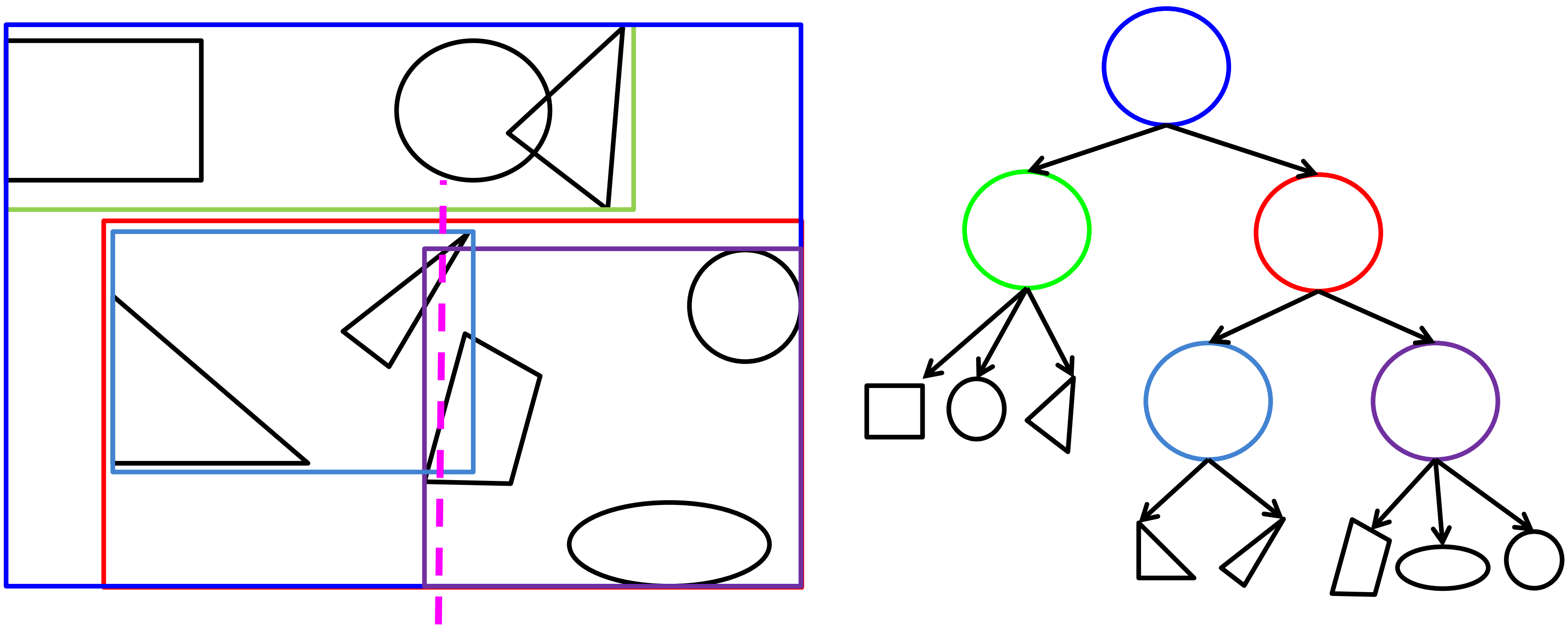
BVH Construction

- **Choose split axis and coordinate**
- **Place each object into a single group**
- **Use position of object's centroid relative to the plane**
  - Estimate centroid as center of object's bounding box in practice



BVH Construction

- **Compute bounding box of the 2 child nodes**
- **Recurse down each individual subtree**
- **Until termination criterion is met**

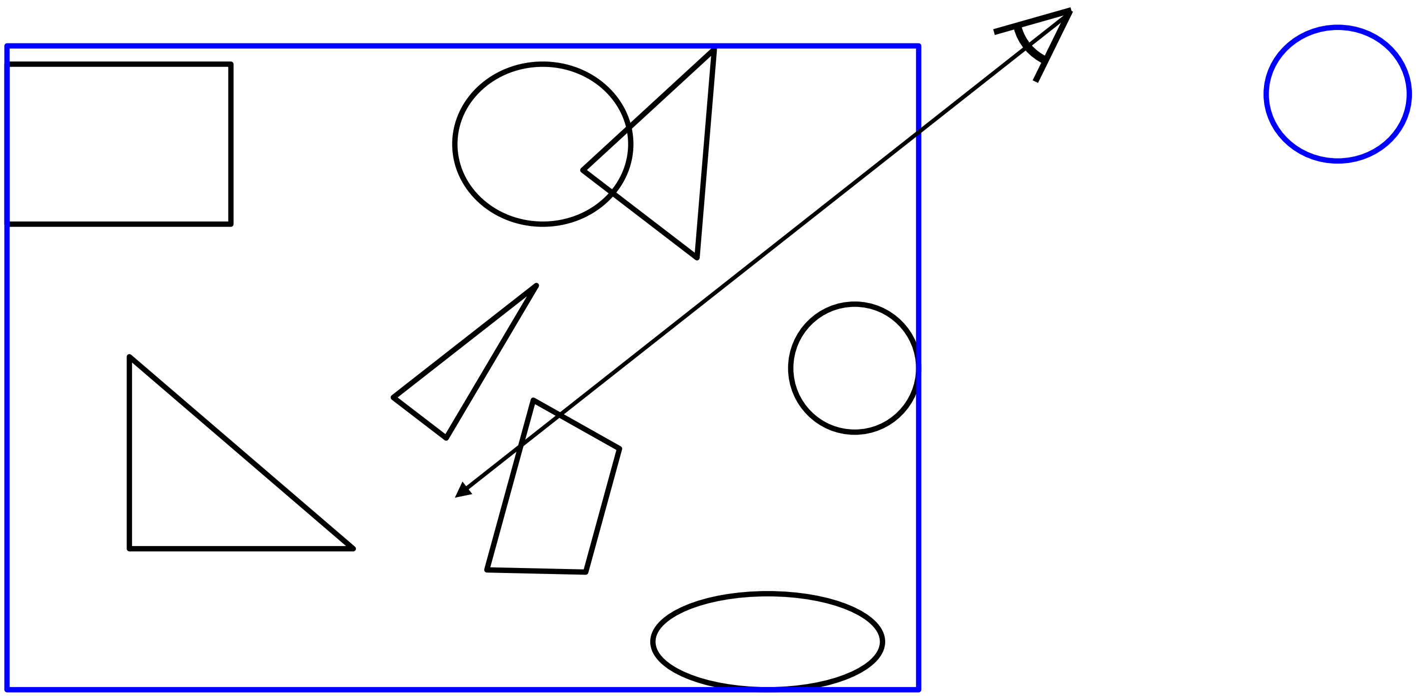


BVH Construction

- **BVHBuild(objects):**
- **MakeSplitDecision();**
- **If split**
  - for(object o in objects)
    - If centroid(o).axis < split
      - Put o into leftGroup;
    - Else
      - Put o into rightGroup;
  - Return new inner node with children
    - BVHBuild(leftGroup);
    - BVHBuild(rightGroup);
- **Else**
  - Return new leaf node with objects;

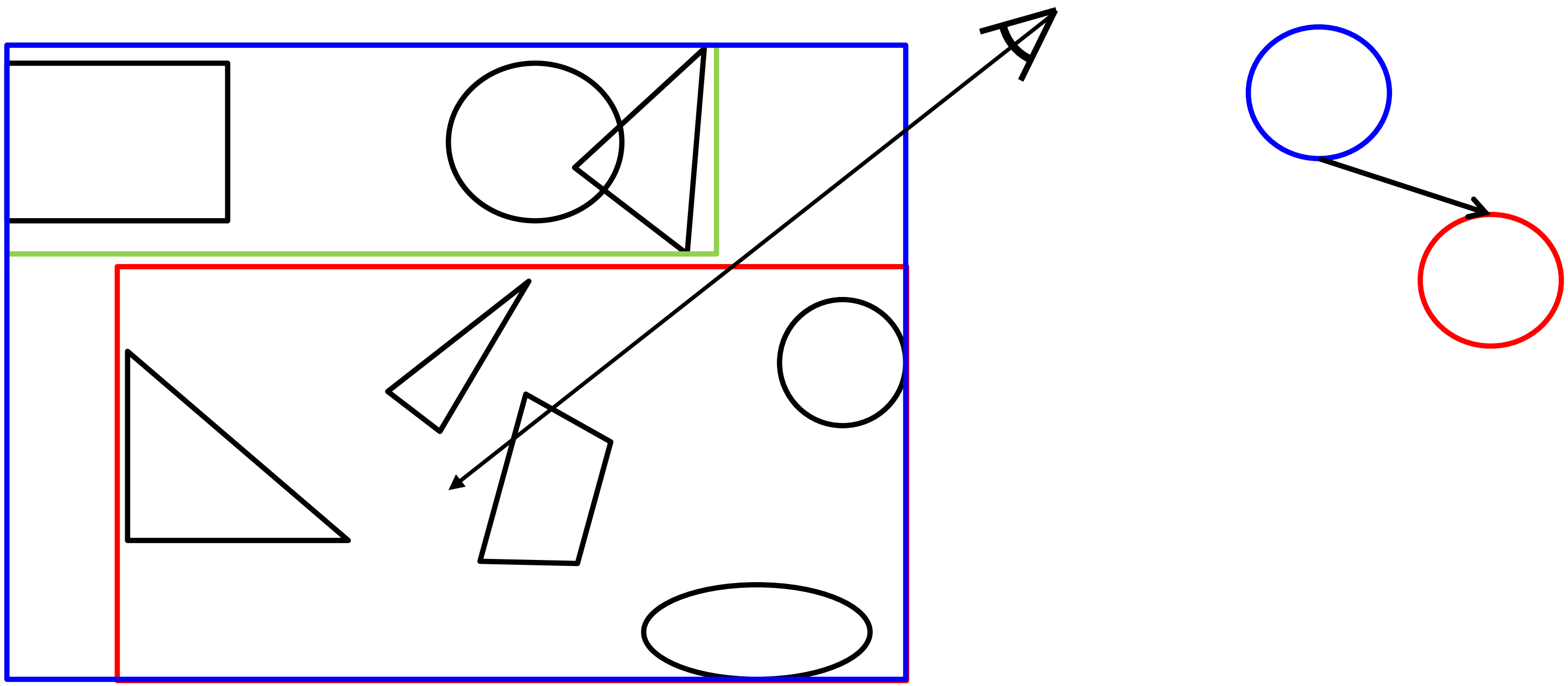
BVH Traversal

- **Check if root node is intersected by the ray**



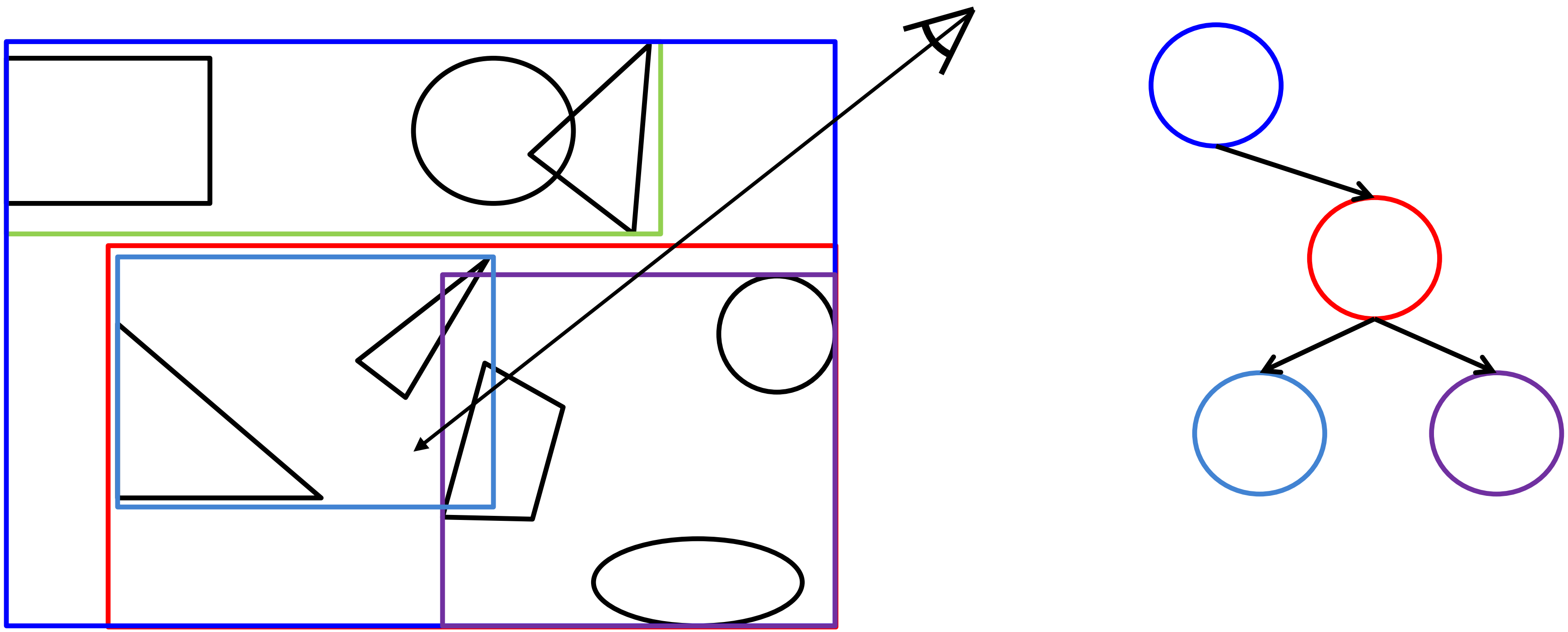
BVH Traversal

- **Recurse only into subtrees intersected by the ray**



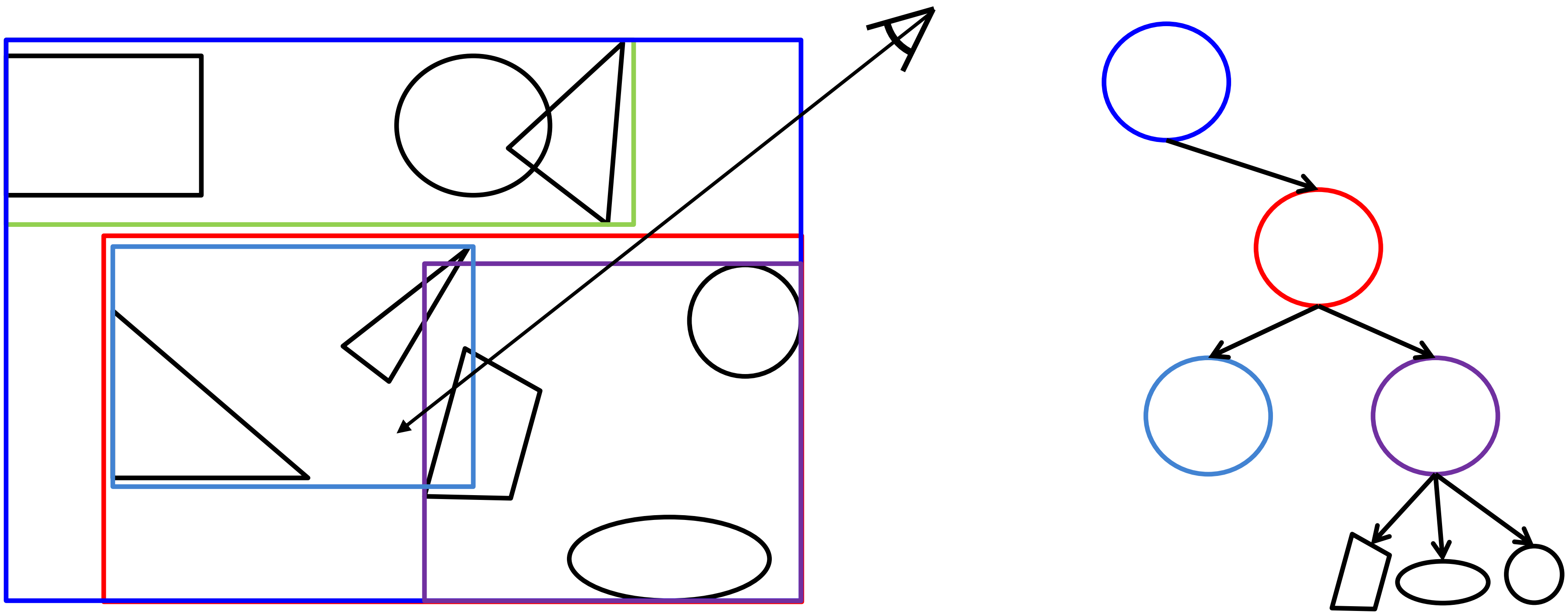
BVH Traversal

- **Cheap traversal instead of costly intersection**
- **Process both subtrees**
  - In random order
  - In order of intersection of their bounding boxes



BVH Traversal

- **For each primitive in current leaf node**
  - Check if ray intersects the primitive
  - If closest (positive) hit so far, record hit
- **Ordered Traversal**
  - Skip 2nd child if non-overlapping intersection found in 1st child



BVH Traversal

- **BVHIntersect(ray, node):**
- **If (node is inner node)**
  - If ray intersects bounding box of near child
    - BVHIntersect(ray, node.nearChild);
  - If ray intersects bounding box of far child (before previous hit)
    - BVHIntersect(ray, node.farChild);
- **Else**
  - Iterate through node.listOfObjects and record closest intersection;

BVH Storage

- **Node Representation**
  - Leaf Flag
    - 1 Boolean
  - Bounding Volume
    - Bounding box: 6 reals
  - Pointers to children / geometry
    - 2 pointers
    - 1 pointer & 1 integer
- **Smarter Node Representation (32 Bytes)**
  - Bounding box
    - *(6 floats, 24 bytes)*
  - Left child index (if interior) OR first primitive index (if leaf)
    - *(1 unsigned integer, 4 bytes)*
  - Number of Triangles (if leaf)
    - *(1 unsigned integer, 4 bytes)*

BVH Discussion

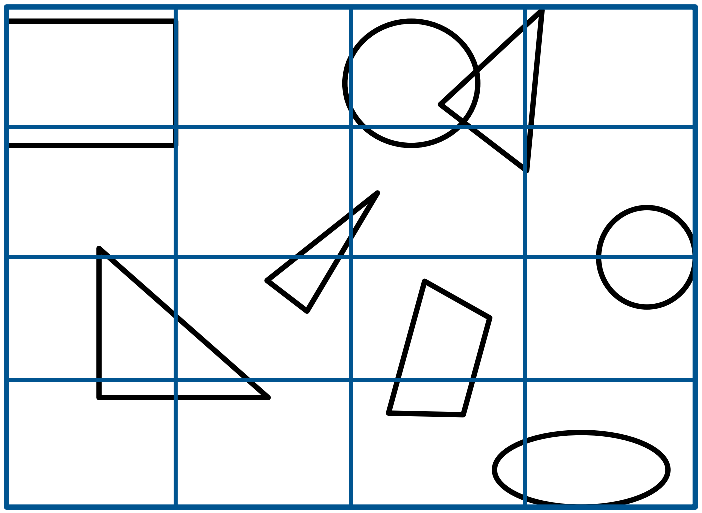
- **Properties**
  - Logarithmic intersection cost
  - Adaptive to local geometric density

- **Trade-Offs**
  - Relatively moderate build cost
  - Relatively moderate traversal cost

SPACE SUBDIVISION

Uniform Grid

- **Regular partitioning of space into equal-size cells**
- **Each cell holds a reference to all overlapping objects**



Uniform Grid Construction

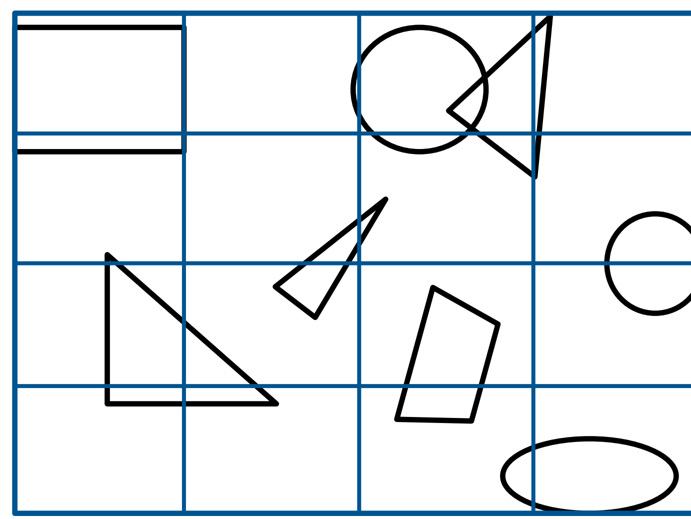
- **Resolution Trade-Offs**
  - Small cells
    - Few intersections tests
    - Many traversal steps
  - Large cells
    - Few traversal steps
    - Many intersections tests

$$res_x = d_x \sqrt[3]{\frac{\lambda n}{d_x d_y d_z}}$$

$$res_y = d_y \sqrt[3]{\frac{\lambda n}{d_x d_y d_z}}$$

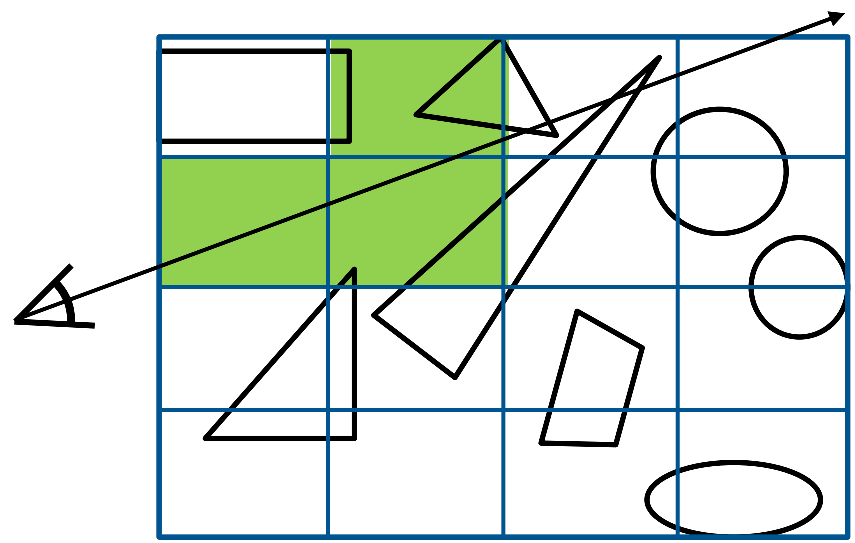
$$res_z = d_z \sqrt[3]{\frac{\lambda n}{d_x d_y d_z}}$$

- **Optimal Resolution**
  - Nb of cells prop. to nb of objects *n*
    - Resolution proportional to $\sqrt[3]{n}$
  - Roughly cubic cells
    - Resolution prop. to scene's extent
  - *d*: diagonal vector pmax - pmin
  - λ: density (user-defined)



Uniform Grid Traversal

- **3D-DDA (Digital Differential Analyzer)**
- **Variant of Bresenham algorithm (see later)**
  - Compute coordinates bx/by/bz to closest cell boundaries
  - Initialize parametric distances px/py/pz to closest cell boundaries
  - Compute parametric distances tx/ty/tz between cells



Mailboxing

- **Avoid redundant intersections**
  - Single primitive can be inserted in many cells

- **Keep track of intersection tests**
  - Per-object cache of ray IDs
    - Concurrent access of multiple rays in multi-threaded environment!!
  - Per-ray cache of object IDs
    - Can only track the N most recent intersections

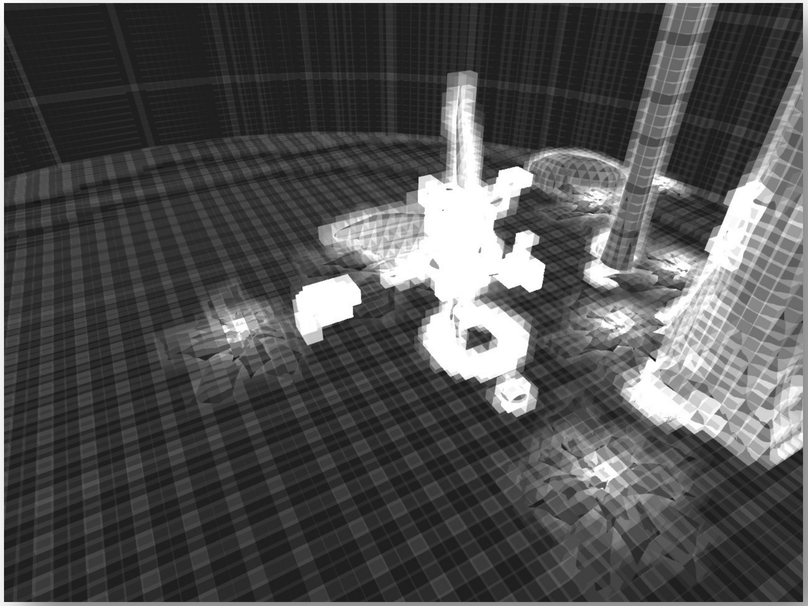
Uniform Grid Discussion

- **Properties**
  - Cube-root intersection cost
  - Non-adaptive to local geometric density: “Teapot in the stadium”
- **Trade-Offs**
  - Relatively cheap build cost
  - Relatively expensive traversal cost

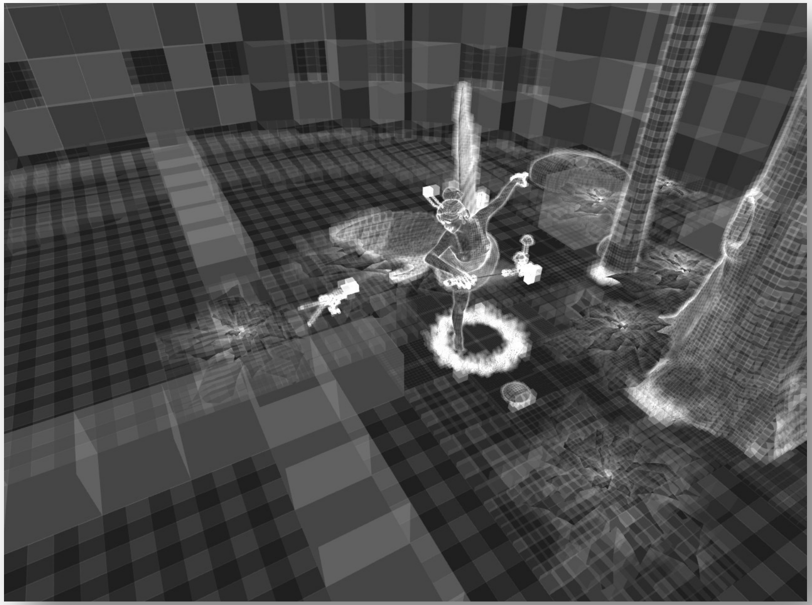
© www.scratchapixel.com

Hierarchical Grids

- **Hierarchy of uniform grids**
  - Each grid cell might be subdivided into a finer grid
- **Properties**
  - Adaptive subdivision: adjust depth to local scene complexity
  - Fixed split positions



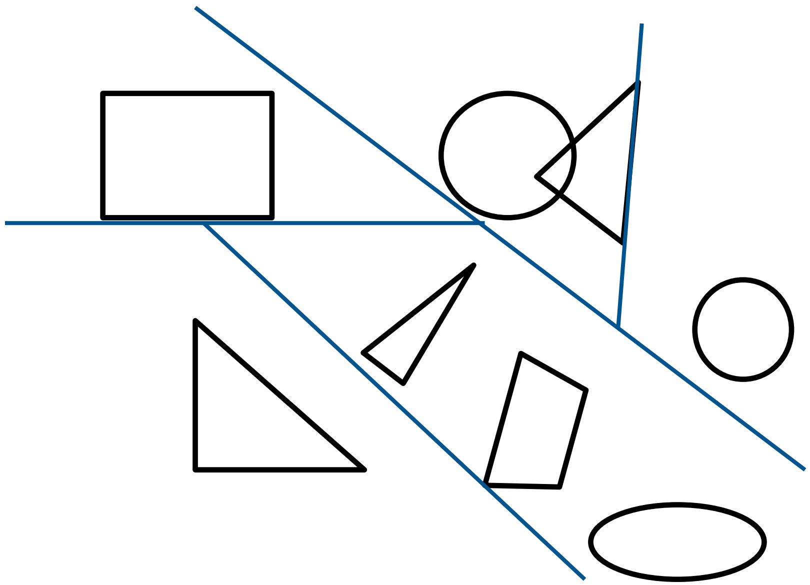
Cells of uniform grid
(colored by # of intersection tests)



Same for two-level grid

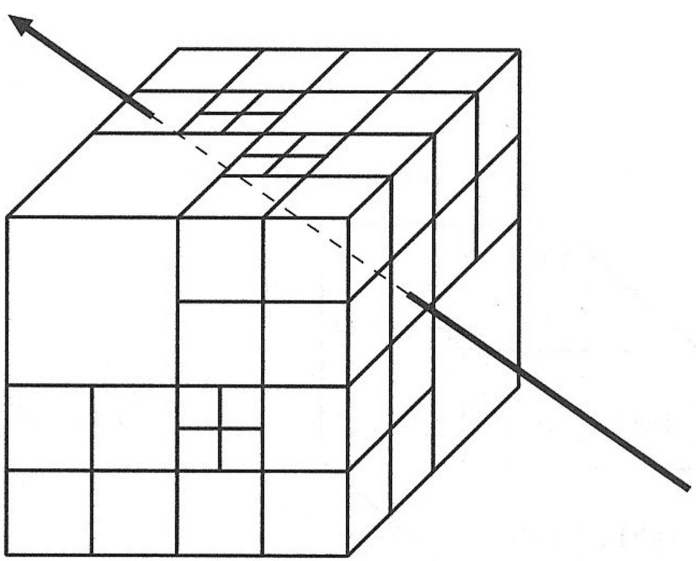
BSP Trees

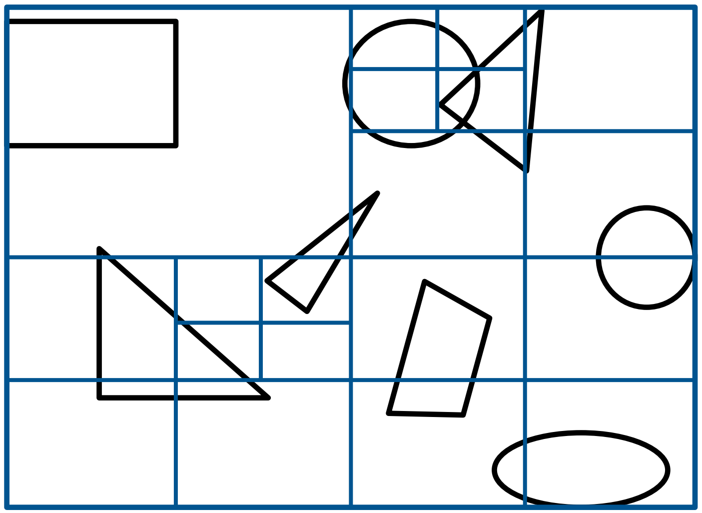
- **Binary Space Partition Tree (BSP)**
  - Recursively split space with planes
    - Arbitrary split positions
    - Arbitrary orientations
  - How much flexibility?
    - Restricted BSP: predefined set of directions
    - Unrestricted BSP: unlimited flexibility



Octree

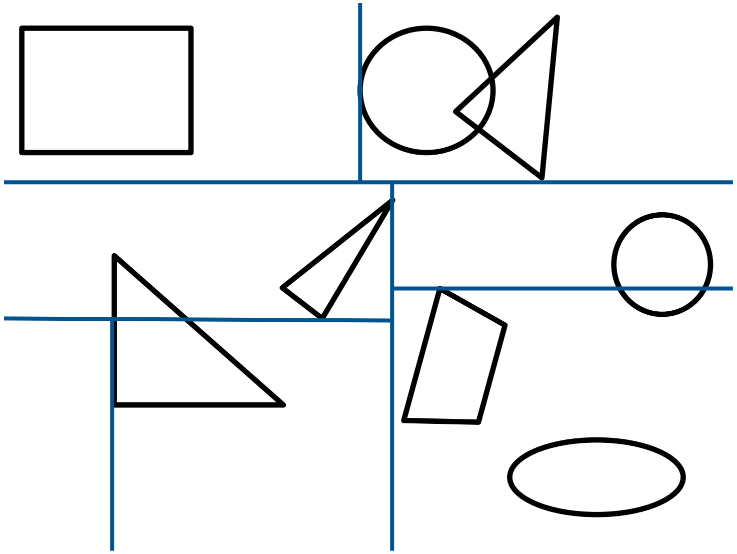
- **Hierarchical Structure**
  - 3D extension of 2D quadtree
  - Each inner node contains 8 equally sized voxels
- **Properties**
  - Adaptive subdivision: adjust depth to local scene complexity
  - Fixed branching factor and split positions





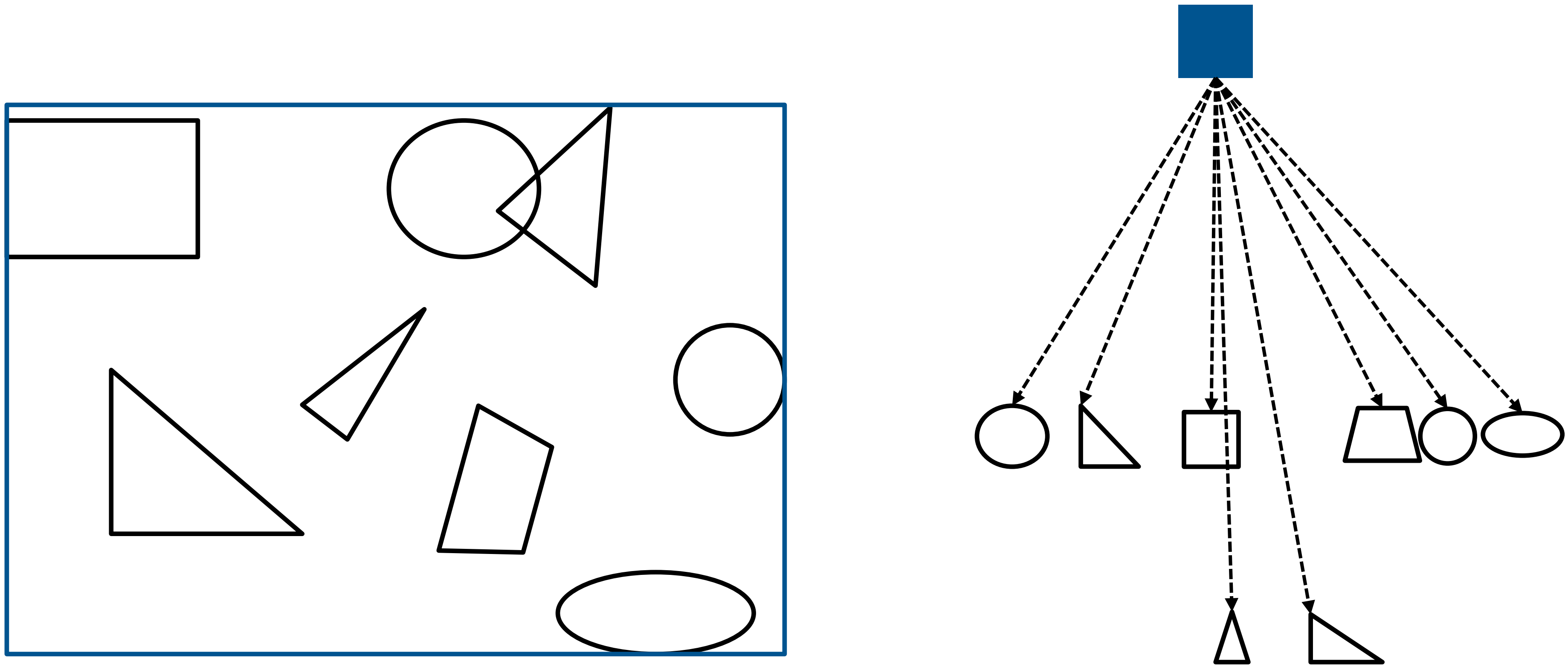
kD-Tree

- **Definition**
  - **Axis-Aligned** Binary Space Partition Tree
  - Recursively split space with axis-aligned planes
    - Arbitrary split positions
    - X, Y or Z orientations
- **Adaptive**
  - Can handle the “Teapot in a Stadium” well



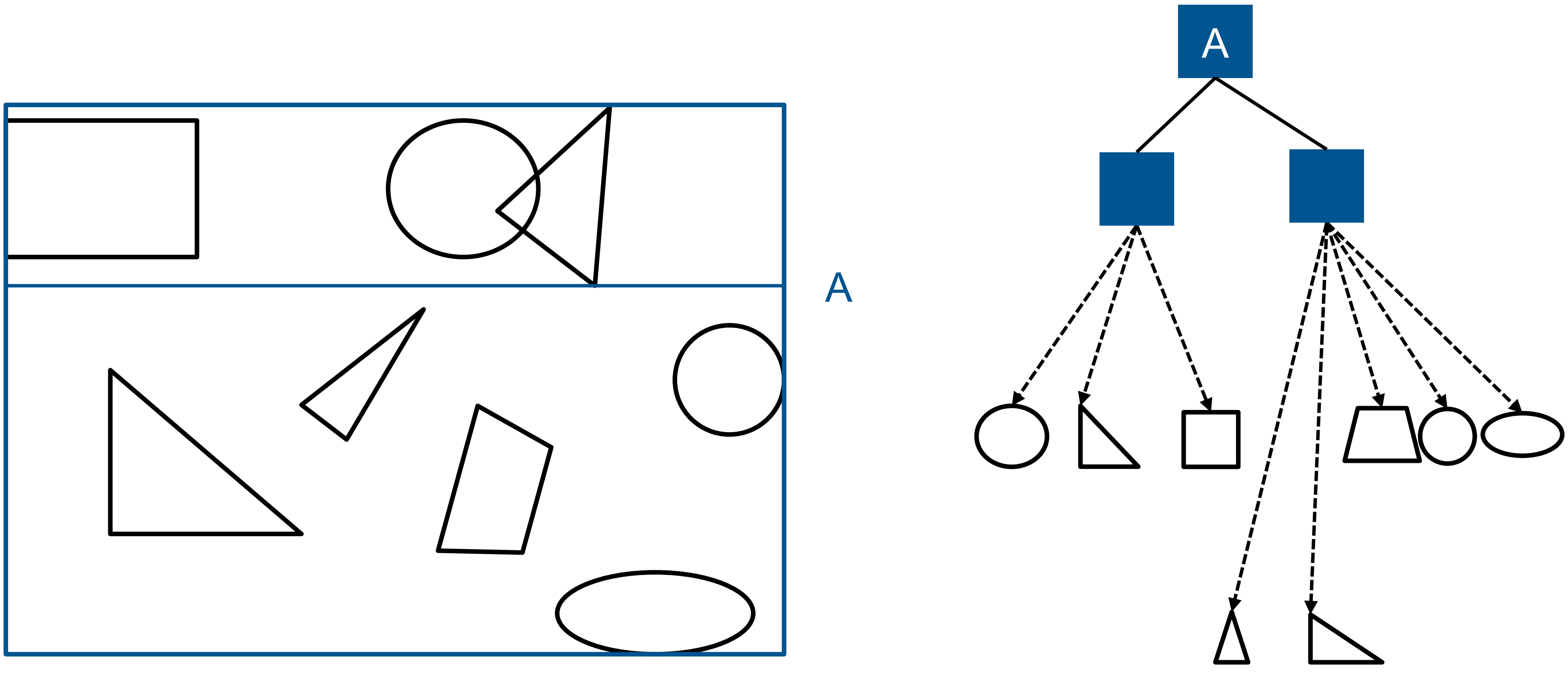
kD-Tree Construction

- **Compute overall bounding box**



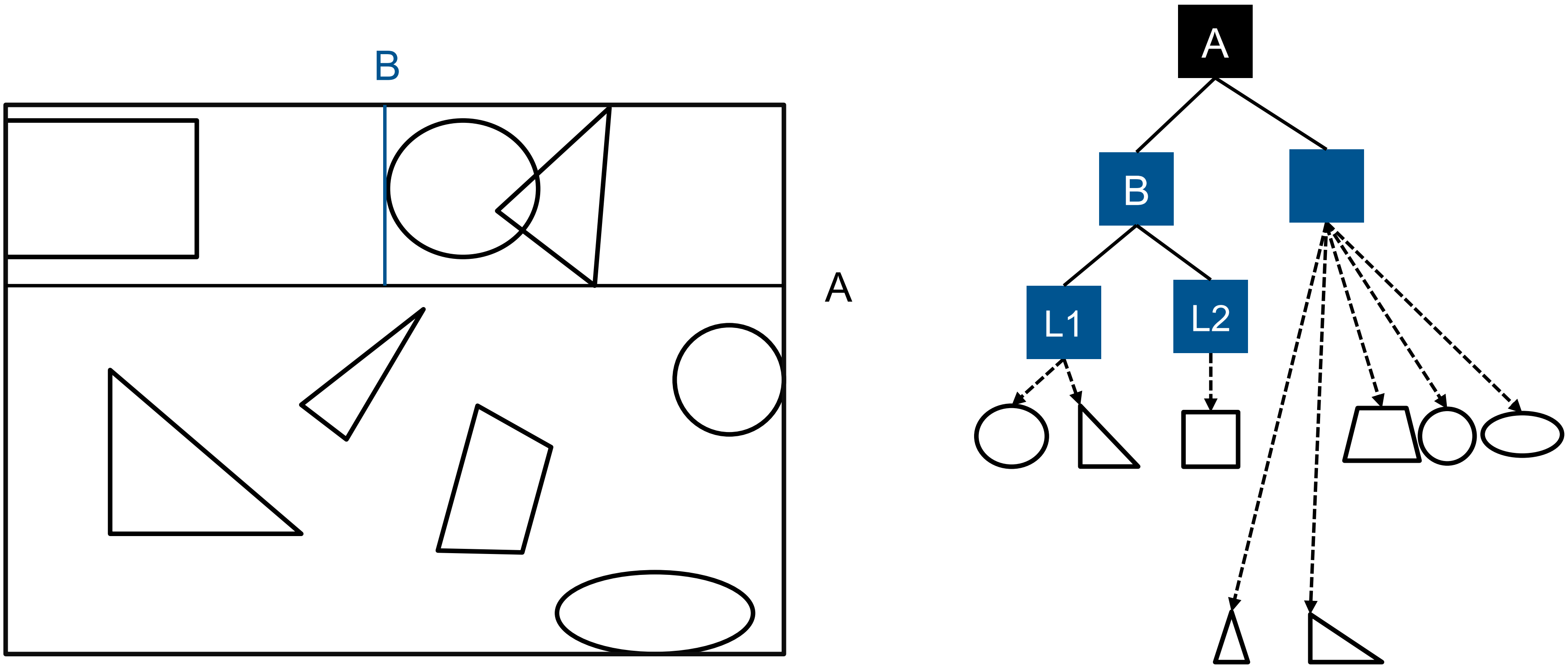
kD-Tree Construction

- **Choose split axis and coordinate**
- **Place each object into non-exclusive groups**

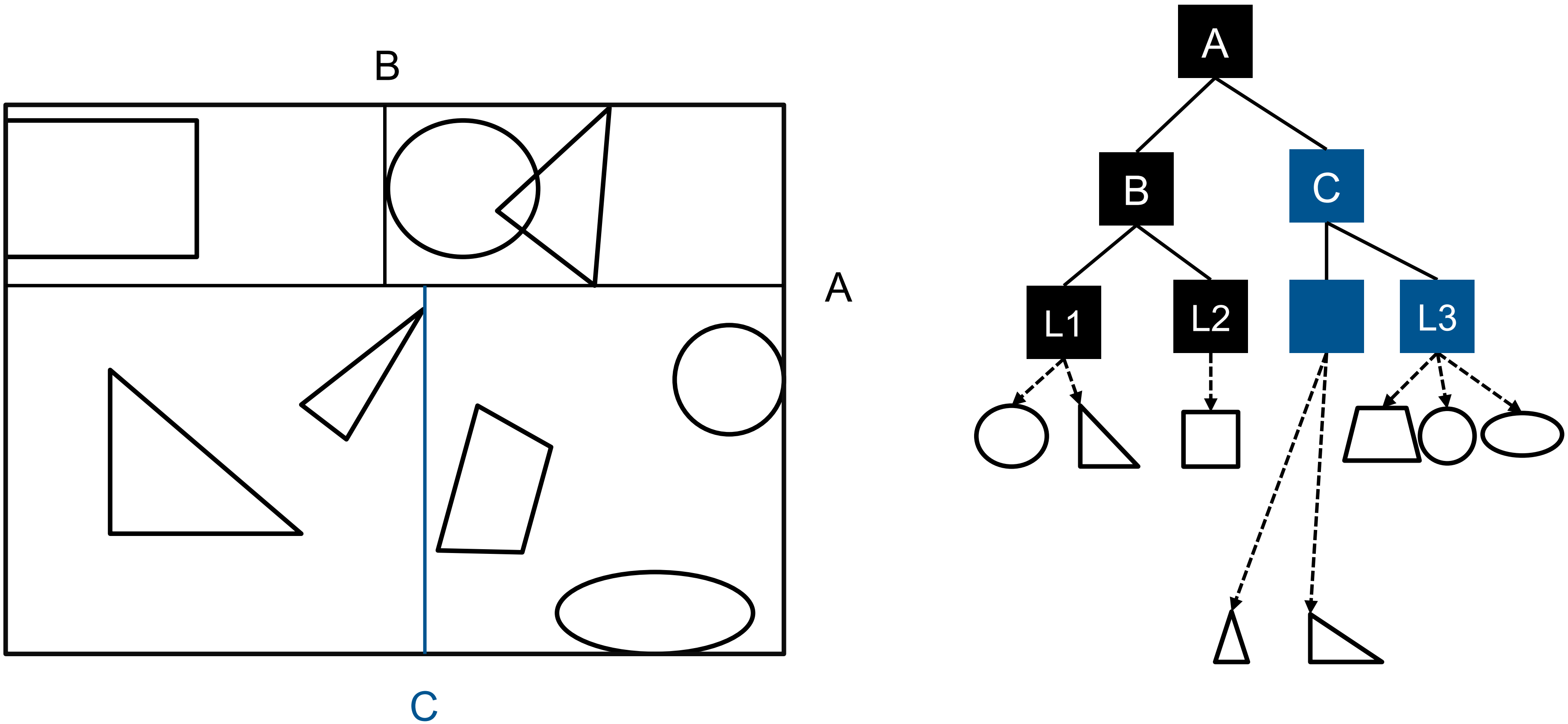


kD-Tree Construction

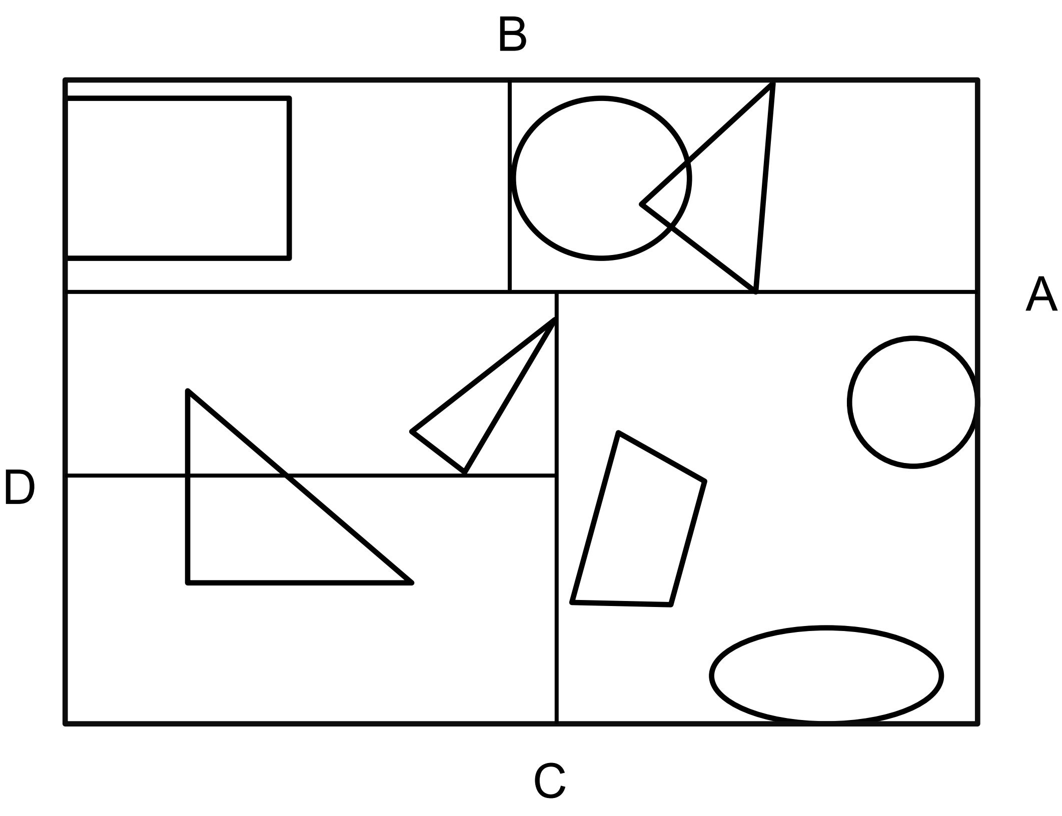
- **Recurse down each individual subtree**
- **Until termination criterion is met**

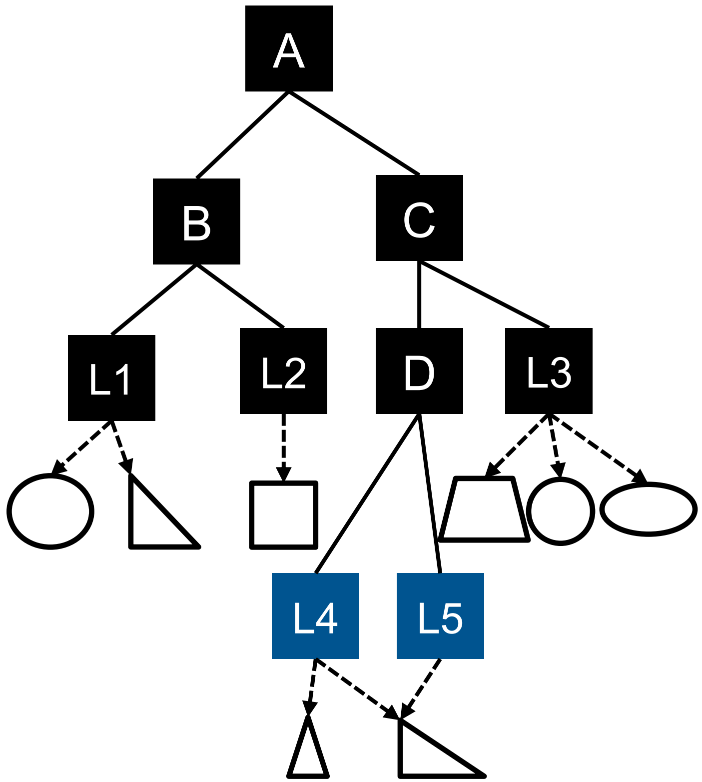


kD-Tree Construction



kD-Tree Construction



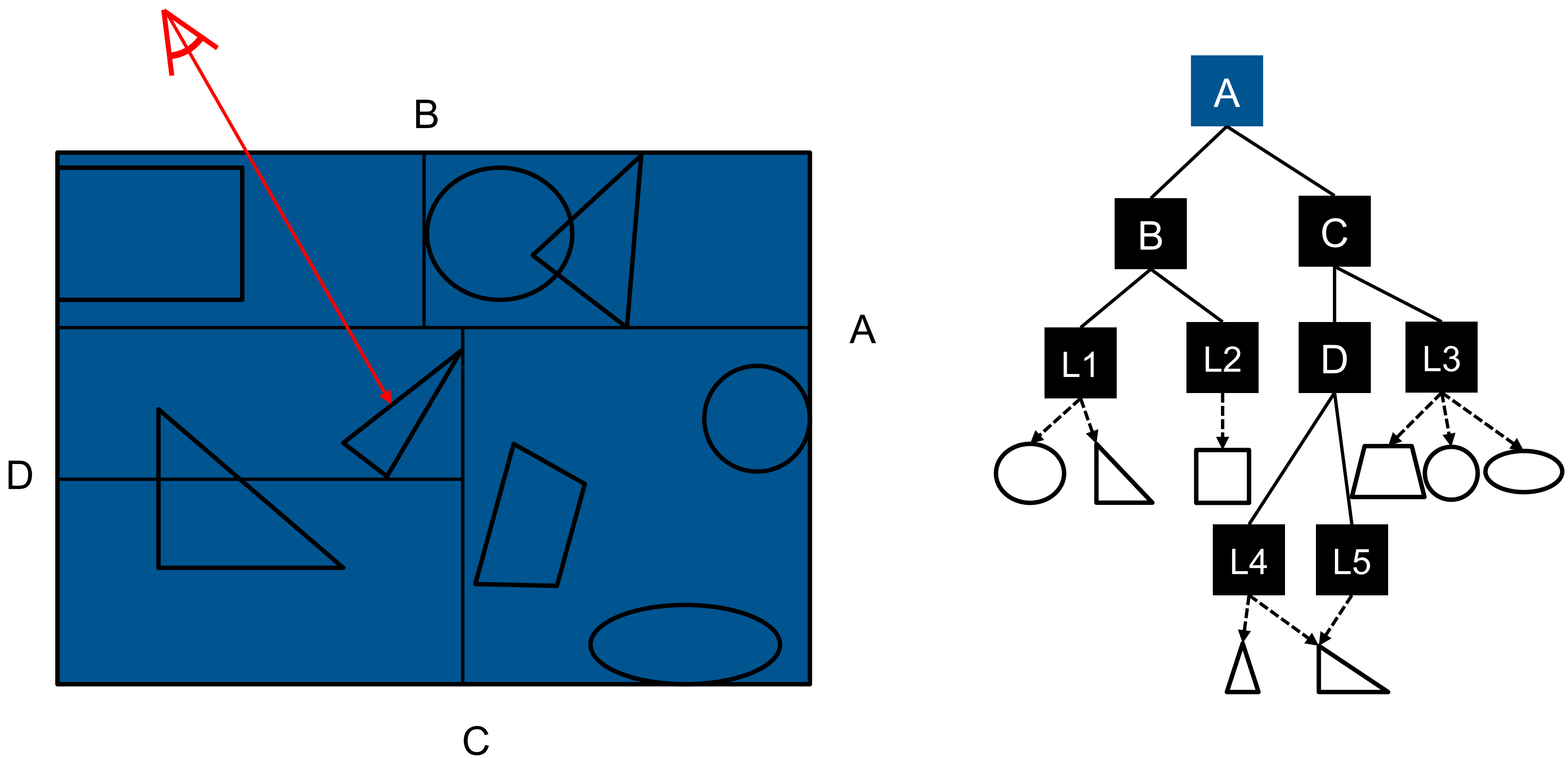


kD-Tree Construction

- **KDTreeBuild(objects):**
- **MakeSplitDecision();**
- **If split**
  - for(object o in objects)
    - If minBound(o).axis < split
      - Add o to leftGroup;
    - If maxBound(o).axis > split
      - Add o to rightGroup;
  - Return new inner node with children
    - KDTreeBuild(leftGroup);
    - KDTreeBuild(rightGroup);
- **Else**
  - Return new leaf node with objects;

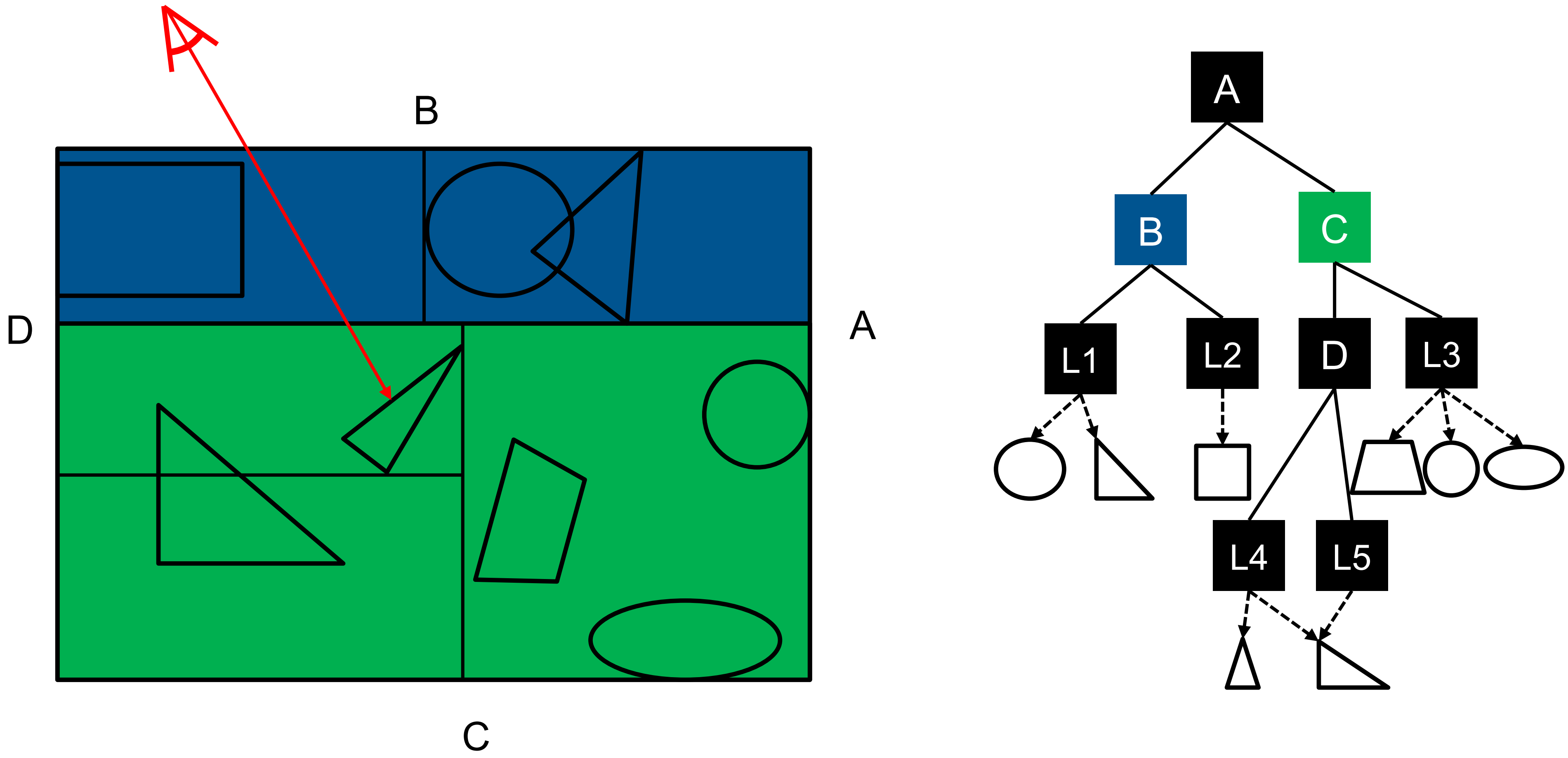
kD-Tree Traversal

- **Check if root node is intersected by the ray**



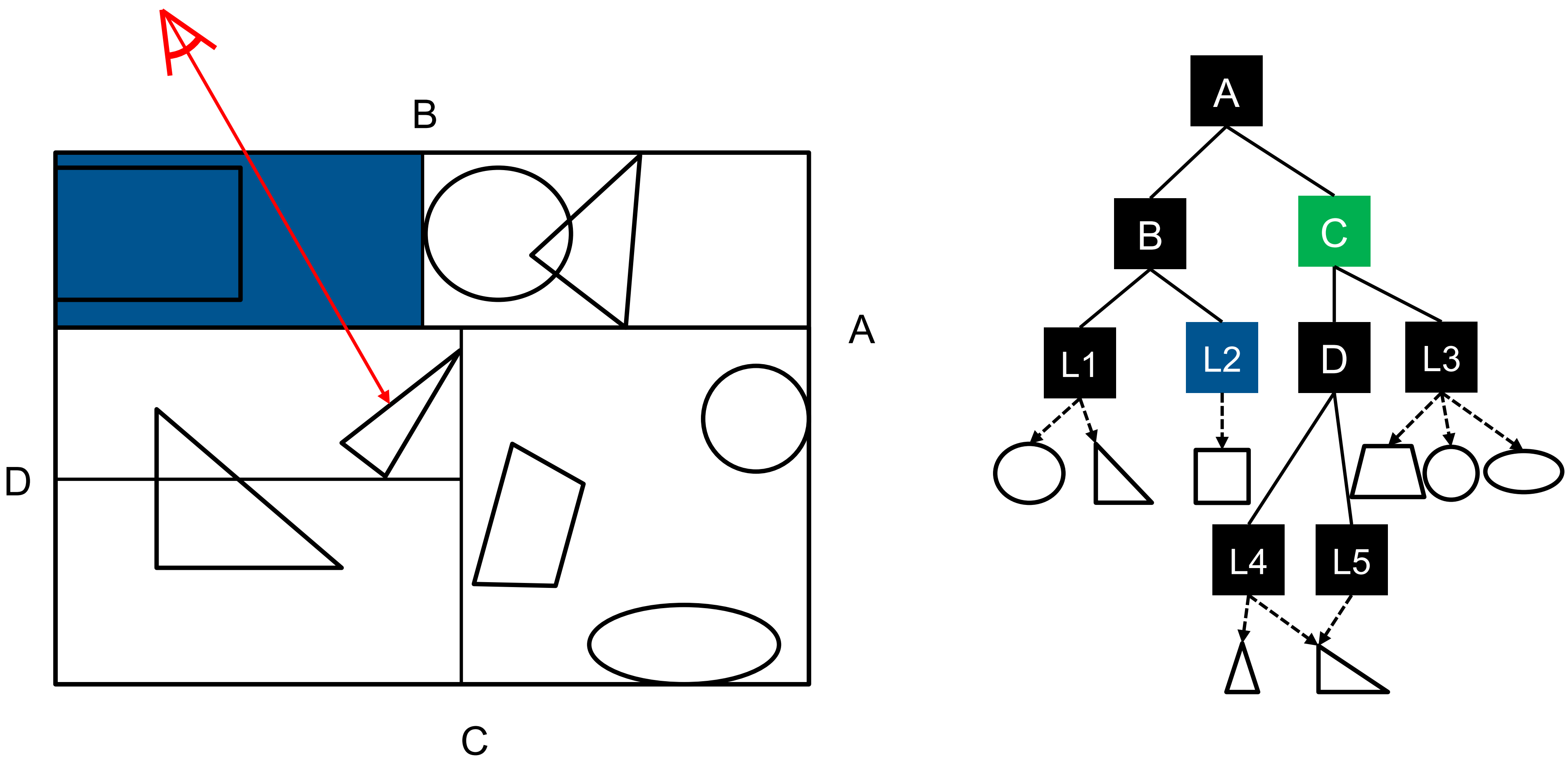
kD-Tree Traversal

- **Process both subtrees**
  - In random order
  - In order of traversal



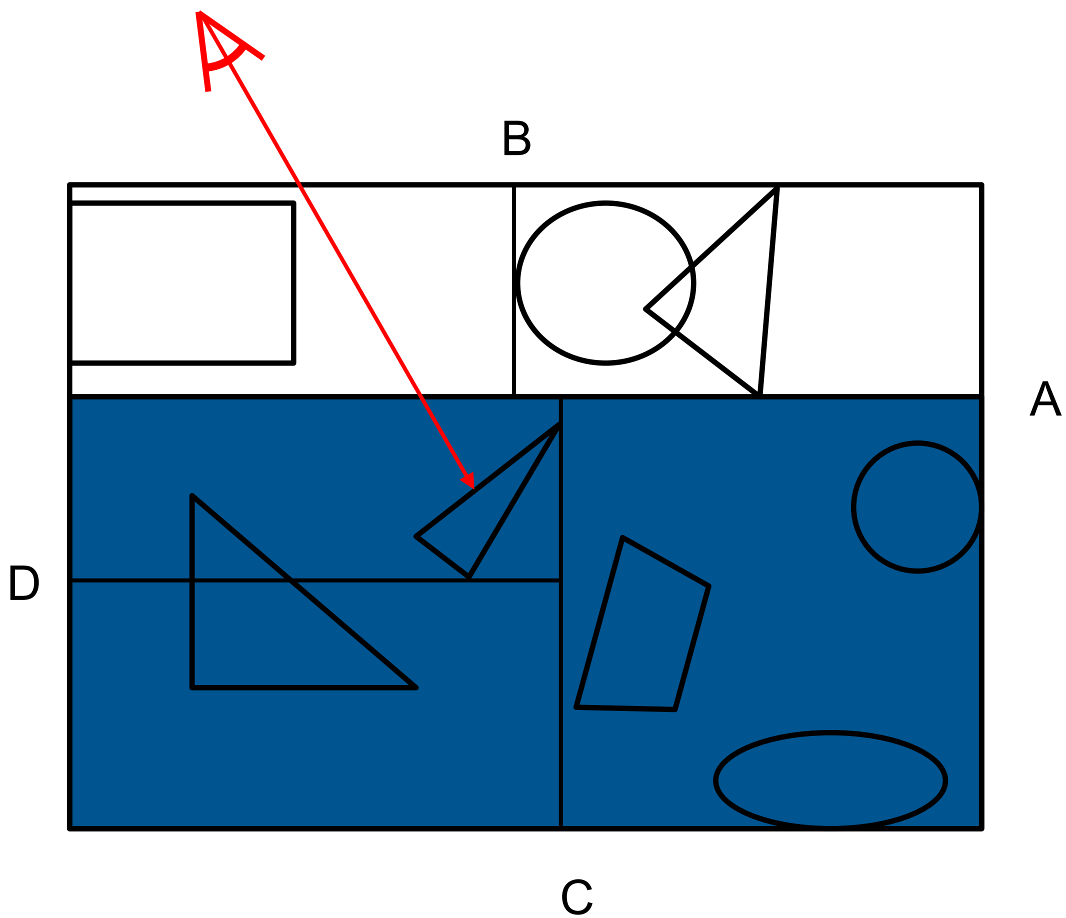
kD-Tree Traversal

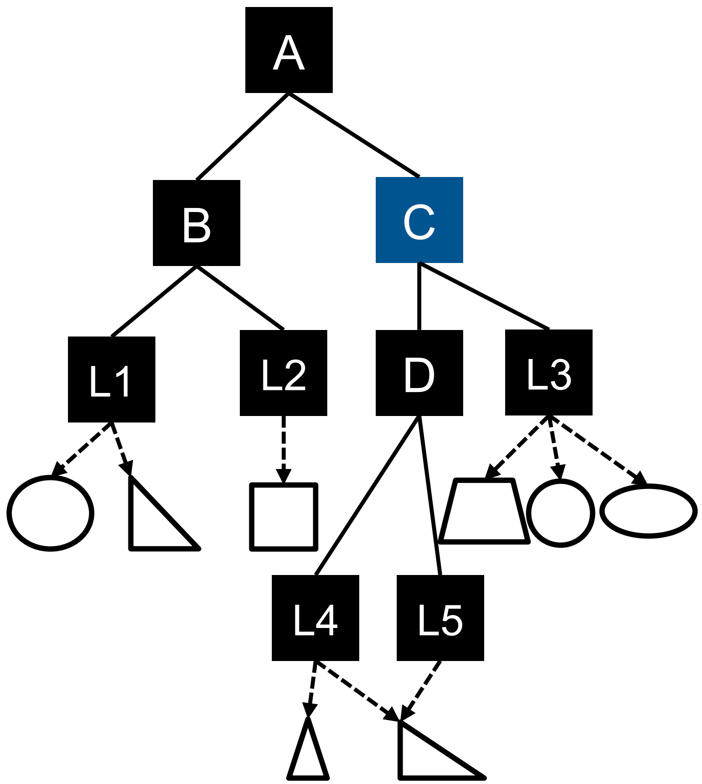
- **Cheap traversal instead of costly intersection**
- **Recurse only into subtrees intersected by the ray**



kD-Tree Traversal

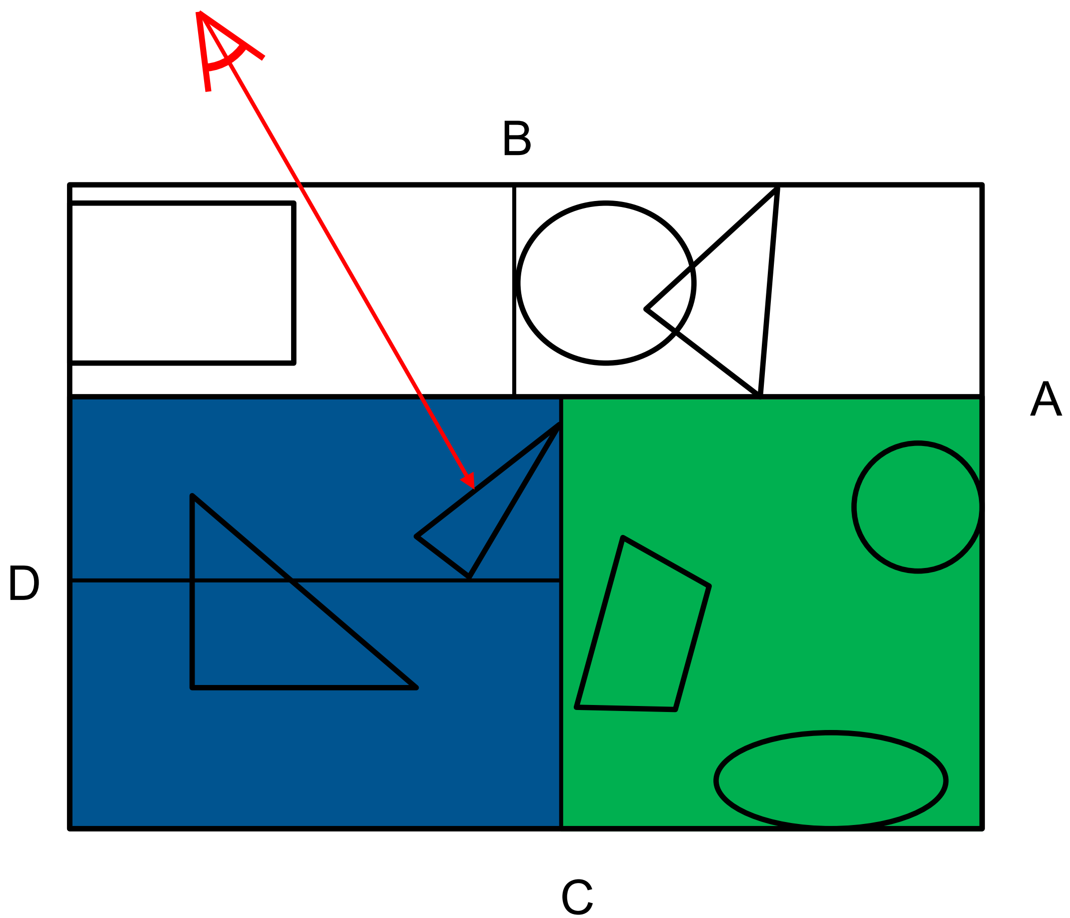
- **For each primitive in current leaf node**
  - Check if ray intersects the primitive
  - If closest (positive) hit so far, record hit

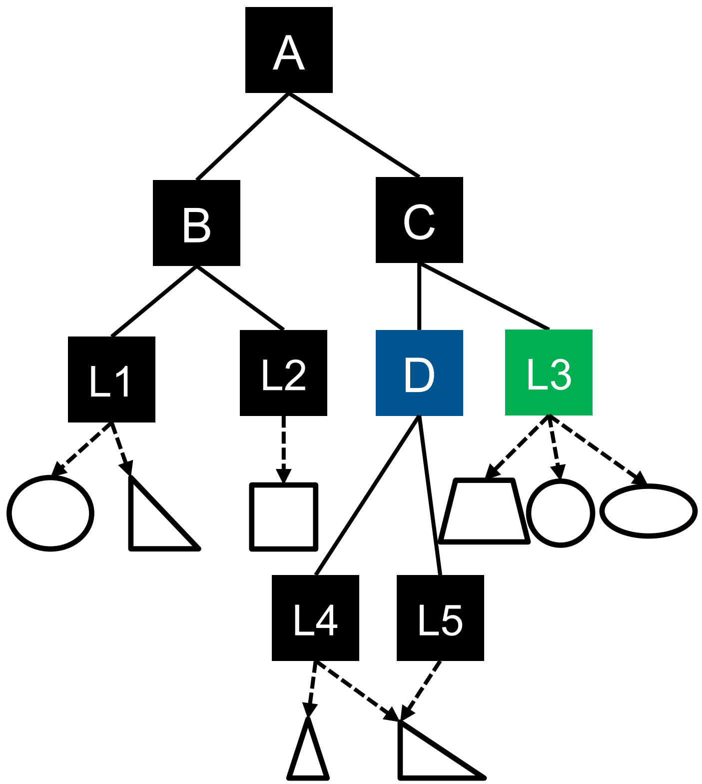




kD-Tree Traversal

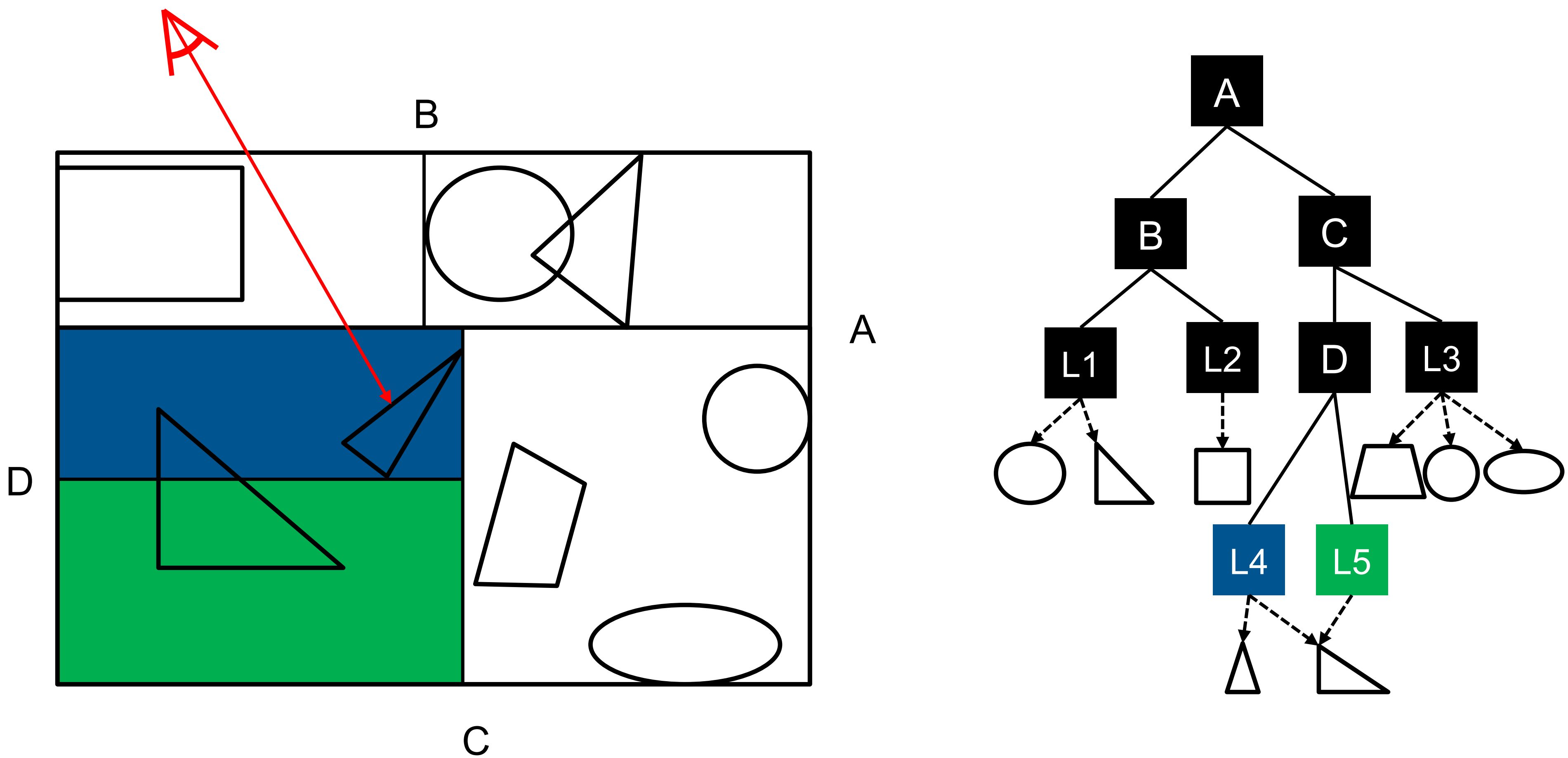
- **Front-to-back traversal**
  - Traverse child nodes in order along rays





kD-Tree Traversal

- **Ordered Traversal**
  – Skip 2nd child if intersection found in 1st child belongs to the cell



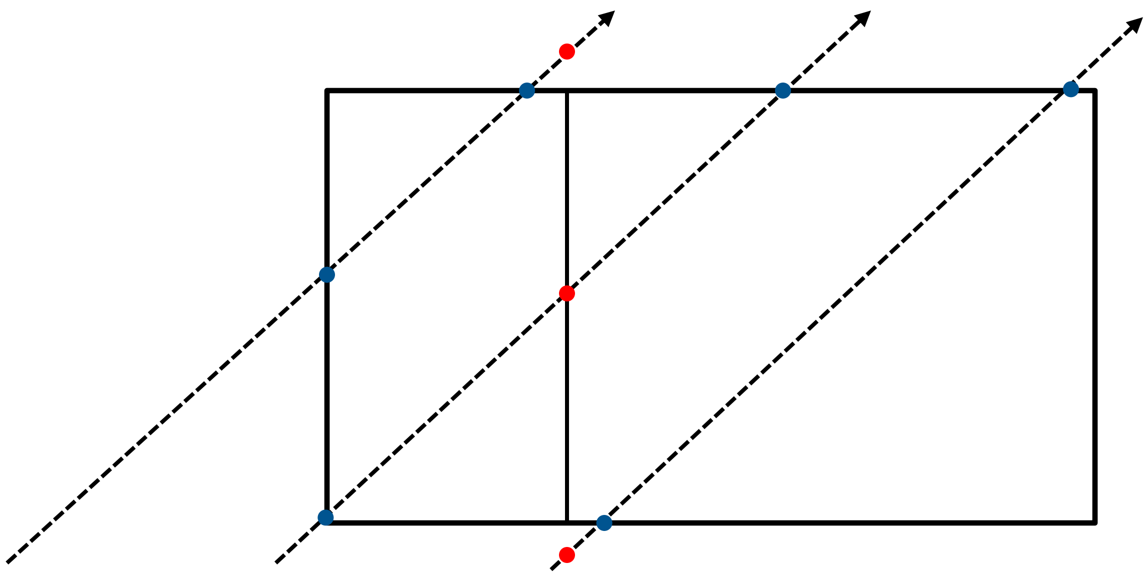
kD-Tree Storage

- **Node Representation**
  - Leaf flag + split axis (x, y, z or leaf)
    - 2 bits
  - Split location (1D)
    - 1 real
  - Pointers to children / geometry
    - 2 pointers
    - 1 pointer & 1 integer

- **Bounding Box**
  - Nodes of k-D tree represent axis-aligned bounding boxes
  - Do not need to be explicitly stored
  - Ray interval can instead be implicitly updated

kD-Tree Traversal

- **Initialize entry/exit distances at root's bounding box**
  - t_near & t_far
- **Compare split distance to node's entry/exit distances**
  - t_split >= t_far Go only to near node
  - t_near < t_split < t_far Go to both
  - t_split <= t_near Go only to far node
- **Near and far depend on direction of ray!**



kD-Tree Traversal

```
KDTreeIntersect(ray, node, t_near, t_far):
If (node is inner node)
    t_split = (node.splitCoord – ray.pos[node.splitAxis] ) / ray.dir[node.splitAxis];
    if (t_split >= t_far)
        KDTreeIntersect(ray, node.near_child, t_near, t_far); // near child only
    else if  (t_split <= t_near)
        KDTreeIntersect(ray, node.far_child, t_near, t_far);      // far child only
    else // hit both children
        KDTreeIntersect(ray, node.near_child, t_near, t_split);
        KDTreeIntersect(ray, node.far_child, t_split, t_far);
else
    Iterate through node.listOfObjects and record closest intersection;
    If (intersection is within [t_near, t_far]) abort traversal;
```

- **Computationally Inexpensive**
  - One subtraction, division, decision, and fetch
  - But many more cycles due to dependencies

KD Tree Discussion

- **Properties**
  - Logarithmic intersection cost
  - Adaptive to local geometric density

- **Trade-Offs**
  - Relatively expensive build cost
  - Relatively cheap traversal cost

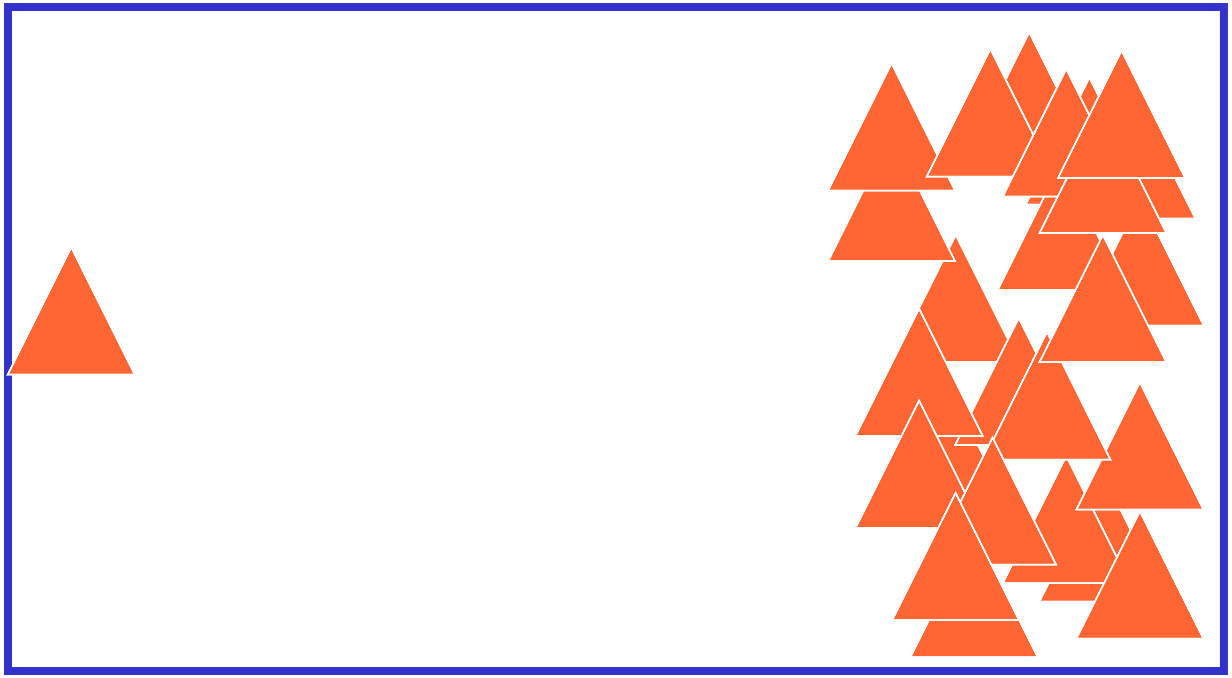
TREE OPTIMIZATIONS

Building Trees

- **Given**
  - Axis-aligned bounding box (“cell”)
  - List of geometric primitives (e.g. triangles) touching cell

- **BVH and kD-Tree Core Operations**
  - Pick an axis-aligned plane to split the cell into two parts
    - kD tree: use extent of bounding box of geometry
    - BVH: use extent of bounding box of centroids instead
      - E.g. yields infinitely thin bounding box for 3 aligned centroids
  - Sift geometry into two (possibly redundant) batches
  - Recurse
  - Stop when termination criterion is met

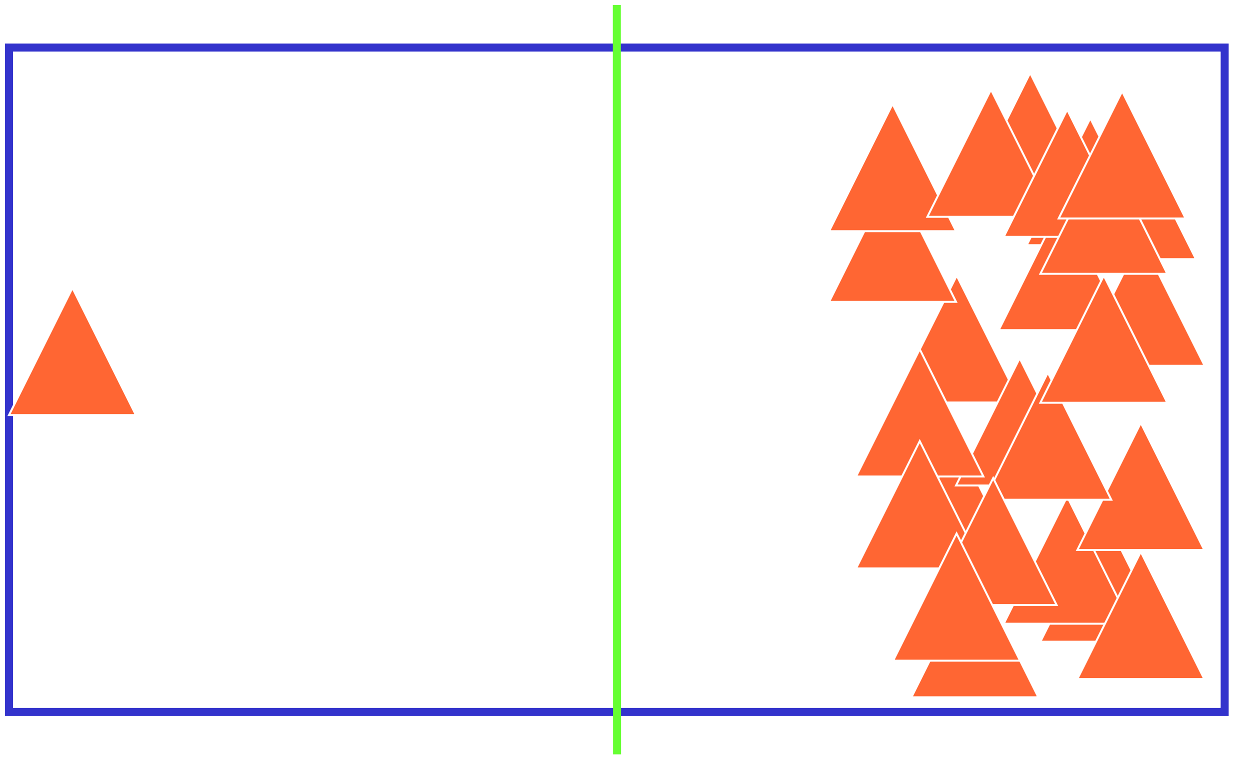
Splitting



Minimize Extents

- **Split Axis**
  - Largest extent
- **Split Location**
  - Middle of extent
- **Termination Criterion**
  - Size of (non-empty) cell < predefined threshold

Split in the Middle

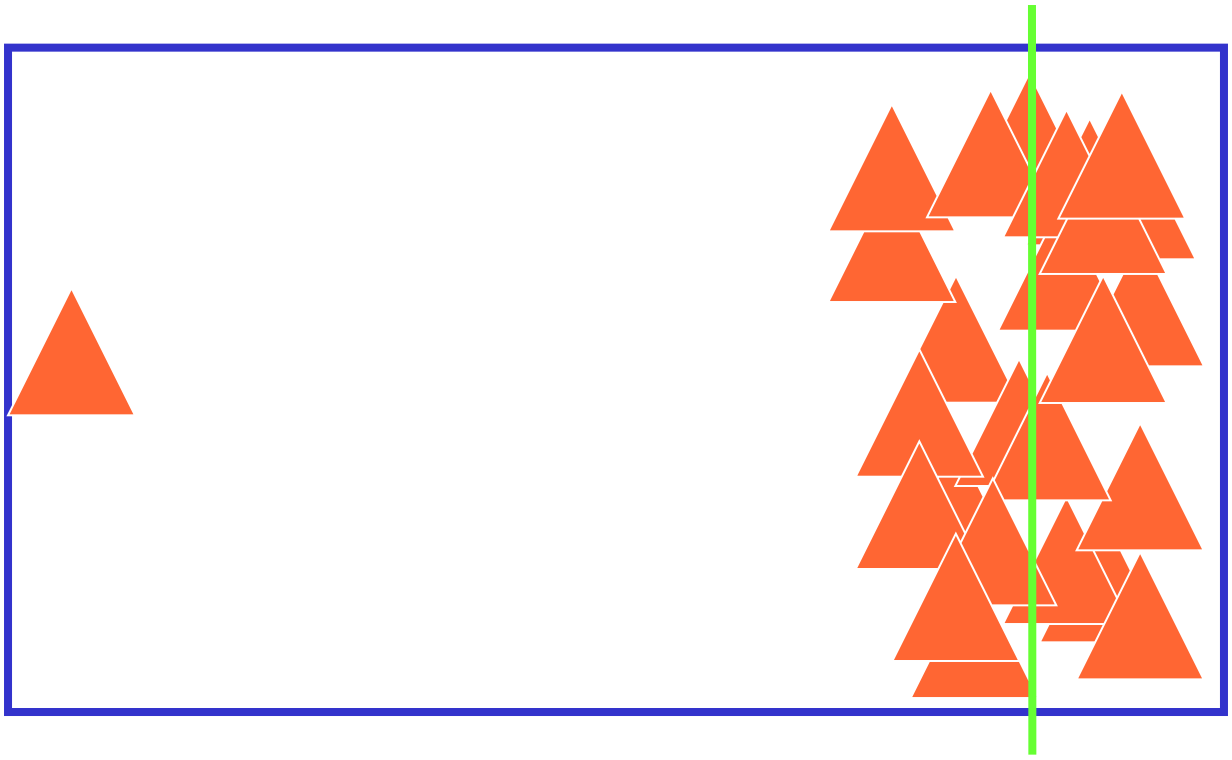


- **Makes the L & R probabilities equal**
- **Pays no attention to the L & R costs**

Minimize Primitives

- **Split Axis**
  - Round-robin
- **Split Location**
  - Median of geometry (balanced tree)
- **Termination Criterion**
  - Number of objects < predefined threshold

Split at the Median



- **Makes the L & R costs equal**
- **Pays no attention to the L & R probabilities**

Minimize Extents and Primitives

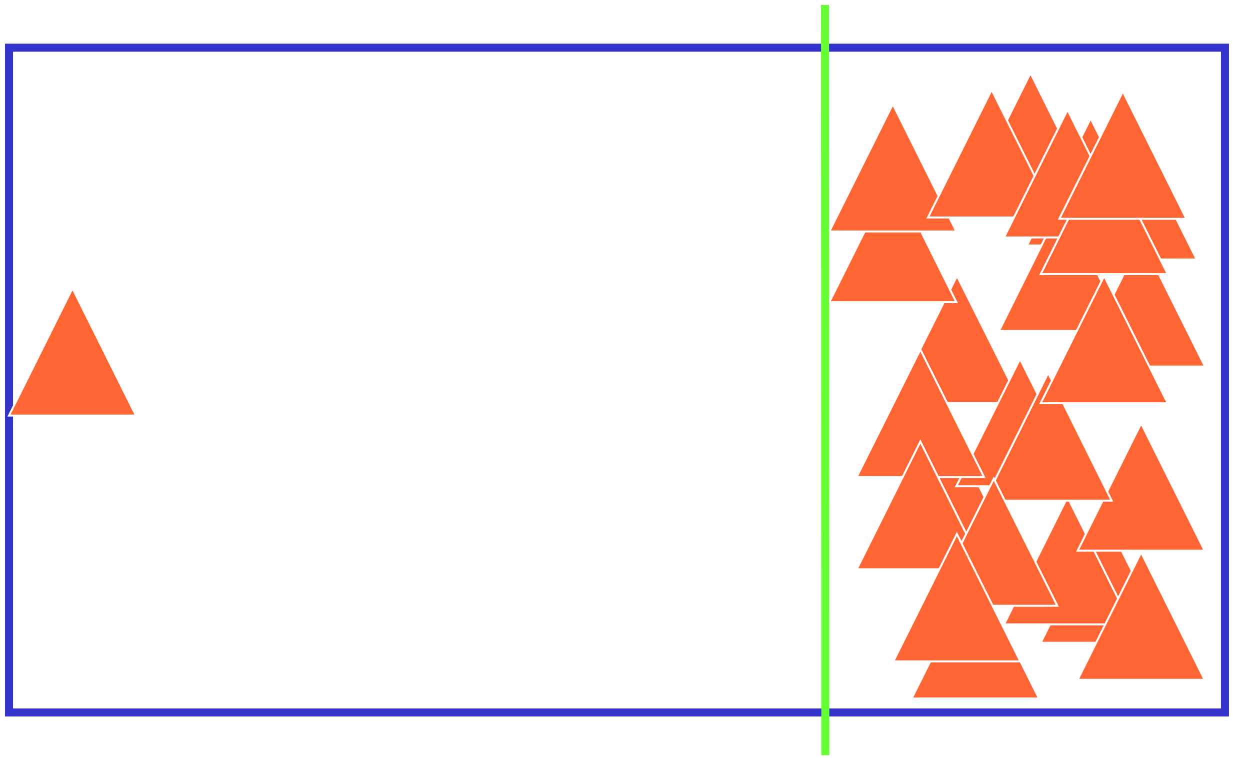
- **What split do we really want?**
  - The one that makes ray tracing cheap
  - Formulate an expression of cost and minimize it
  - Cost optimization

- **What is the cost of tracing a ray through a node?**
  - Cost(cell) = traversalCost + Prob(hit L | hit P) * Cost(L)
    + Prob(hit R | hit P) * Cost(R)

Surface Area Heuristic

- **Compute the Probability**
  - Assume uniform directional distribution of rays
  - Probability turns out to be proportional to *surface area*
  - Area of outer surface of bounding box, not its volume
  - Prob(hit N | hit P) = SA(N) / SA(P)
- **Compute the Cost**
  - Should recursively compute the cost of the subtree
  - Use the cost of a leaf node as a greedy approximation
  - Cost(N) = ObjectCount(N) * intersectionCost
- **Tuning Parameters**
  - traversalCost
  - intersectionCost
  - Build behavior only depends on their relative ratio

Cost-Optimized Split



- **Automatically isolates complexity**
- **Produces large chunks of empty space**

Minimize Extents and Primitives

- **Split Axis**
  - Iterate over each of the 3 axes in turn
  - Record axis whose split location with minimal cost is optimal
- **Split Location**
  - Iterate over all candidates along the axis currently considered
    - BVH: Centroid intervals
    - K-D tree: Extrema of cost function at boundaries of bounding boxes
  - Compute cost for each and record candidate with minimal cost
    - Naively compute cost individually → N^2 operations
      - Iterate over the set of objects
      - Compute left/right bounding volumes and primitive numbers
    - Sort the candidates along the given axis → N log(N) operations
      - BVH: incrementally compute surface areas both forward and backward
      - KD tree: incrementally compute primitive numbers from min/max bounds

Minimize Extents and Primitives

- **Termination Criterion**
  - When optimal Cost(cell) of splitting > Cost(N) of creating a leaf

- **Additional Criteria**
  - Avoid infinite loops
    - E.g. all objects of a BVH node put in single child
  - Bound memory consumption
    - Limited tree depth

Minimize Extents and Primitives

- **MakeSplitDecision(objects):**

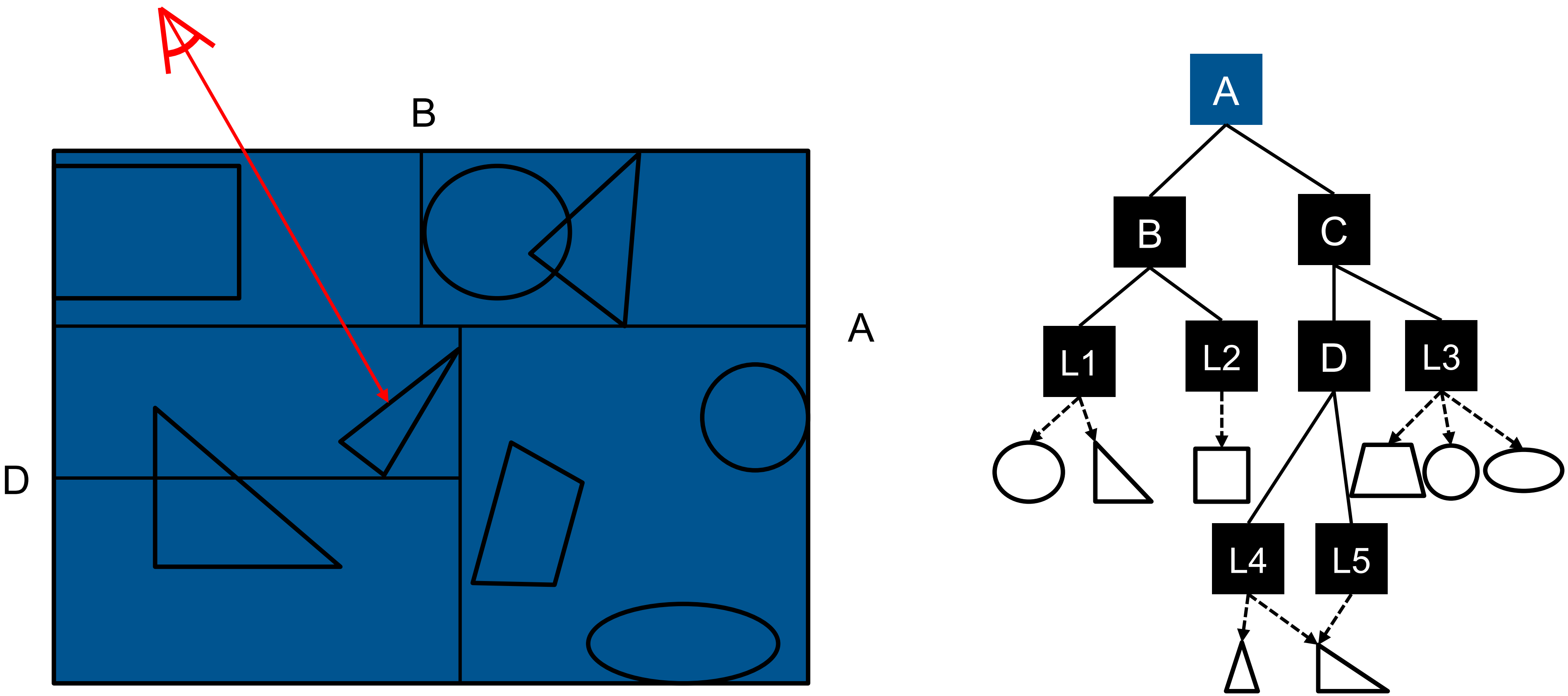
- **For(axis a in x/y/z)**
  - For(candidate c in [sorted] candidates along a)
    - Compute left and right object counts;
    - Compute left and right surface areas;
    - splitCost = Cost(c);
    - If (splitCost < bestCost)
      - bestCost = splitCost;
      - bestSplit = c;
      - bestAxis = a;
- **splitDecision = (bestCost < LeafCost(objects));**

Stack-Based Traversal

- **Avoid overhead of recursive function calls**
  - No need for a true recursion

- **Explicitly maintain stack of sub-trees to traverse**
  - Optimize by minimizing stack operations

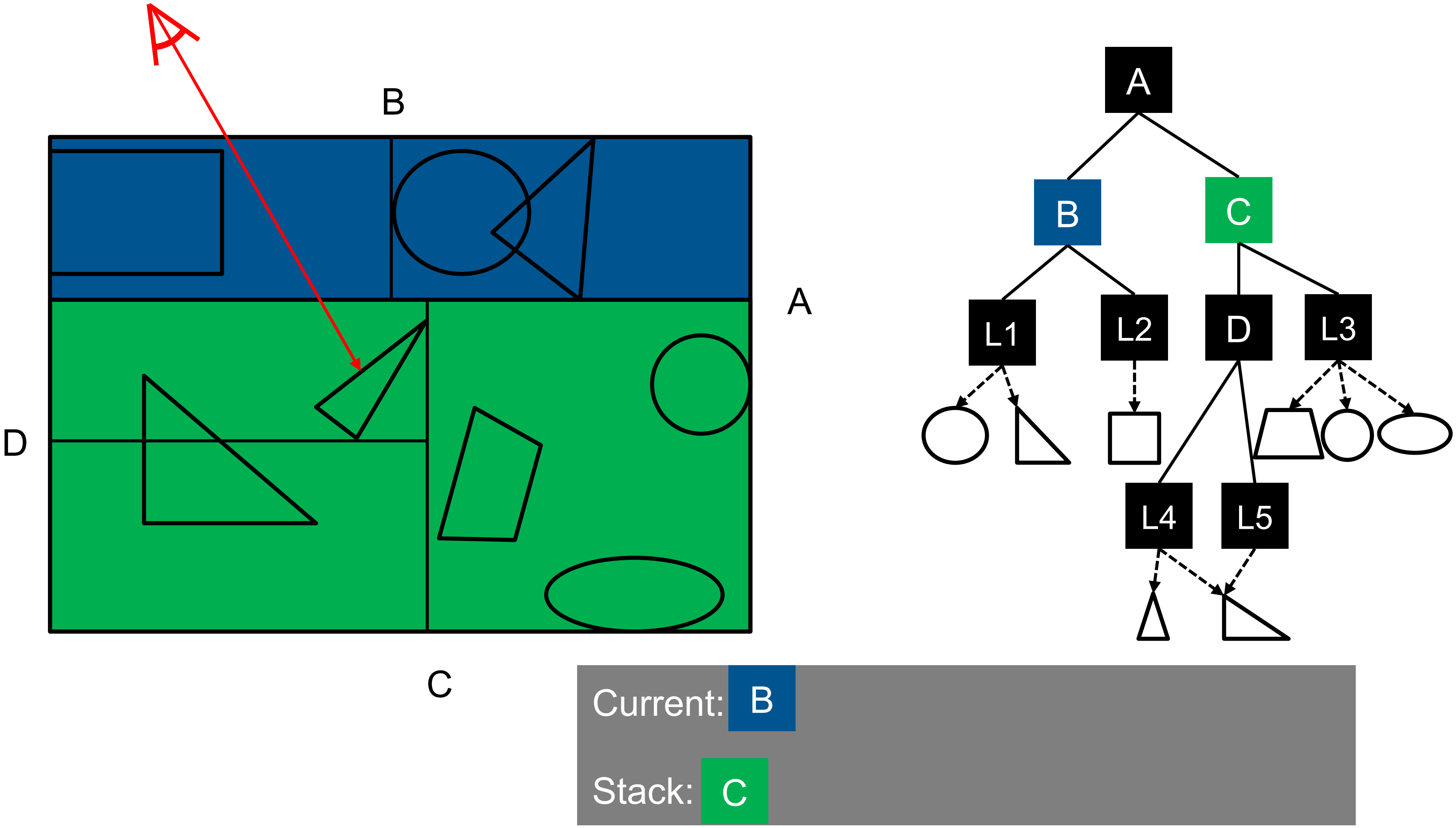
Stack-Based Traversal



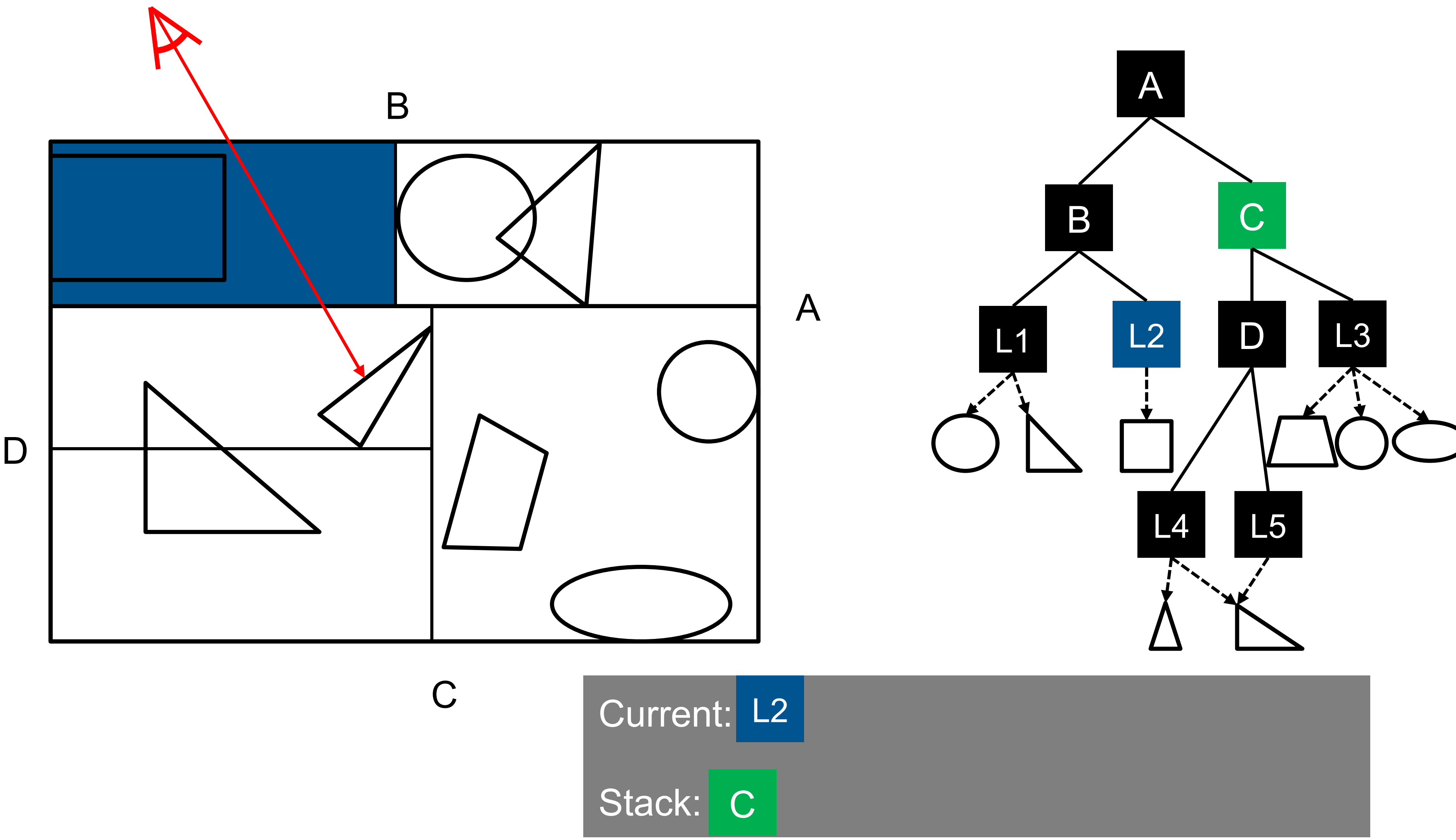
Current: A

Stack:

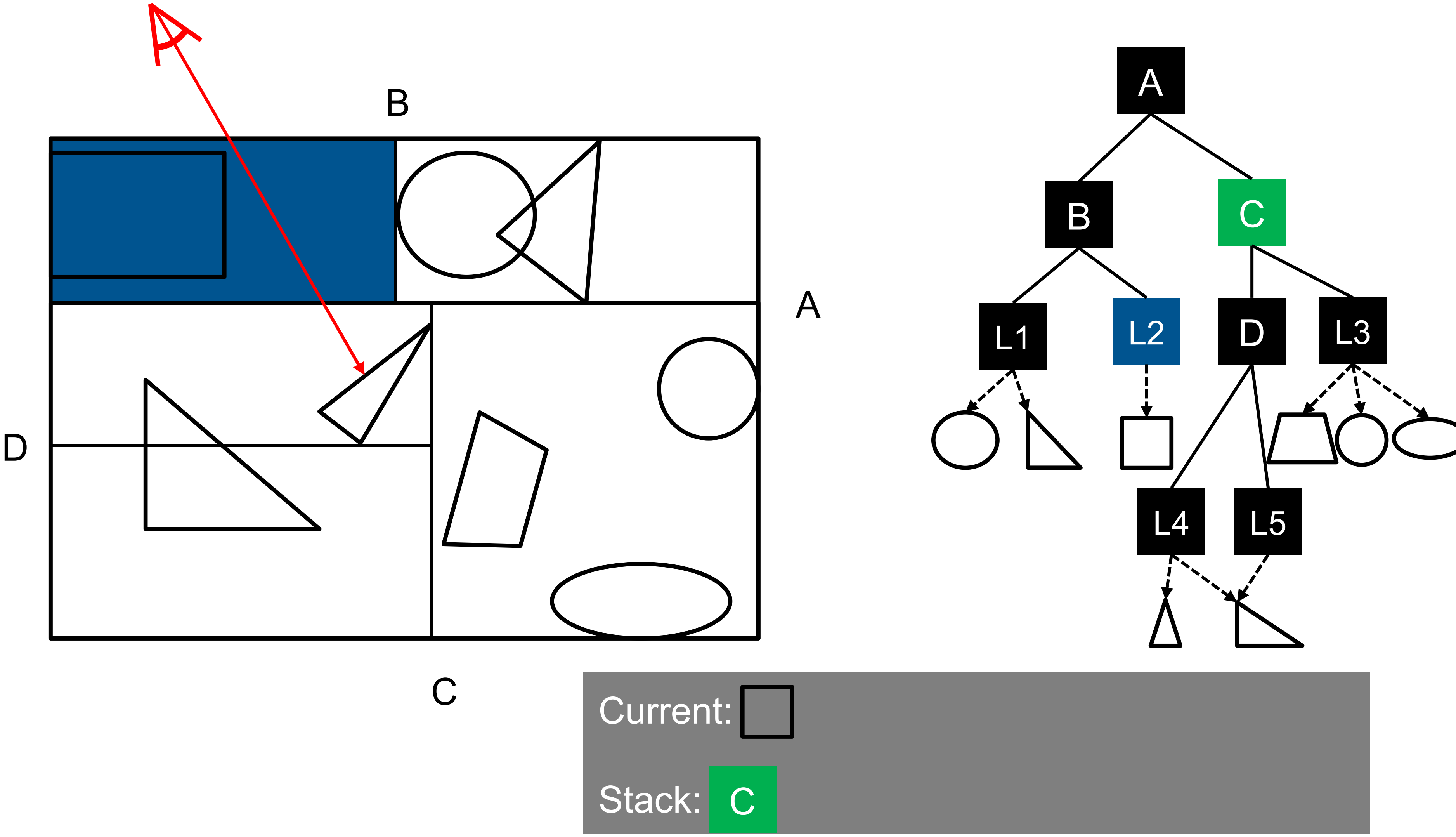
Stack-Based Traversal



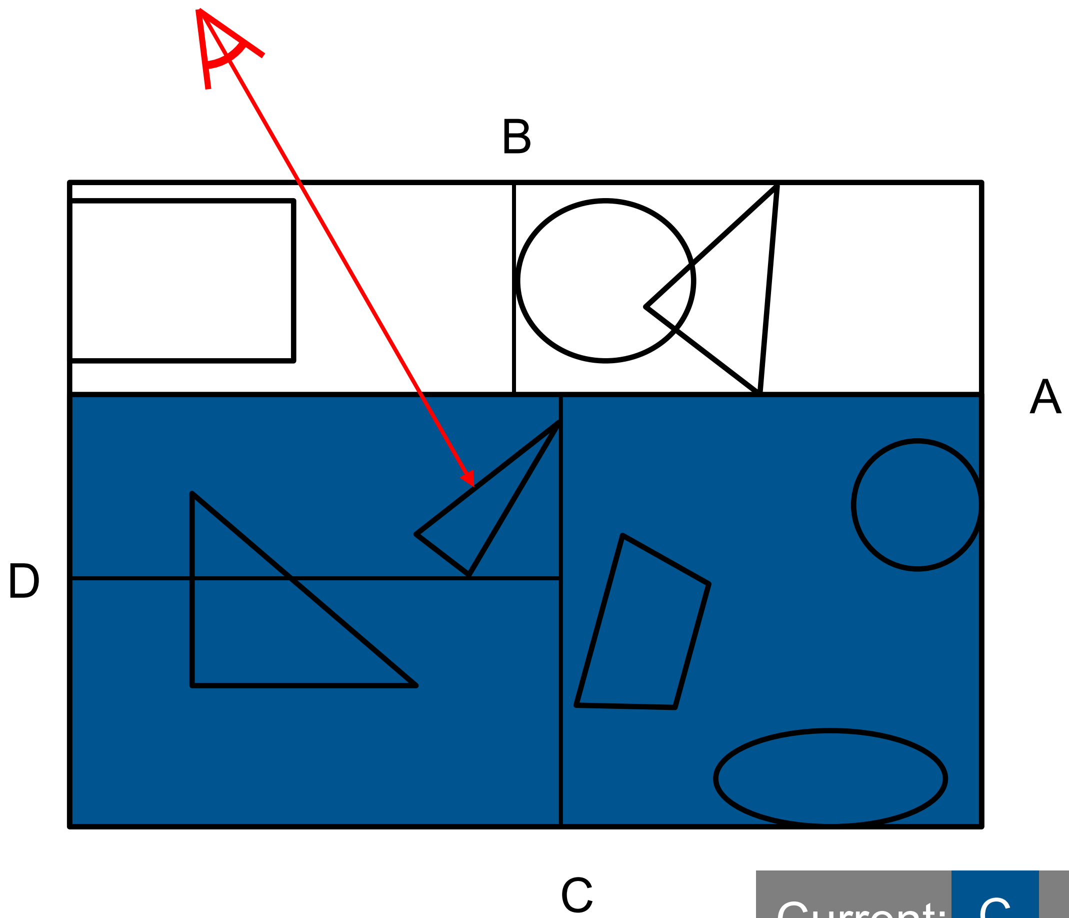
Stack-Based Traversal

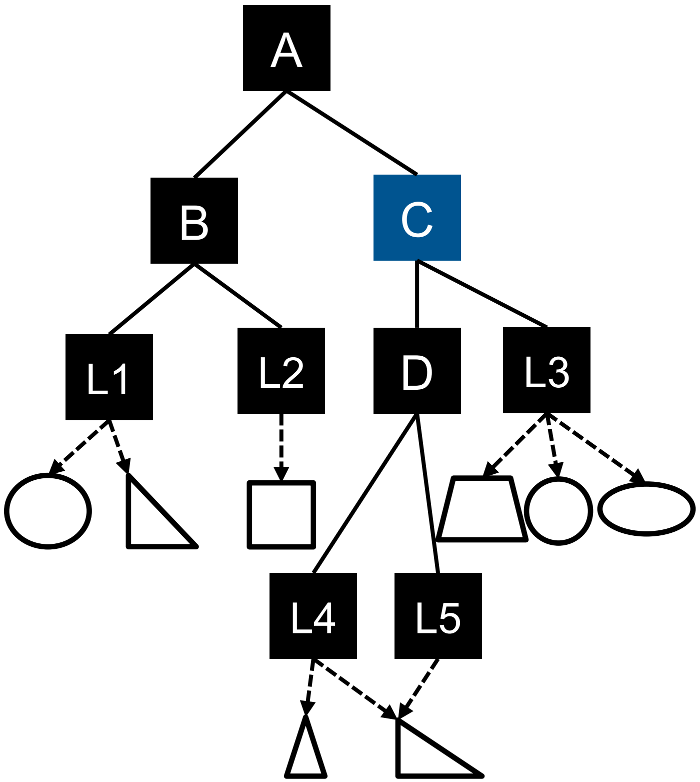


Stack-Based Traversal



Stack-Based Traversal

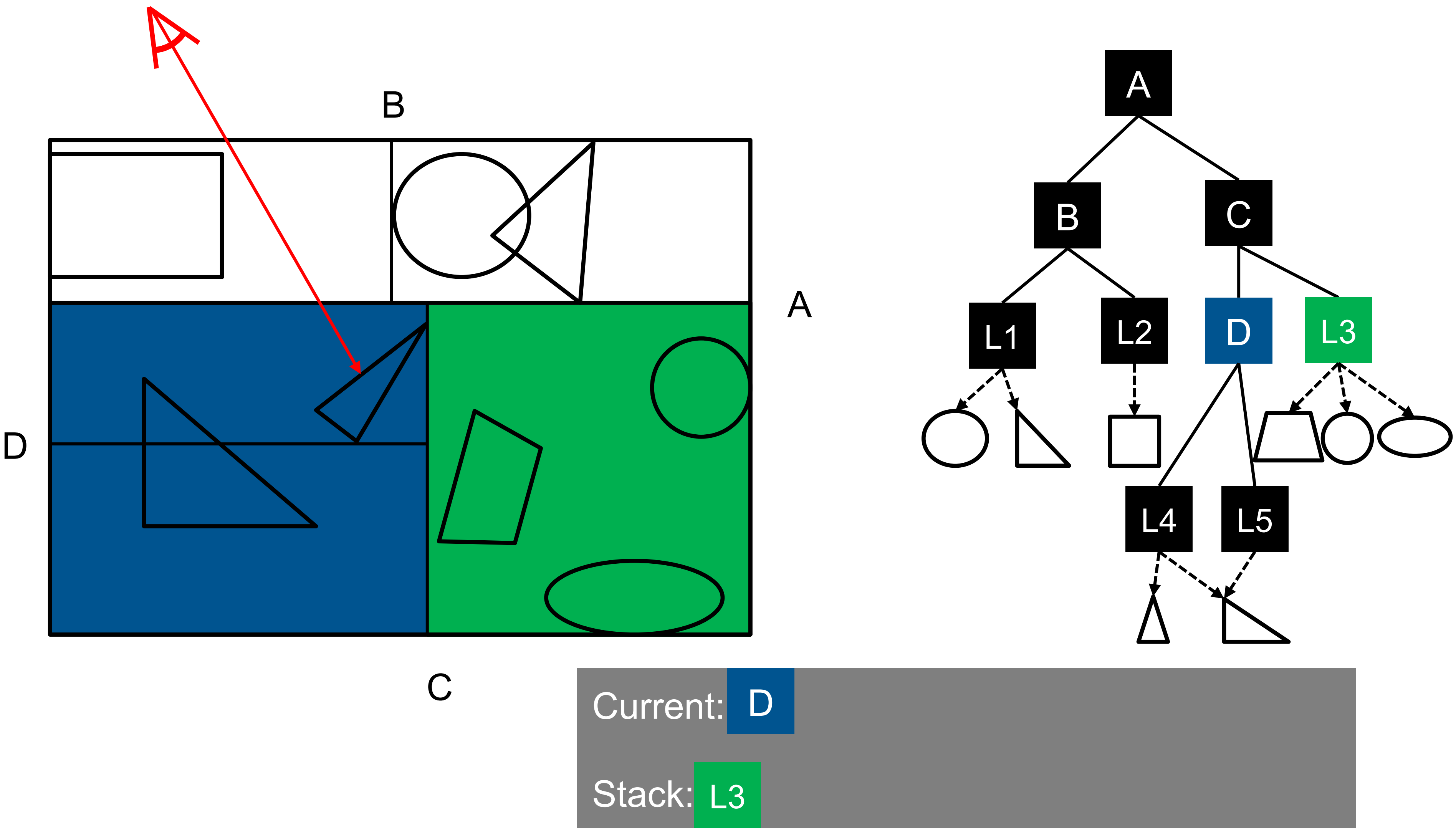




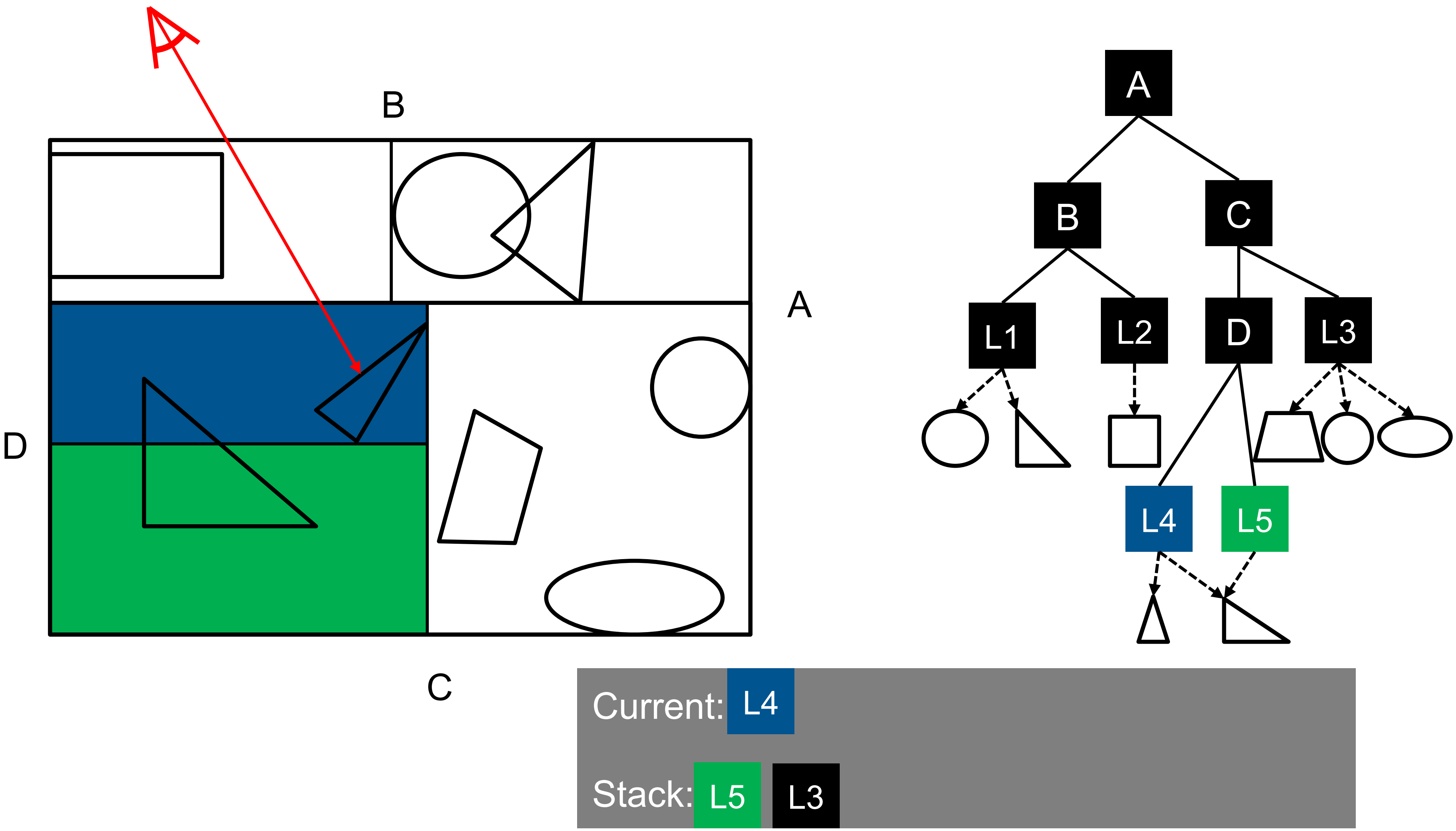
Current: C

Stack:

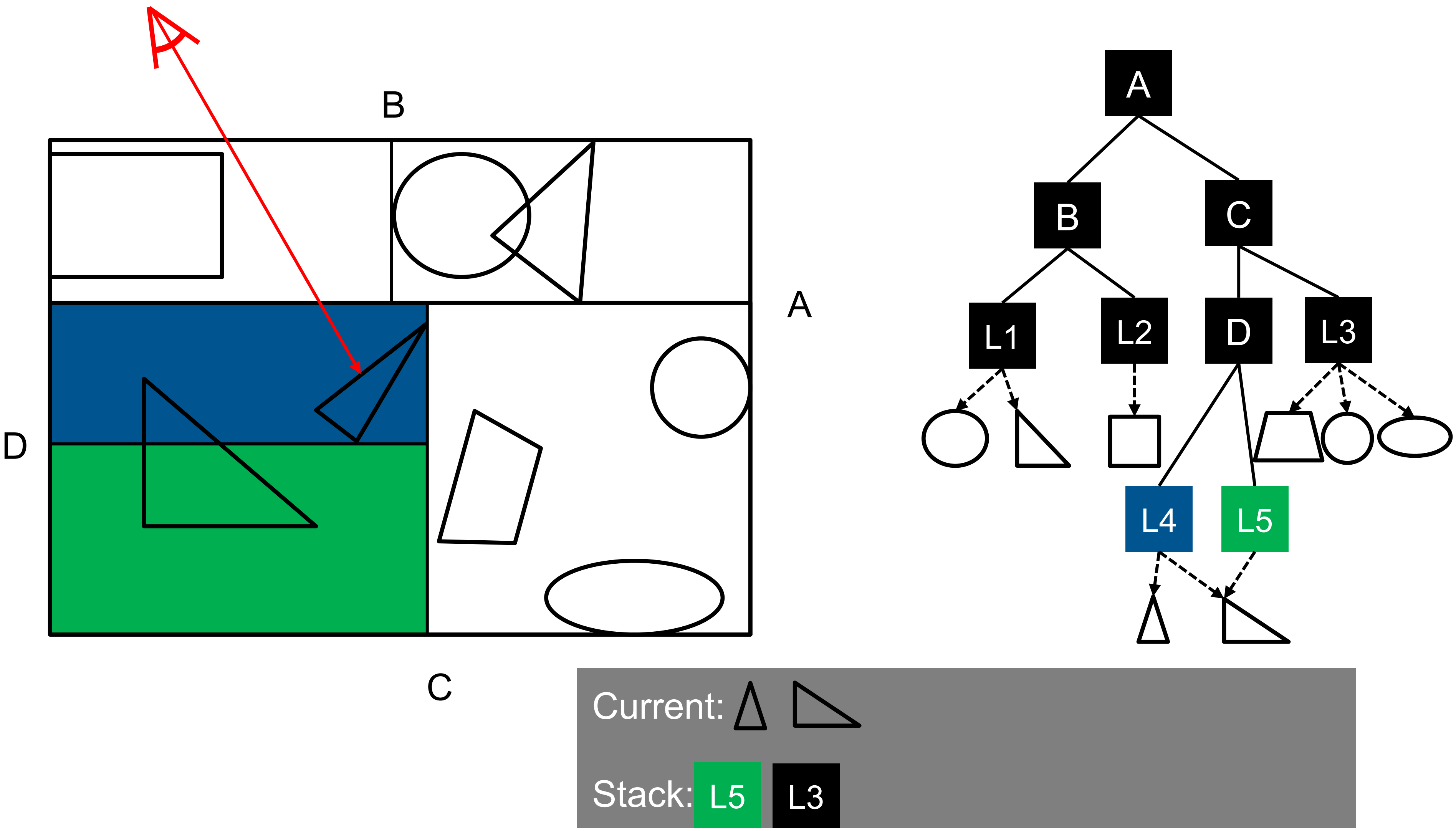
Stack-Based Traversal



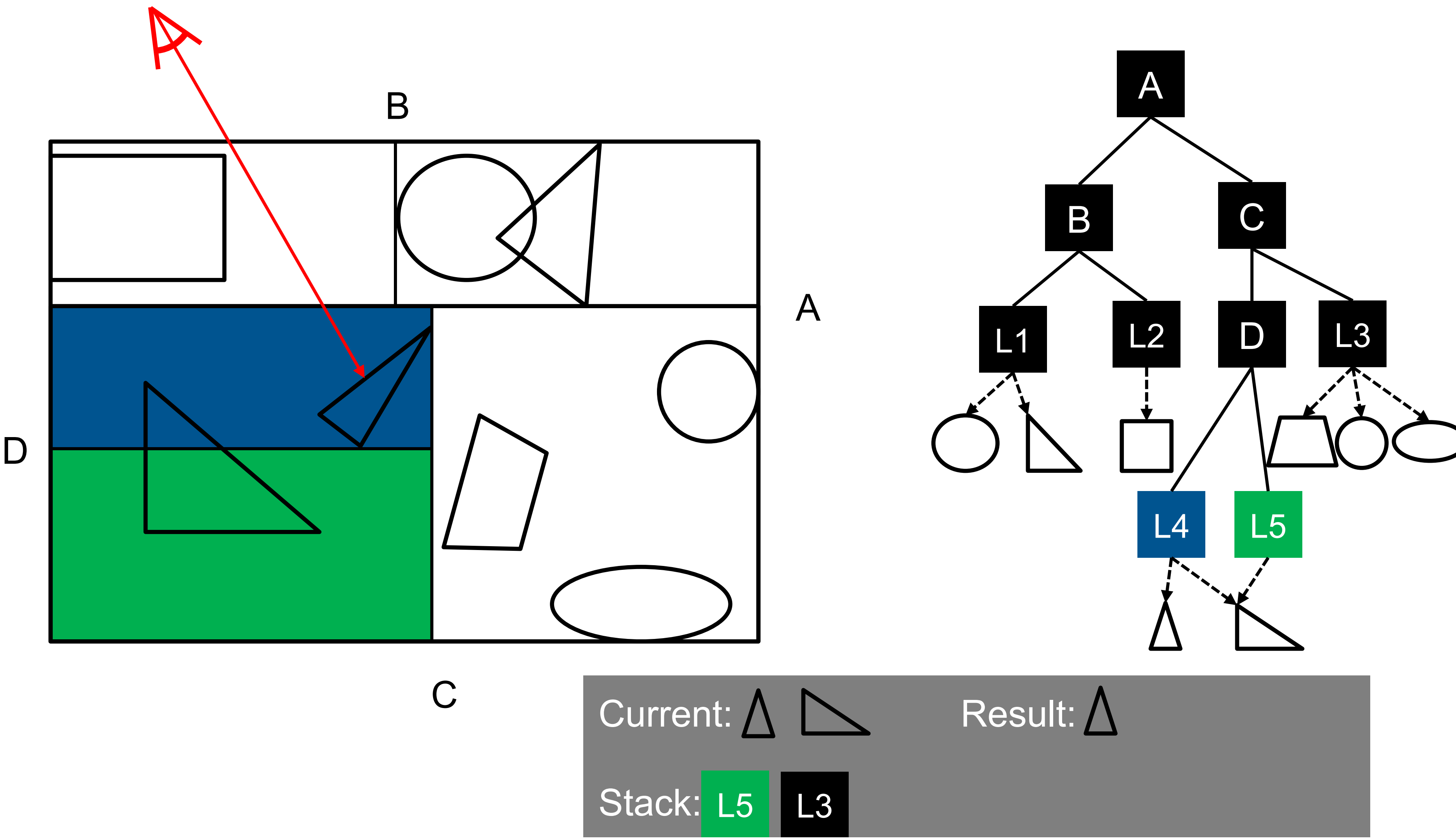
Stack-Based Traversal



Stack-Based Traversal



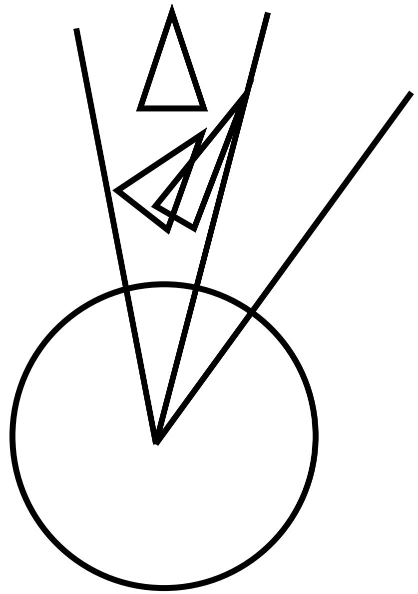
Stack-Based Traversal



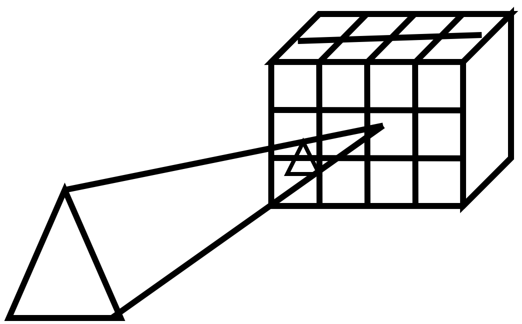
DIRECTIONAL APPROACHES

Directional Partitioning

- **Applications**
  - Useful only for rays that start from a fixed point
    - Camera (assuming no camera movement)
    - Point light sources
  - Preprocessing of visibility
    - For each object locate where it is visible

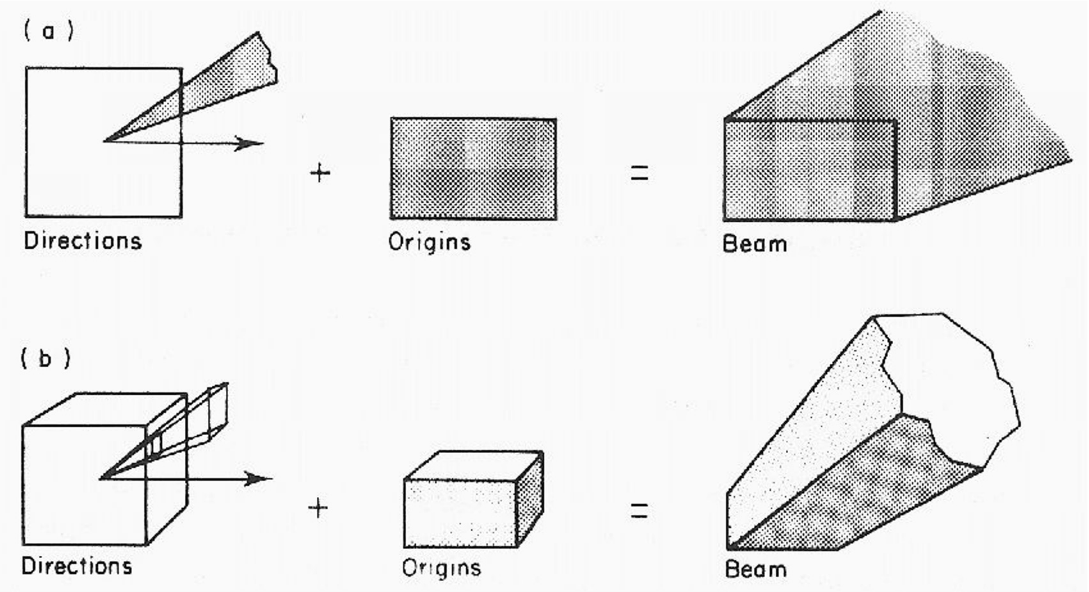


- **Variation: “light buffer”**
  - Lazy and conservative evaluation
  - Store occluder that was found in directional structure
  - Test entry first for next shadow test



5D Partitioning

- **Partitioning of space and direction**
- **Roughly pre-computes visibility for the entire scene**
    - What is visible from each point in each direction?
  - Very costly preprocessing, cheap traversal
    - Improper trade-off between preprocessing and run-time
  - Memory hungry, even with lazy evaluation
  - Seldom used in practice



WRAP-UP

Battle of Acceleration Structures

- **Trade-Off**
  - Build vs. Traversal Cost
  - Target time to image
    - Preprocessing time + rendering time
    - Preprocessing time depends on geometry
    - Rendering time depends on total number of rays

- **Some Structures better than Others**
  - Depending on input geometry
  - Depending on rendering task
  - No absolute best!

Nested Acceleration Structures

- **Acceleration structure is a geometric primitive**
- **Can be used inside another acceleration structure**
- **Build meta-acceleration structure**