Computer Graphics

- Spatial Index Structures -

Philipp Slusallek

Motivation

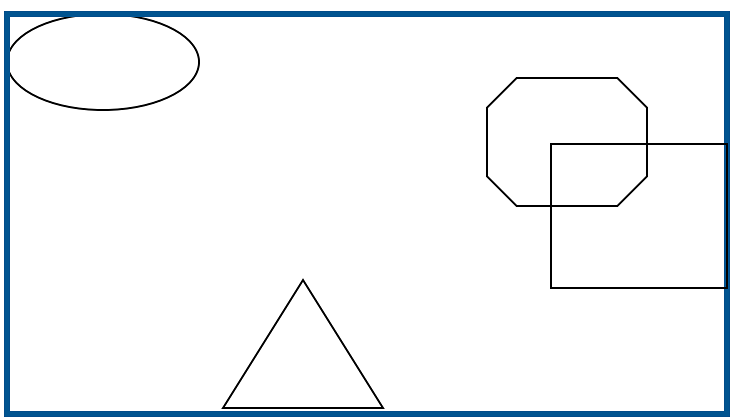
- **Tracing rays in O(n) is too expensive**
  - Need hundreds of millions rays per second
  - Scenes consist of millions of triangles
- **Reduce complexity through pre-sorting data**
  - Spatial index structures
    - Dictionaries of objects in 3D space
  - Eliminate intersection candidates as early as possible
    - Can reduce complexity to $O(\log n)$ on average
  - Worst case complexity is still $O(n)$
    - *Private exercise: Come up with a worst case example*

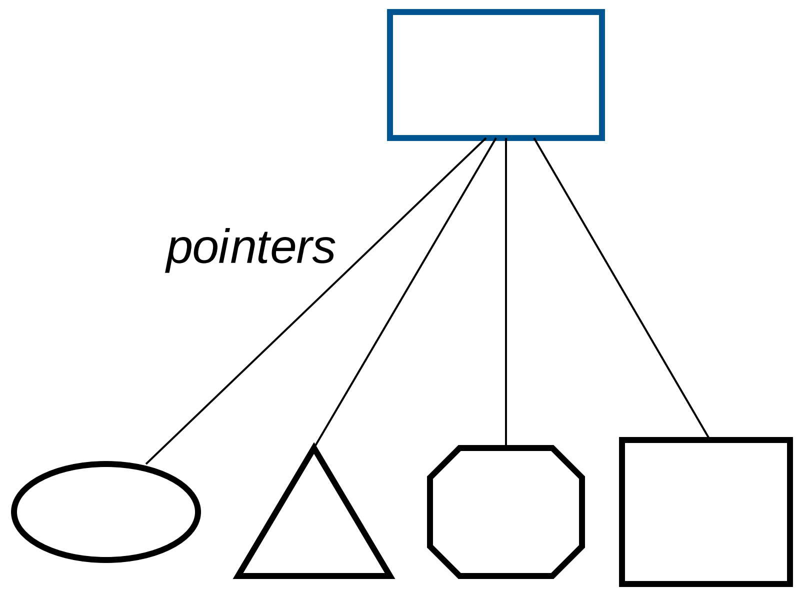
Acceleration Strategies

- **Faster ray-primitive intersection algorithms**
  - Does not reduce complexity, “only” a constant factor (but relevant!)
- **Less intersection candidates**
  - Spatial indexing structures
  - (Hierarchically) partition space *or* partition the set of objects, e.g.:
    - Grids, hierarchies of grids
    - Octrees
    - Binary space partitions (BSP) or kd-trees
    - Bounding volume hierarchies (BVH)
  - Directional partitioning (not very useful)
  - 5D partitioning (partition space *and* direction, once a big hype)
    - Close to pre-computing visibility for all points and all directions
- **Tracing of continuous bundles of rays**
  - Exploits coherence of neighboring rays, amortize cost among them
    - Frustum tracing, cone tracing, beam tracing, ...

Aggregate Objects

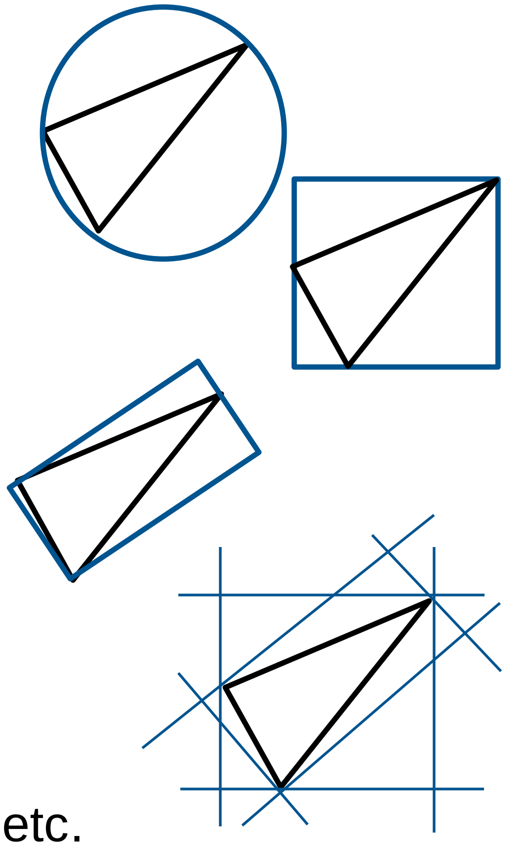
- **Object that holds groups of objects**
- **Conceptually stores bounding volume (e.g. box) & a list of children**
- **Useful for instancing (placing collection of objects repeatedly) & for Bounding Volume Hierarchies (BVHs)**





Bounding Volumes

- **Observation**
  - BVs (tightly) bound geometry, ray must intersect BV first
  - Only compute intersection if ray hits BV
- **Sphere**
  - Very fast intersection computation
  - Often inefficient because it is too large
- **Axis-aligned bounding box (AABB)**
  - Very simple intersection computation (min-max)
  - Sometimes too large
- **Non-axis-aligned box**
  - A.k.a. „oriented bounding box (OBB)“
  - Often better fit
  - Fairly complex computation
- **Slabs**
  - Pairs of half spaces (in addition to 3 for AABB)
  - Fixed number of orientations/axes: e.g. x+y, x-y, etc.
    - Pretty fast computation, but more expensive then AABB

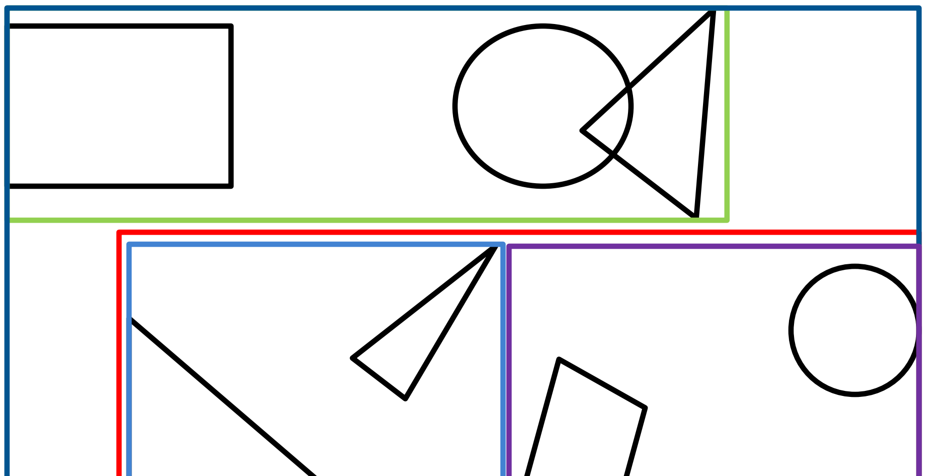


Bounding Volume Hierarchies (BVHs)

- **Definition**
  - Hierarchical ***partitioning of a set of objects***
- **BVHs form a tree structure**
  - Each inner node stores a volume enclosing all sub-trees
  - Each leaf stores a volume and pointers to objects
  - All nodes are aggregate objects
  - Usually every object appears once in the tree
    - Except in case of *instancing*

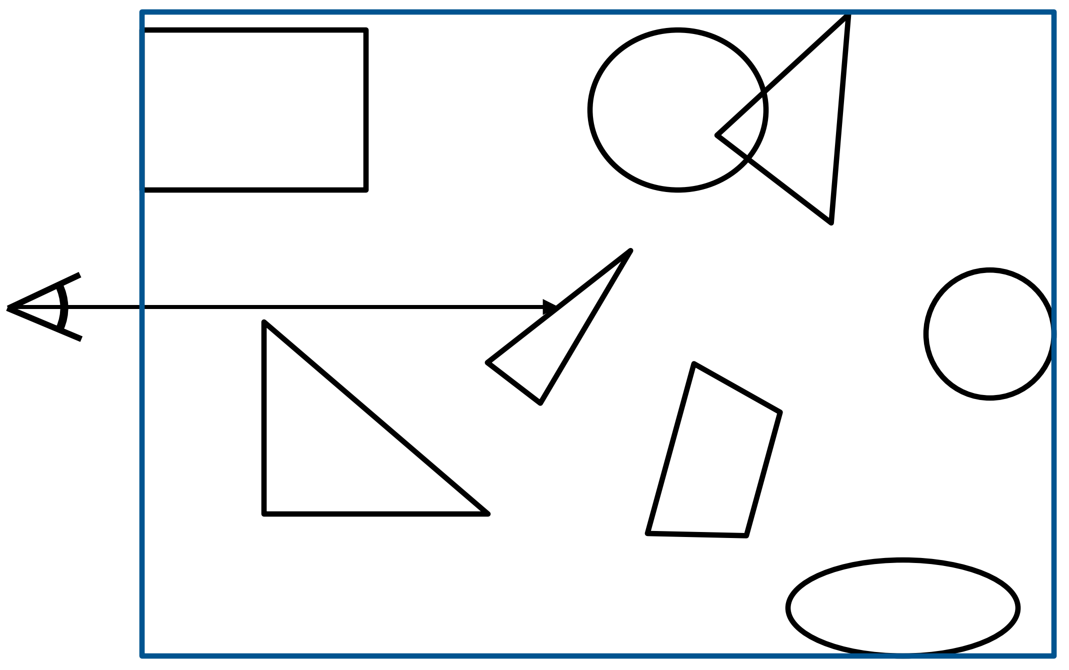
Bounding Volume Hierarchies (BVHs)

- **Hierarchy of groups of objects**



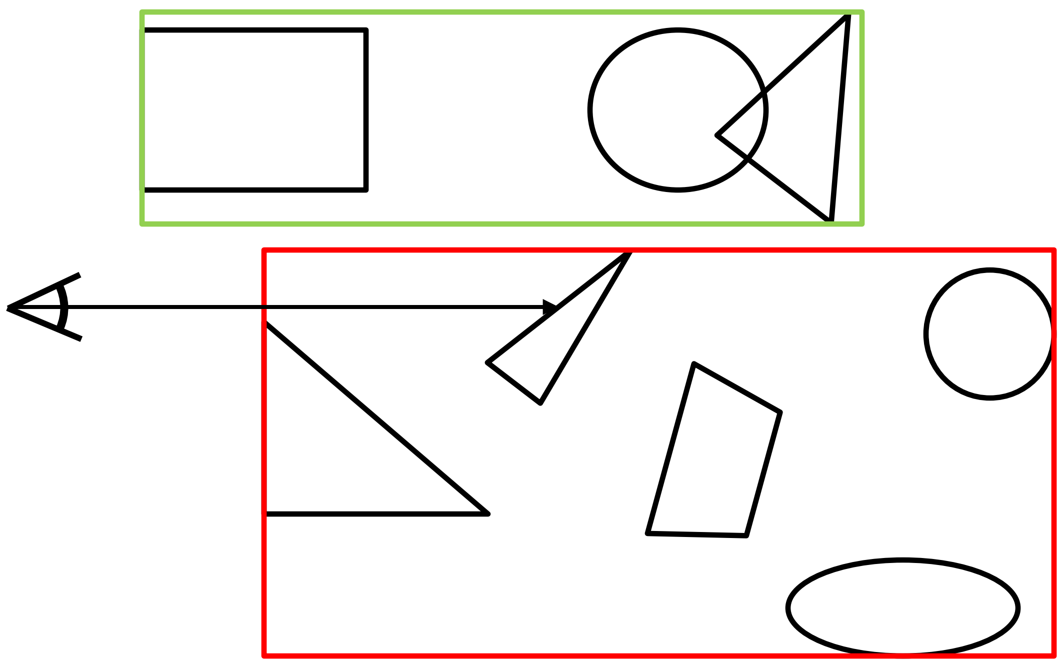
BVH traversal (1)

- **Accelerate ray tracing**
  - By eliminating intersection candidates
- **Traverse the tree**
  - Consider only objects in leaves intersected by the ray



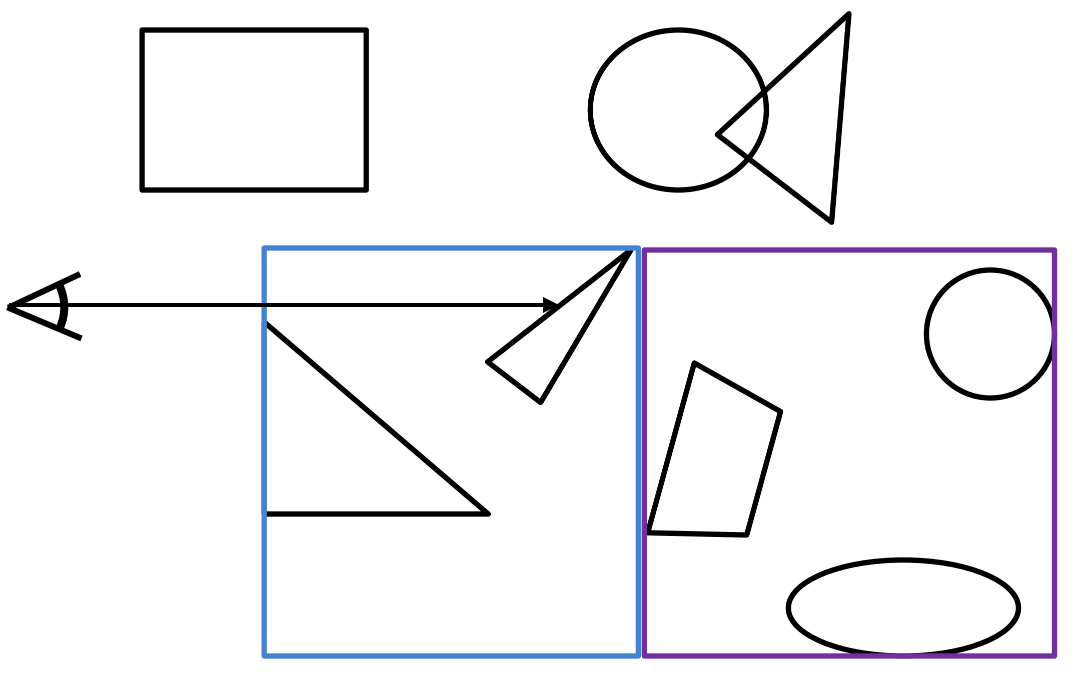
BVH traversal (2)

- **Accelerate ray tracing**
  - By eliminating intersection candidates
- **Traverse the tree**
  - Consider only objects in leaves intersected by the ray



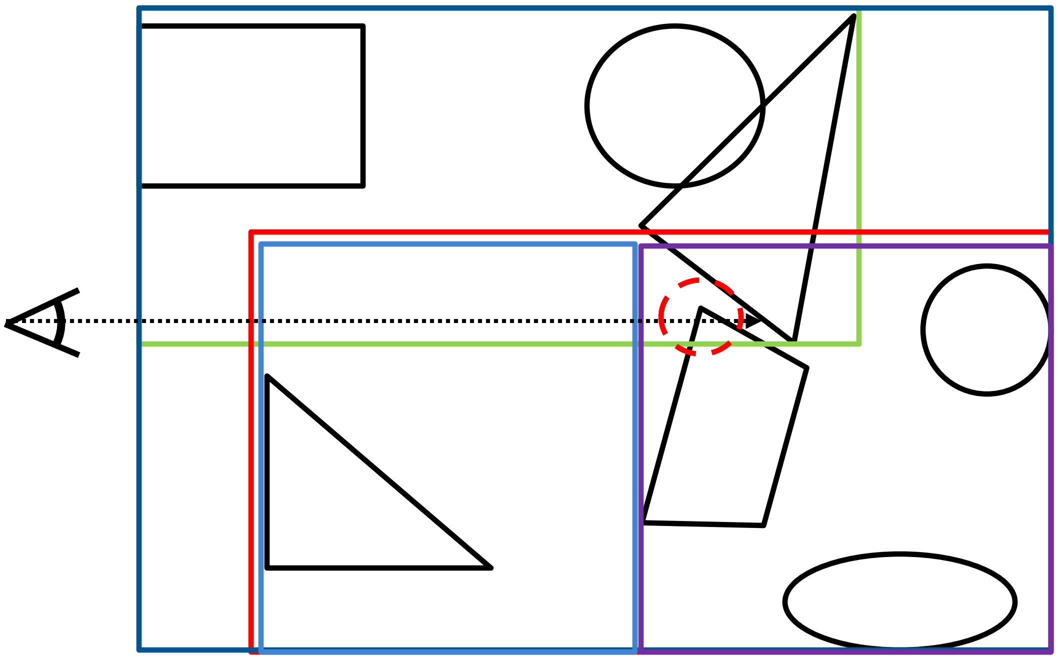
BVH traversal (3)

- **Accelerate ray tracing**
  - By eliminating intersection candidates
- **Traverse the tree**
  - Consider only objects in leaves intersected by the ray
  - Cheap traversal instead of costly intersection



Bounding Volume Hierarchies (BVHs)

- **BV can also overlap**
  - Cannot terminate on first intersection found
  - There could be an earlier object in an overlapping BV
  - Can only terminate, once all remaining BVs are completely behind the intersection



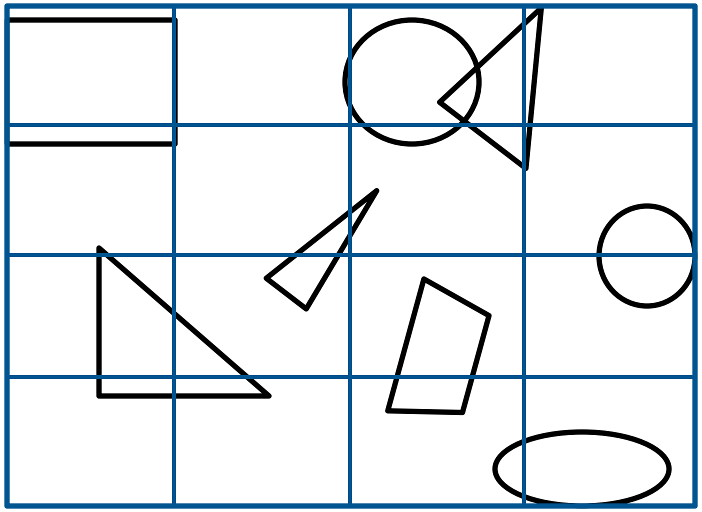
Object vs. Space Partitioning

- **Object partitioning**
  - BVHs hierarchical partition *objects* into groups
  - Create spatial index by spatially bounding each subgroup
  - Subgroups may be overlapping !
- **Space partitioning**
  - (Hierarchically) partitions *space* in subspaces
  - Subspaces are non-overlapping and completely fill parent space
  - Organize them in a structure (tree or table)

- **Next: Space partitioning**

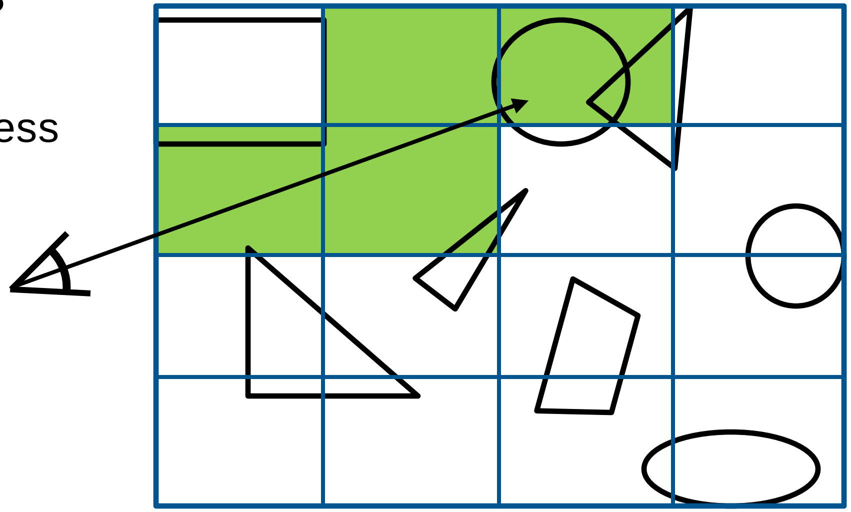
Uniform Grids

- **Definition**
  - Regular partitioning of space into equal-size cells
  - Non-hierarchical structure
- **Resolution**
  - Want: number of cells is $O(n)$
  - Resolution in each dimension proportional to $\sqrt[3]{n}$
  - Usually $R_{x,y,z} = d_{x,y,z}\sqrt[3]{\frac{\lambda n}{V}}$
    - *d: diagonal of box (a vector)*
    - *n: #objects*
    - *V: volume of Bbox*
    - *λ: density (user-defined)*



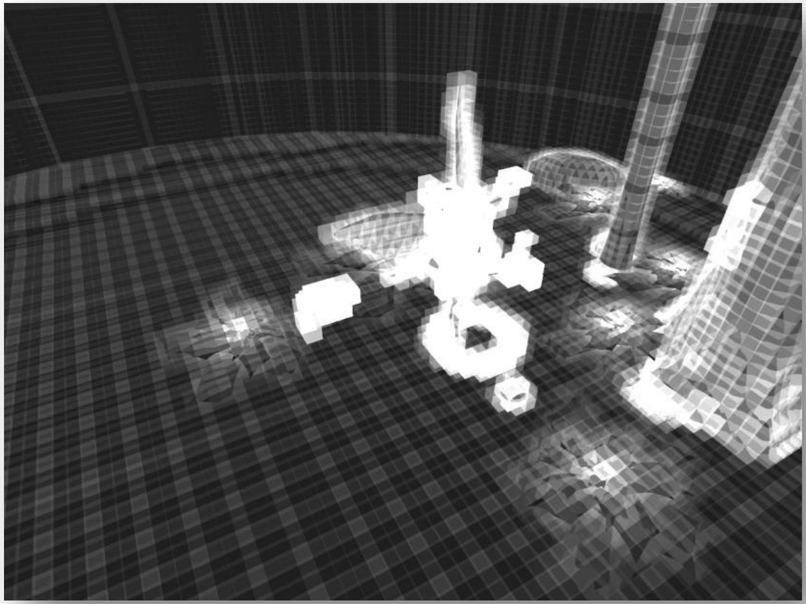
Uniform Grid Traversal

- **Grids are cheap to traverse**
  - E.g. 3D-DDA or modified Bresenham algorithm (see later)
  - Step through the structure cell by cell
  - Intersect with primitives inside non-empty cells
- **Mailboxing**
  - Single primitive can be referenced in many cells
  - Avoid multiple intersections
  - Keep track of intersection tests
    - Per-object cache of ray IDs
      - Problem with concurrent access
    - Per-ray cache of object IDs
      - Data local to a ray (better!)

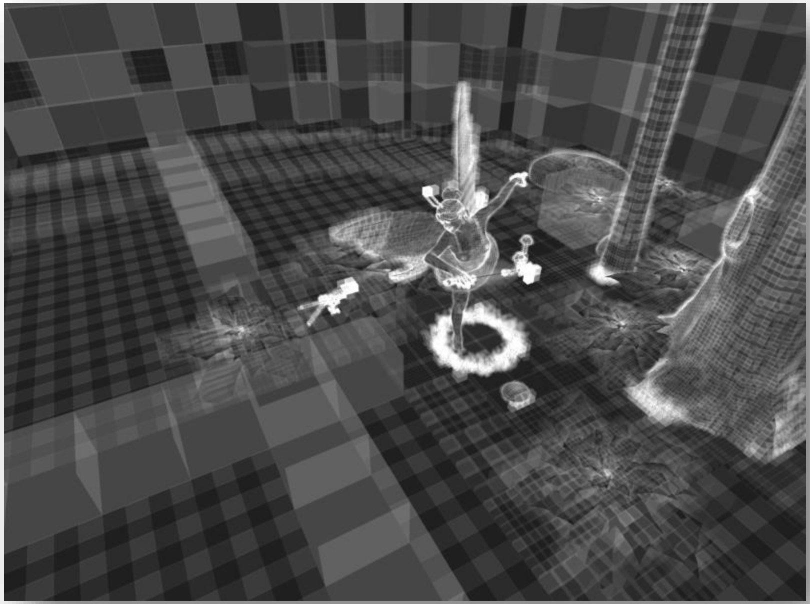


Nested Grids

- **Problem: „Teapot in a stadium"**
  - Uniform grids cannot adapt to local density of objects
- **Nested Grids**
  - Hierarchy of uniform grids: Each cell is itself a grid
  - Fast algorithms for building & traversal (Kalojanov et al. ´09,´11)



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

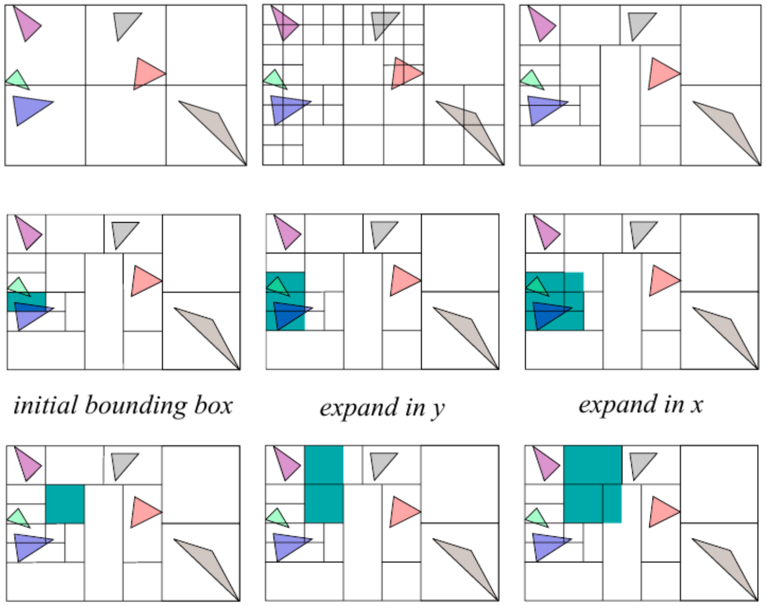


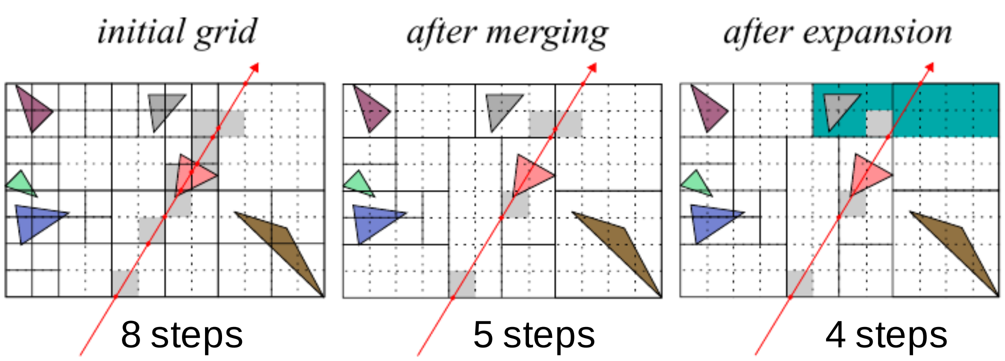
Same for two-level grid

Irregular Grids

- **Irregular grids can accel traversal [Perard-Gayot´17]**
  - Build (hierarchical) base grid (power of 2, adapts to scene)
    - Base grid defines minimum resolution for computation
  - Neighboring cells can be *merged* (eagerly)
    - As long as no change in set of primitives
  - Can also *expand* cells (for exit operations)
    - As long as neighbors contain only subset of cells primitives
    - Allows for making larger steps
  - Approach needs more memory

Construction (merge & expand)

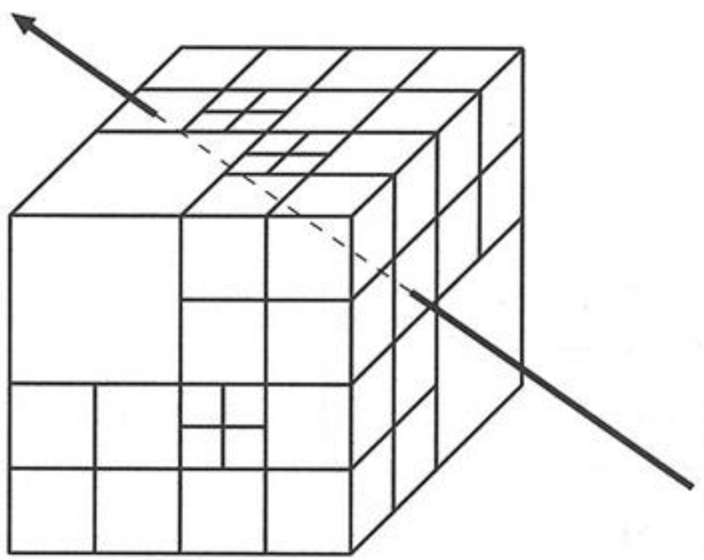


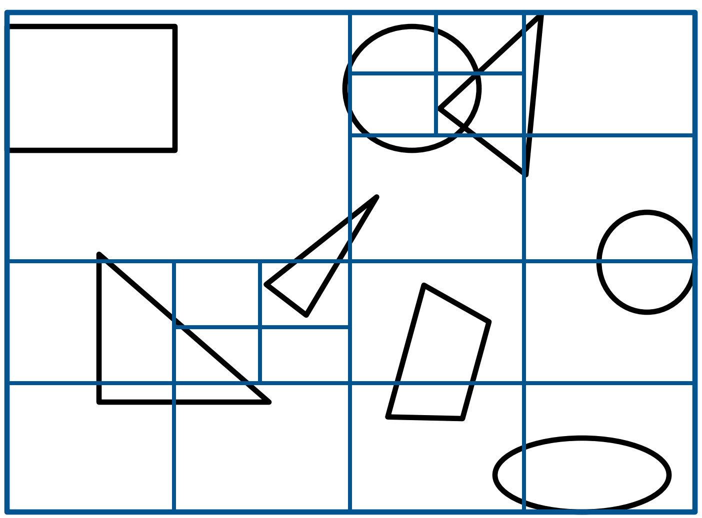


Traversal (simplified, finest level: 12 steps)

Octrees and Quadtrees

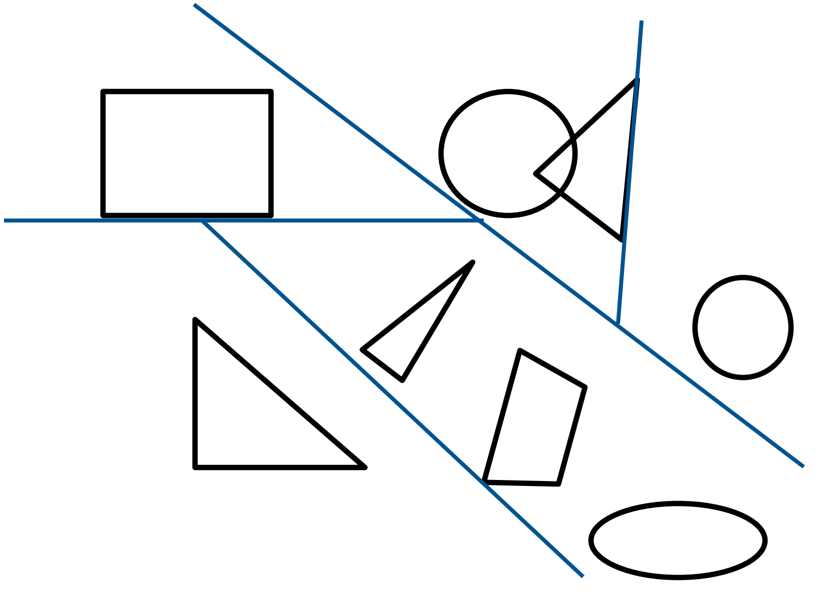
- **Octree**
  - Hierarchical space partitioning ("simplest hierarchical grid")
  - Each inner node contains 8 equally sized voxels (2 x 2 x 2 grid)
- **Quadtree**
  - 2D "octree"
- **Adaptive subdivision**
  - Adjust depth to local scene complexity





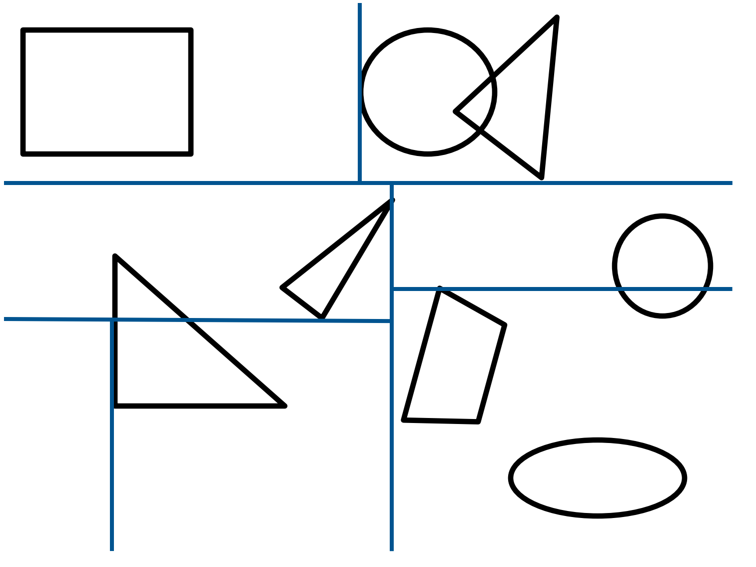
BSP Trees

- **Definition**
  - Binary Space Partition Tree (BSP)
  - Recursively split space with planes
    - Arbitrary split positions
    - Arbitrary orientations
- **Used for visibility computation**
  - E.g. in games (Doom!)
  - Enumerating objects in back to front order

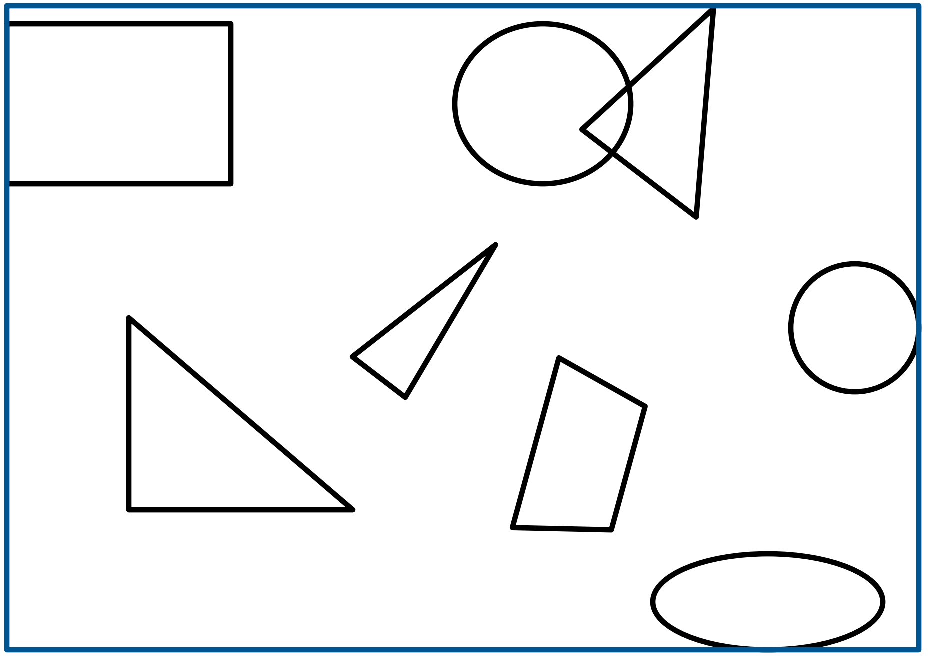


kD-Trees

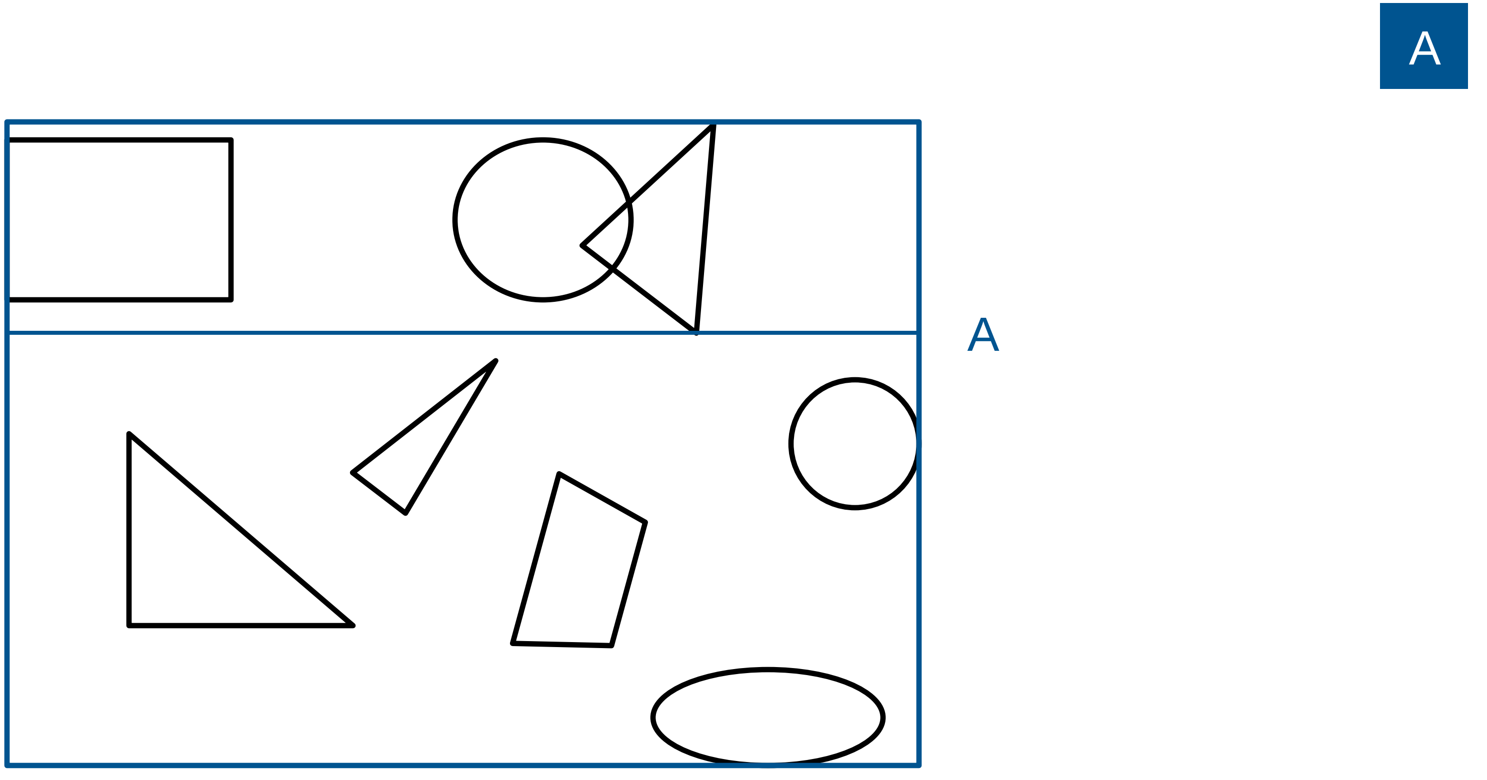
- **Definition**
  - **Axis-Aligned** Binary Space Partition Tree
  - Recursively split space with axis-aligned planes
    - Arbitrary split positions
    - Greatly simplifies/accelerates computations



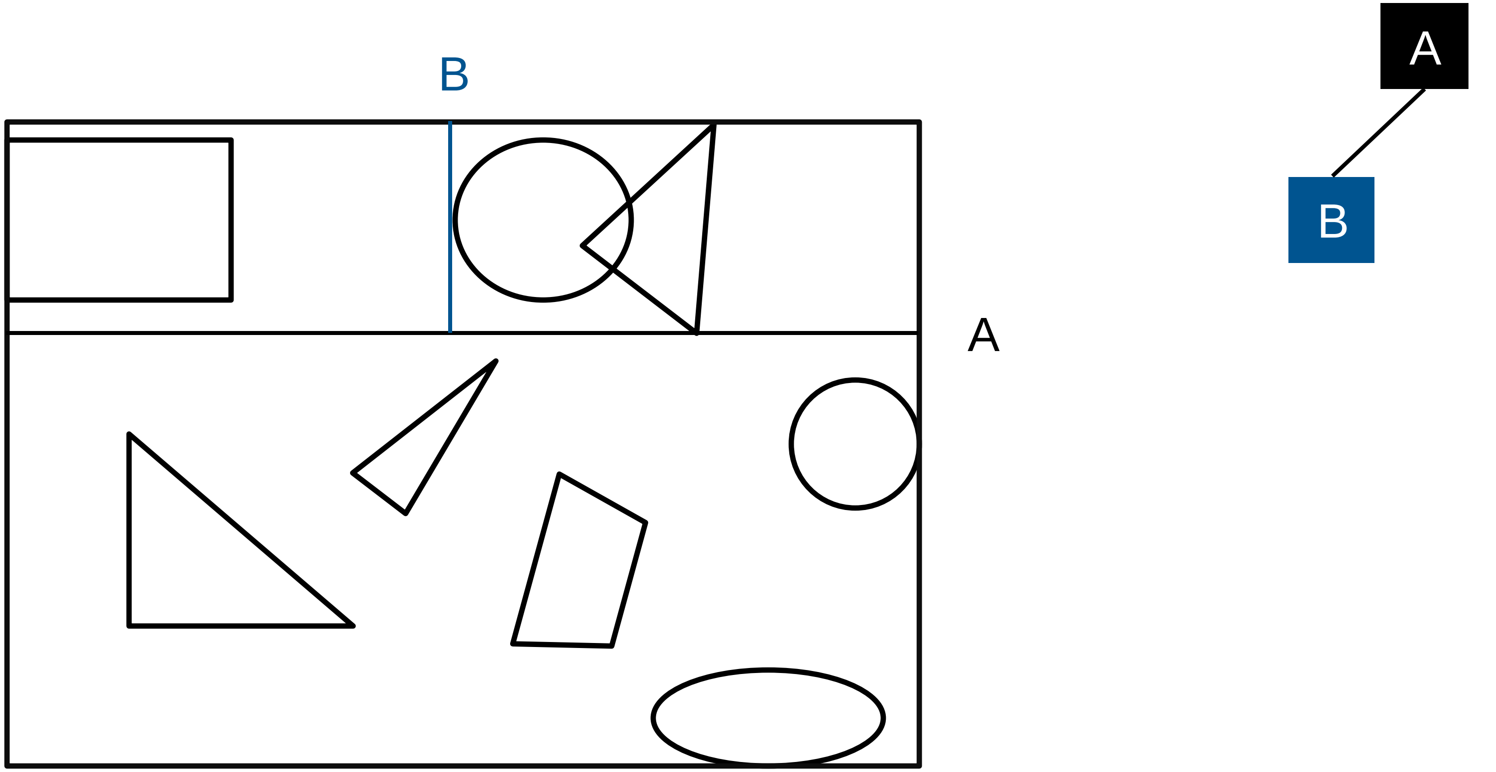
kD-Tree Example (1)



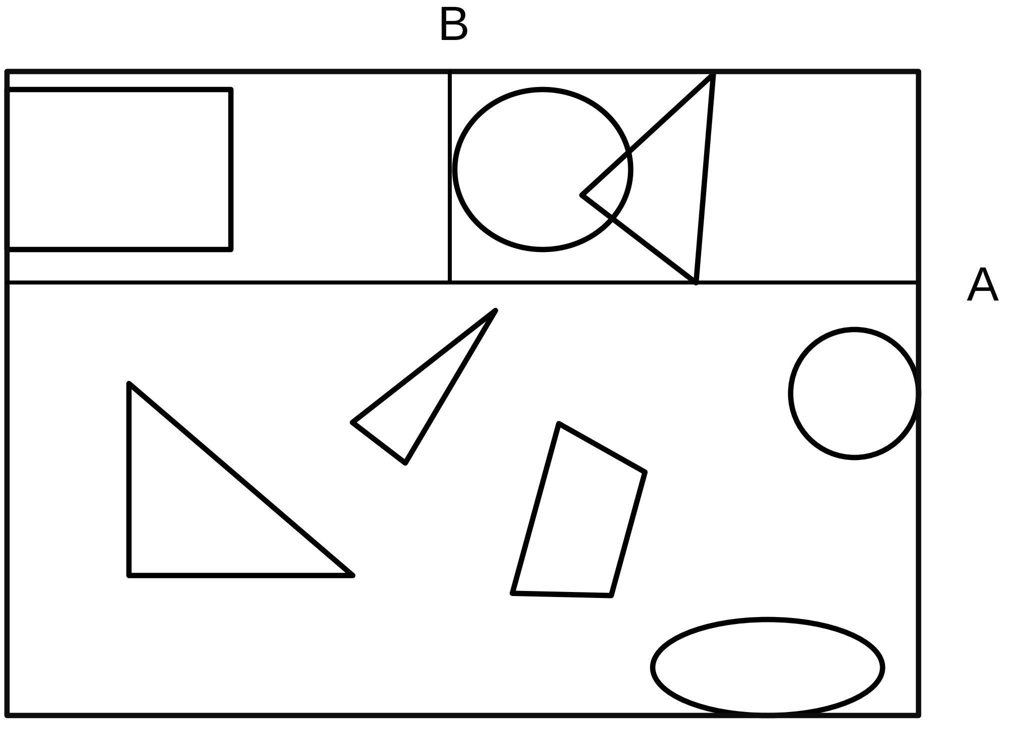
kD-Tree Example (2)

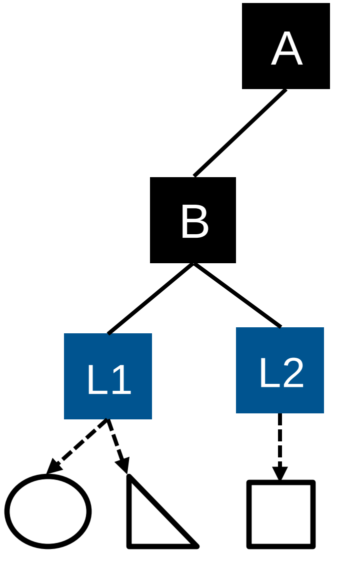


kD-Tree Example (3)

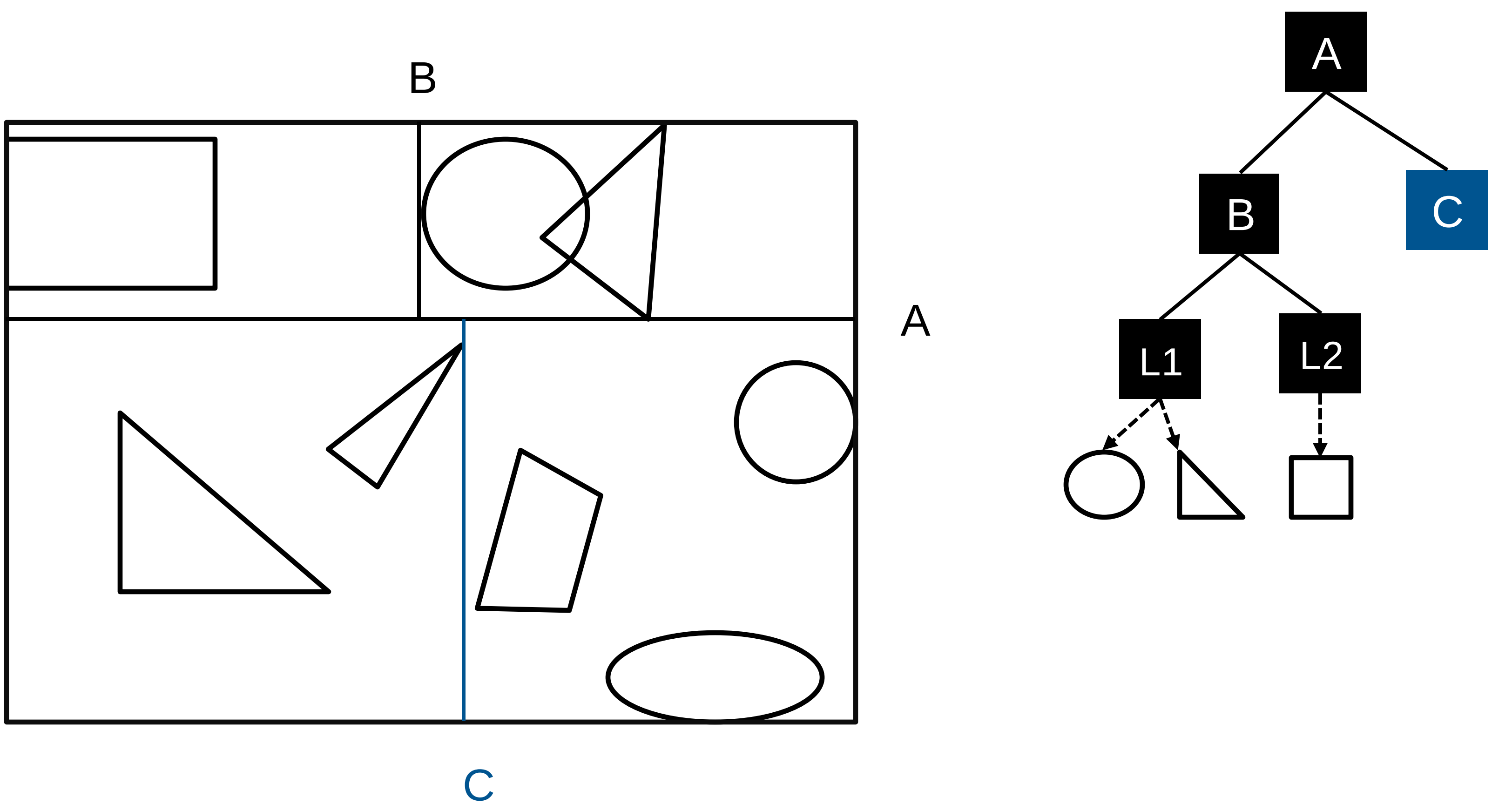


kD-Tree Example (4)

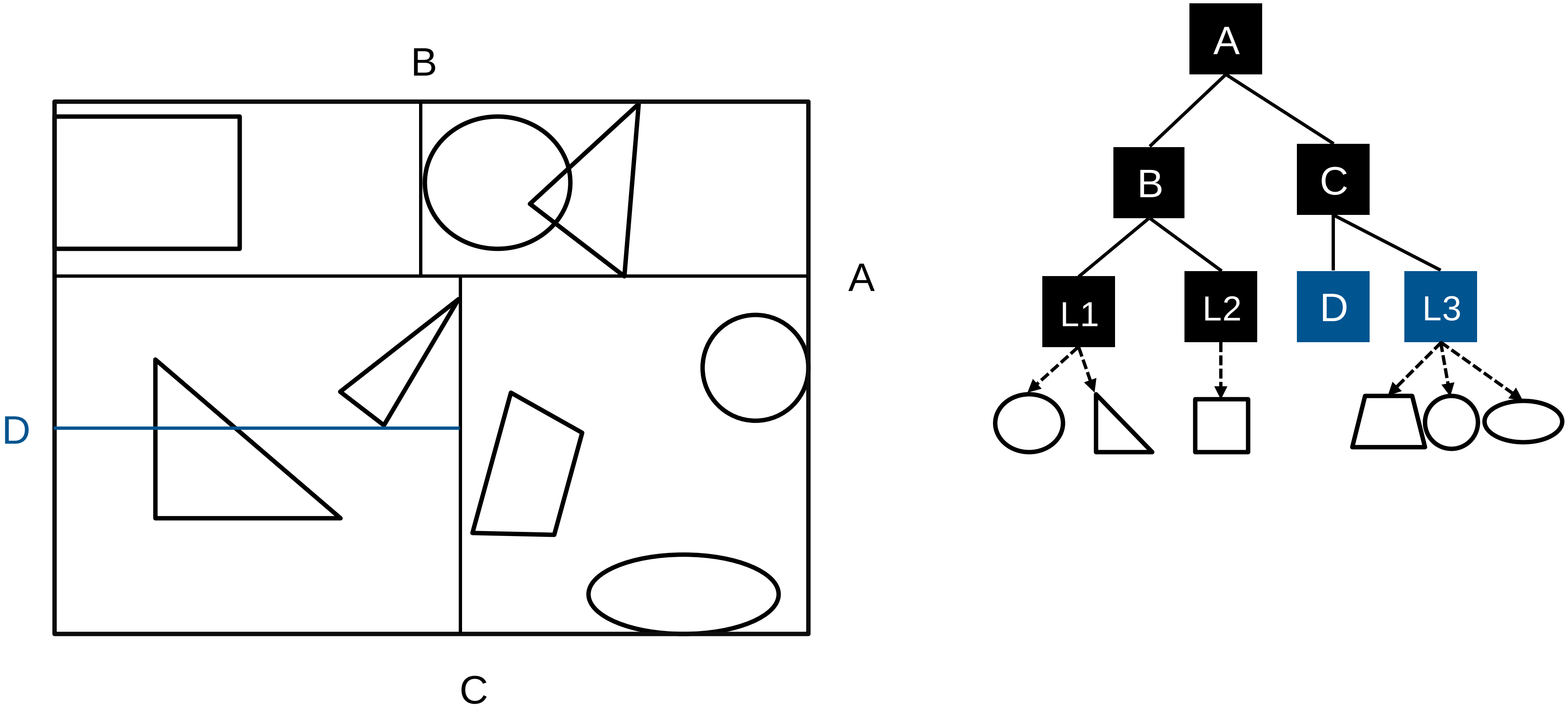




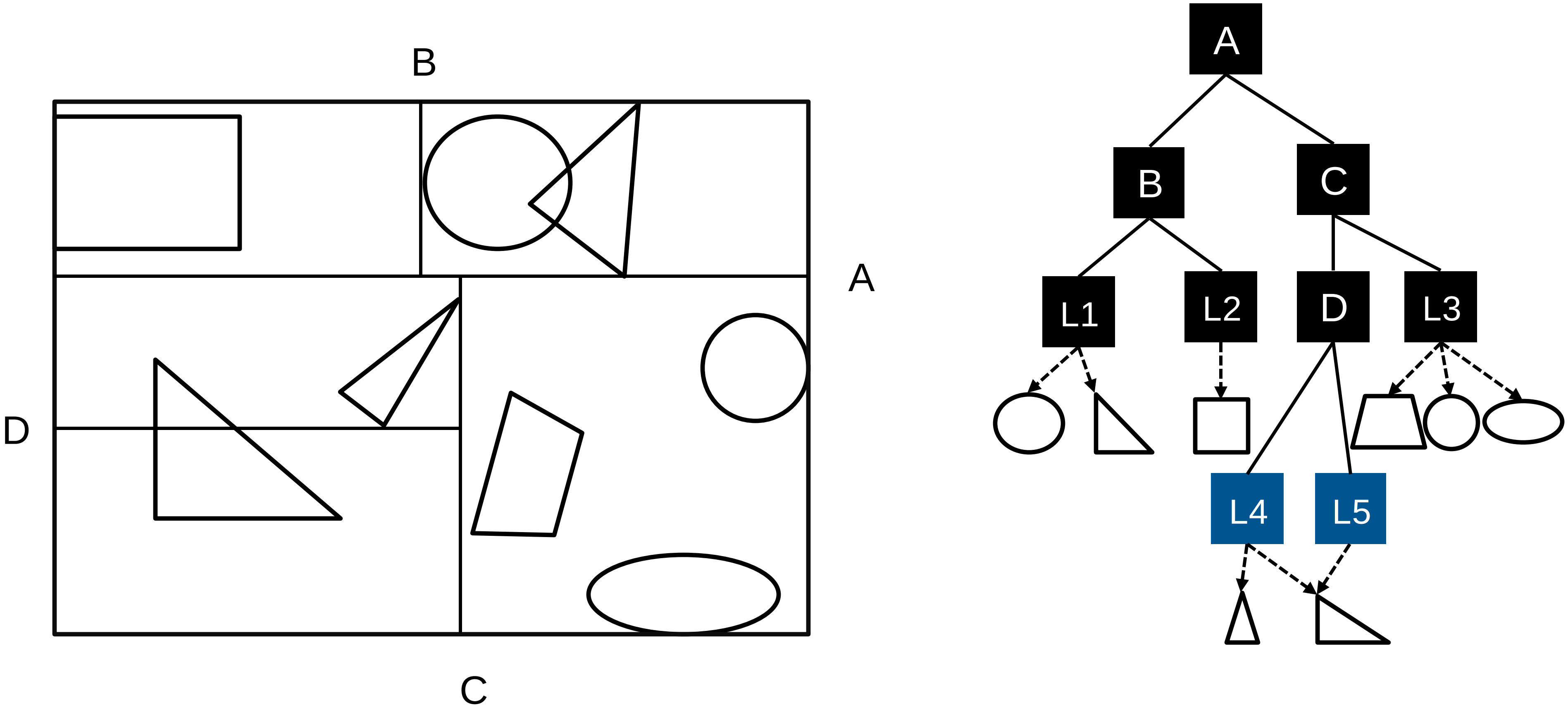
kD-Tree Example (5)



kD-Tree Example (6)



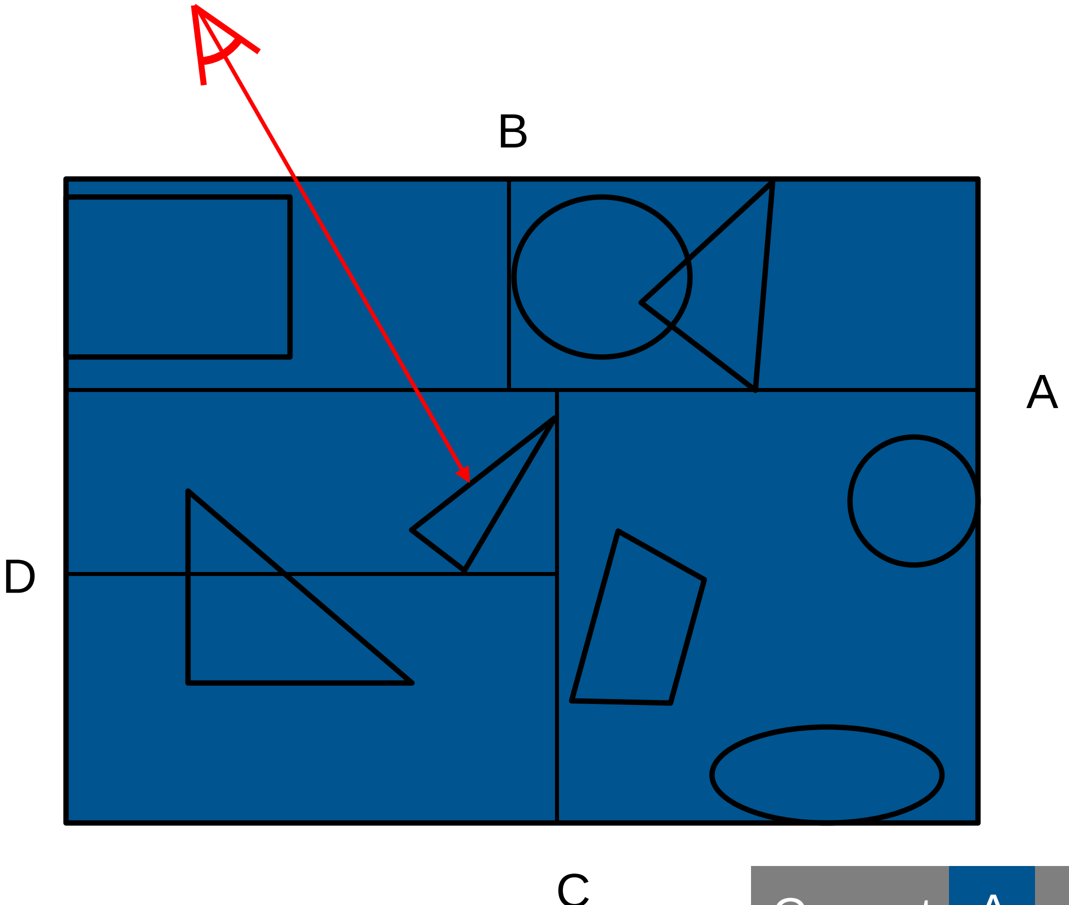
kD-Tree Example (7)

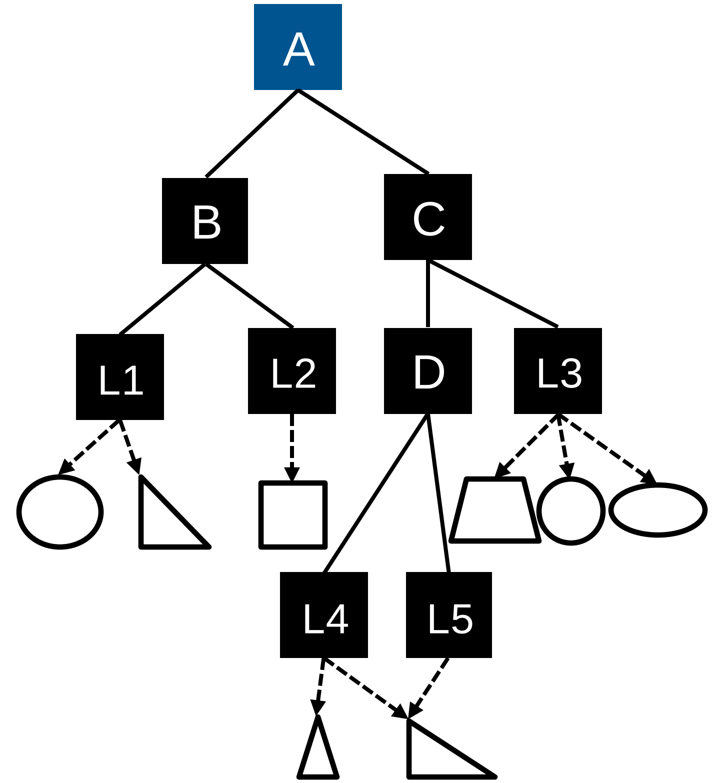


kD-Tree Traversal

- **“Front-to-back” traversal**
  - Traverse child nodes in order along rays
- **Termination criterion**
  - Traversal can be terminated as soon as surface intersection is found in the current node
- **Maintain stack of sub-trees still to traverse**
  - More efficient than recursive function calls
  - Algorithms with no or limited stacks are also available (for GPUs)

kD-Tree Traversal (1)

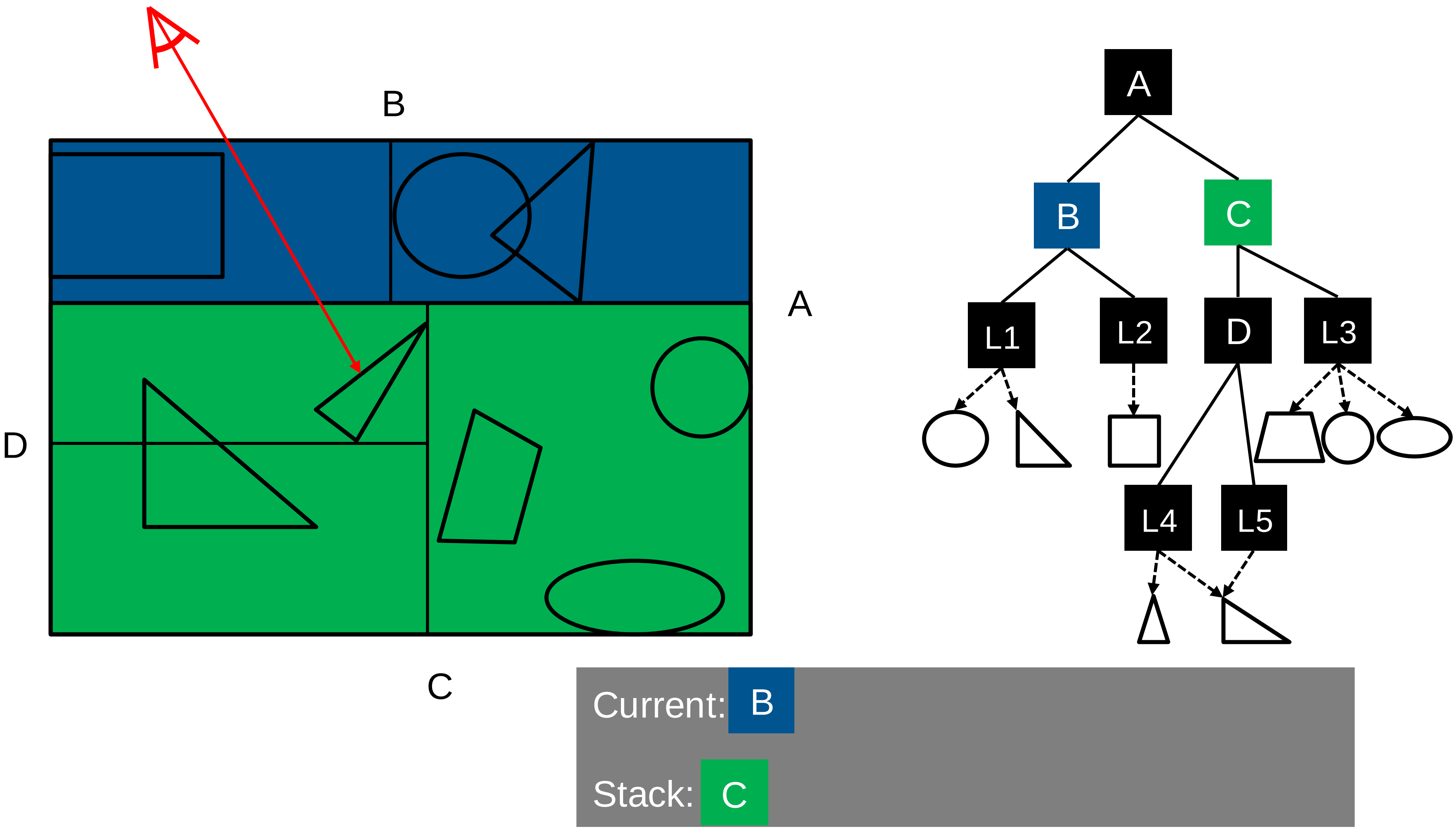




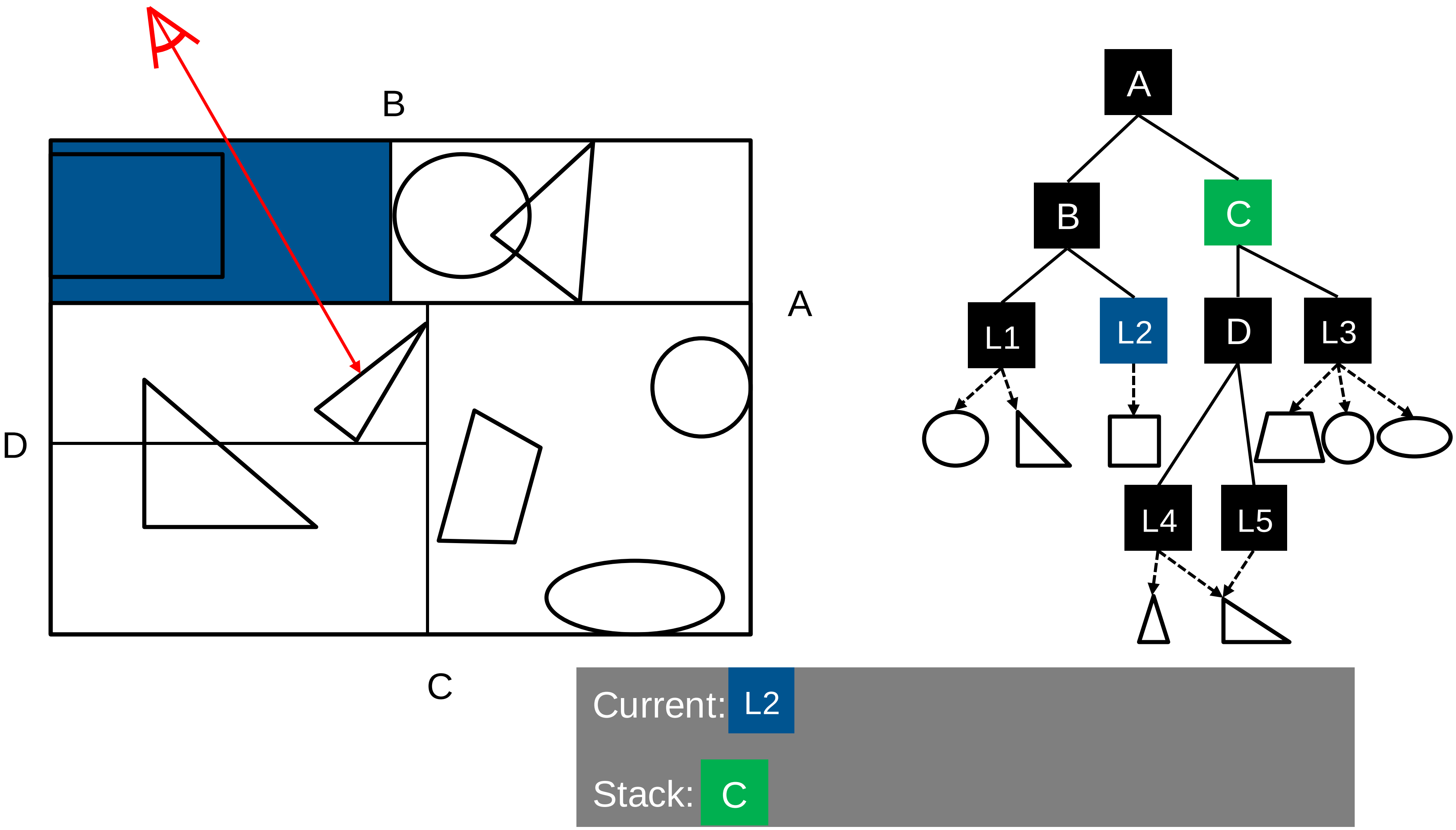
Current: A

Stack:

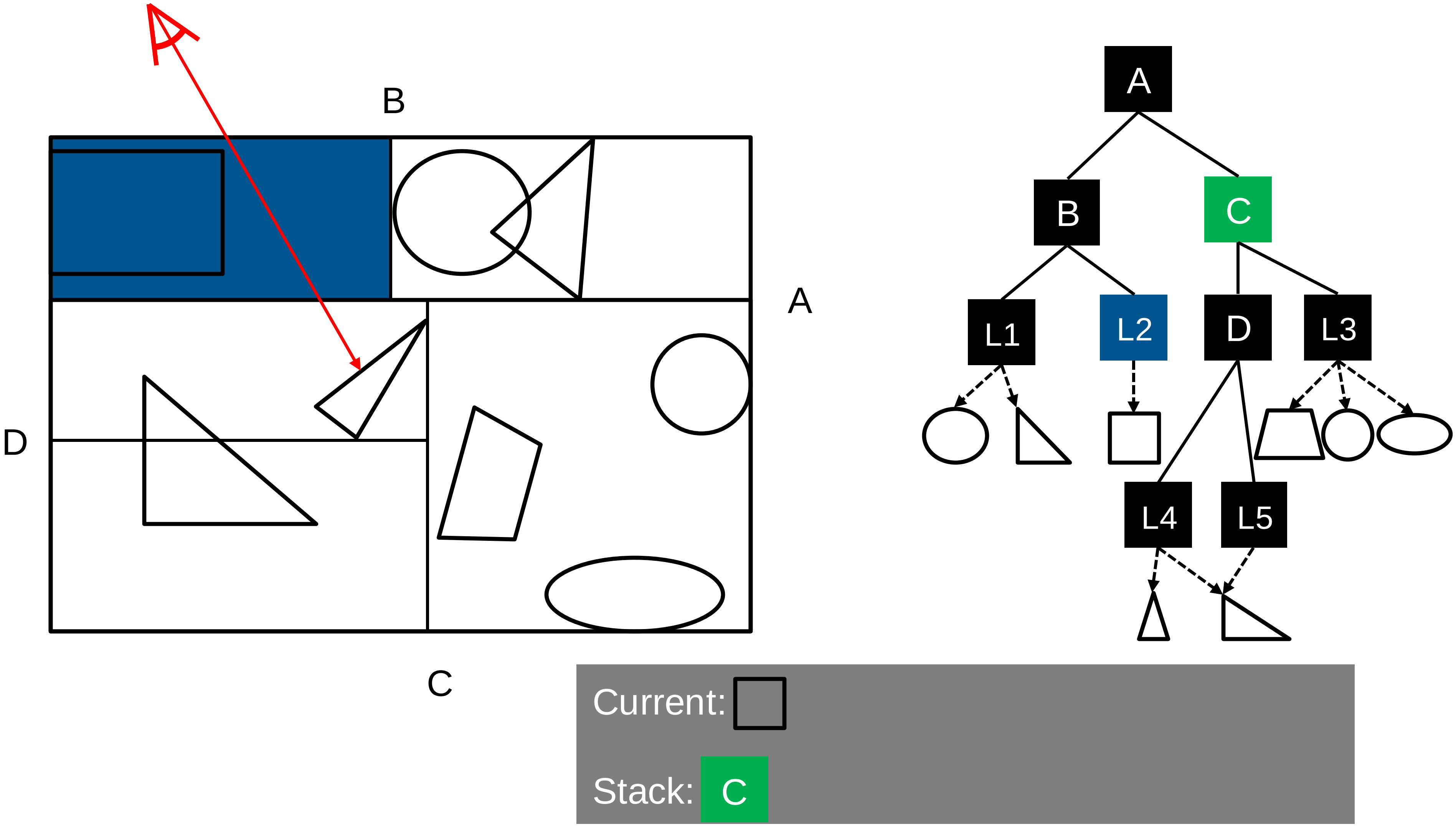
kD-Tree Traversal (2)



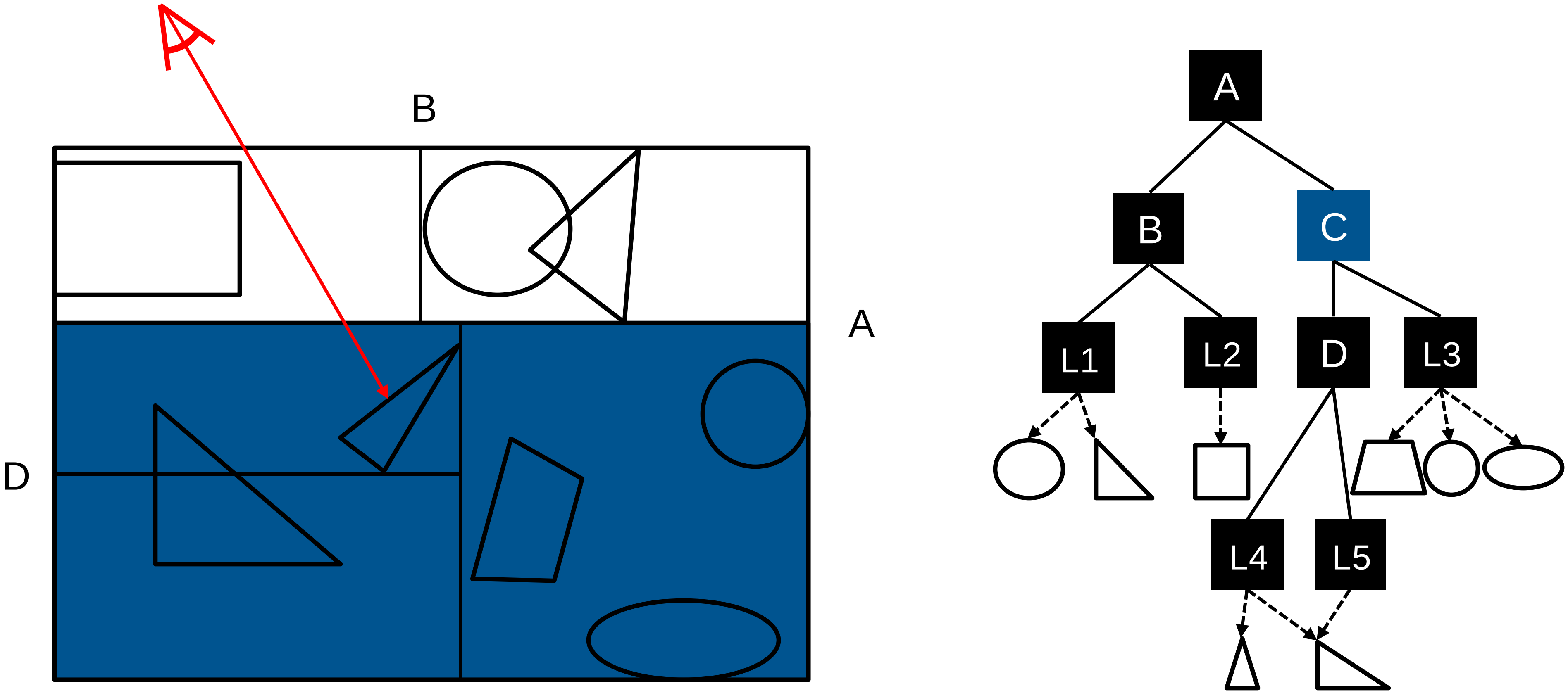
kD-Tree Traversal (3)



kD-Tree Traversal (4)



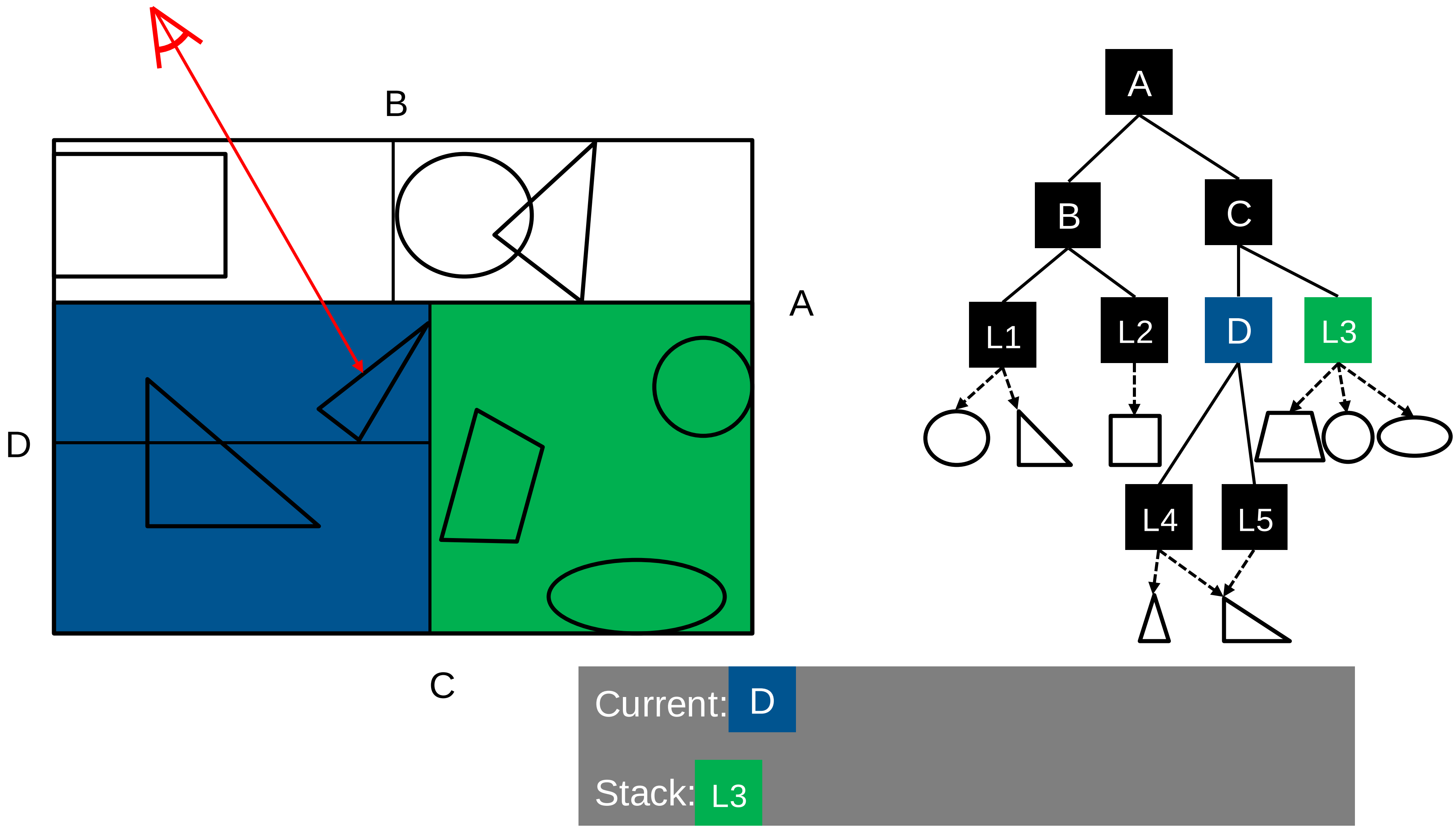
kD-Tree Traversal (5)



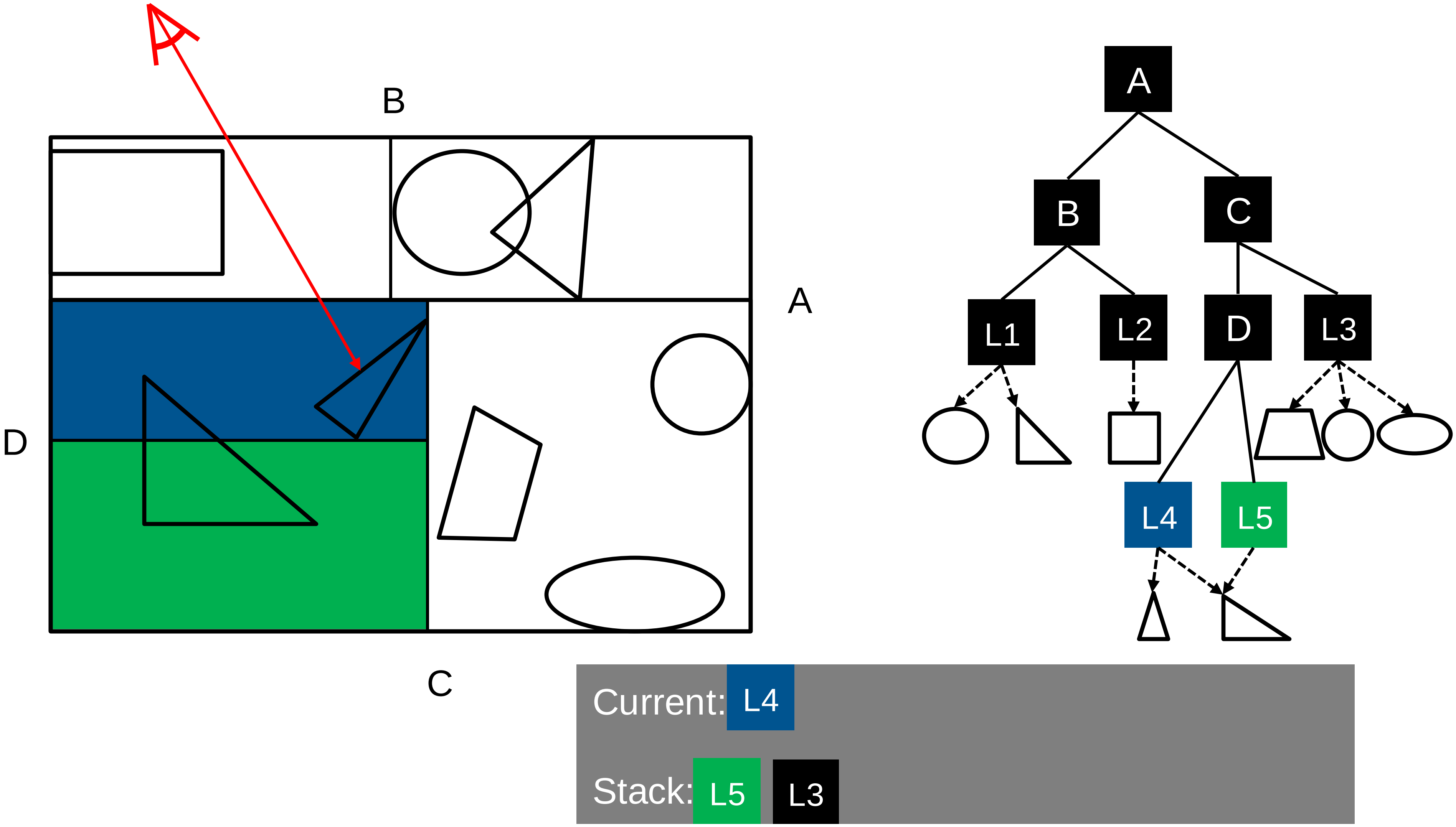
Current: C

Stack:

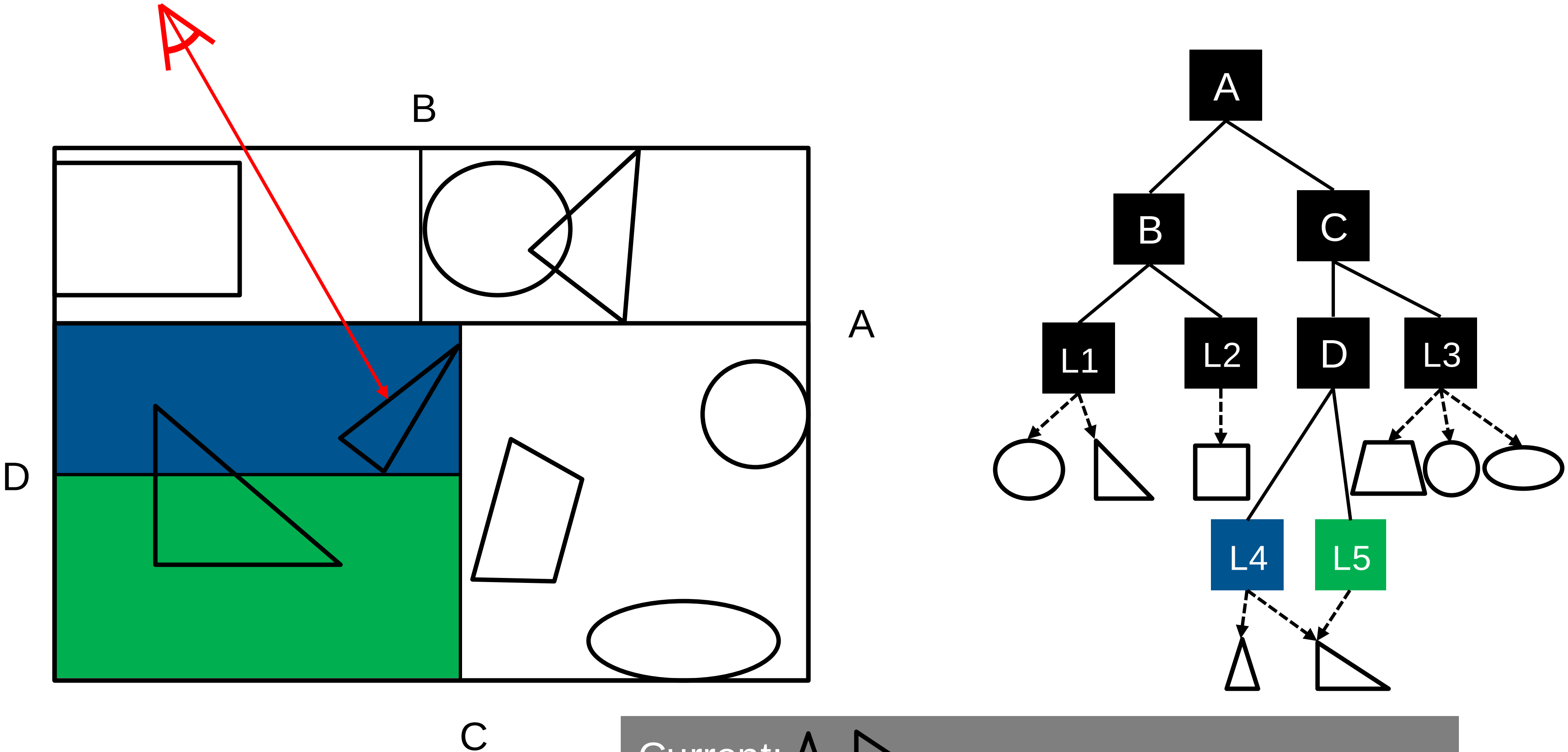
kD-Tree Traversal (6)



kD-Tree Traversal (7)



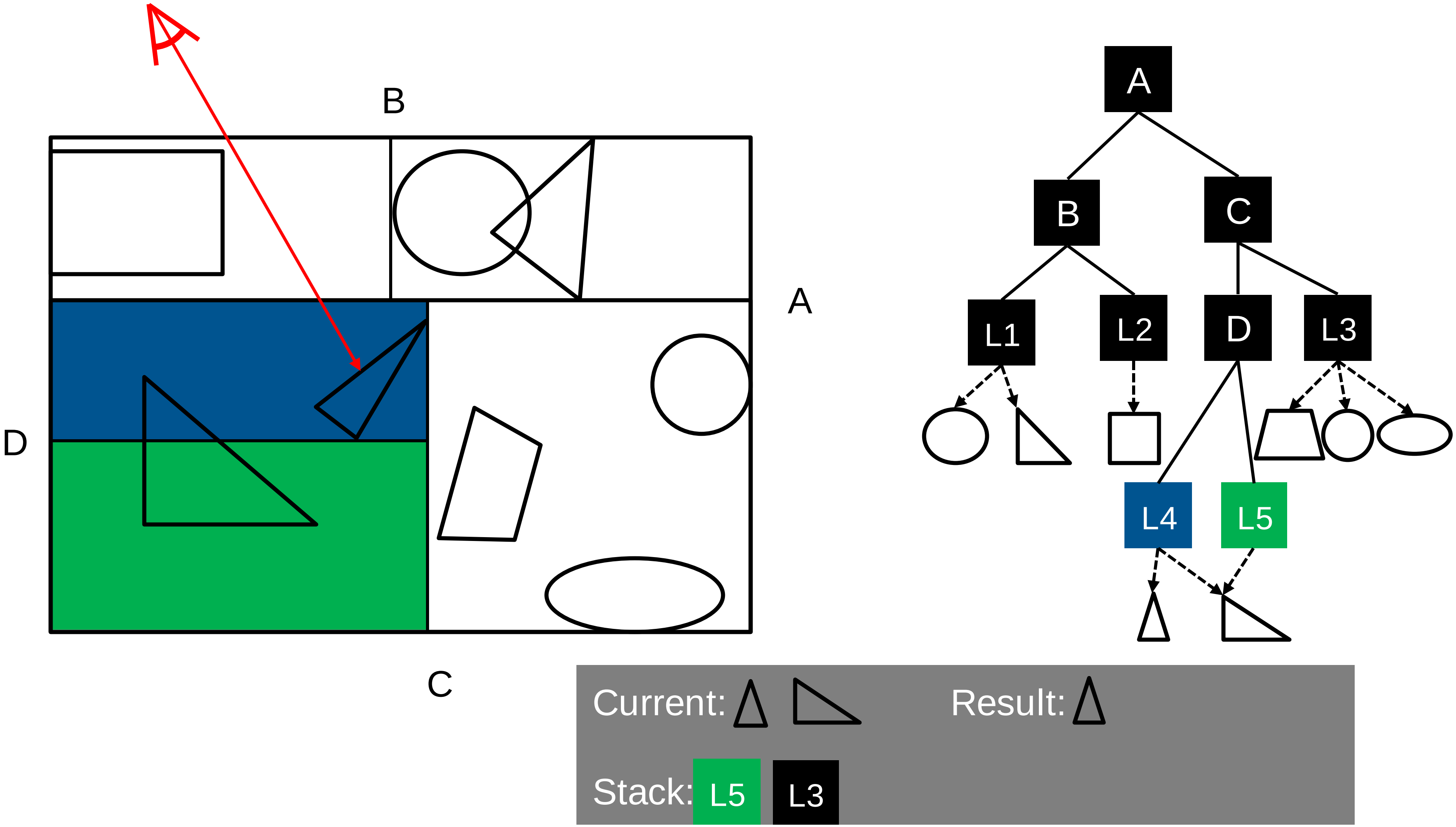
kD-Tree Traversal (8)



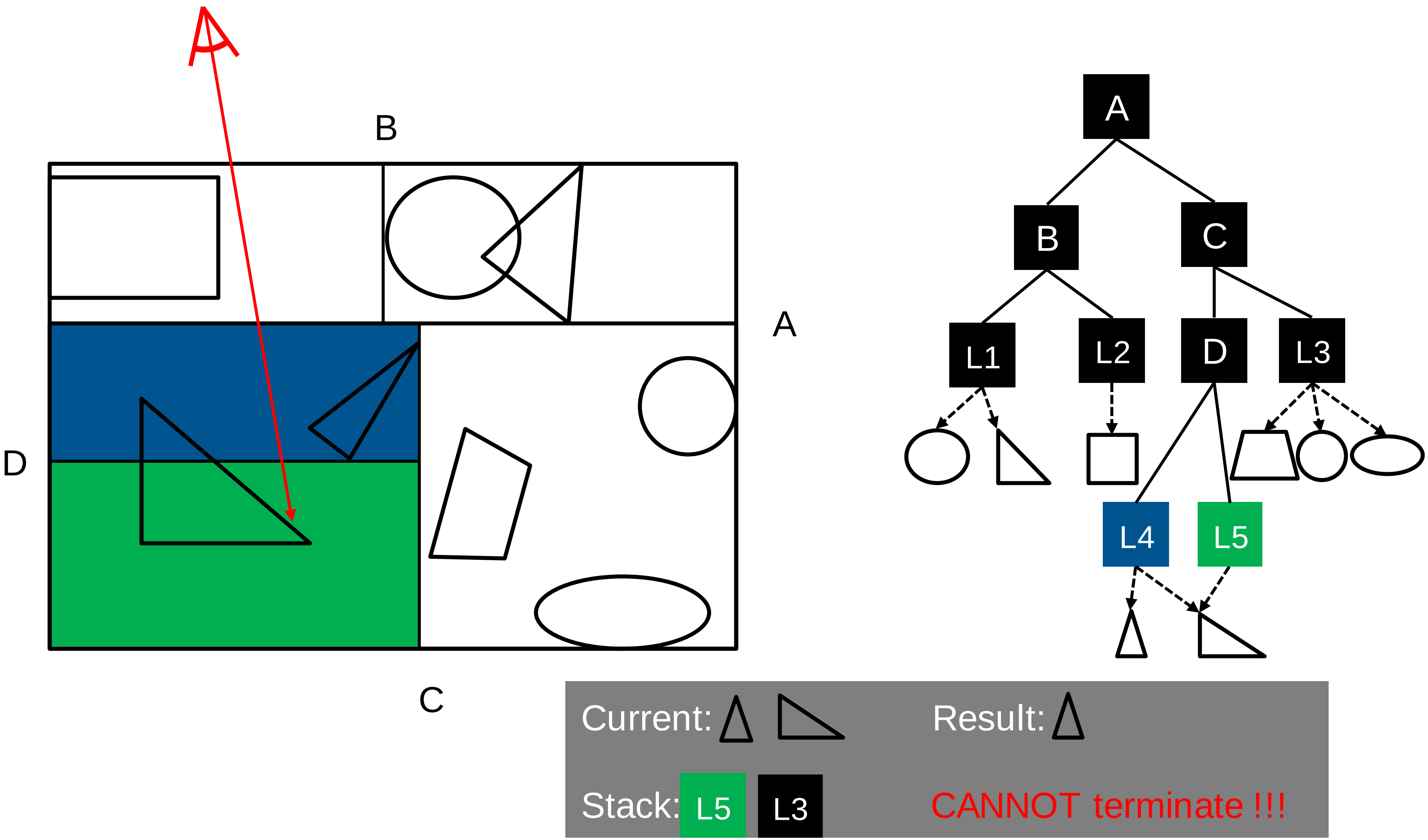
Current:

Stack: L5 L3

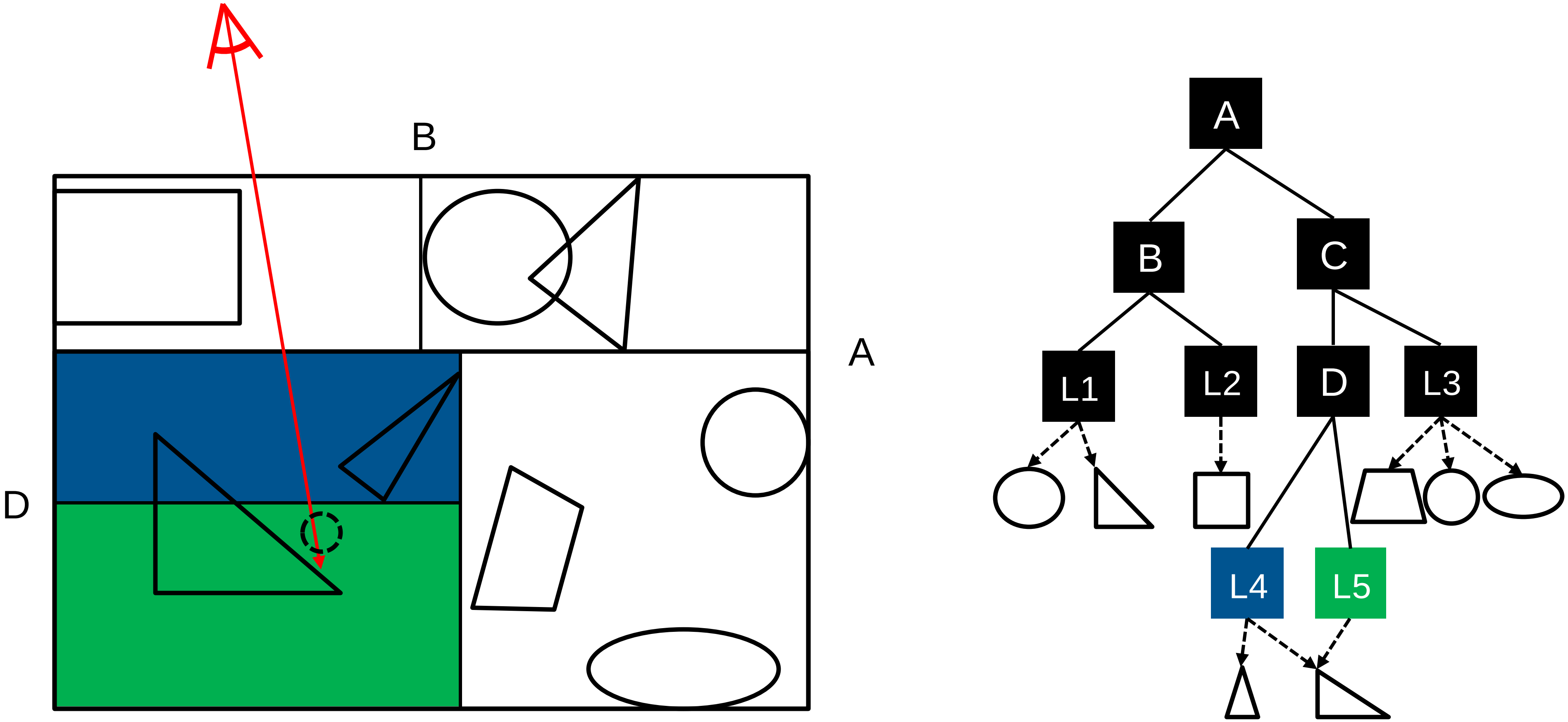
kD-Tree Traversal (9)

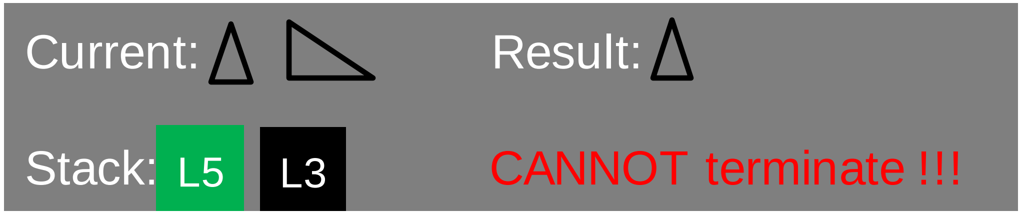


kD-Tree Traversal (10)



kD-Tree Traversal (11)





kD-Tree Properties

- **kD-Trees**
  - Split space instead of sets of objects
  - Split into disjoint, fully covering regions
- **Adaptive**
  - Can handle the “Teapot in a Stadium” well
- **Compact representation**
  - Relatively little memory overhead per node
  - Node stores:
    - Split location (1D), child pointer (to array with both children), axis-flag (often merged into pointer)
    - Can be compactly stored in 8 bytes
  - But replication of objects in (possibly) many nodes
    - Can greatly increase memory usage
- **Cheap Traversal**
  - One subtraction, multiplication, decision, and fetch
  - But many more cycles due to data dependencies
    - Latency can harm you!

Overview: kD-Trees Construction

- **Adaptive**
- **Compact**
- **Cheap traversal**

Exploit Advantages

- **Adaptive**
  - You have to build a good tree
- **Compact**
  - At least use the compact node representation (8-byte)
  - You can’t be fetching whole cache lines every time
- **Cheap traversal**
  - No sloppy inner loops! (one subtract, one multiply!)

Building kD-trees

- **Given:**
  - Axis-aligned bounding box (“cell”)
  - List of geometric primitives (triangles?) touching cell
- **Core operation:**
  - Pick an axis-aligned plane to split the cell into two parts
  - Sift geometry into two batches (possible some duplication)
  - Recurse

Building kD-trees

- **Given:**
  - Axis-aligned bounding box (“cell”)
  - List of geometric primitives (triangles?) touching cell
- **Core operation:**
  - Pick an axis-aligned plane to split the cell into two parts
  - Sift geometry into two batches (some redundancy)
  - Recurse
  - Termination criteria!

“Intuitive” kD-Tree Building

- **Split Axis**
  - Round-robin; largest extent
- **Split Location**
  - Middle of extent; median of geometry (balanced tree)
- **Termination**
  - Target # of primitives, limited tree depth

“Intuitive” kD-Tree Building

- **Split Axis**
  - Round-robin; largest extent
- **Split Location**
  - Middle of extent; median of geometry (balanced tree)
- **Termination**
  - Target # of primitives, limited tree depth
- **All of these techniques are NOT very clever**

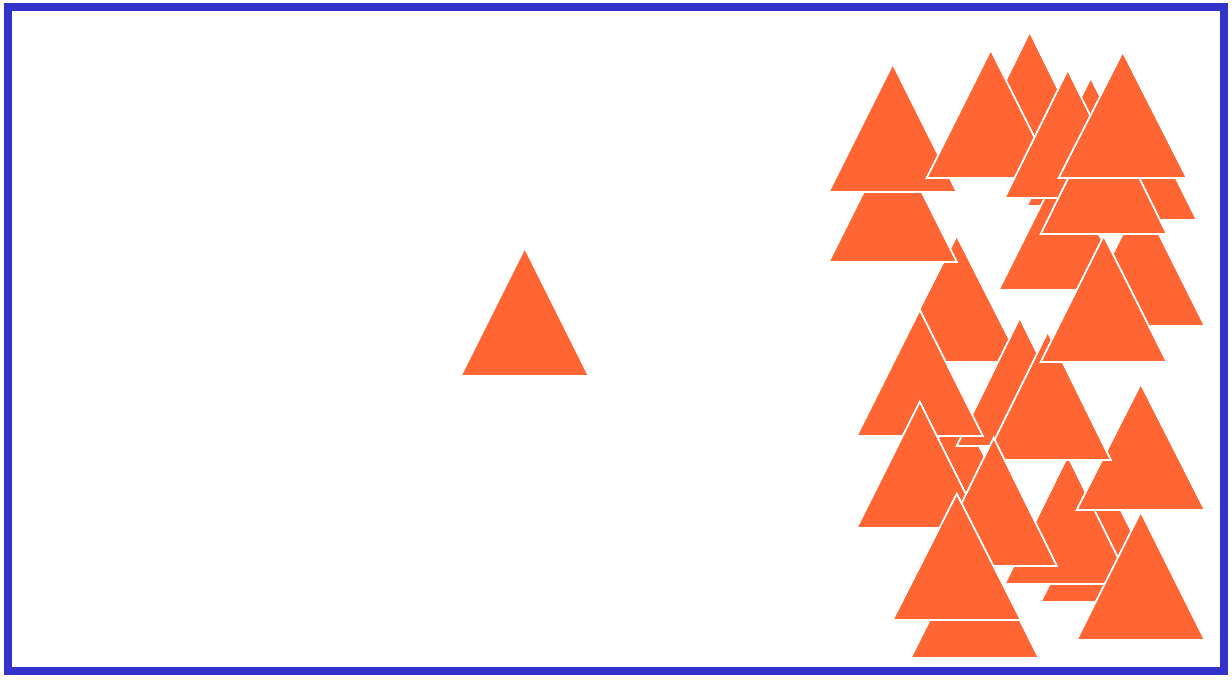
Building good kD-trees

- **What split do we really want?**
  - Clever Idea: The one that makes ray tracing cheap
  - Write down an expression for the cost and minimize it
    - ➔ *Cost Optimization*
- **What is the cost of tracing a ray through a cell?**
  - Surface Area Heuristic (SAH)

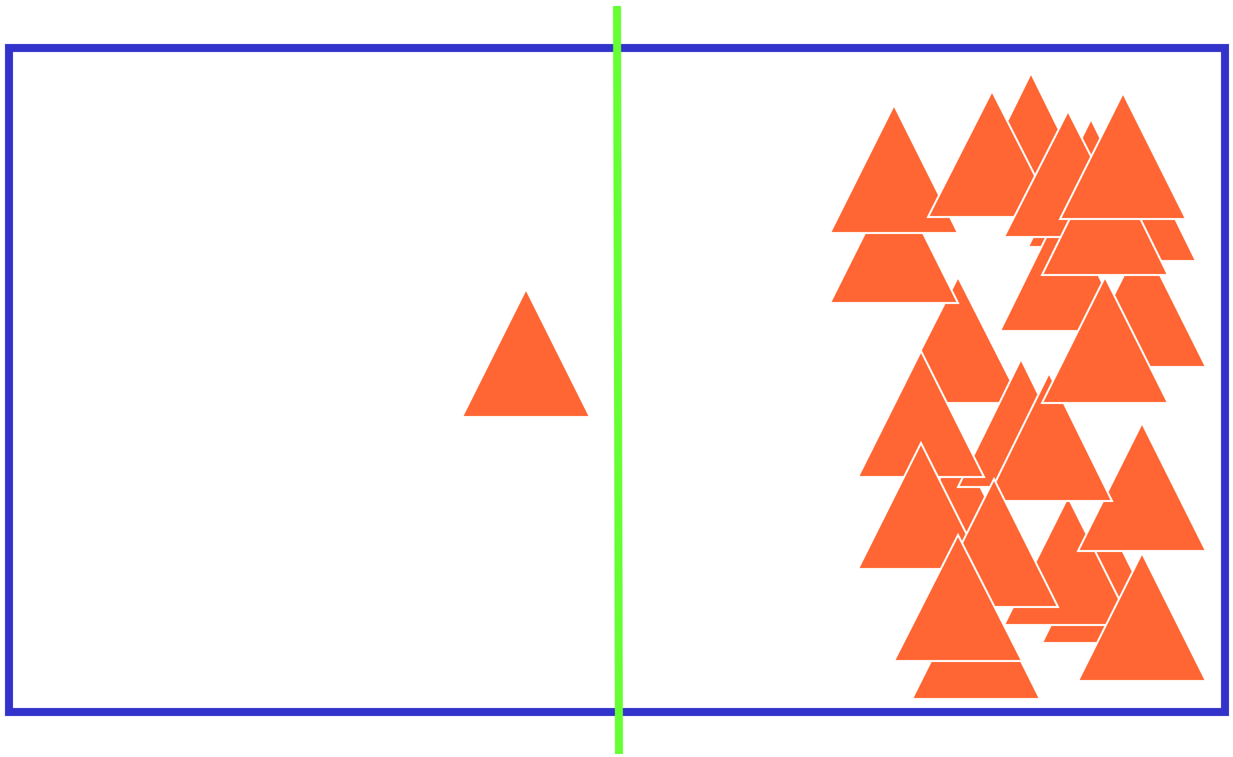
Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

- Cost of traversal of the inner node itself, plus
- Relative probability of hitting one child, times
- Cost of intersecting with that child
- Same for other child

Splitting with Cost in Mind

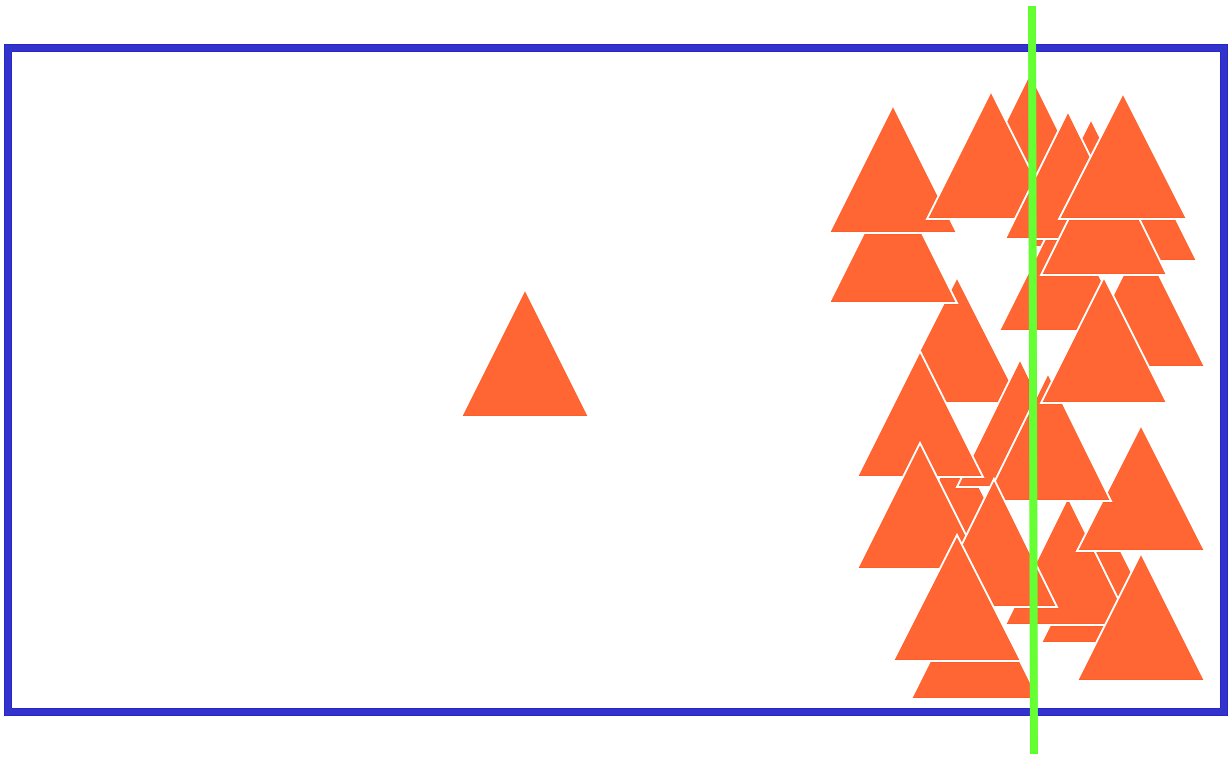


Split in the middle



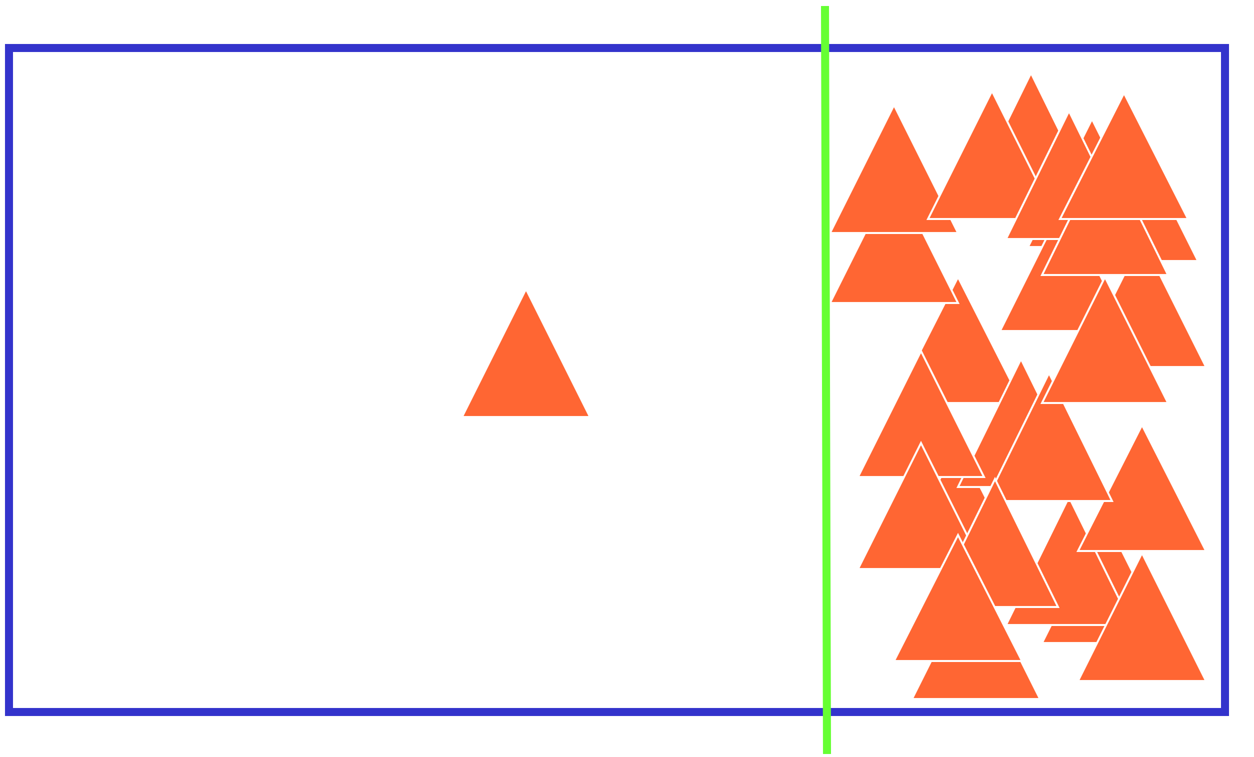
- **Makes the L & R probabilities equal**
- **Pays no attention to the L & R costs**

Split at the Median



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

Cost-Optimized Split



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

Building good kD-trees

- **Need the probabilities**
  - Turns out to be proportional to *surface area* (SA)
    - Sum of area of all sides
    - True for random rays
    - Proof: Left as an exercise for the reader
  - *Not* the volume
- **Need the child cell costs**
  - Simple *triangle count* works great (very rough approx.)
  - Many attempts to improve this did not work out

Cost(c) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

= C_trav + SA(L)/SA(c) * TriCount(L) + SA(R)/SA(c) * TriCount(R)

Termination Criteria

- **When should we stop splitting?**
  - Another clever idea: When splitting does not help any more.
  - Use the cost estimates in your termination criteria
- **Threshold of cost improvement**
  - But stretch decision over multiple levels, to avoid local minima
- **Threshold of cell size**
  - Absolute (!) probability so small there is no point in going on

Building good kD-trees

- **Basic build algorithm**
  - Pick an axis, or optimize across all three
  - Build a set of candidate split locations
    - Based on BBox of triangles (in/out events)
      - One can show that SAH cannot have minima unless #triangles changes
    - Or predefined locations (fixed number of bins across bbox axis)
  - Sort the triangle events or bin them
  - Walk through candidates to find minimum cost split
- **Characteristics of the tree you are looking for**
  - Deep and thin
  - Typical depth of 50-100,
  - About 2 triangles per leaf,
  - Big empty cells

Building kD-trees quickly

- **Very important to build good trees first**
  - Otherwise you have no basis for comparison
- **Don’t give up cost optimization!**
  - Use the math, Luke…
- **Luckily, lots of flexibility…**
  - Axis picking (“hack” pick vs. full optimization)
  - Candidate picking (bboxes, exact; binning, sorting)
  - Termination criteria (“knob” controlling tradeoff)

Building kD-trees quickly

- **Remember, profile first! Where's the time going?**
  - Split personality
    - Memory traffic all at the top (NO cache misses at bottom)
  - Sifting through bajillion triangles to pick one split (!)
  - Hierarchical building?
    - Computation mostly at the bottom
  - Lots of leaves, need more exact candidate info
  - Lazy building?
    - Change criteria during the build?

Fast Ray Tracing w/ kD-Trees

- **Adaptive**
  - Build a cost-optimized kD-tree w/ the surface area heuristic
- **Compact**
- **Cheap traversal**

What's in a node?

- **A kD-tree internal node needs:**
  - Am I a leaf?
  - Split axis
  - Split location
  - Pointers to children

Compact (8-byte) Nodes

- **kD-Tree node can be packed into 8 bytes**
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - Only one 32-bit pointer
  - Leaf flag + Split axis
    - 2 bits

Compact (8-byte) Nodes

- **kD-Tree node can be packed into 8 bytes**
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - Only one 32-bit pointer
  - Leaf flag + Split axis
    - 2 bits
- **So close! Sweep those 2 bits under the rug…**
  - Encode bits in lowest 2 bits of pointer
  - Bits are not used as structure is multiple of 8, anyway

No Bounding Box!

- **kD-Tree node corresponds to an AABB**
- **Does not mean it has to *contain* one**
  - Would be 24 bytes: 4X explosion (!)

Memory Layout

- **Cache lines are much bigger than 8 bytes!**
  - Advantage of compactness lost with poor layout
- **Pretty easy to do something reasonable**
  - Building depth first, watching memory allocator

Other Data

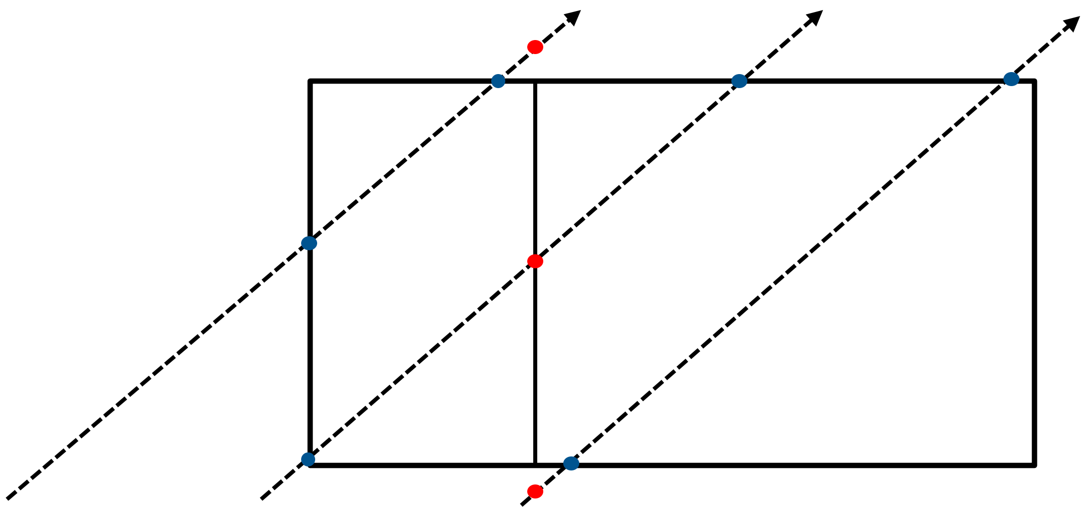
- **Memory should be separated by rate of access**
  - Frames
  - << Pixels
  - << Samples [Ray Trees]
  - << Rays [Shading (not quite)]
  - << Triangle intersections
  - << Tree traversal steps
- **Example:**
  - Store pre-processed triangle data
  - Store shading info of triangle separately
    - Object-orientation comes to bite you!
  - …

Fast Ray Tracing w/ kD-Trees

- **Adaptive**
  - Build a cost-optimized kD-tree w/ the surface area heuristic
- **Compact**
  - Use an 8-byte node
  - Lay out your memory in a cache-friendly way
- **Cheap traversal**

kD-Tree Traversal Operation

- **Implicitly maintain the bounds of the current node**
- **Store only necessary info on the stack**
  - Entry and exit distance to node (t_near and t_far)
- **Three cases**
  - t_split > t_far: Go only to near node
  - t_near < t_split < t_far Go to both (use stack)
  - t_split < t_near Go only to far node
- **Near and far depend on direction of ray!**



kD-Tree Traversal: Inner Loop

```
Given (node, t_near, t_far)
while ( ! node.isLeaf() )
{
    t_at_split = ( split_location - ray->origin[split_axis] ) * ray->inv_dir[split_axis]
    if  (t_split <= t_min)
            continue with (far child, t_split, t_far)     // hit either far child or none
    if (t_split >= t_max)
            continue with (near child, t_min, t_split)     // hit near child only
    // hit both children
    push (far child, t_split, t_max) onto stack
    continue with (near child, t_min, t_split)
}
```

Optimize Your Inner Loop

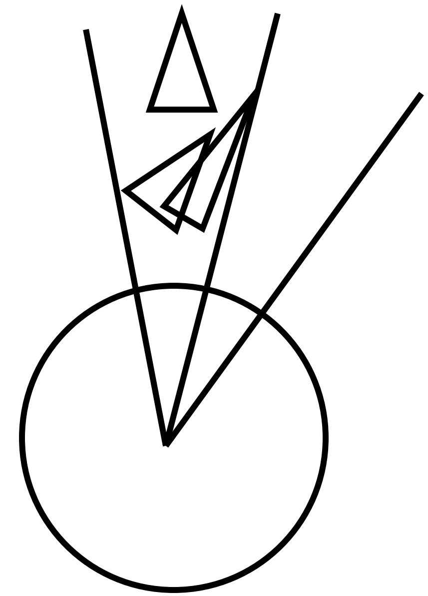
- **kD-Tree traversal is the most critical kernel**
  - It happens about a zillion times
  - It's tiny
  - Sloppy coding *will* show up
- **Optimize, Optimize, Optimize**
  - Remove recursion and minimize stack operations
  - Other standard tuning & tweaking

Can it go faster?

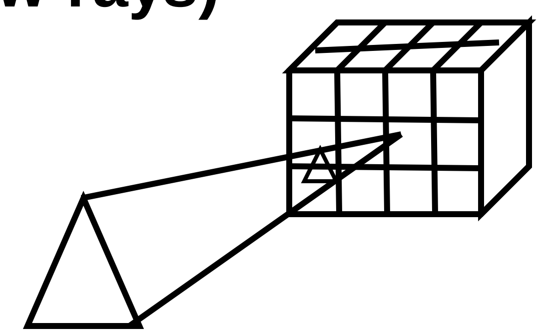
- **How do you make fast code go faster?**
- **Parallelize it!**
  - – Trace rays on multiple cores in parallel
    - Ray tracing is “embarrassingely parallel”
  - – Use SIMD instructions
    - Traverse many rays (packets), test with one BV (for BVHs)
    - Traverse one ray, but intersect with many BVs (needs wide BVH!)
    - Hybrid mix of both with adaptive switch
  - – Not covered here

Directional Partitioning

- **Applications**
  - Useful only for rays that start from a single point
    - Camera
    - Point light sources
  - Preprocessing of visibility
  - Requires scan conversion of geometry (see later)1
    - For each object locate where it is visible
    - Expensive and linear in # of objects
- **Generally not used for primary rays**

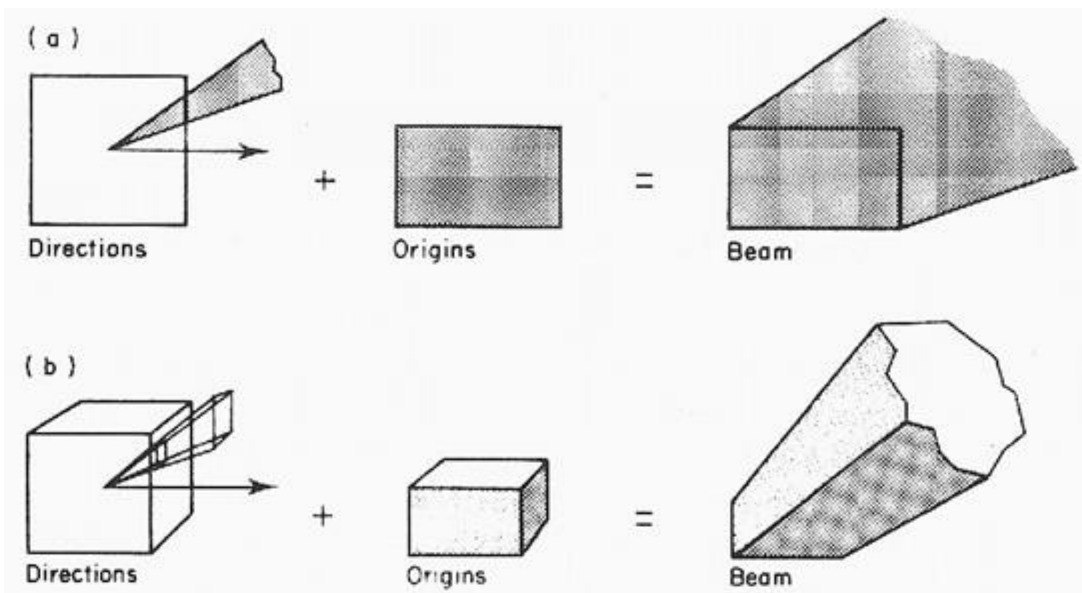


- **Variation: Light buffer (for shadow rays)**
  - Lazy and conservative evaluation
  - Store last found occluder in directional structure
  - Test entry first for next shadow test



Ray Classification

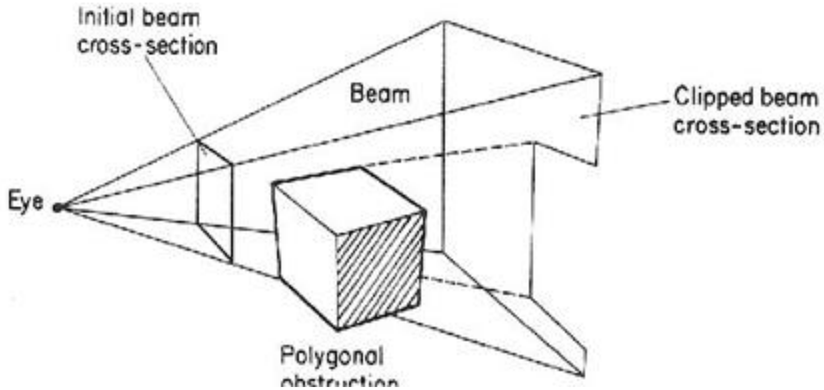
- **Partitioning of space and direction [Arvo & Kirk´87]**
  – Roughly pre-computes visibility for the entire scene
    - What is visible from each point in each direction?
  – Very costly preprocessing, cheap traversal
    - Improper trade-off between preprocessing and run-time
  – Memory hungry, even with lazy evaluation
  – Seldom used in practice

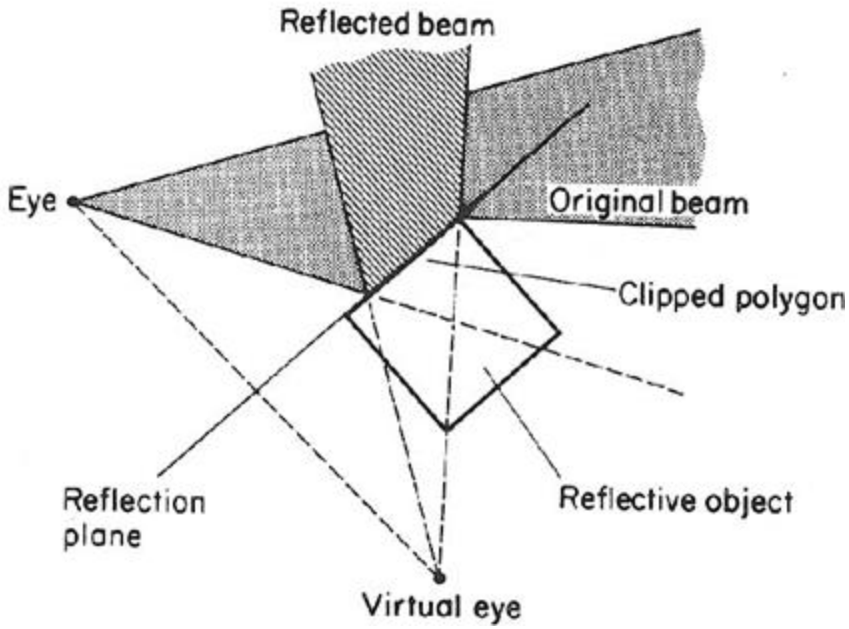


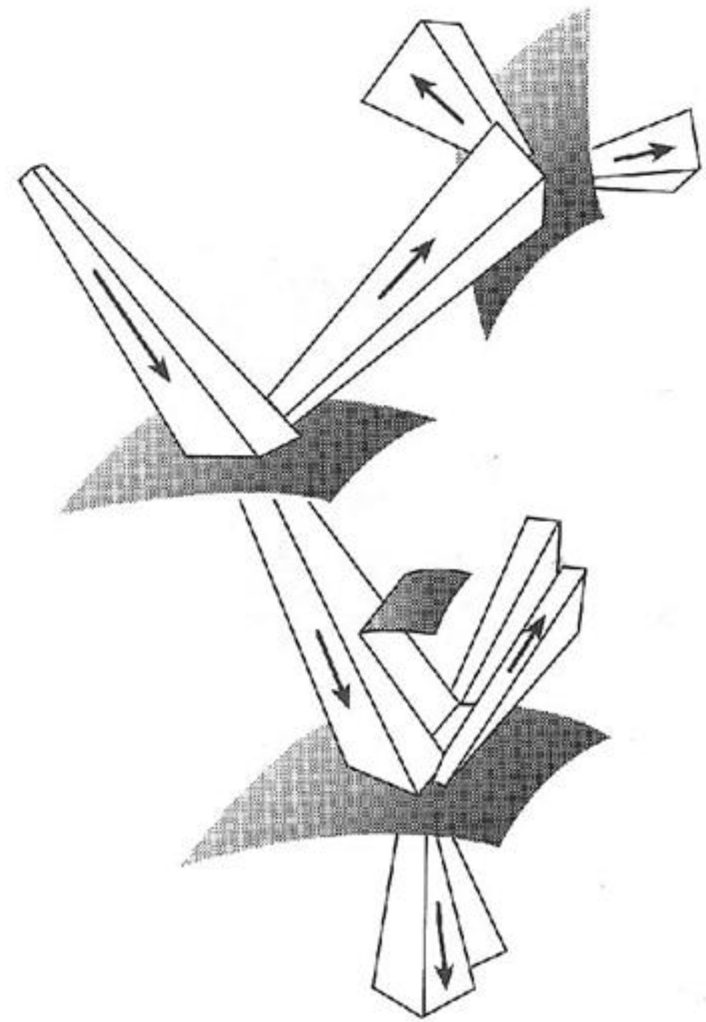
Packet Tracing

- **Approach**
  - Combine many similar rays (e.g. primary or shadow rays)
  - Trace them together in SIMD fashion
    - All rays perform the same traversal operations
    - All rays intersect the same geometry
    - Can use SIMD instructions in modern processors
  - Exposes coherence between rays
    - All rays touch similar spatial indices
    - Loaded data can be reused (in registers & cache)
    - More computation per recursion step → better optimization
  - Overhead
    - Rays will perform unnecessary operations
    - Overhead low for coherent and small set of rays (e.g. up to 4x4 rays)
- **Needs an API that provides coherent sets of rays**

Beam Tracing

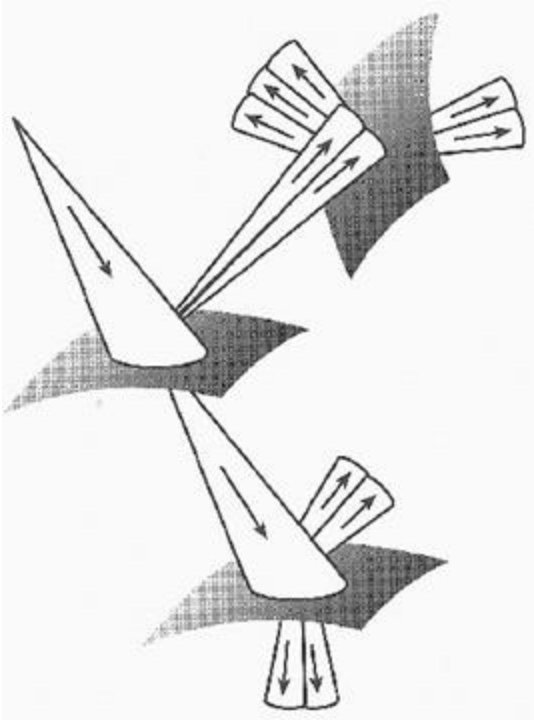






Beam and Cone Tracing

- **General idea:**
  - Trace continuous bundles of rays
- **Cone Tracing:**
  - Approximate collection of ray with cone(s)
  - Subdivide into smaller cones if necessary
- **Beam Tracing:**
  - Exactly represent a ray bundle with pyramid
  - Create new beams at intersections (polygons)
- **Problems:**
  - Clipping of beams?
  - Good approximations?
  - How to compute intersections?
- **Not really practical !!**



Frustum Tracing

- **Bound set of rays with frustum (NOT frustrum!!)**
  - Only during traversal
  - API needs to provide coherent groups of rays
    - Possibly hierarchically
- **Traverse spatial index with frustum**
  - Small overhead (largely avoided by SIMD)
    - Compute with 4 corner “rays”
  - Avoid traversing many rays individually
    - Particularly beneficial in the upper levels of spatial index
  - Switch to (packets of) rays when needed (intersection)
    - Might be able to only use subset (e.g. based on extend of triangle)
  - Split frustum hierarchically and traverse separately in lower levels
    - Avoids overhead of carrying to many rays into small nodes
- **E.g. fast primary ray traversal by W. Hunt (Oculus)**