Accelerated Ray Casting

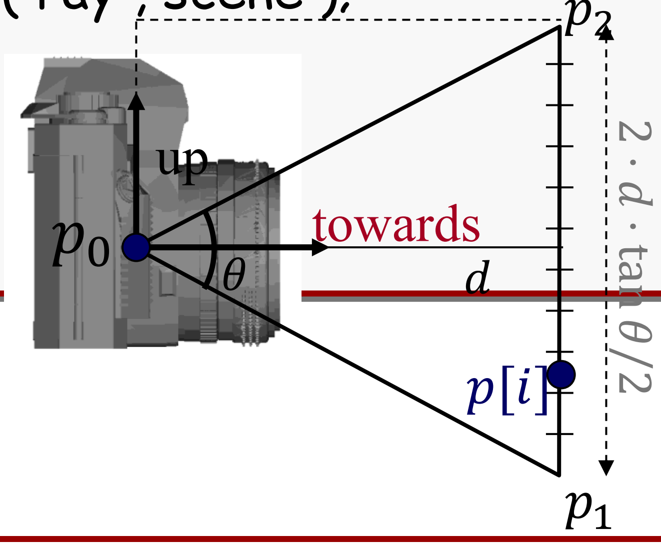
Misha Kazhdan

(601.457/657)

Ray Casting

Simple implementation:

```
Image RayCast( Camera camera , Scene scene , int width , int height)
{
    Image image( width , height );
    for( int j=0 ; j<height ; j++ ) for( int i=0 ; i<width ; i++ )
    {
        Ray< 3 > ray = ConstructRayThroughPixel( camera , i , j );
        Intersection hit = FindIntersection( ray , scene );
        image[i][j] = GetColor( hit );
    }
    return image;
}
```



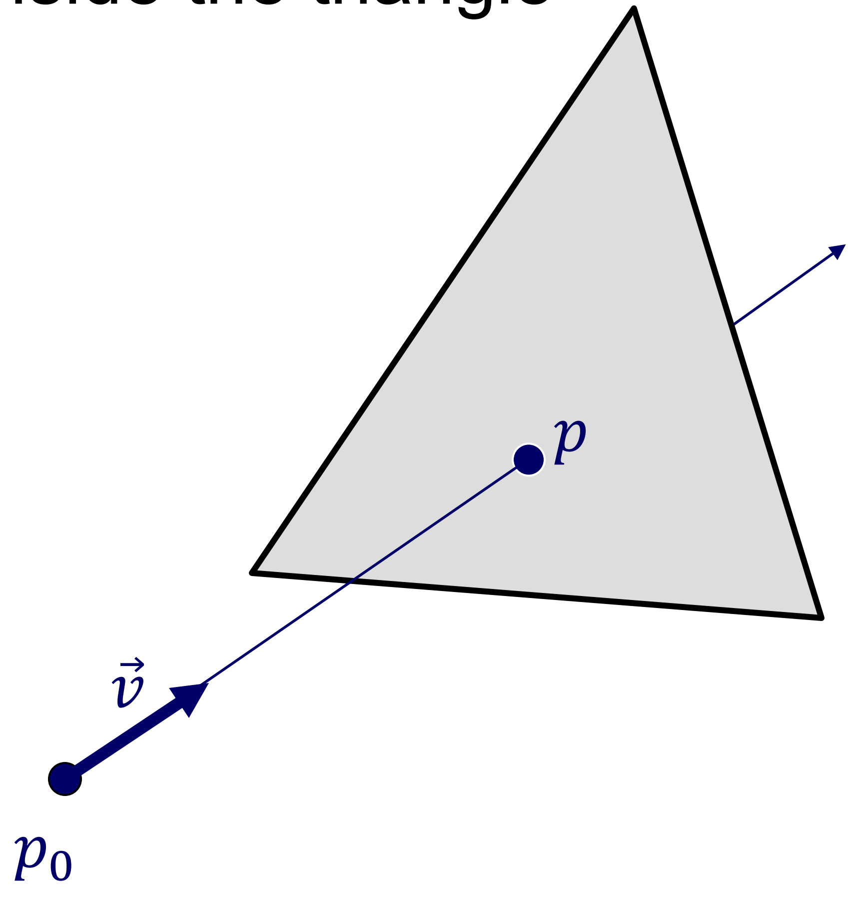
Ray Casting

Simple implementation:

```
Image RayCast( Camera camera , Scene scene , int width , int height)
{
    Image image( width , height );
    for( int j=0 ; j<height ; j++ ) for( int i=0 ; i<width ; i++ )
    {
        Ray< 3 > ray = ConstructRayThroughPixel( camera , i , j );
        Intersection hit = FindIntersection( ray , scene );
        image[i][j] = GetColor( hit );
    }
    return image;
}
```

Ray-Triangle Intersection

1. Intersect ray with plane
2. Check if the point is inside the triangle



Ray-Plane Intersection

Ray: $p(t) = p_0 + t \cdot \vec{v}, \quad (0 \le t < \infty)$
Plane: $\Phi(p) = \langle p, \vec{n} \rangle - d = 0$

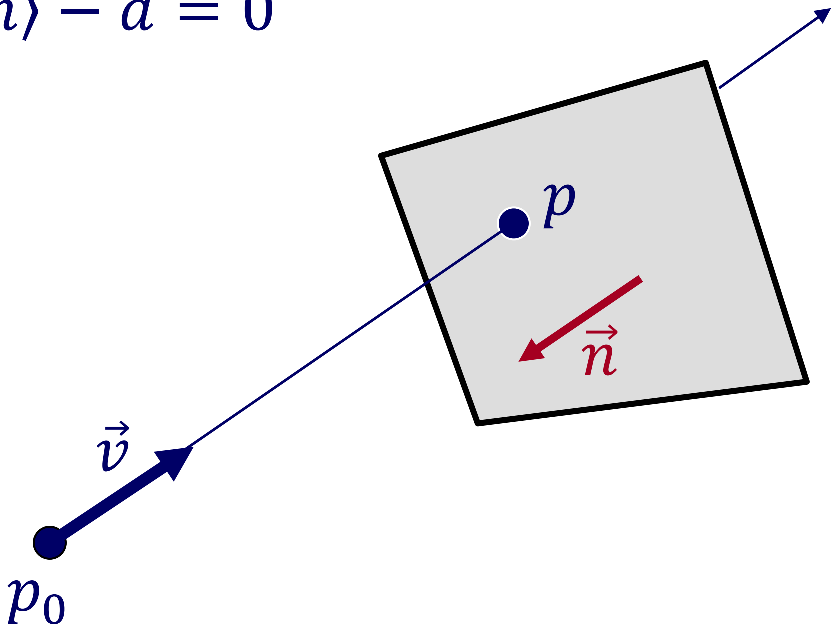
Algebraic Method

Substituting for p we get:

$$\Phi(t) \equiv \Phi\big(p(t)\big) = \langle e + t \cdot \vec{v}, \vec{n} \rangle - d = 0$$

Solution:

$$t = -\frac{\langle p_0, \vec{n} \rangle - d}{\langle \vec{v}, \vec{n} \rangle}$$

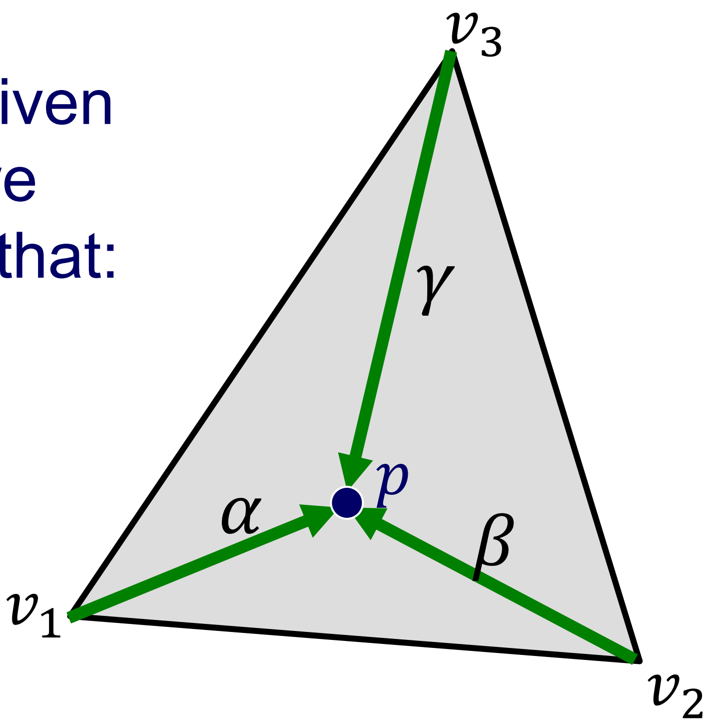


Ray-Triangle Intersection

Check for point-triangle intersection parametrically

In general, given $p \in \mathbb{R}^3$ and given three points $\{v_1, v_2, v_3\} \subset \mathbb{R}^3$ we can solve for $\alpha, \beta, \gamma \in \mathbb{R}$ such that:

$$p = \alpha v_1 + \beta v_2 + \gamma v_3$$



p is in the plane containing $\{v_1, v_2, v_3\}$ if and only if (iff.):

$$\alpha + \beta + \gamma = 1$$

p is inside the triangle with vertices $\{v_1, v_2, v_3\}$ iff.:

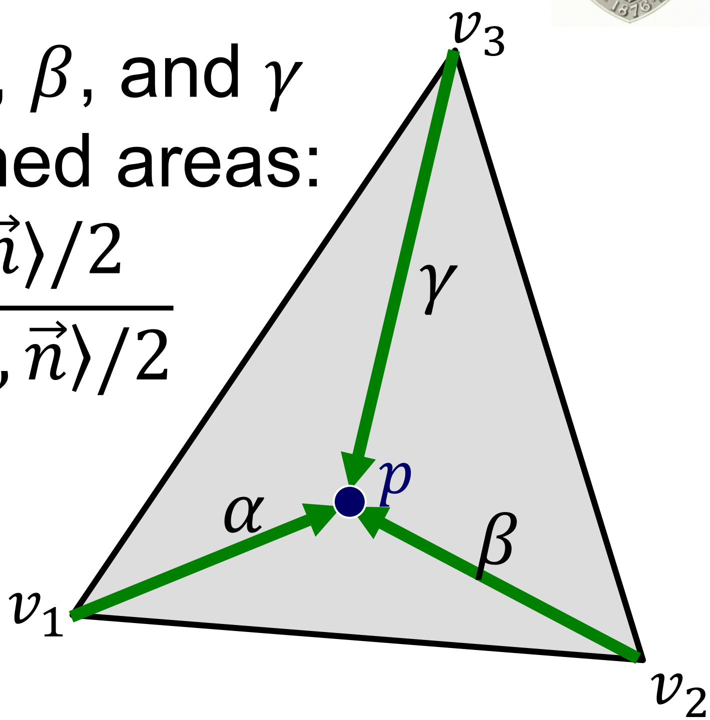
$$\alpha, \beta, \gamma \geq 0$$

Ray-Triangle Intersection

We can compute the weights α, β, and γ by considering the ratios of signed areas:

$$\alpha = \frac{\langle (v_2 - p) \times (v_3 - p), \vec{n} \rangle / 2}{\langle (v_2 - v_1) \times (v_3 - v_1), \vec{n} \rangle / 2}$$

$$\vdots$$



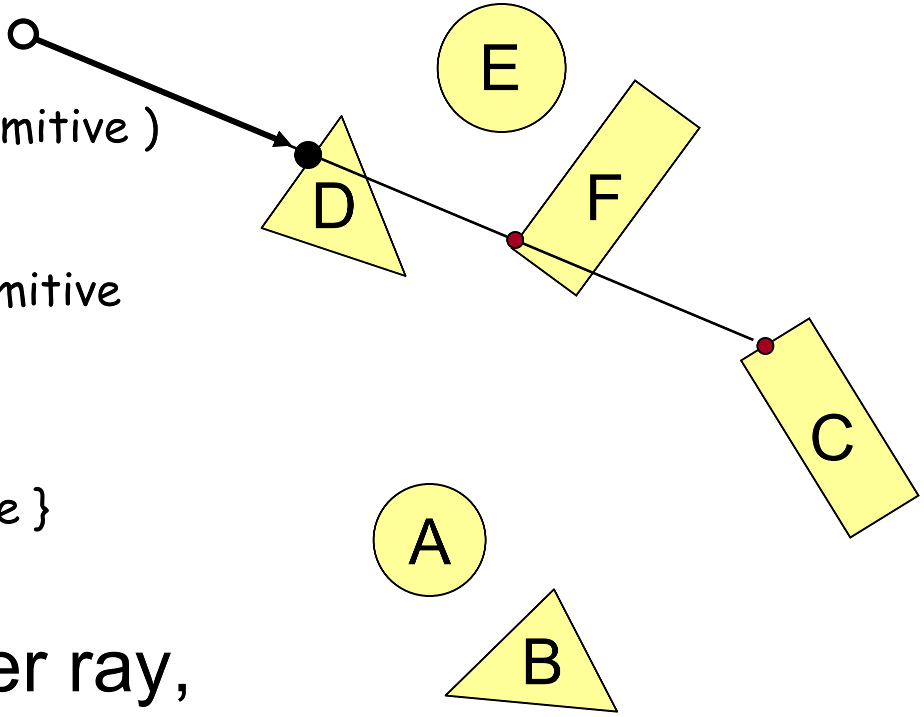
where $\vec{n}$ is a unit vector that is perpendicular to the triangle:

$$\vec{n} = \pm \frac{(v_2 - v_1) \times (v_3 - v_1)}{|(v_2 - v_1) \times (v_3 - v_1)|}$$

Ray-Scene Intersection

A direct (naïve) approach:

```
Intersection FindIntersection( Ray< 3 > ray, Scene scene )
{
    { min_t , first_shape } = { ∞ , NULL }
    for each primitive in scene
    {
        t = Intersect( ray , primitive )
        if( t<min_t )
        {
            first_shape = primitive
            min_t = t
        }
    }
    return { min_t , first_shape }
}
```



Complexity is $O(N)$ per ray, with N the number of primitives.

Overview

Acceleration techniques

- Data-Partitions
  - » Bounding volume hierarchy (BVH)
- Space-Partitions
  - » Uniform (voxel) grid
  - » Octree
  - » Binary space partition (BSP) tree

Acceleration techniques

Both data- and space-partitions accelerate intersections testing by leveraging:

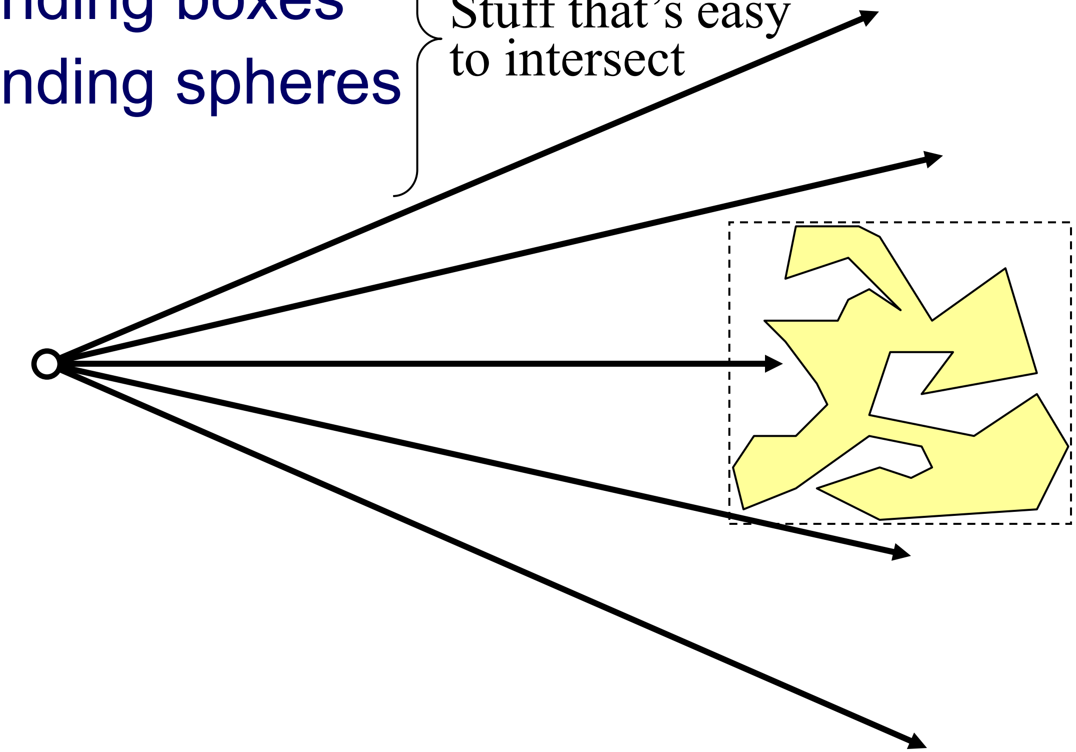
- Trivial Rejection:
  Discard groups of primitives / regions of space that can be (easily) guaranteed to be missed by the ray.
- Ordering:
  Test (likely) nearer primitives / regions of space first, allowing for early termination if there is a hit.

Data-Partition: Bounding Volume

Surround the shape with a bounding volume and check for intersection with the volume:

- Bounding cubes
- Bounding boxes
- Bounding spheres
- Etc.

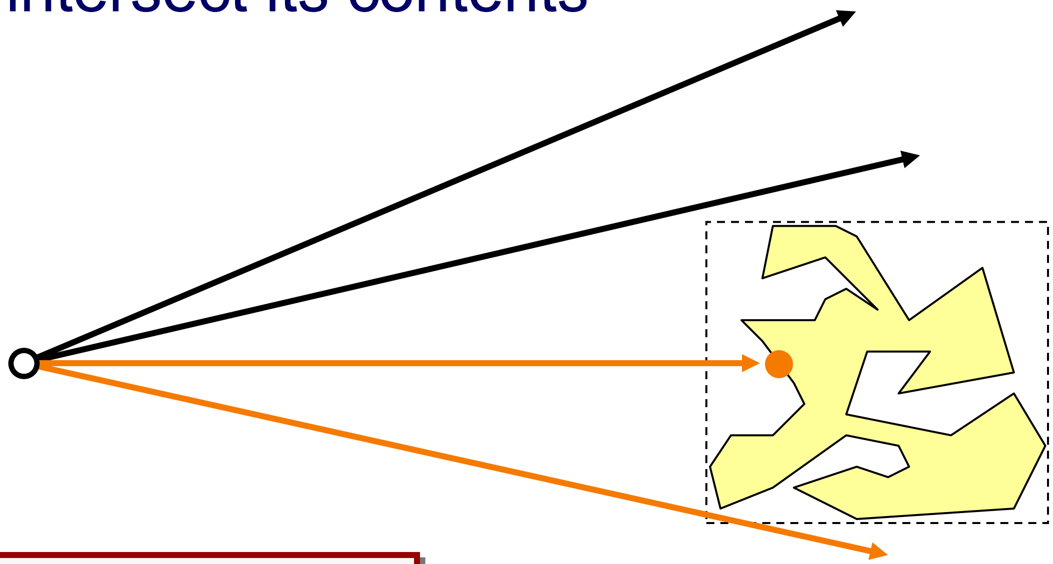
Stuff that's easy to intersect



Data-Partition: Bounding Volume

Surround the shape with a bounding volume and check for intersection with the volume:

If the ray misses the bounding volume, it can't intersect its contents

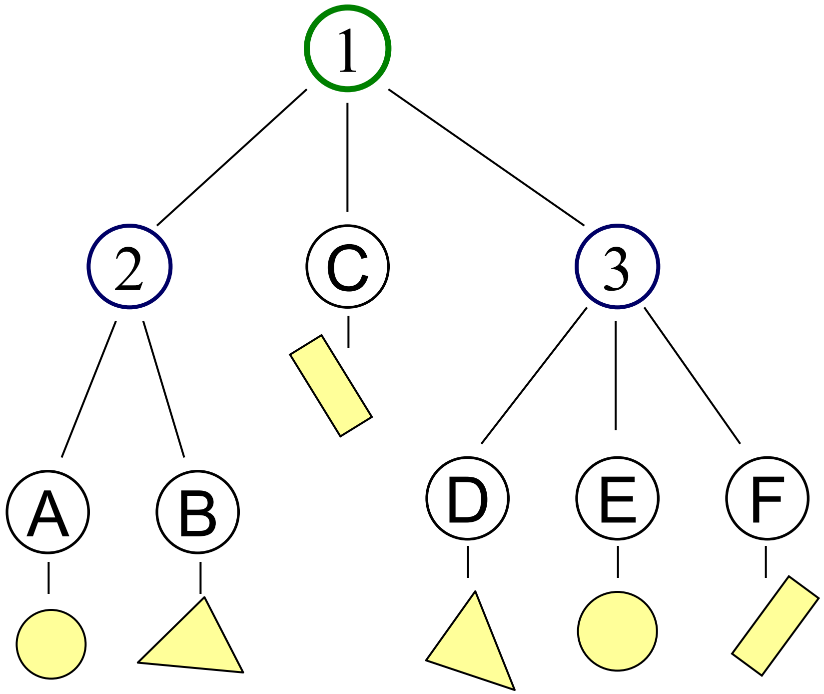


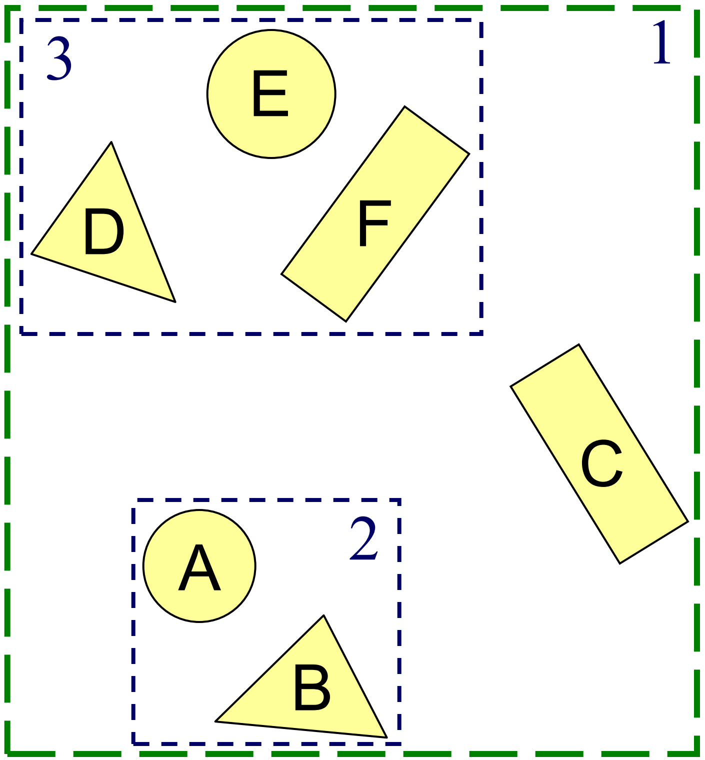
Still need to check for intersections with shape.

Data-Partition: BVH

Build a bounding volume hierarchy (BVH) tree

- A node in the tree consists of a bounding-box as well as:
  - » A subset of (enclosed) shapes
  - » Child nodes





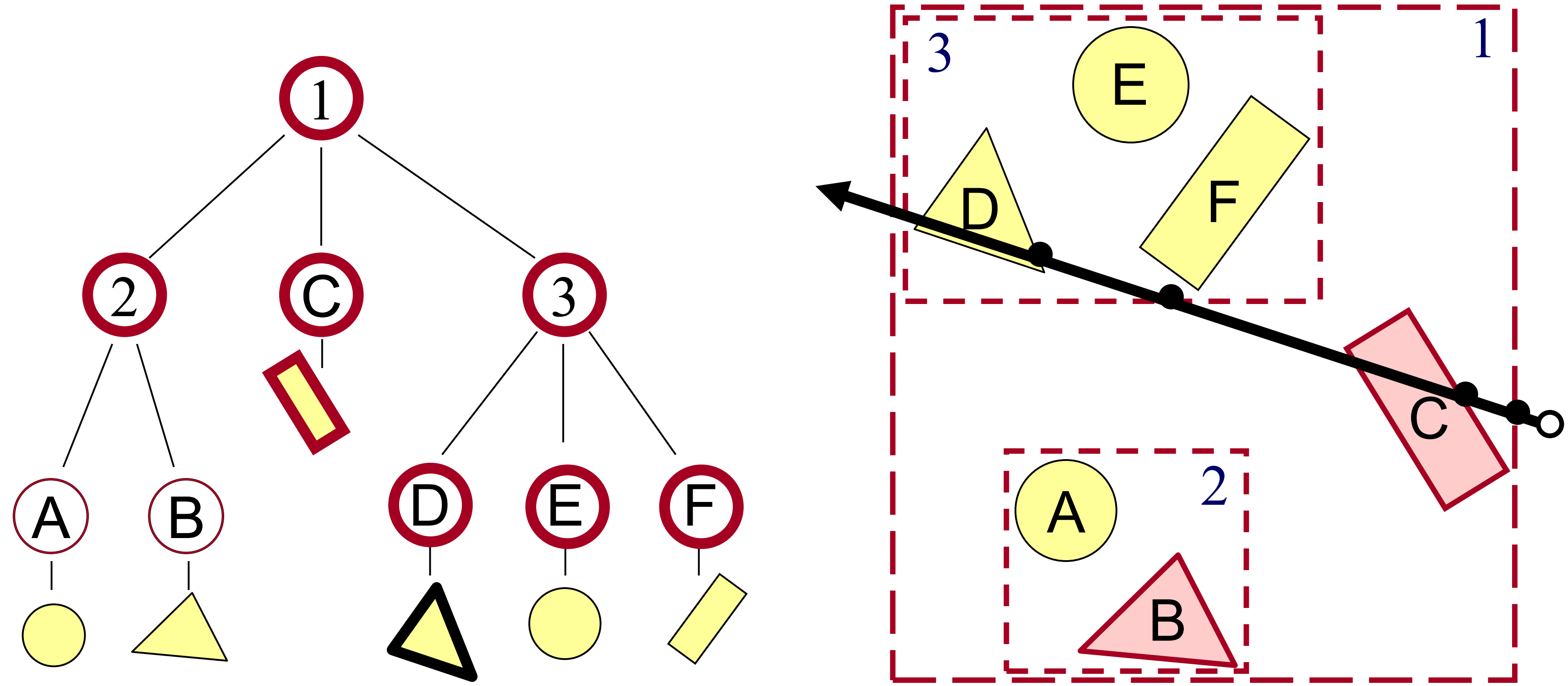
Data-Partition: BVH

Trivial Rejection:

```
Intersection FindIntersection( Ray< 3 > ray , BVHNode< 3 > node )
{
	{ min_t , first_shape } = { ∞ , NULL }

	if( !intersect( ray , node.bBox ) )			// Test Bounding-box
		return { ∞ , NULL }

	foreach shape in node					// Test node's shape
	{
		t = Intersect( ray , shape )
		if( t<min_t ) { min_t , first_shape } = { t , shape }
	}

	for each child_node of node				// Test node's children
	{
		{ t , shape } = FindIntersection( ray , child_node )
		if( t<min_t ) { min_t , first_shape } = { t , shape }
	}
	return { min_t , first_shape }
}
```

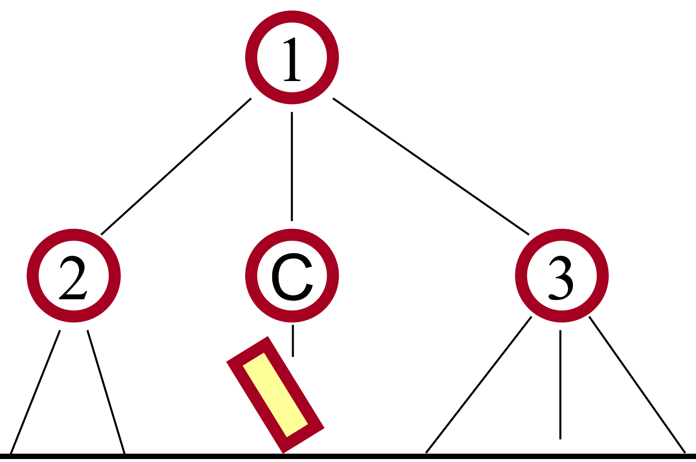
Data-Partition: BVH

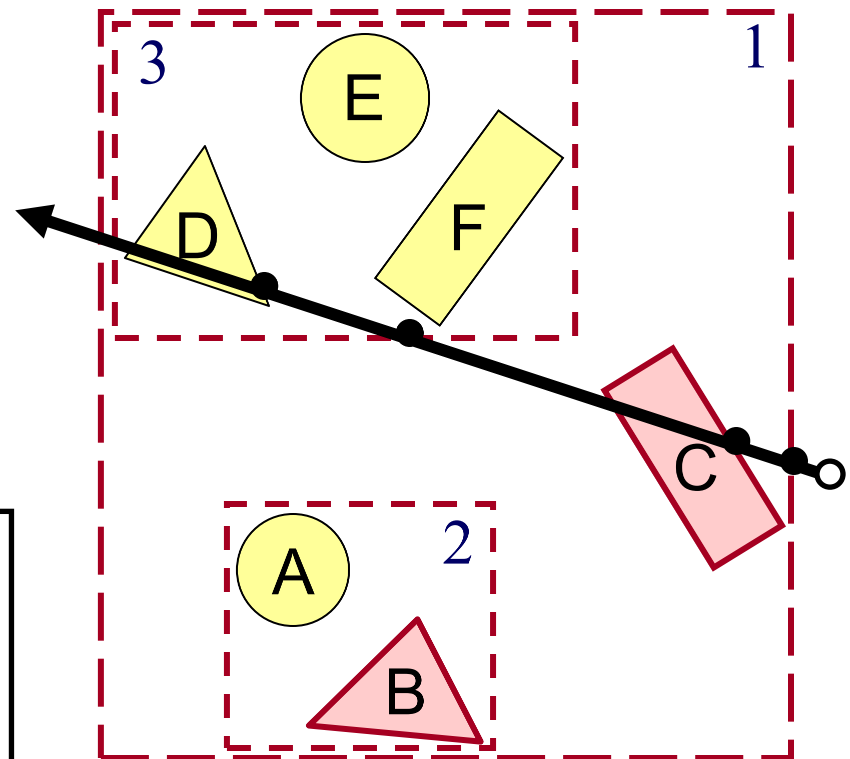
Trivial Rejection:
Discard groups of primitives that can be (easily) guaranteed to be missed by the ray.



Data-Partition: BVH

Trivial Rejection:
Discard groups of primitives that can be (easily) guaranteed to be missed by the ray.





Don't need to test shapes A or B

Need to test volumes 1, 2, and 3

Need to test shapes C, D, E, and F

Data-Partition: BVH

Ordering:

```
Intersection FindIntersection( Ray< 3 > ray , BVHNode< 3 > node )
{
	// Find intersections with the nearest shape stored in node
	…
	// Find intersections with all child bounding-boxes
	…
	// Sort child BVH node intersections front to back
	// and store distances to child bounding-boxes in bv_t[]
	…

	// Process intersections
	for each intersected child bounding volume
	{

		{ t , shape } = FindIntersection( ray , child_bBox )
		if( t<min_t )  { min_t , min_shape } = { t , shape }
	}
	return { min_t , min_shape }
}
```

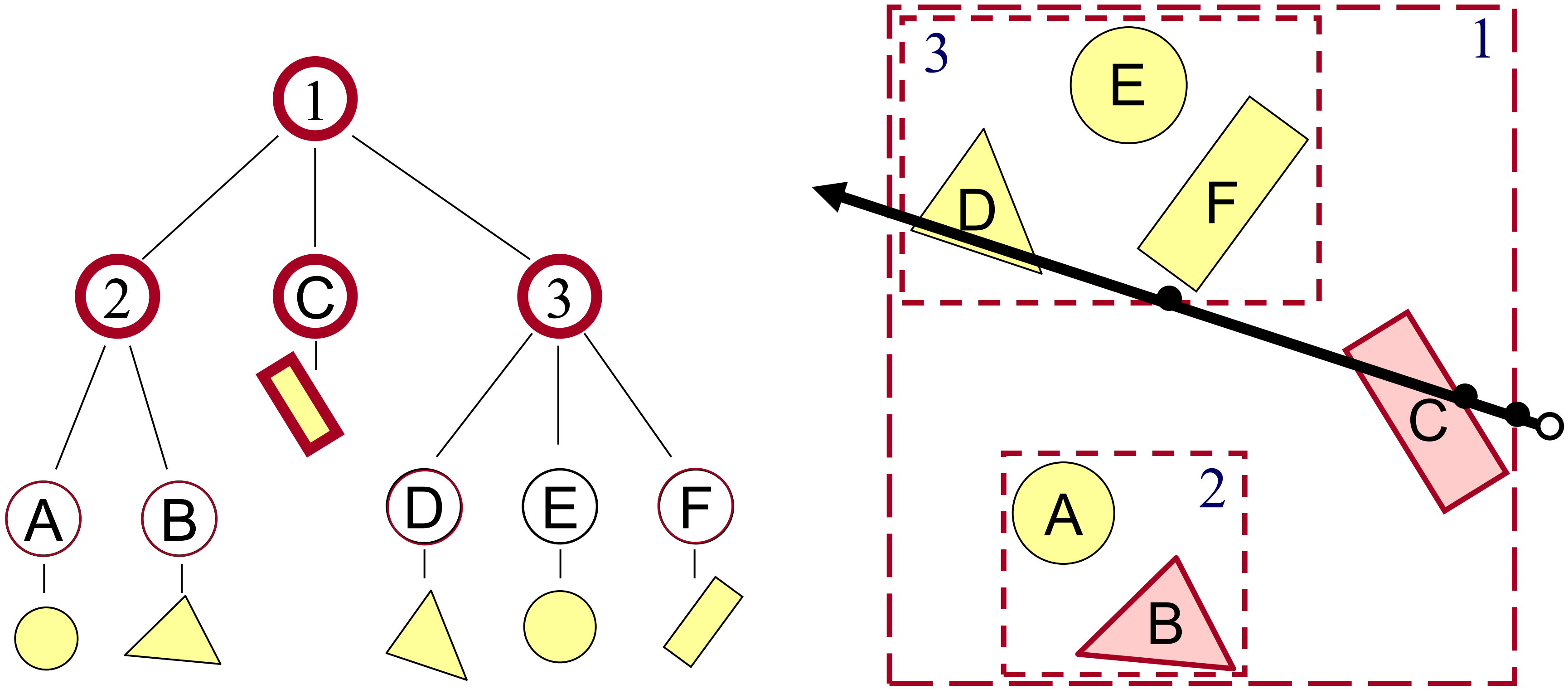
Data-Partition: BVH

Ordering:

```
Intersection FindIntersection( Ray< 3 > ray , BVHNode< 3 > node )
{
	// Find intersections with the nearest shape stored in node
	…
	// Find intersections with all child bounding-boxes
	…
	// Sort child BVH node intersections front to back
	// and store distances to child bounding-boxes in bv_t[]
	…

	// Process intersections
	for each intersected child bounding volume
	{
		if( min_t<bv_t[child_bVolume] ) break
		{ t , shape } = FindIntersection( ray , child_bBox )
		if( t<min_t ) { min_t , min_shape } = { t , shape }
	}
	return { min_t , min_shape }
}
```

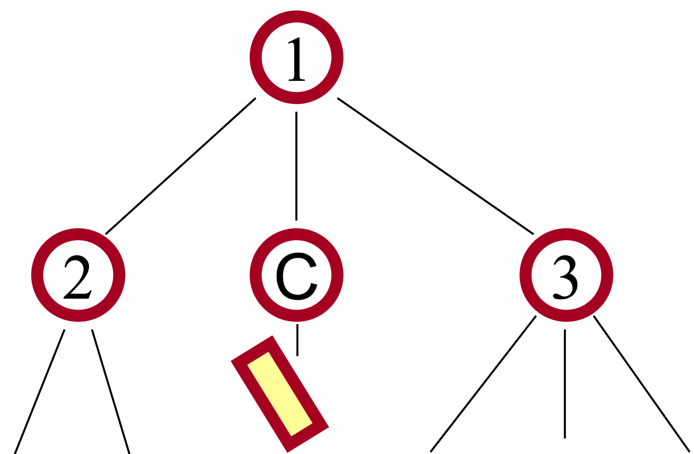
Data-Partition: BVH

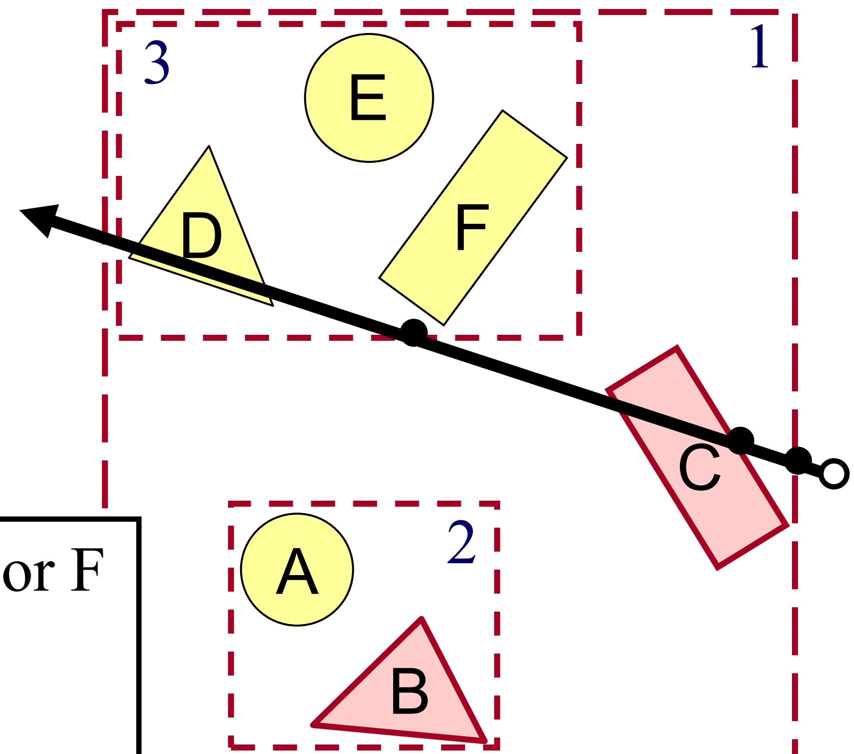
Ordering:
Test (likely) nearer intersections first, allowing for early termination if there is a hit.



Data-Partition: BVH

<u>Ordering</u>:
Test (likely) nearer intersections first, allowing for early termination if there is a hit.





Don’t need to test shapes A, B, D, E, or F

Need to test groups 1, 2, and 3

Need to test shape C

Data-Partition: BVH

[WARNING]:
A bounding-box may be singular – e.g. if it incapsulates planar, axis-aligned geometry.

To avoid potential numerical issues, you can thicken the bounding-box by a small amount.

Overview

Acceleration techniques

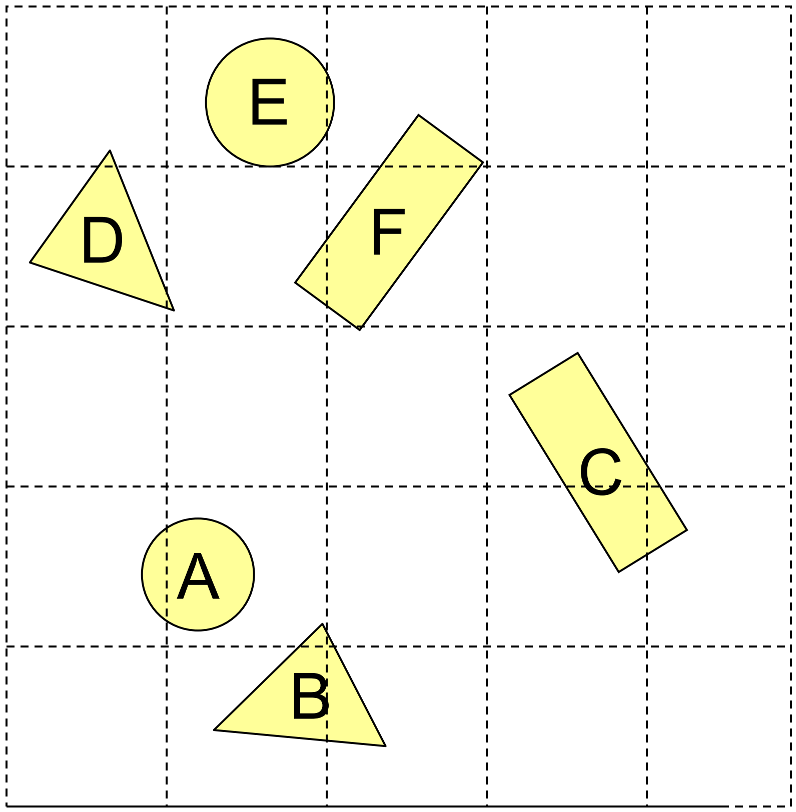
- Data-partitions
  - » Bounding volume hierarchy (BVH)
- Space-Partitions
  - » Uniform (Voxel) grid
  - » Octree
  - » Binary space partition (BSP) tree

Space-Partitions: Uniform Grid

Construct uniform grid over the scene

Store a list of intersected primitive with each grid cell

- A primitive may belong to multiple cells
- A cell may have multiple primitives

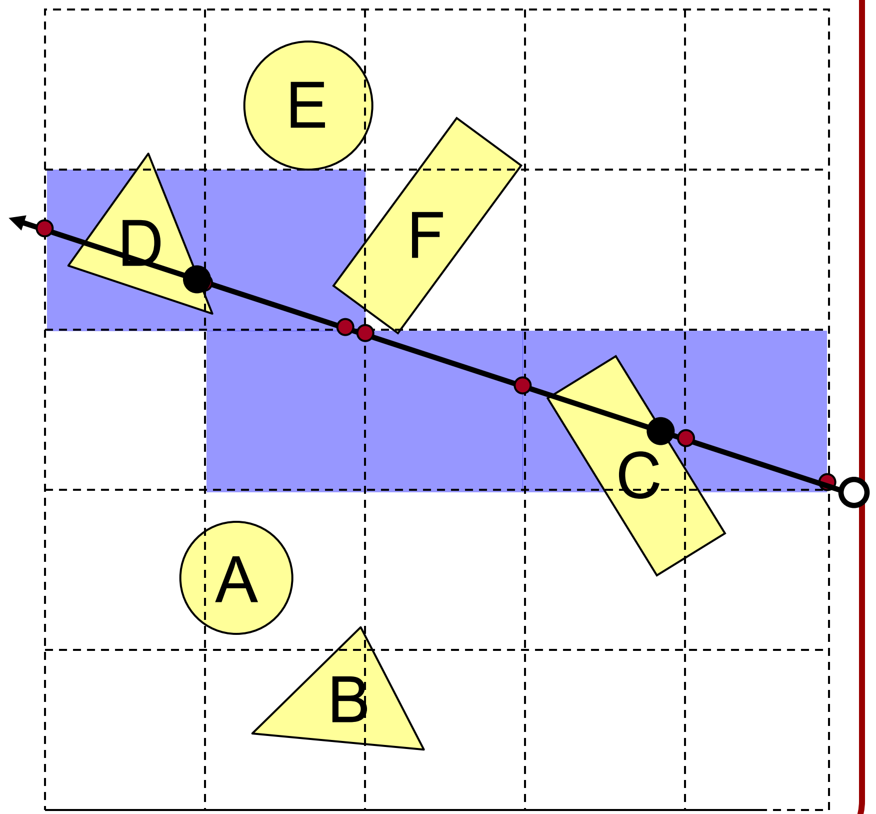


Space-Partitions: Uniform Grid

Trace rays through grid cells

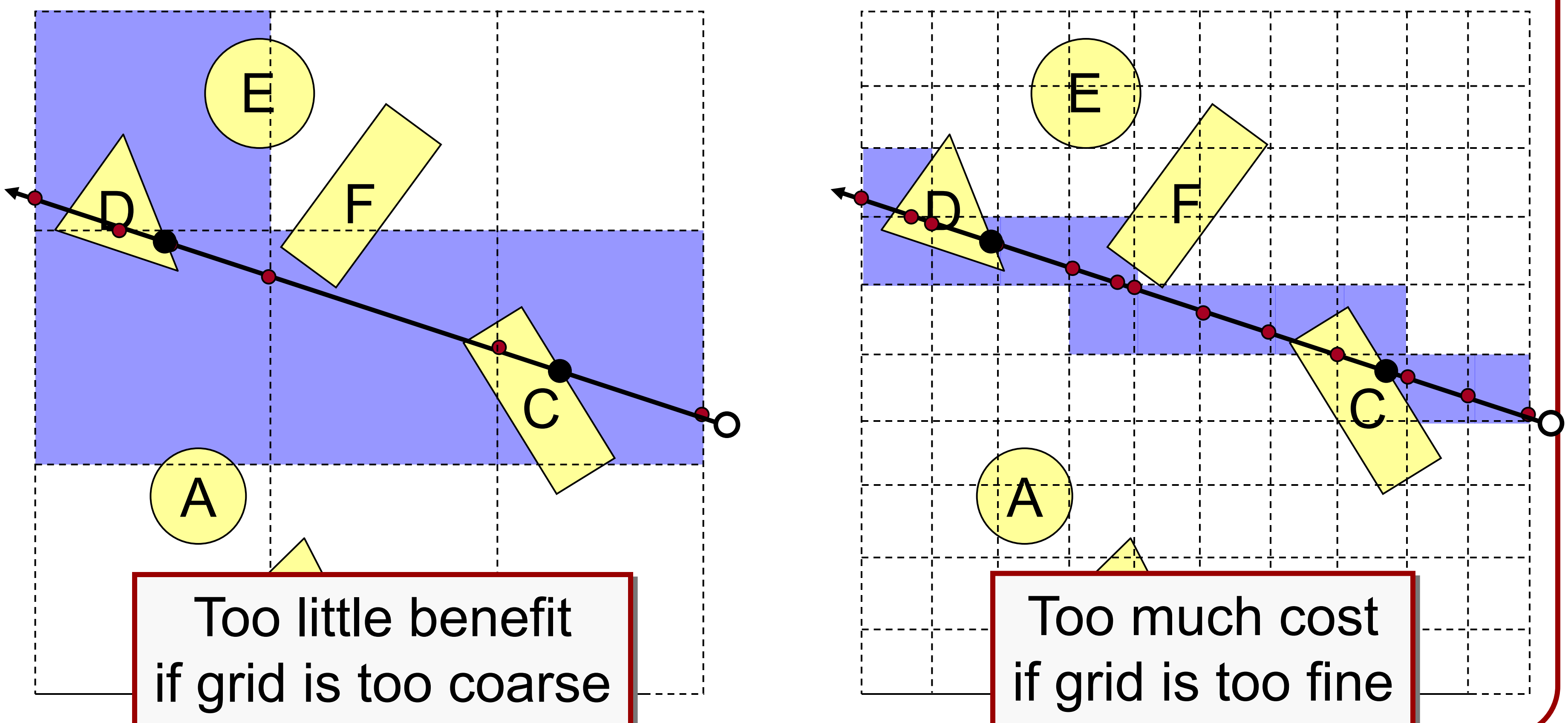
- Trivial Rejection:
  Avoid testing parts of space the ray doesn't go through
- Ordering:
  Traverse the cells in closest-to-furthest order

Only check primitives in intersected grid cells



Space-Partitions: Uniform Grid

Challenge: How to choose suitable grid resolution?



Overview

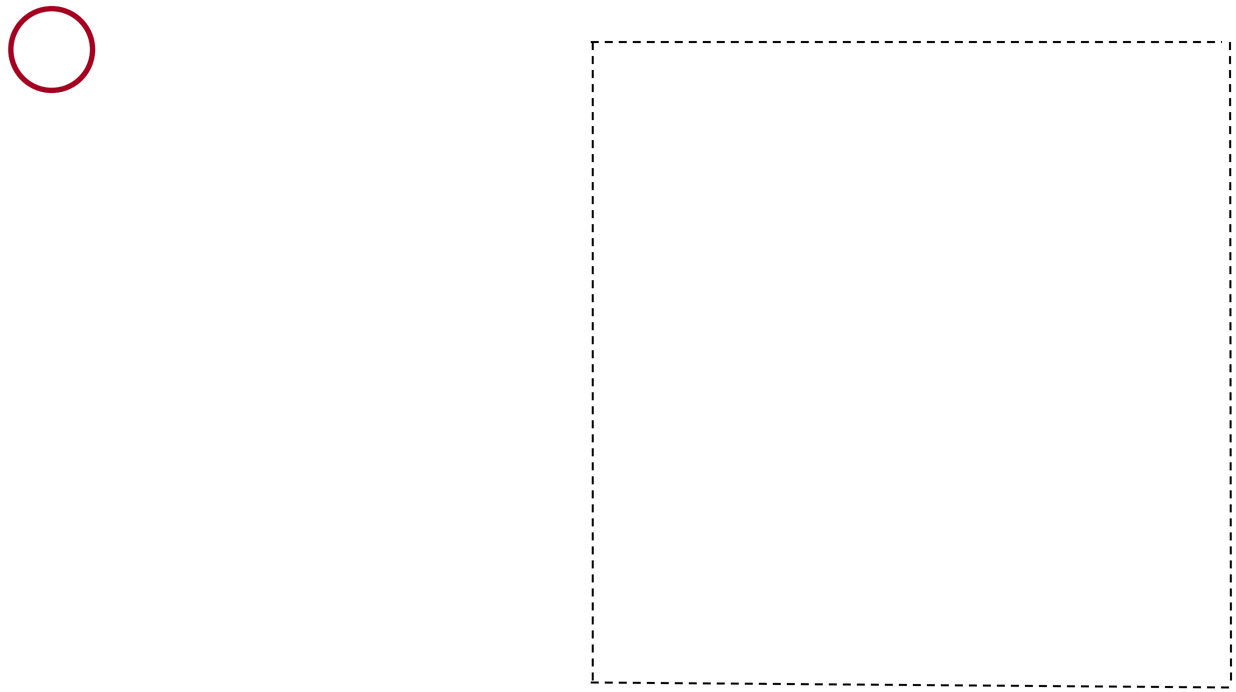
Acceleration techniques

- Data-Partitions
  - » Bounding volume hierarchy (BVH)
- Space-Partitions
  - » Uniform (Voxel) grid
  - » Octree
  - » Binary space partition (BSP) tree

Space-Partition: Octree

Think of a (power-of-two) voxel grid as a tree.

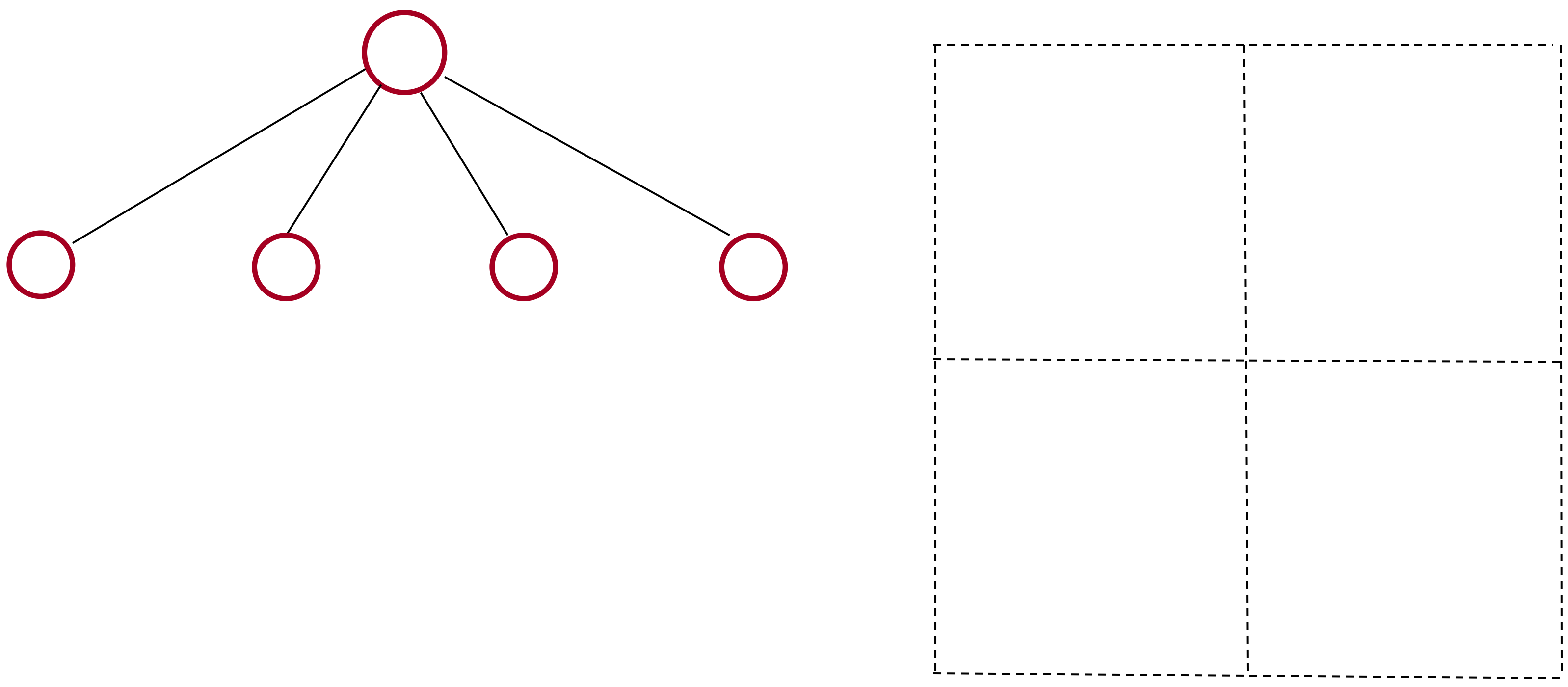
- The root node is the entire region
- Each node has eight children obtained by subdividing the parent into eight equal regions



Space-Partition: Octree

Think of a (power-of-two) voxel grid as a tree.

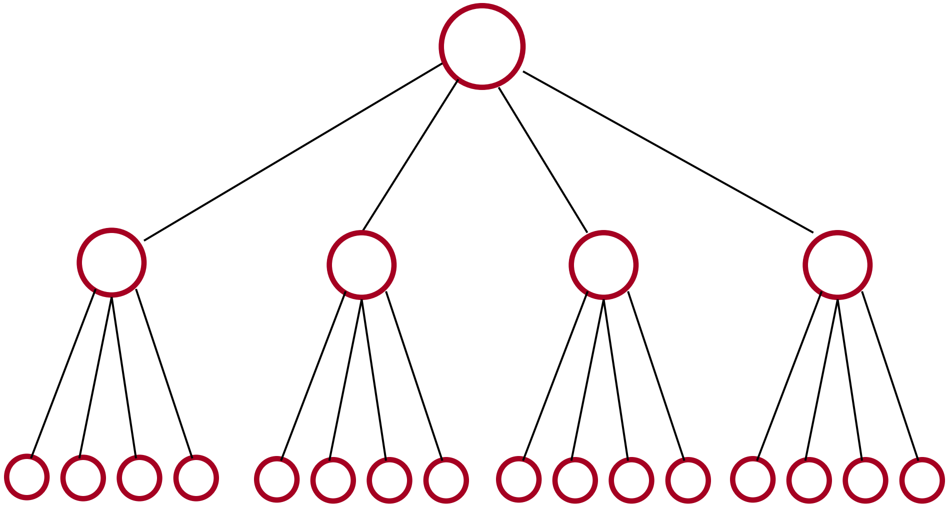
- The root node is the entire region
- Each node has eight children obtained by subdividing the parent into eight equal regions

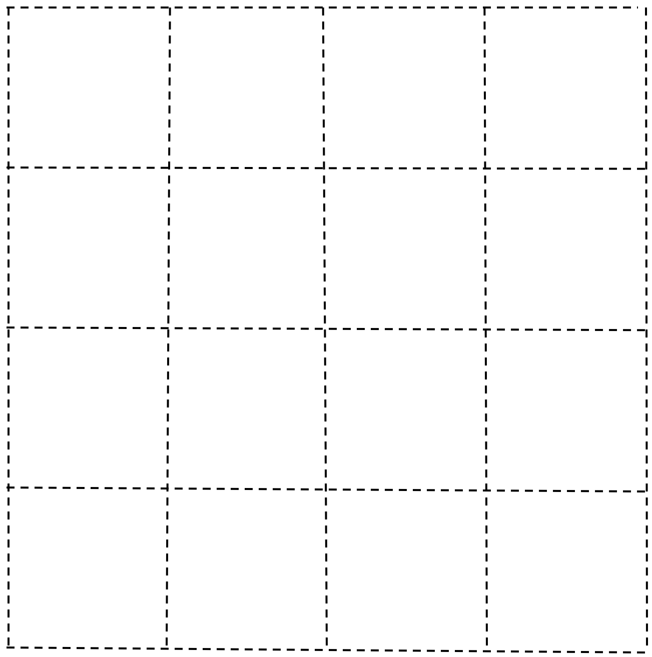


Space-Partition: Octree

Think of a (power-of-two) voxel grid as a tree.

- The root node is the entire region
- Each node has eight children obtained by subdividing the parent into eight equal regions

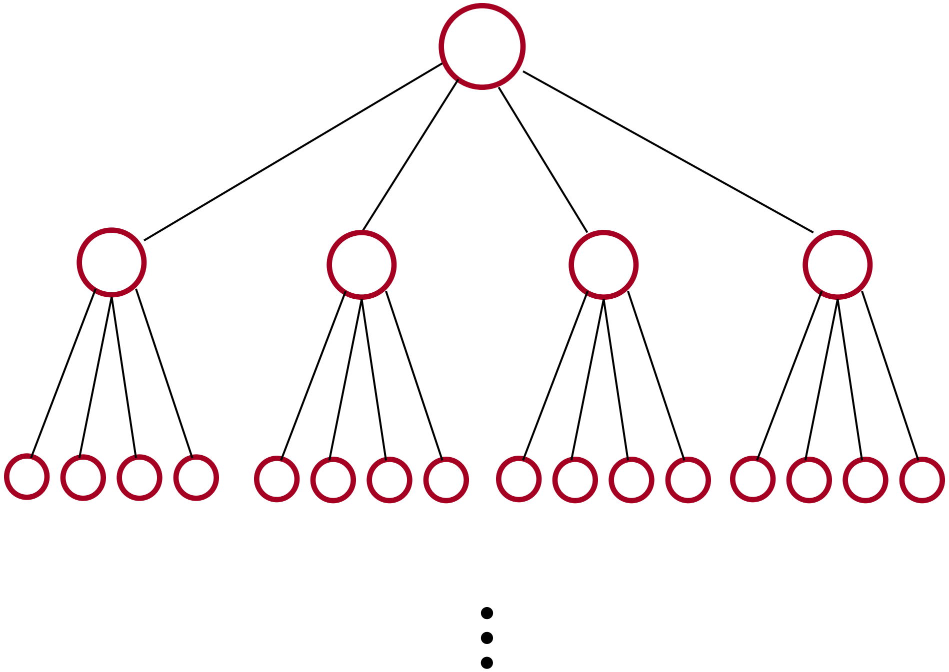


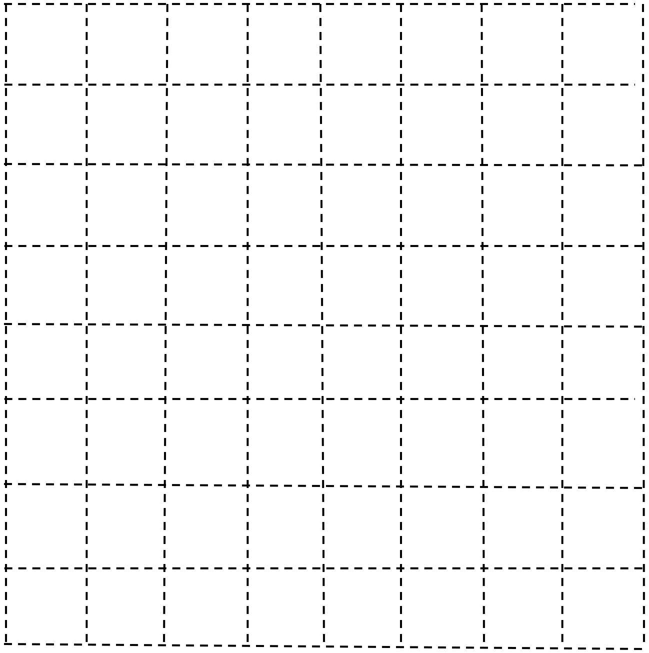


Space-Partition: Octree

Think of a (power-of-two) voxel grid as a tree.

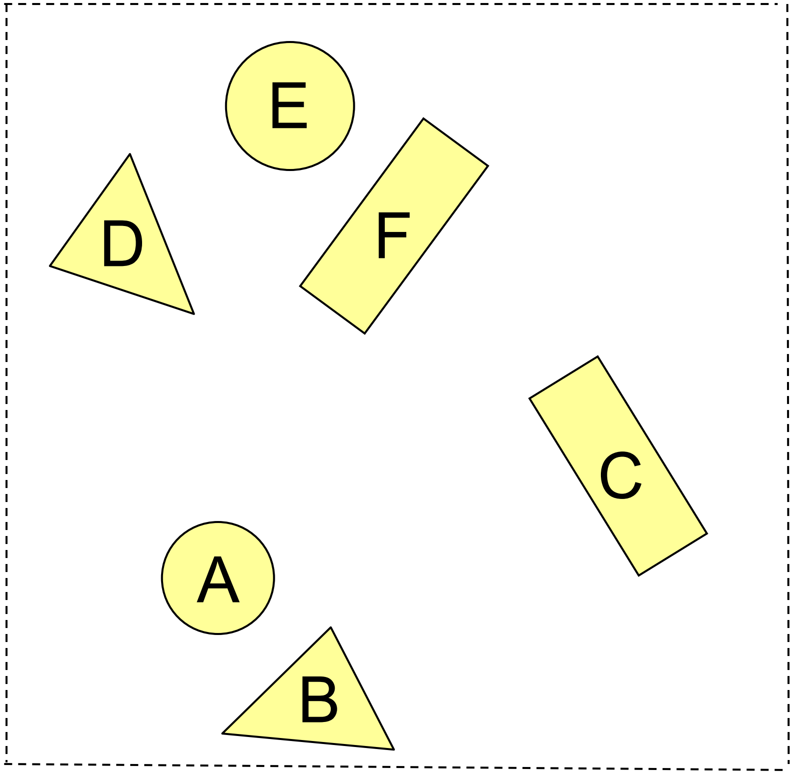
- The root node is the entire region
- Each node has eight children obtained by subdividing the parent into eight equal regions





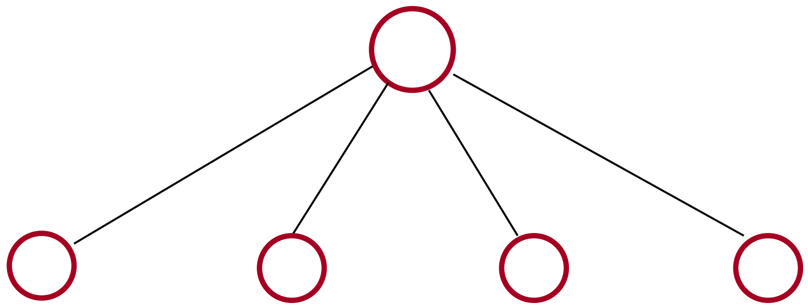
Space-Partition: Octree

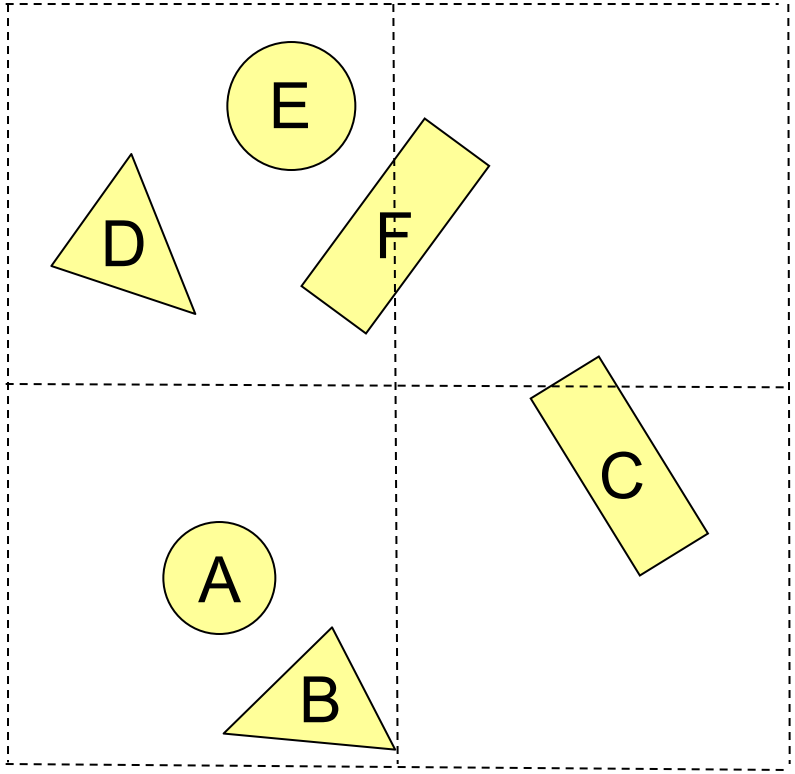
In an octree, subdivide **adaptively** (only refining cells containing more than one shape).



Space-Partition: Octree

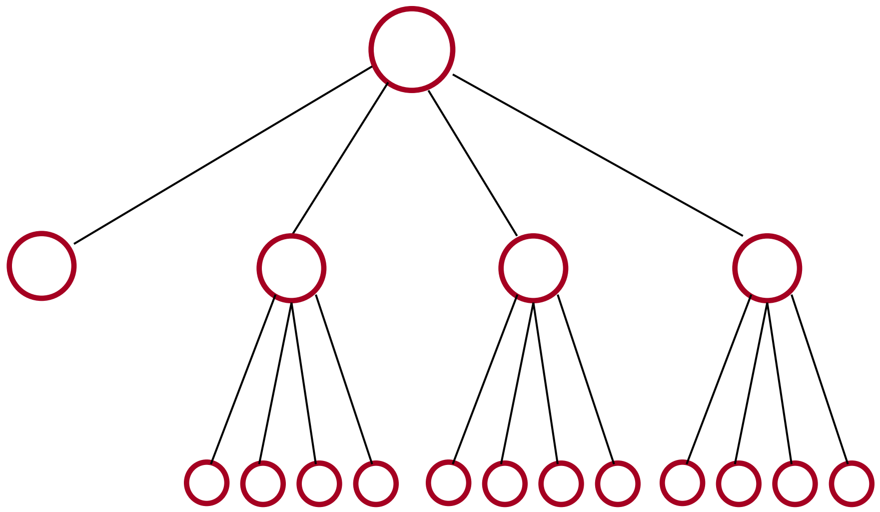
In an octree, subdivide **adaptively** (only refining cells containing more than one shape).

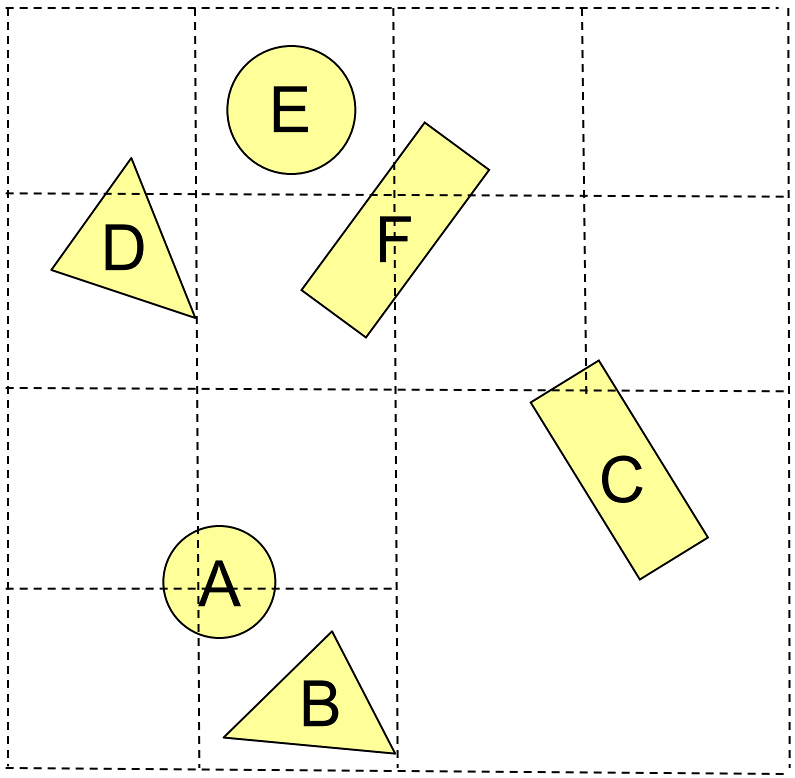




Space-Partition: Octree

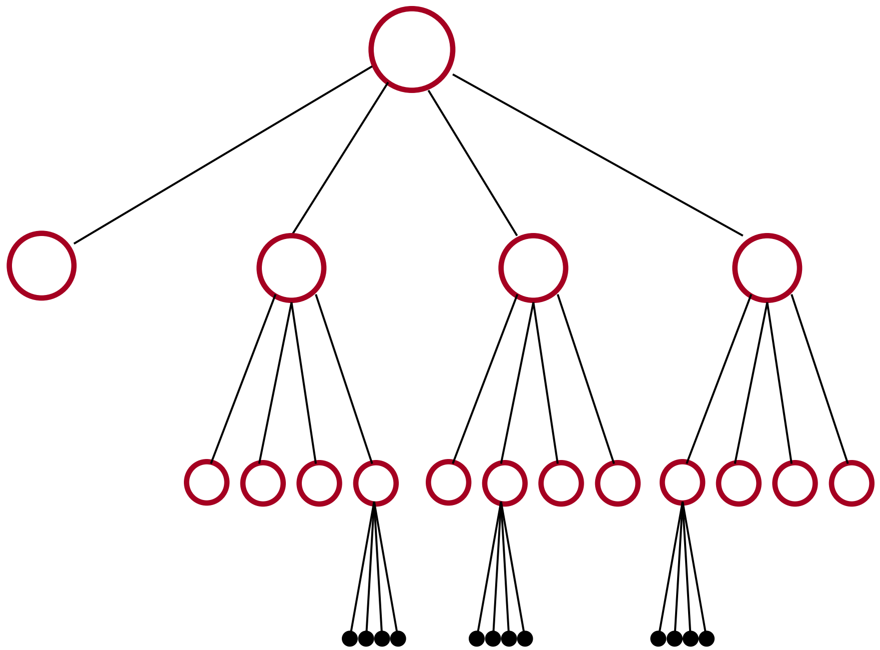
In an octree, subdivide **adaptively** (only refining cells containing more than one shape).

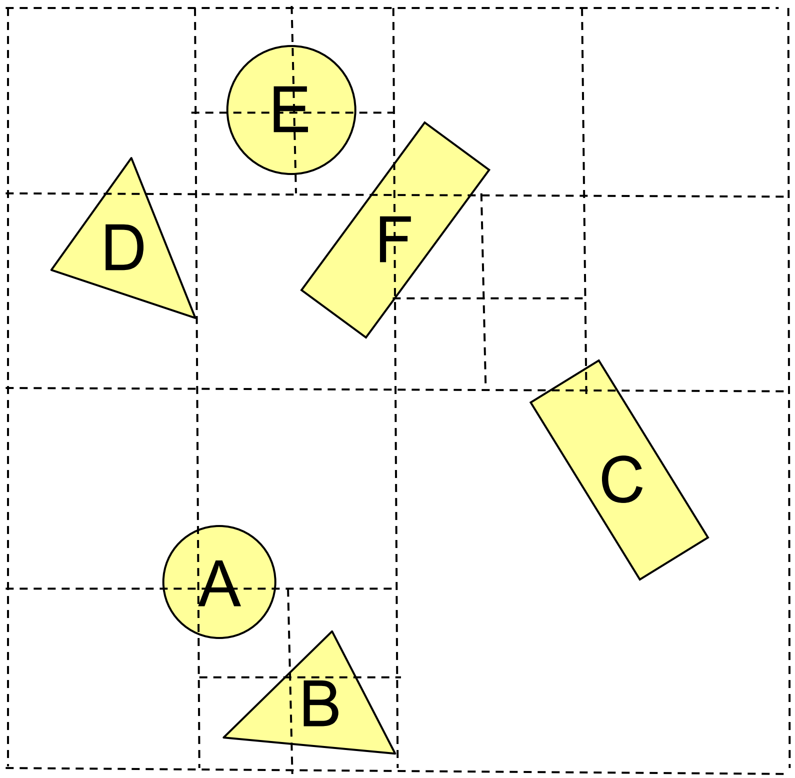




Space-Partition: Octree

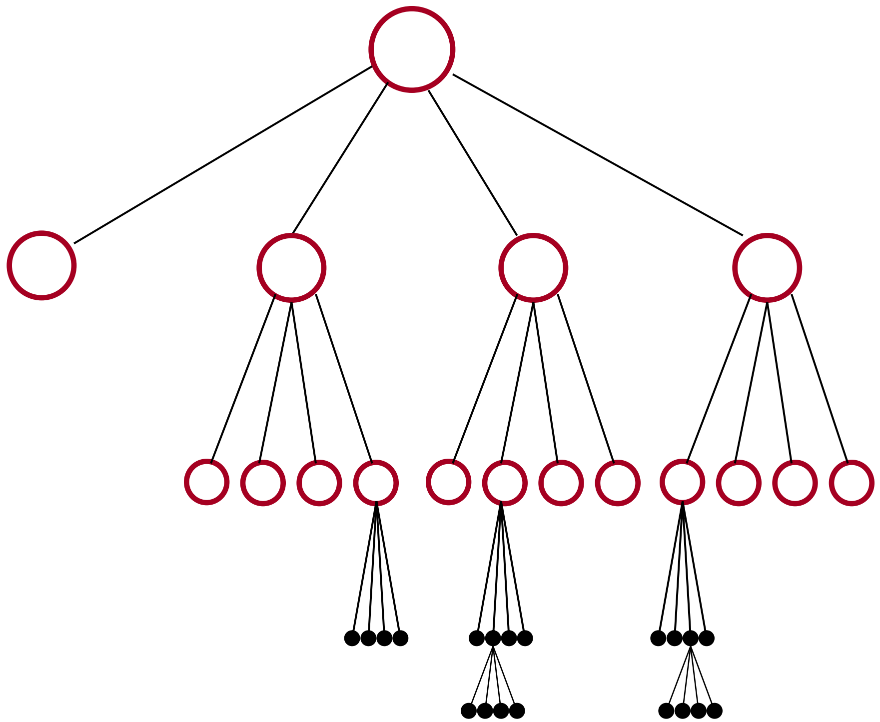
In an octree, subdivide **adaptively** (only refining cells containing more than one shape).

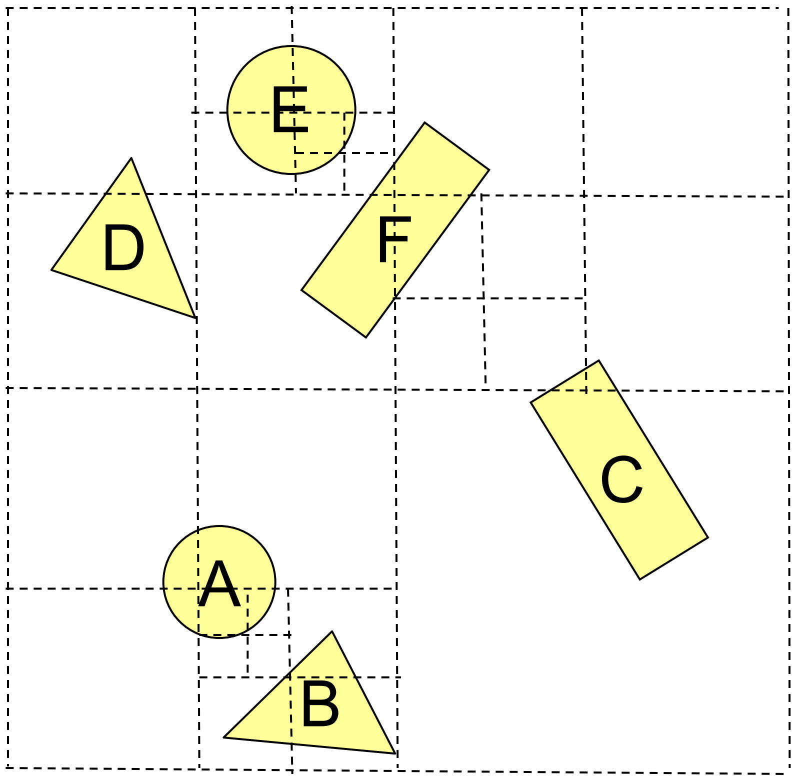




Space-Partition: Octree

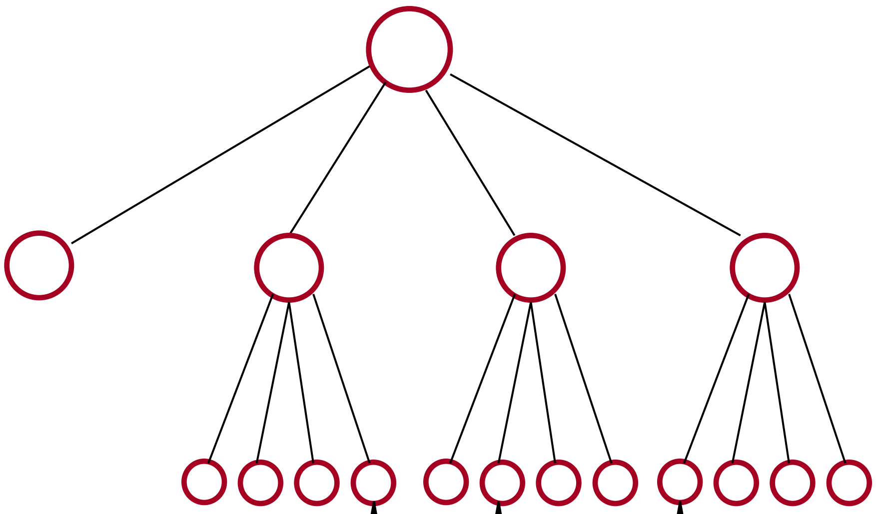
In an octree, subdivide **adaptively** (only refining cells containing more than one shape).

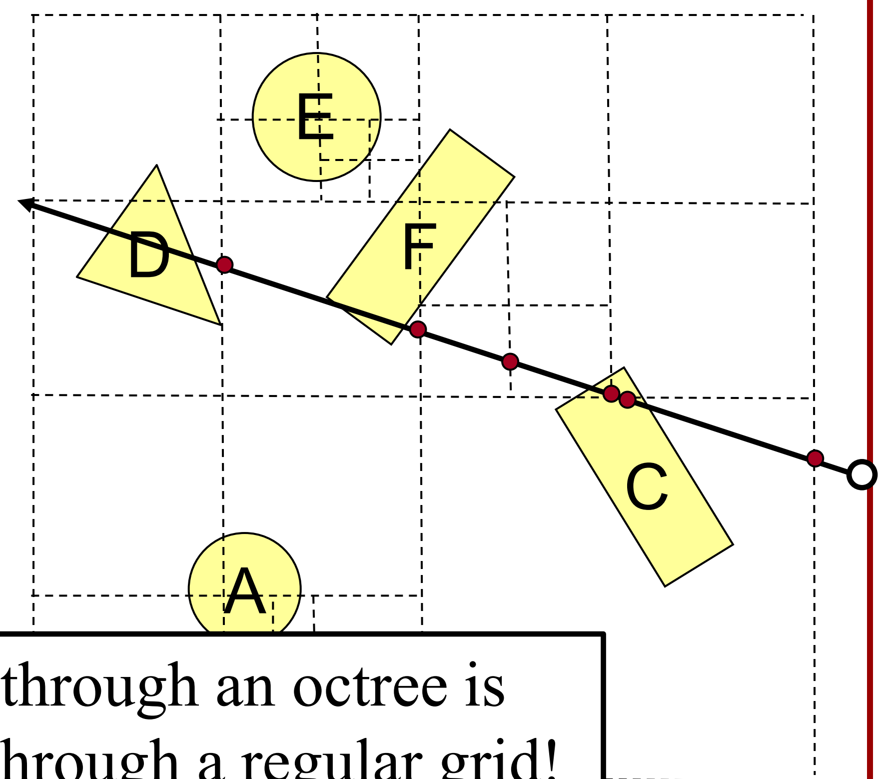




Space-Partition: Octree

In an octree, subdivide **adaptively** (only refining cells containing more than one shape).





Efficiently tracing a ray through an octree is trickier than tracing a ray through a regular grid!

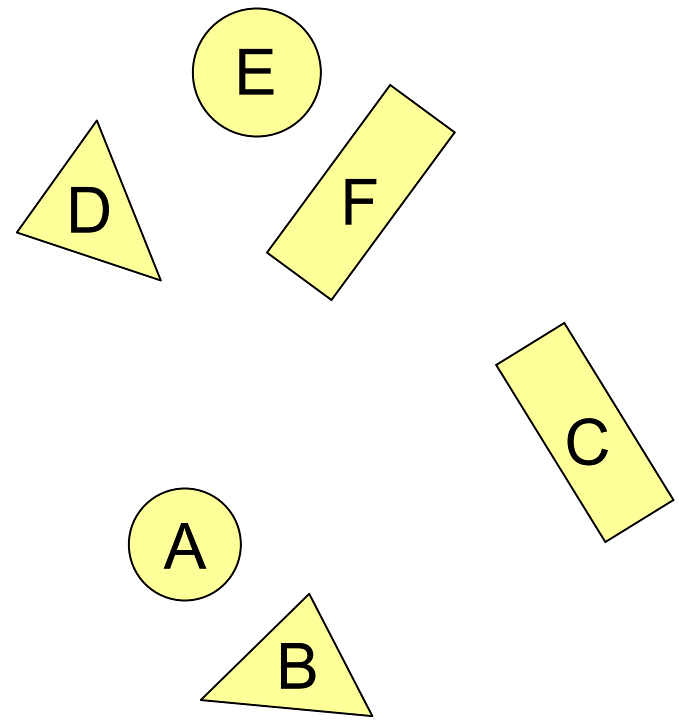
Overview

Acceleration techniques

- Data-Partitions
  - » Bounding volume hierarchy (BVH)
- Space-Partitions
  - » Uniform (Voxel) grid
  - » Octree
  - » Binary space partition (BSP) tree
    - – k-D tree

Space-Partition: *k*-D Trees

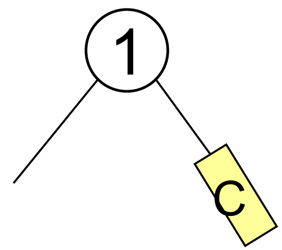
Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

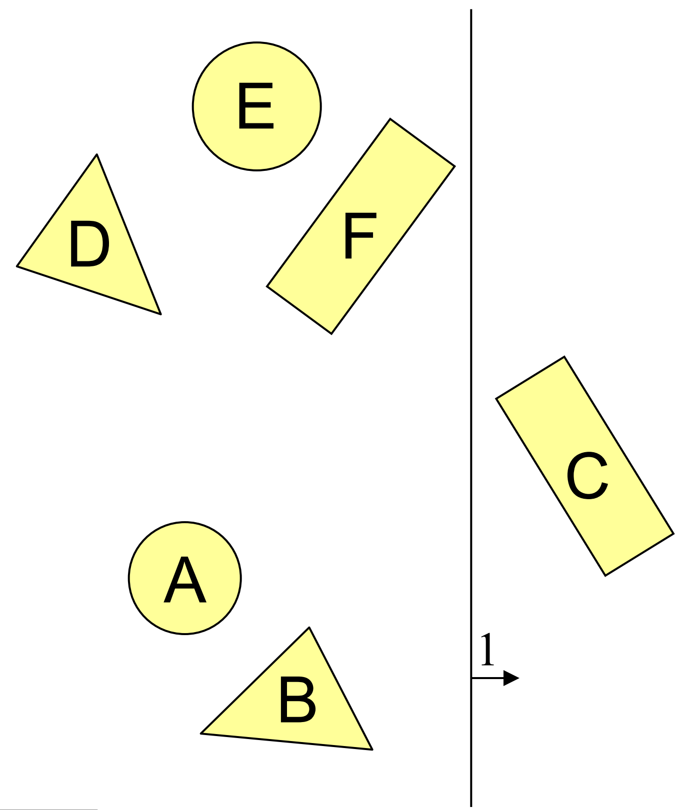


Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

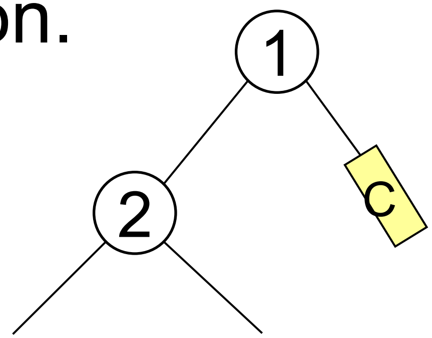


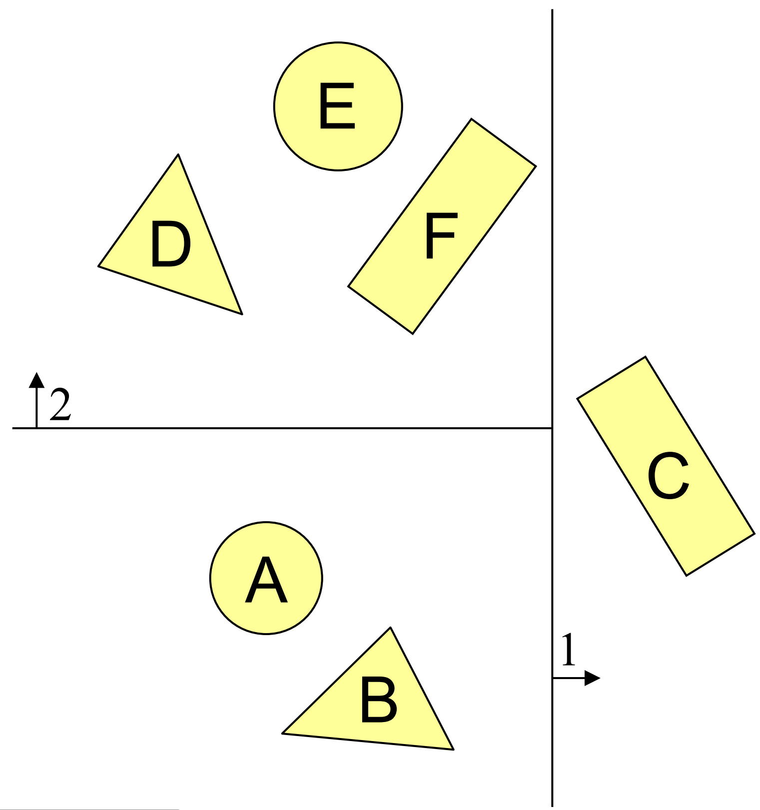


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

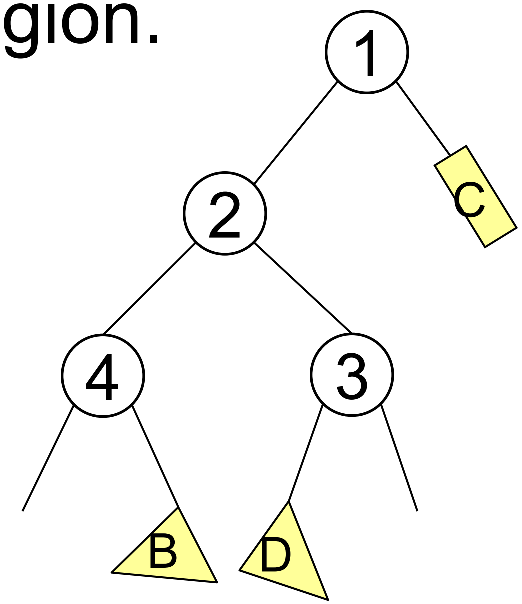


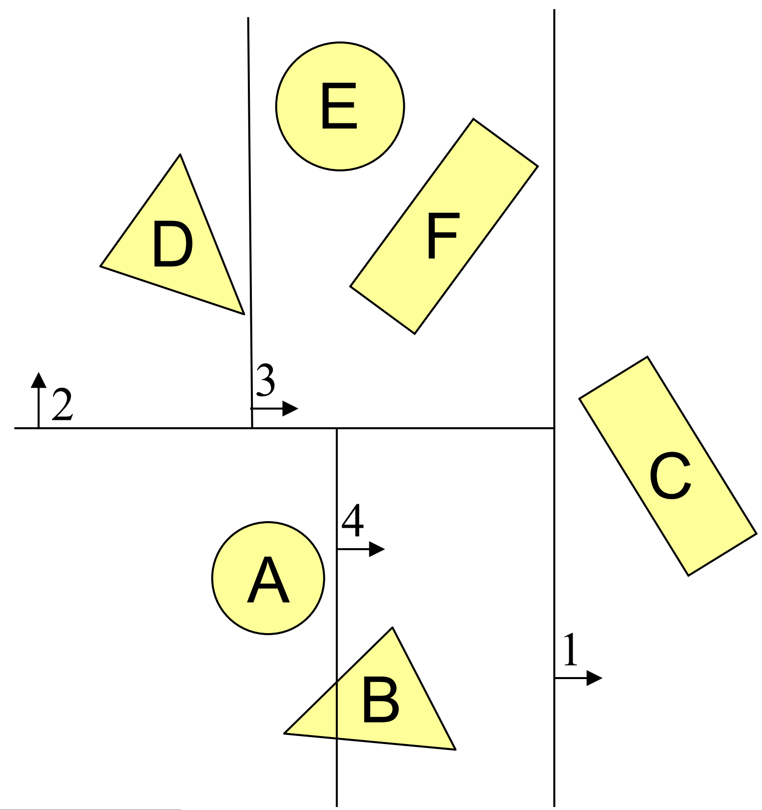


<ins>Note</ins>: Arrows denote the “right” side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

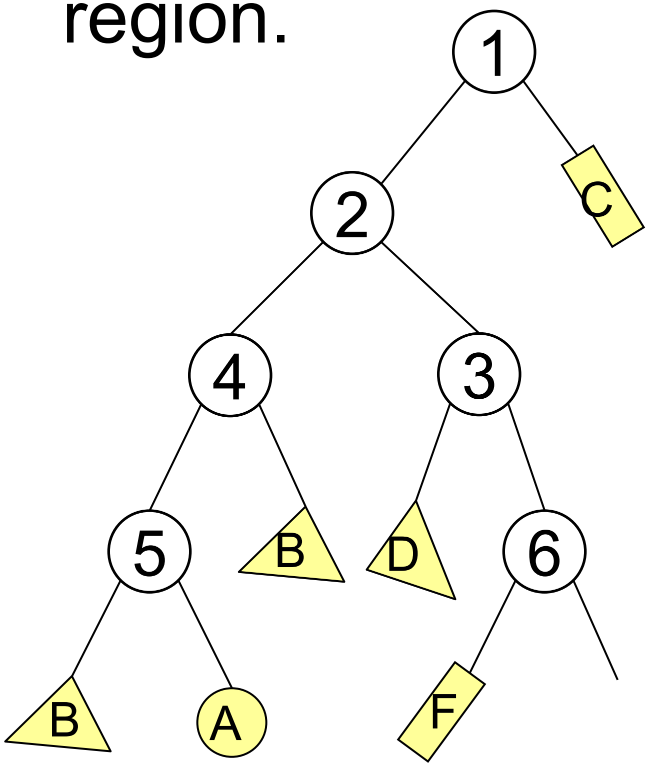


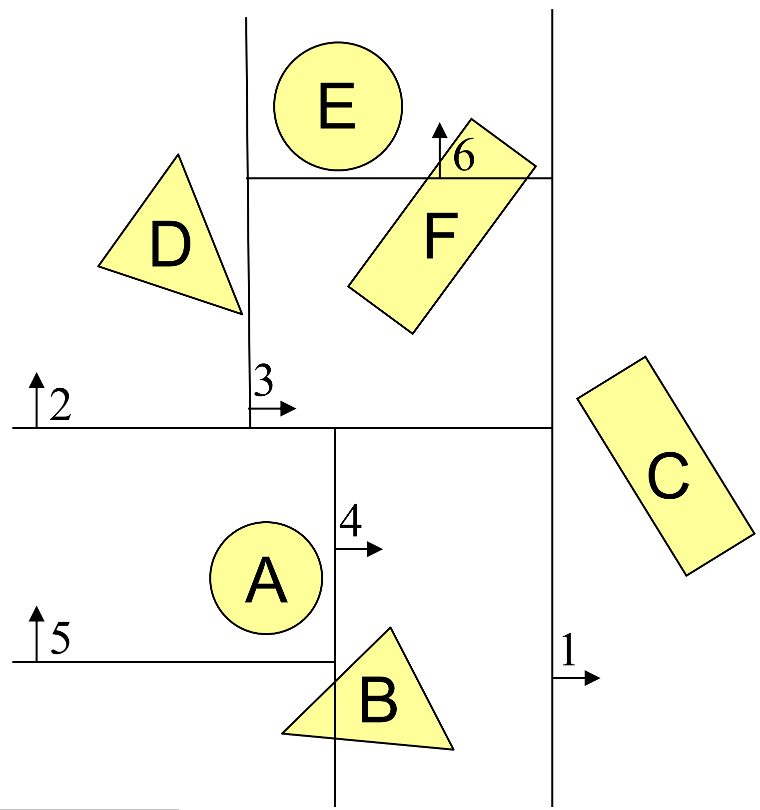


Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

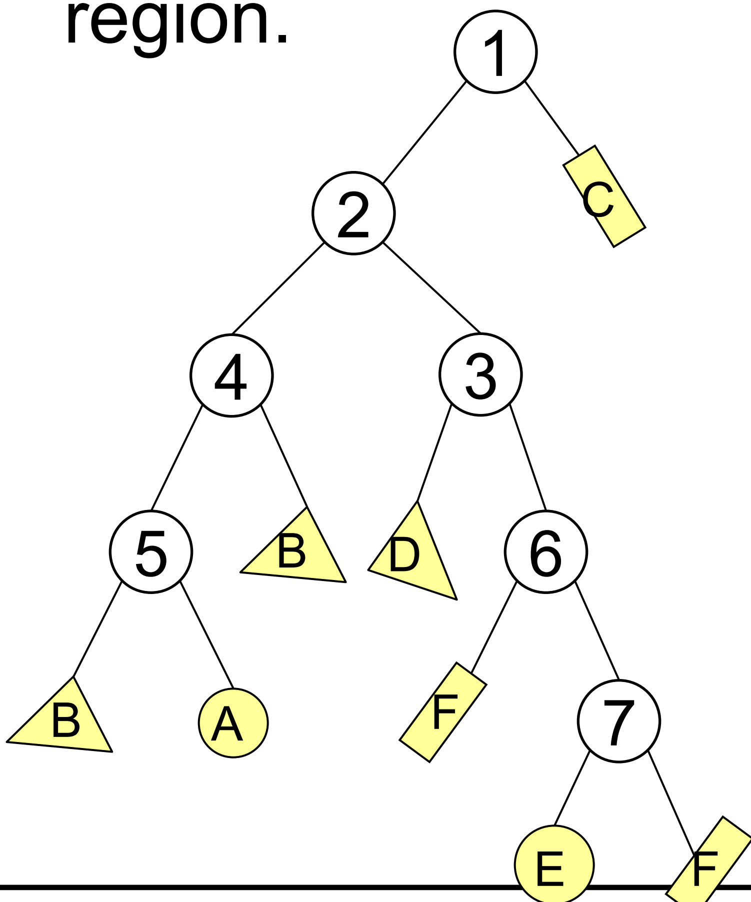


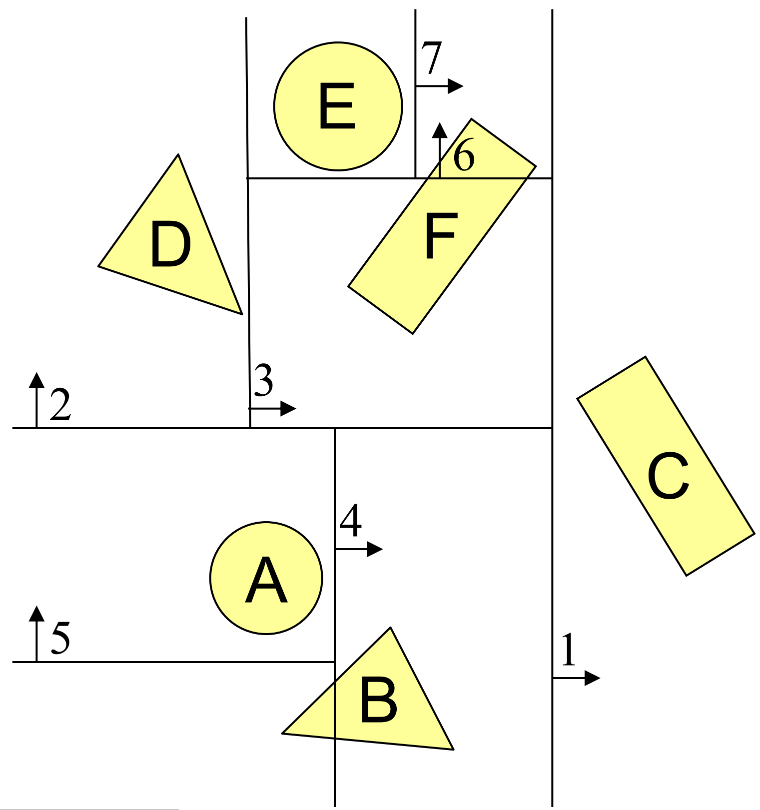


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.

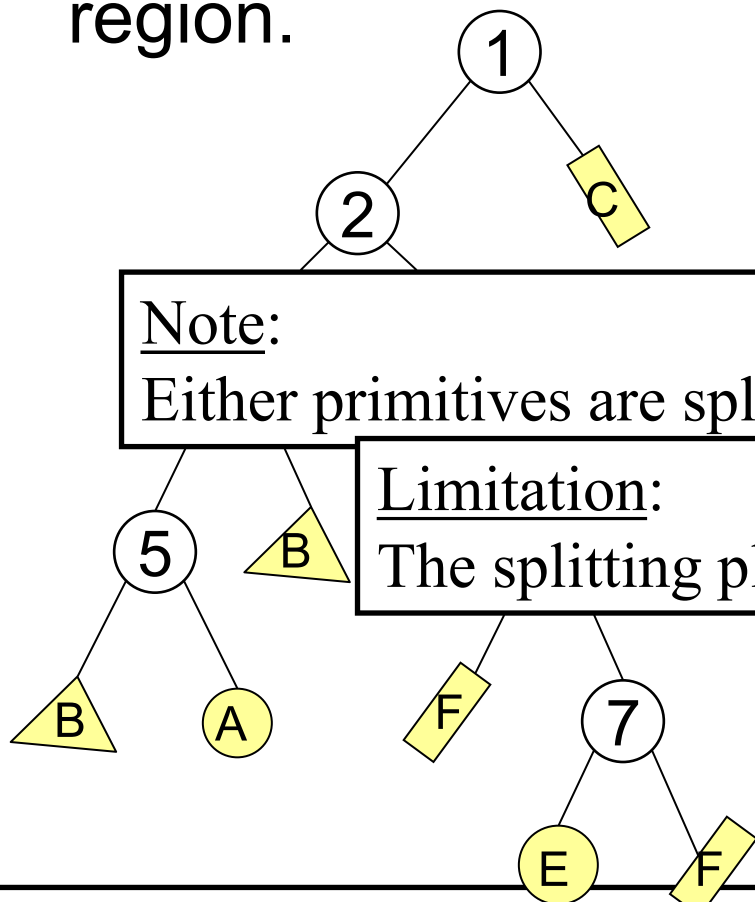


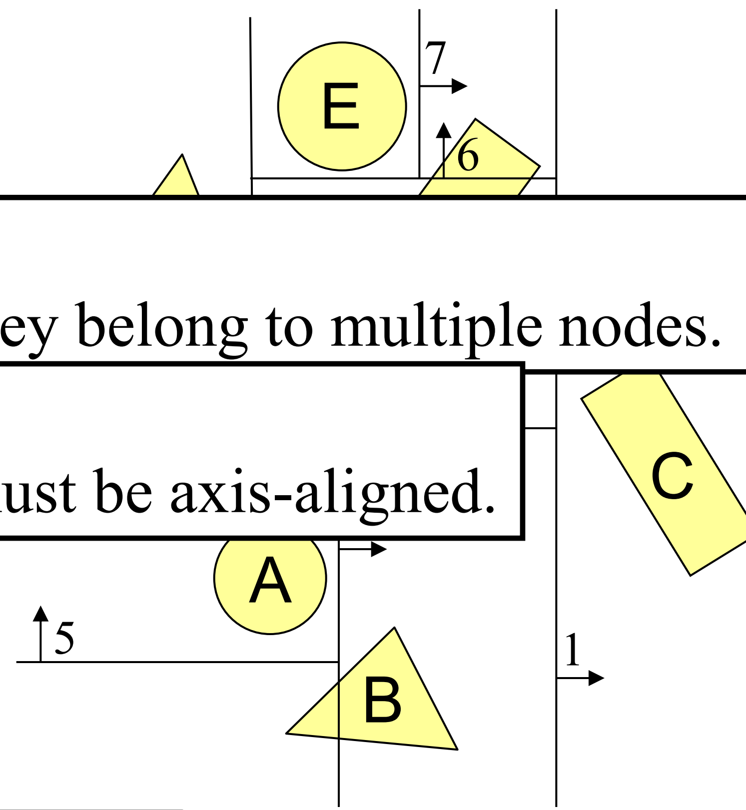


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: *k*-D Trees

Recursively split along the coordinate axes, alternating direction, until (at most) one shape per region.





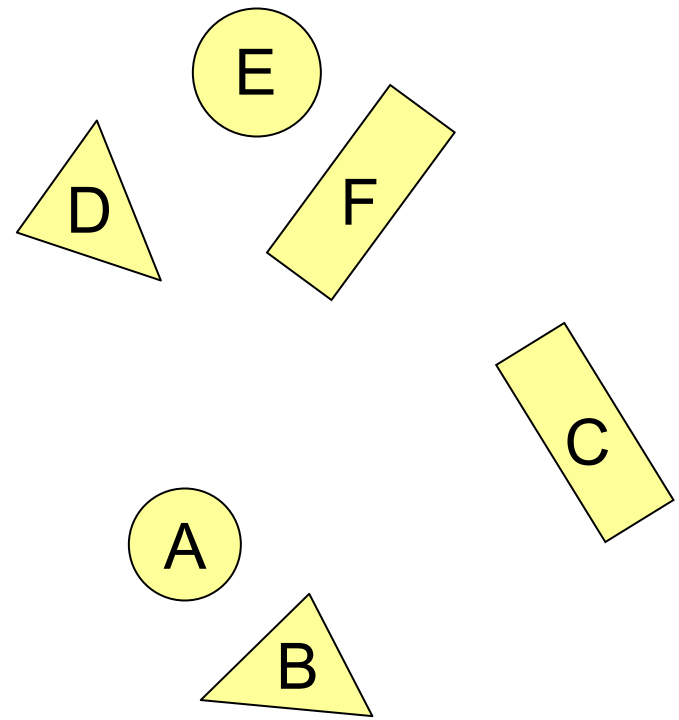
Note:
Either primitives are split, or they belong to multiple nodes.

Limitation:
The splitting planes must be axis-aligned.

Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

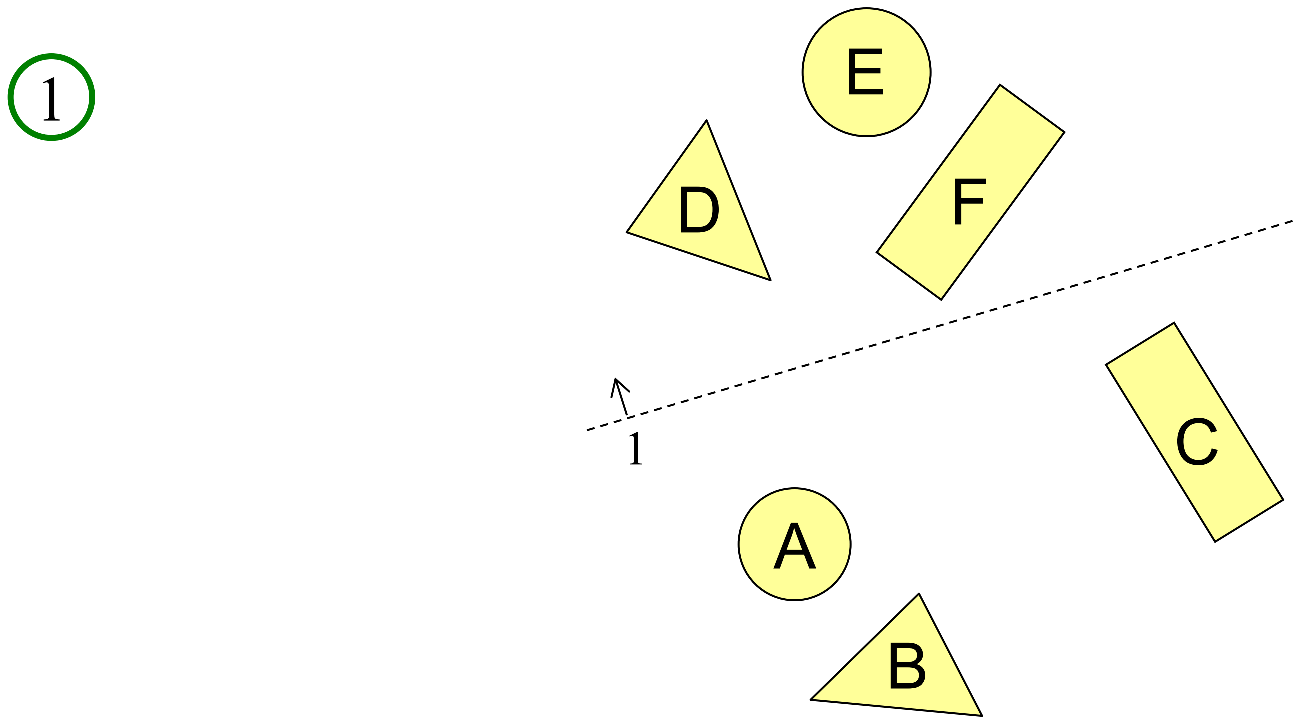


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

Generate a tree structure where leaves store shapes

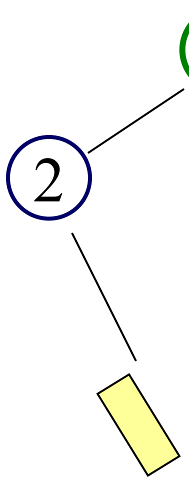


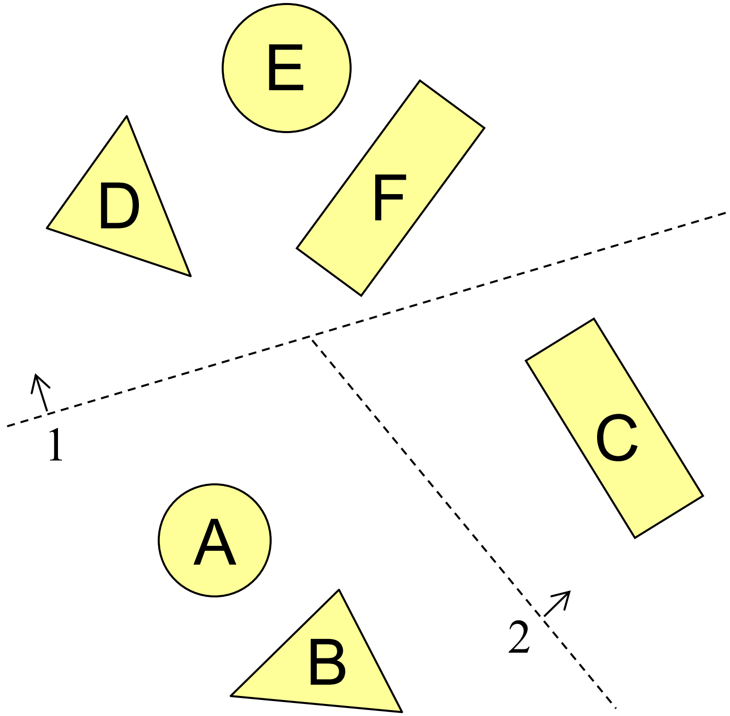
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

Generate a tree structure where leaves store shapes



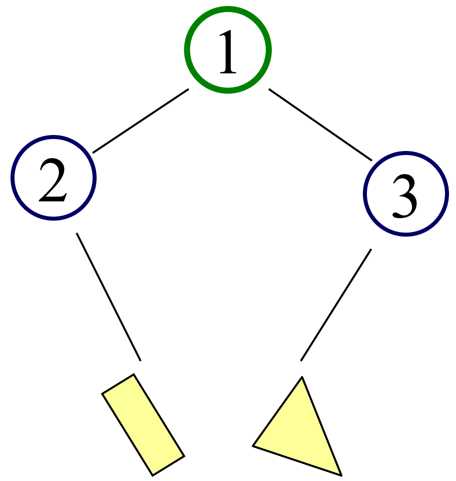


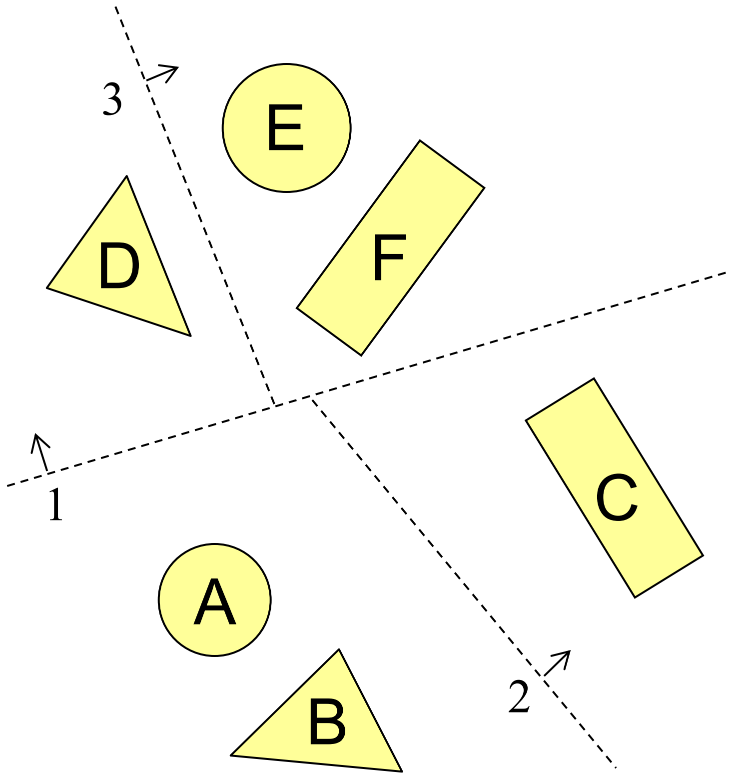
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

Generate a tree structure where leaves store shapes



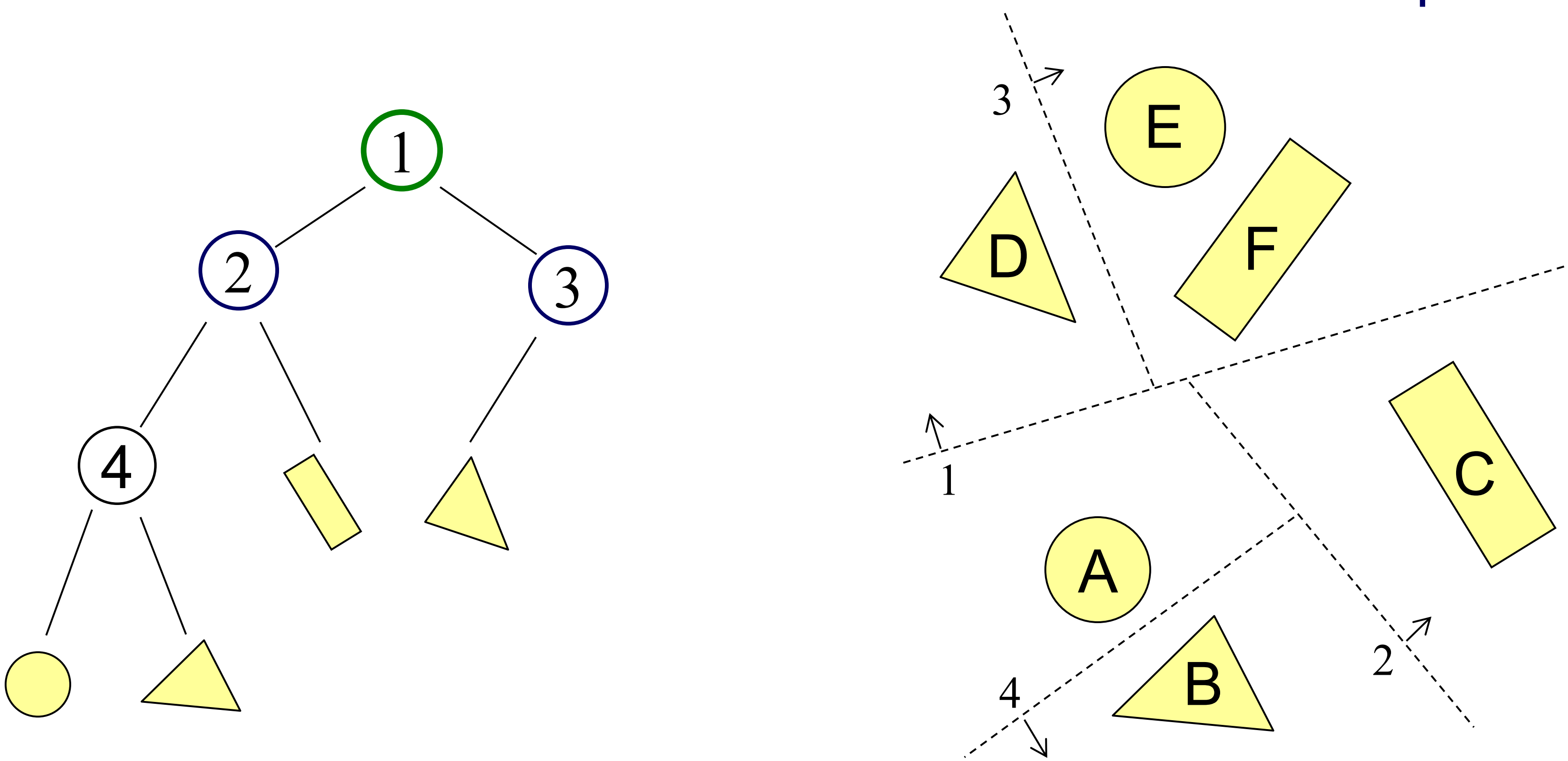


<u>Note</u>: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

Generate a tree structure where leaves store shapes

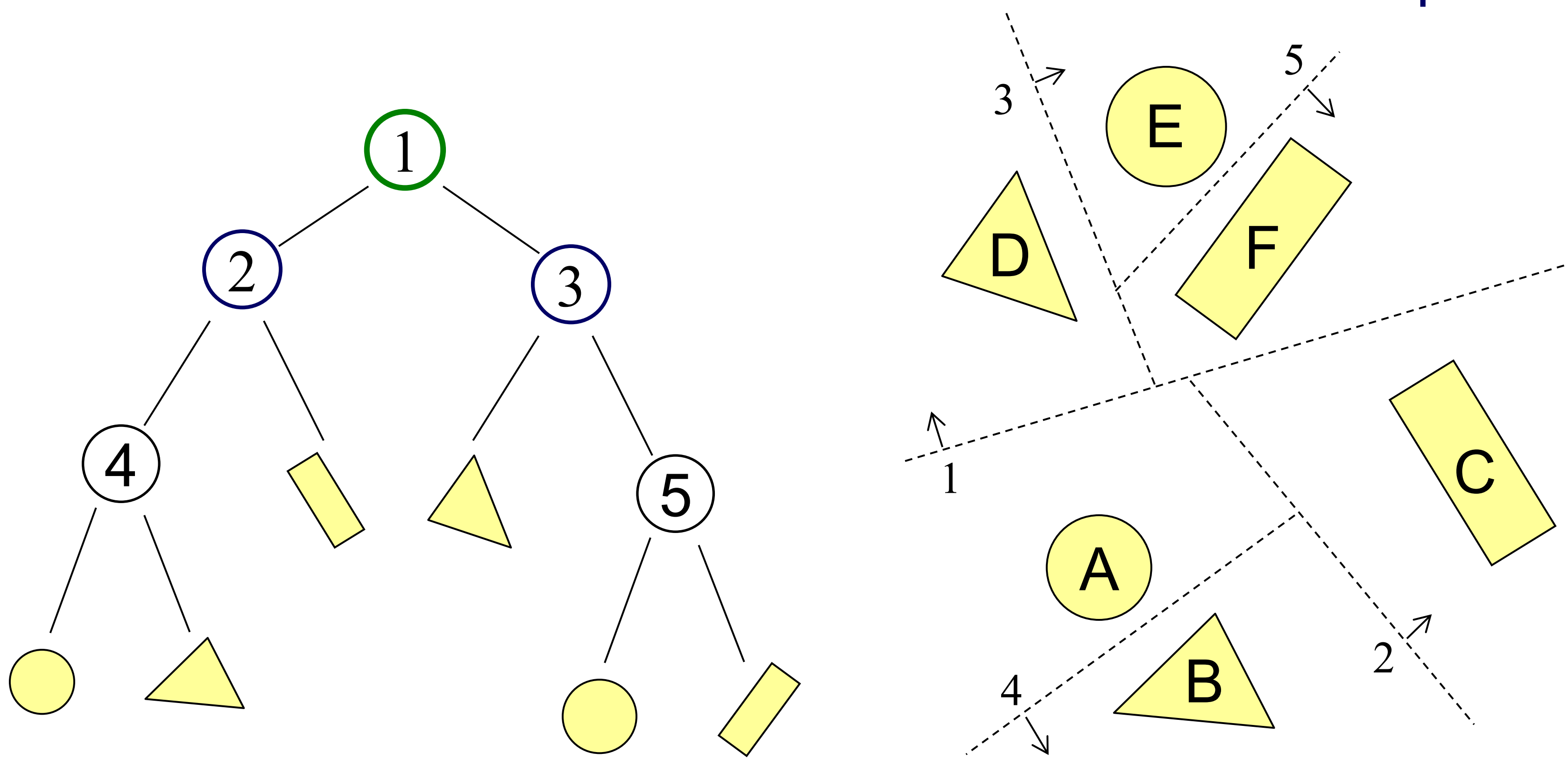


<u>Note</u>: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

With a Binary Space Partition (BSP) we recursively partition space by **arbitrarily** (aligned) planes

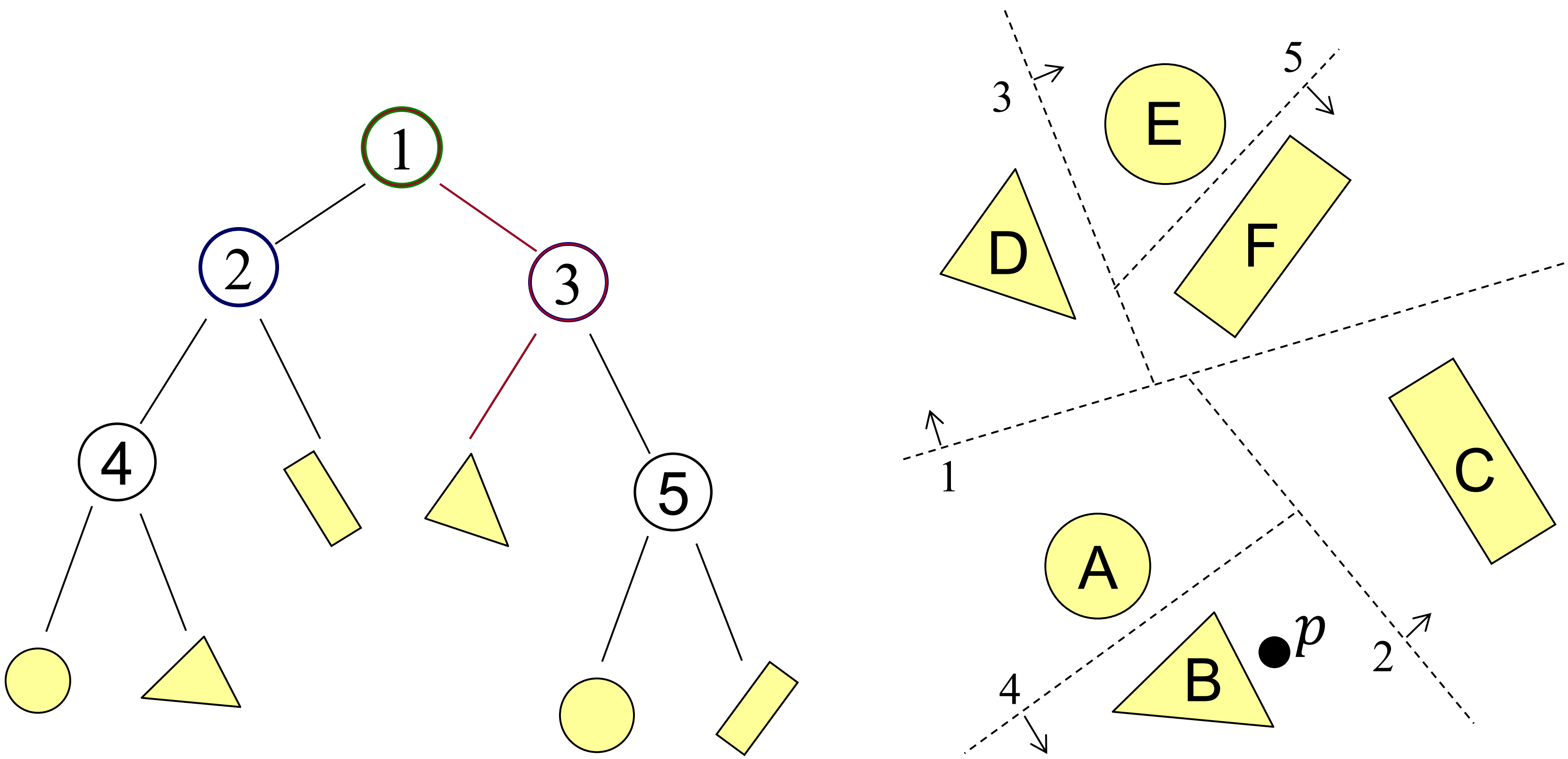
Generate a tree structure where leaves store shapes



Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

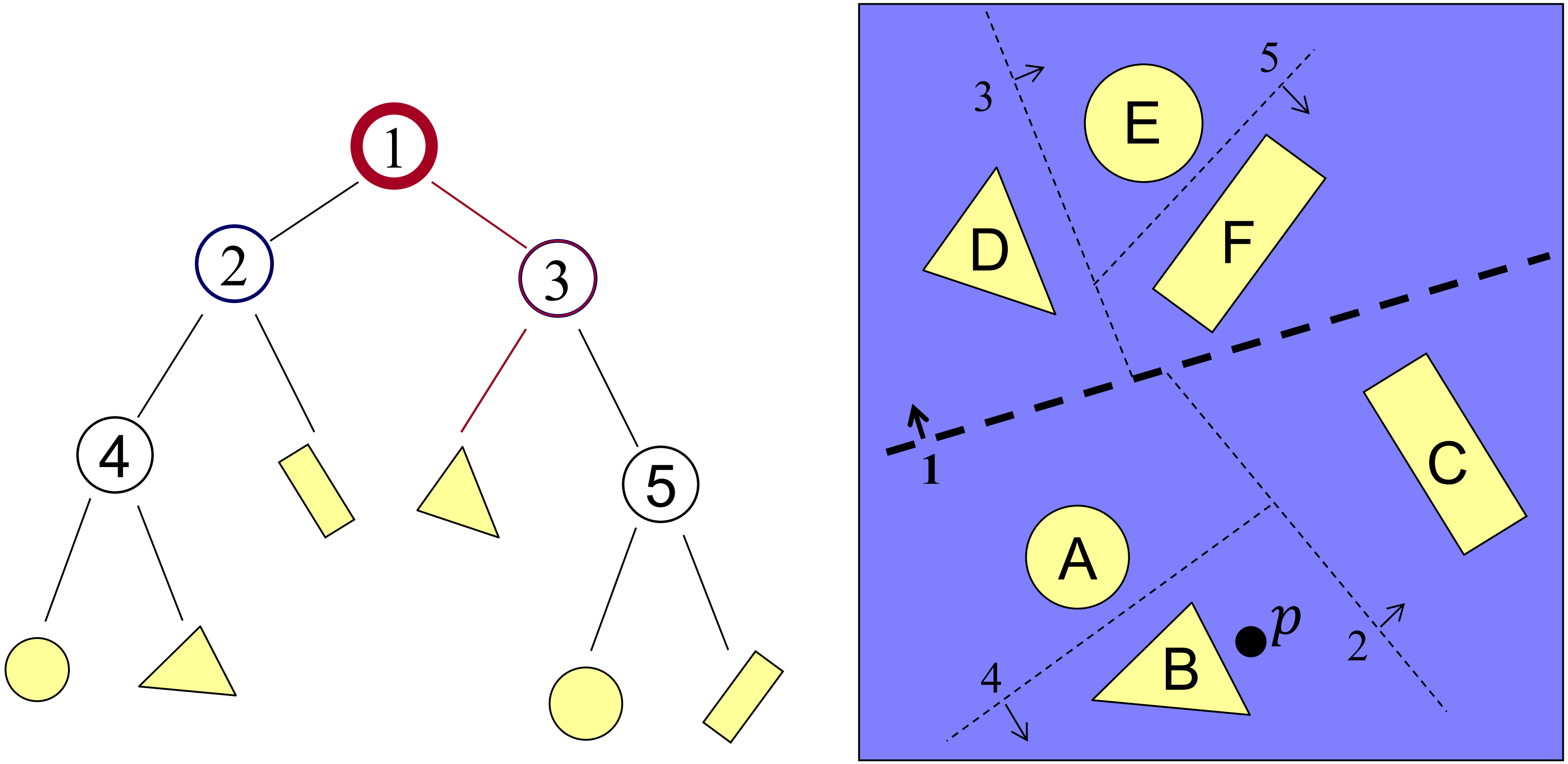


<u>Note</u>: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on

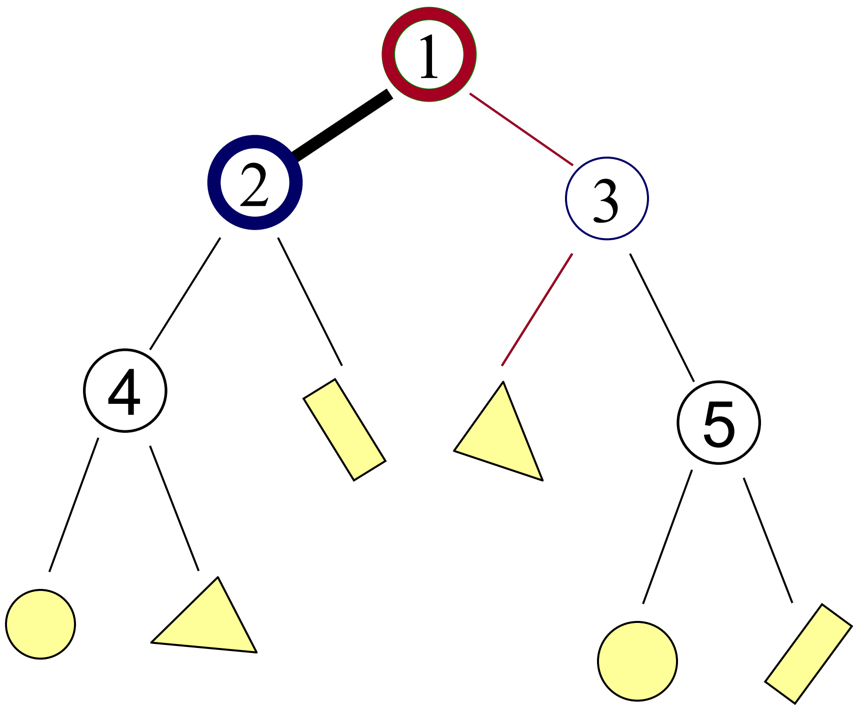


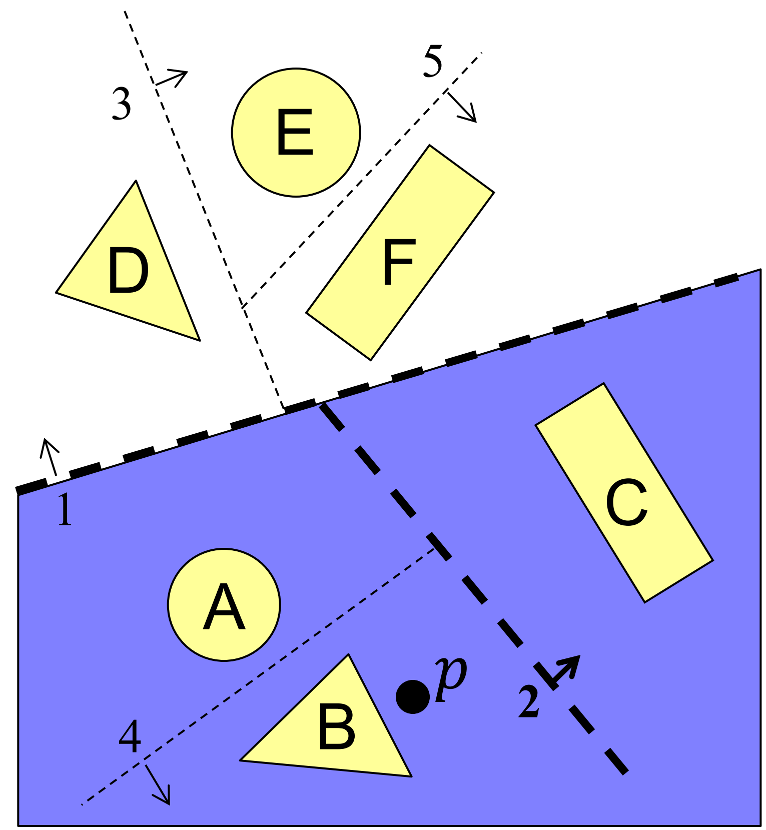
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on
  - » Left of 1 (root) → 2



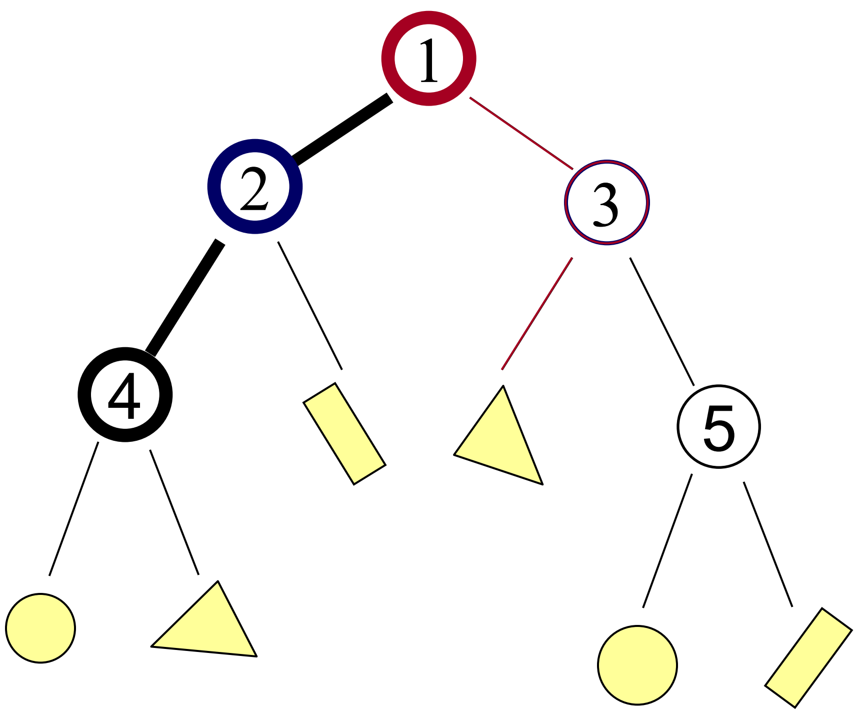


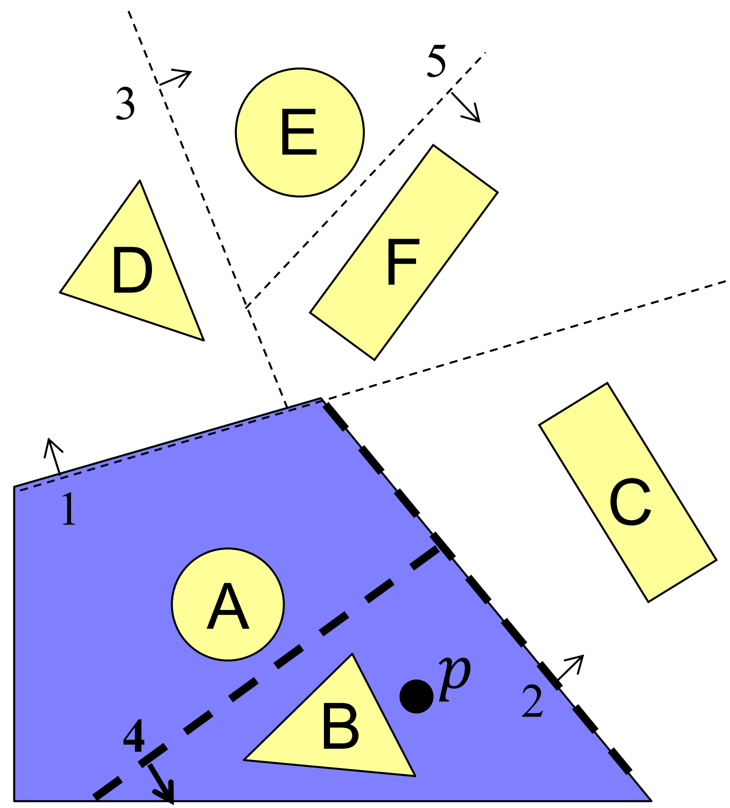
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on
  - » Left of 2 → 4



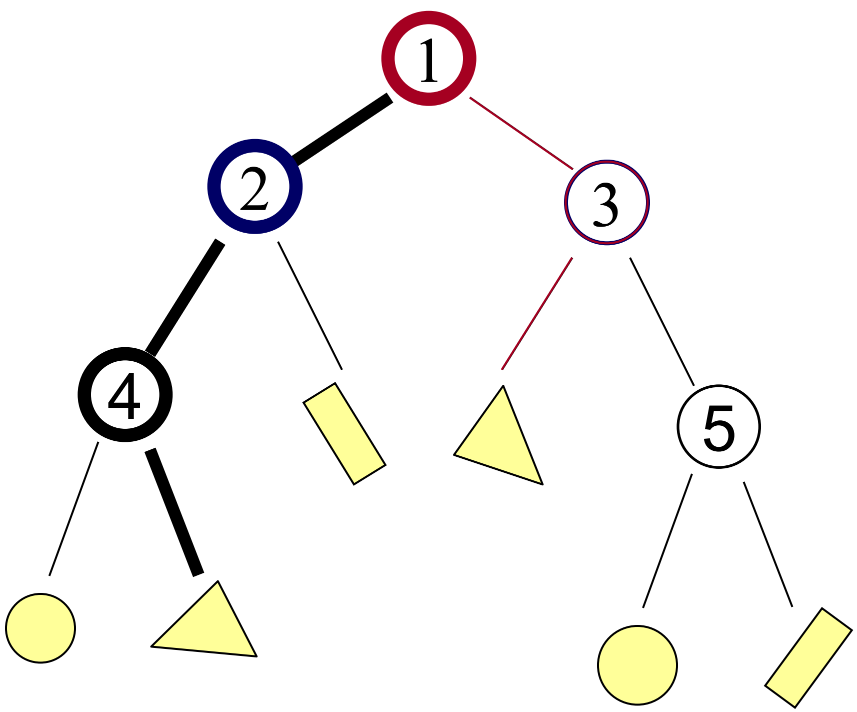


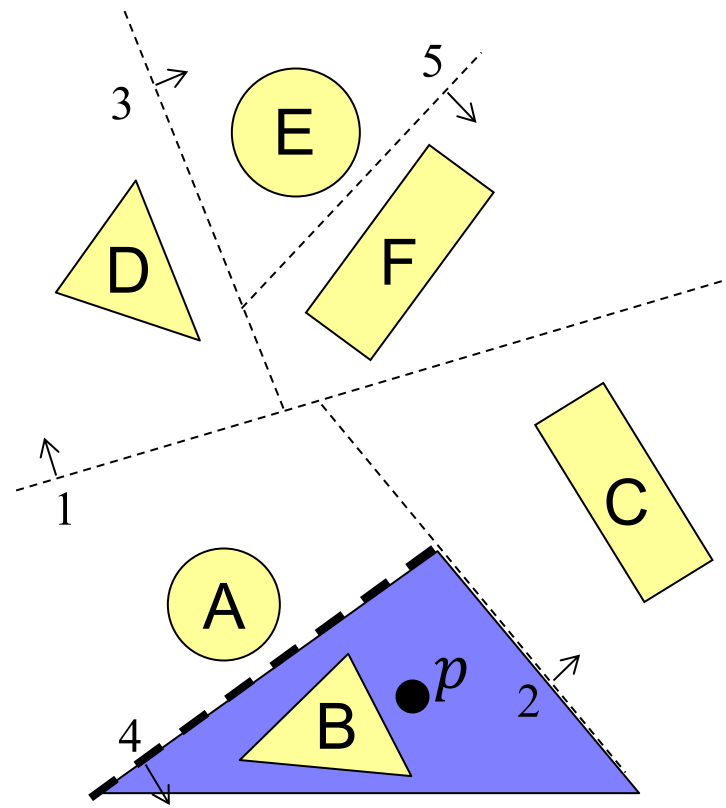
<u>Note</u>: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on
  - » Right of 4 → Test B



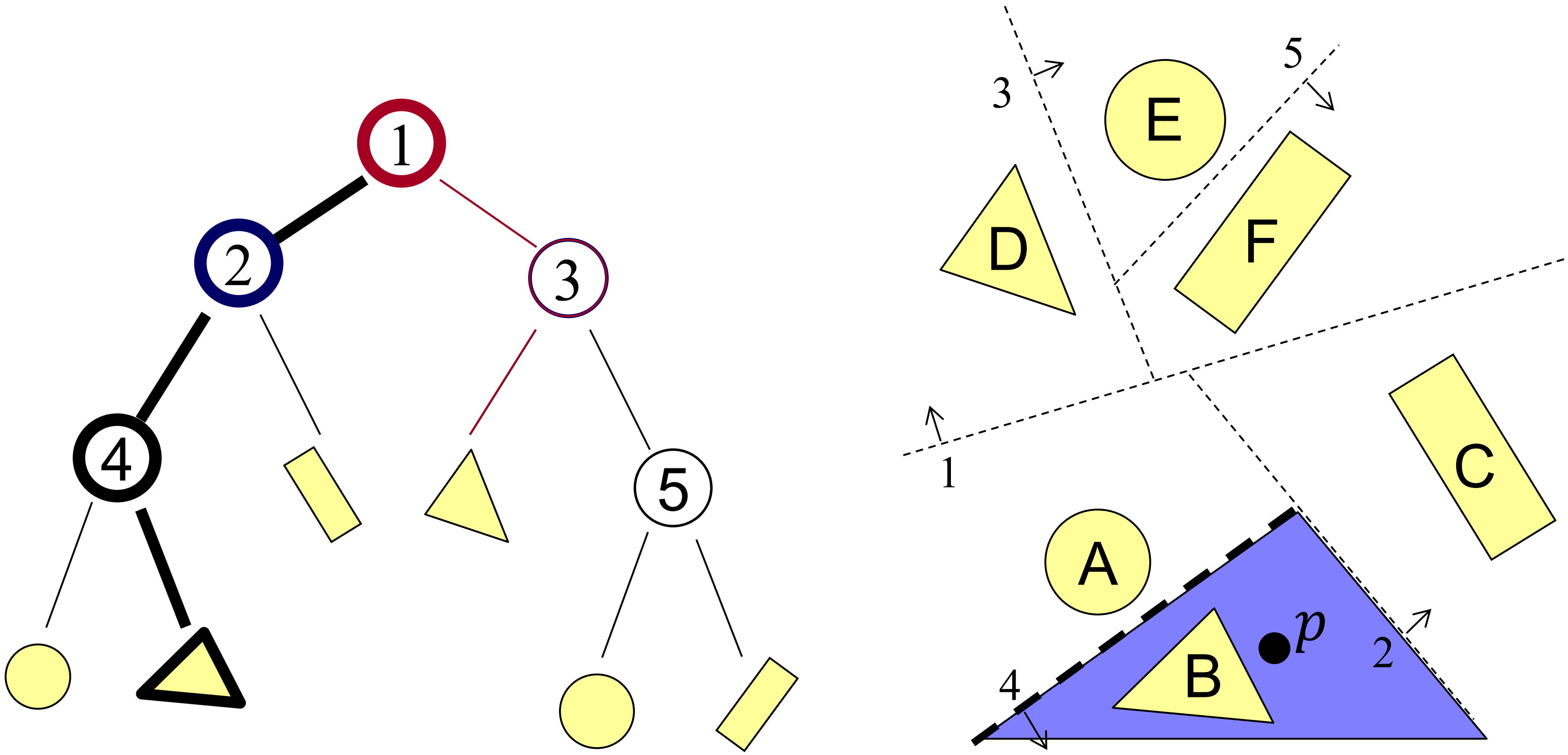


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on
  - » Missed B. No intersection!

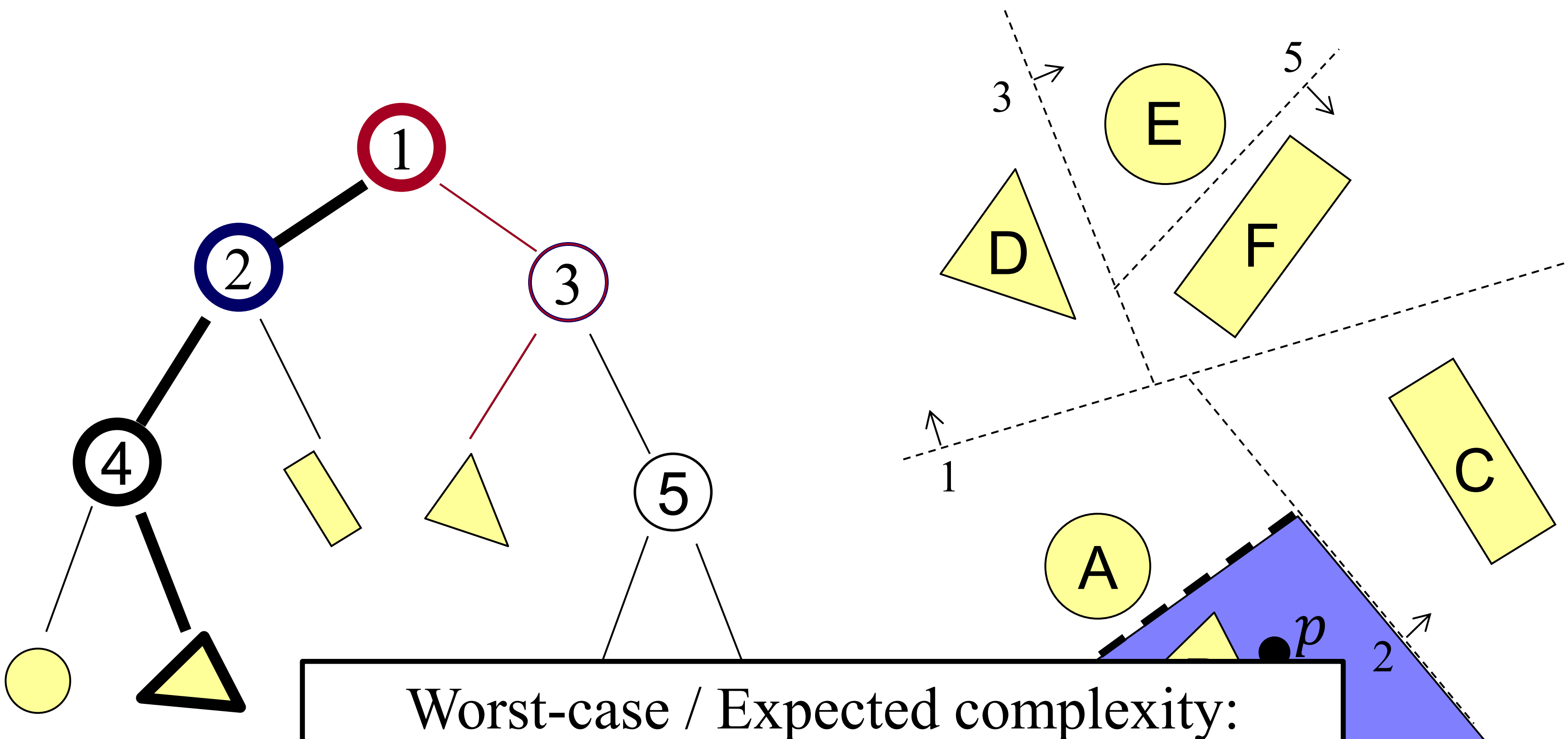


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 1: Point intersection query

- Recursively test what side we are on
  - » Missed B. No intersection!



Worst-case / Expected complexity: proportional to the depth of the tree

Note: Arrows denote

Space-Partition: BSP Tree

<u>Observation</u>:

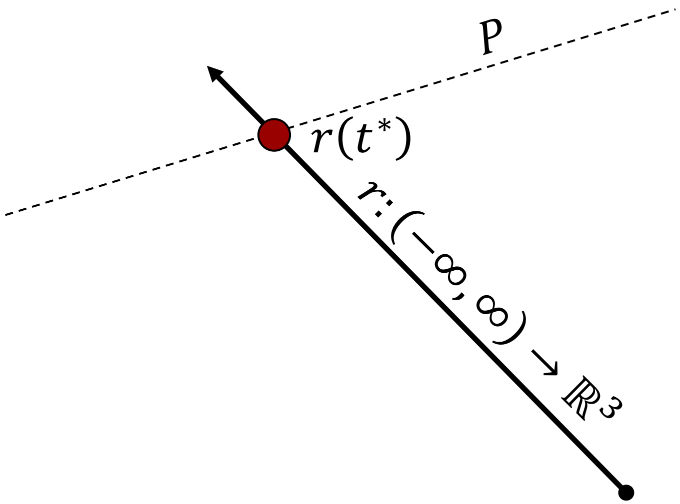
Assume we are given a ray:

$$r: (-\infty, \infty) \to \mathbb{R}^3$$

and a plane P (assuming not parallel).

There exists a time $t^* \in (-\infty, \infty)$ at which the ray passes through the plane:

$$r(t^*) \in P.$$



Space-Partition: BSP Tree

<u>Observation</u>:

Assume we are given a ray:

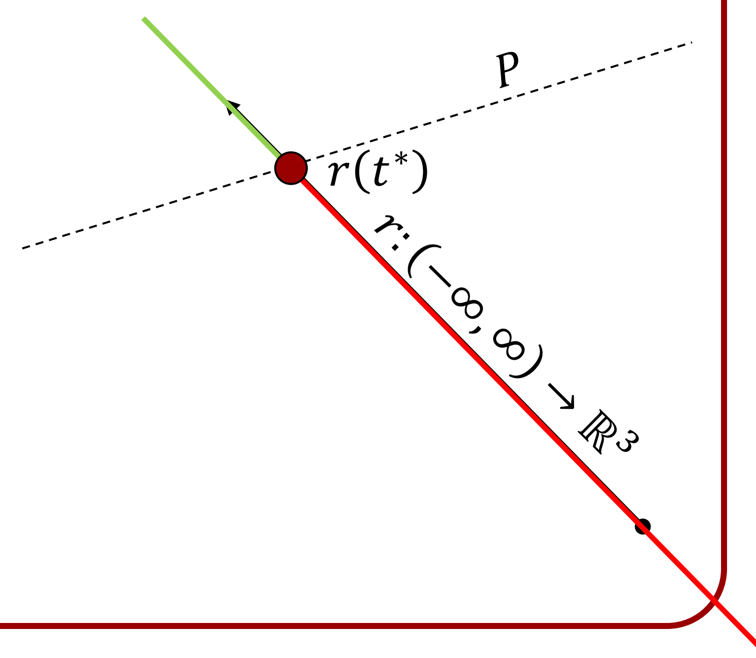
$$r: (-\infty, \infty) \to \mathbb{R}^3$$

and a plane P (assuming not parallel).

- There exists a time $t^* \in (-\infty, \infty)$ at which the ray passes through the plane:

$$r(t^*) \in P.$$

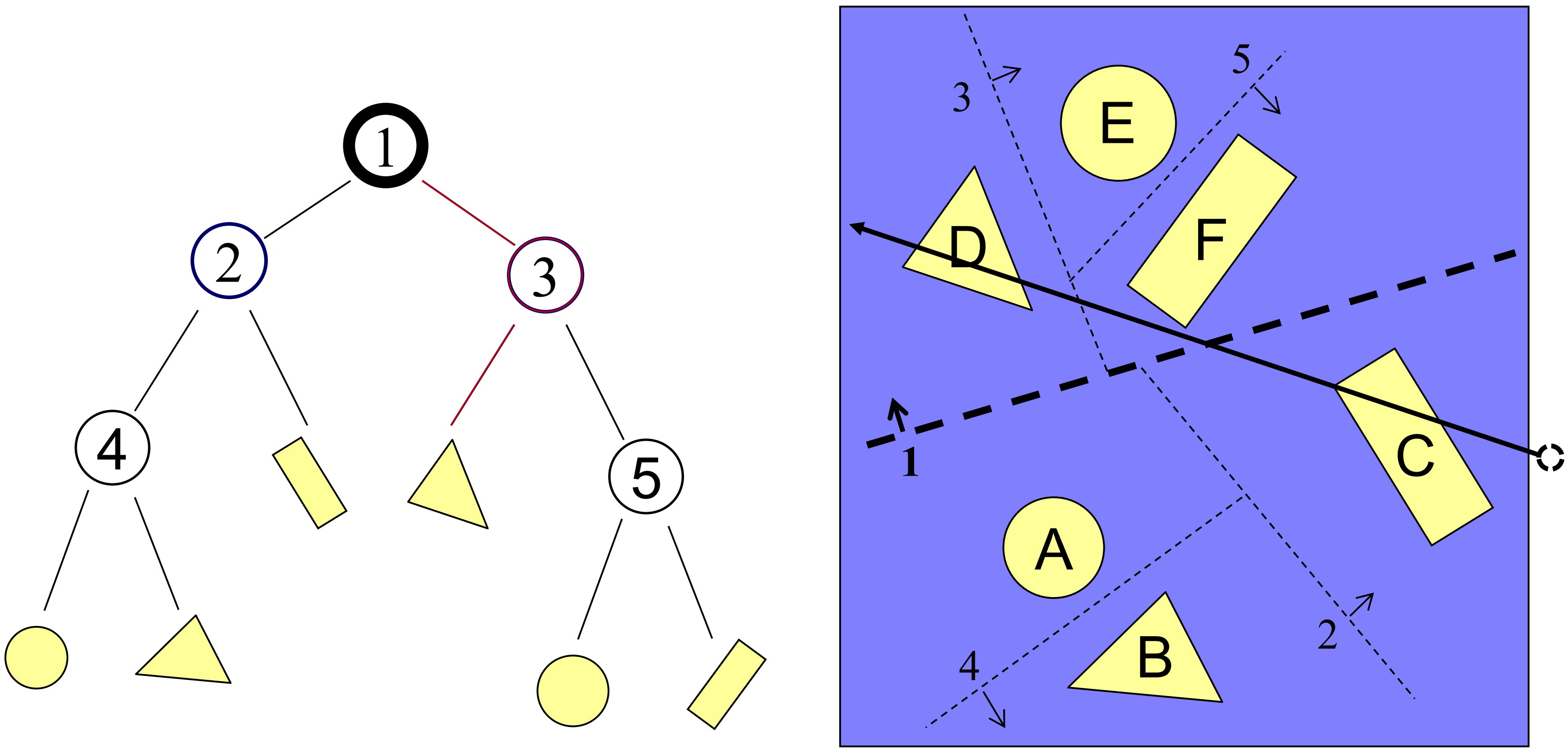
- This partitions the line containing the ray in two parts:
  - » Back: $r(t)$ with $t \in (-\infty, t^*)$
  - » Front: $r(t)$ with $t \in (t^*, \infty)$



Space-Partition: BSP Tree

Example 2: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:



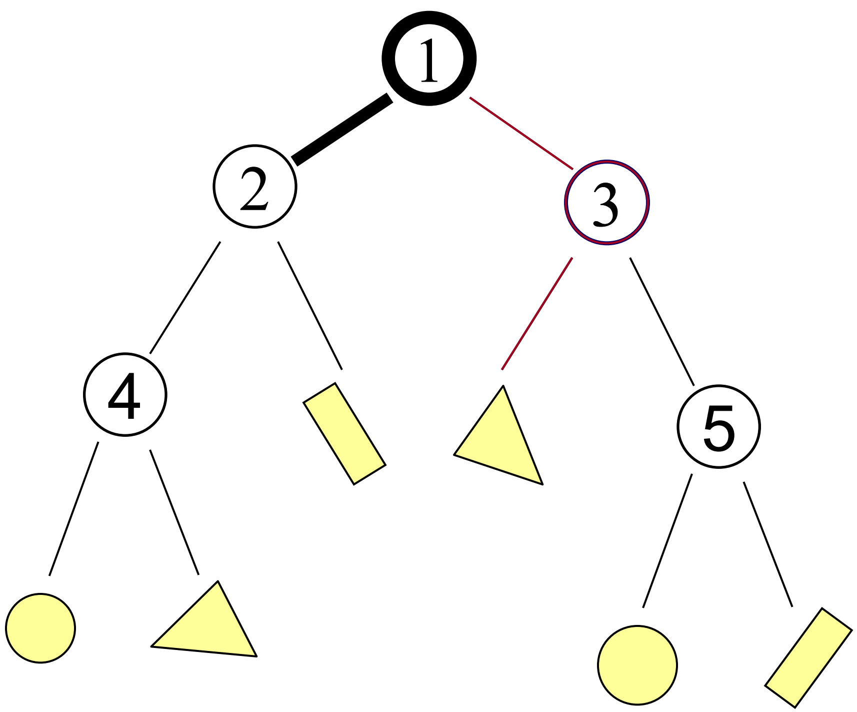
Note: Arrows denote the "right" side of the splitting plane.

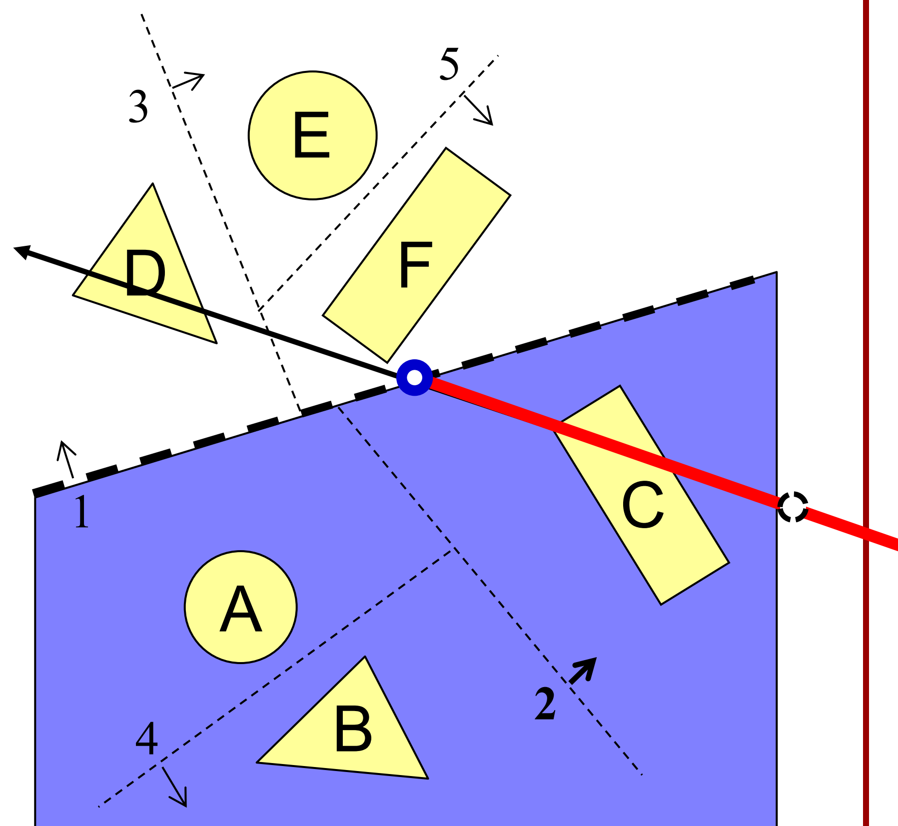
Space-Partition: BSP Tree

Example 2: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Test back (left)



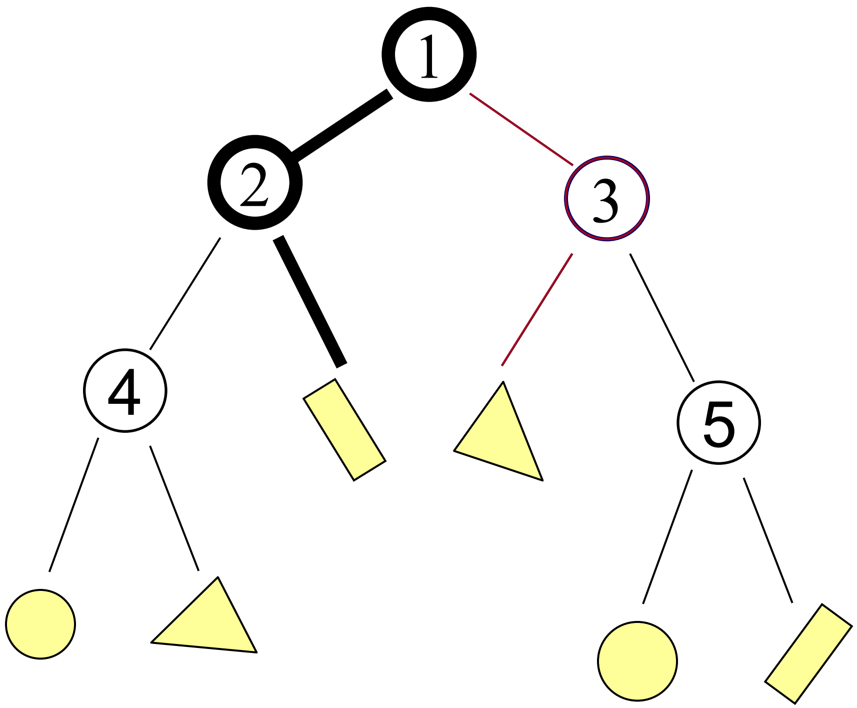


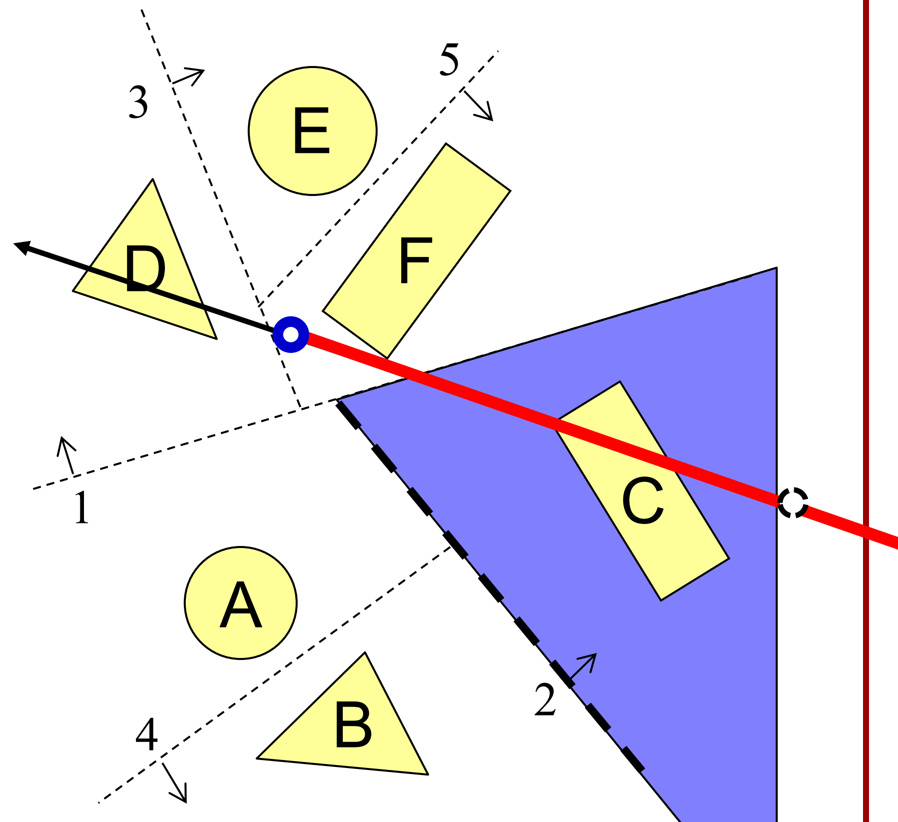
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 2: Ray intersection query

- Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
  - » Test back (right)





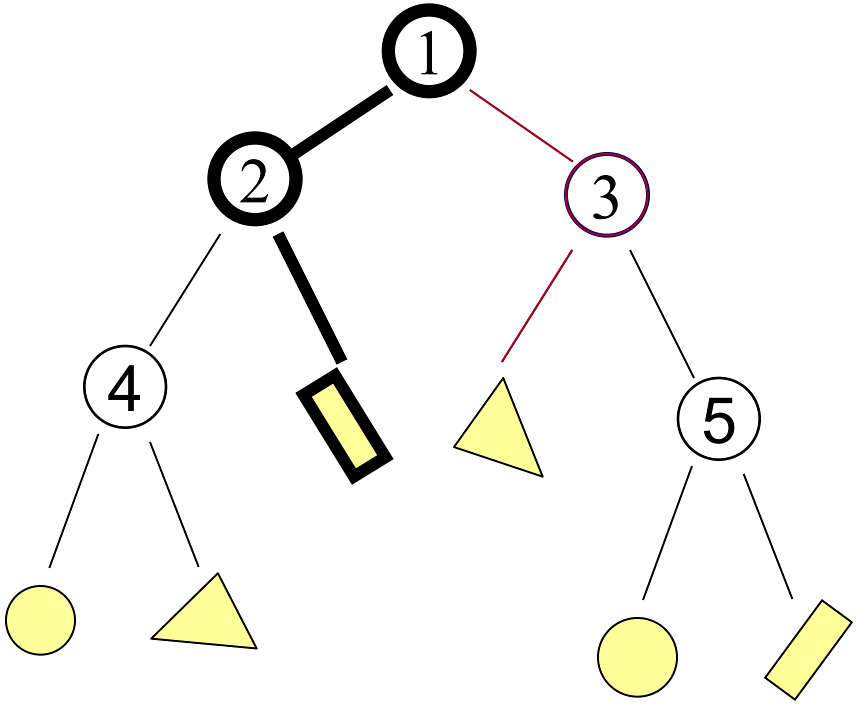
Note: Arrows denote the "right" side of the splitting plane.

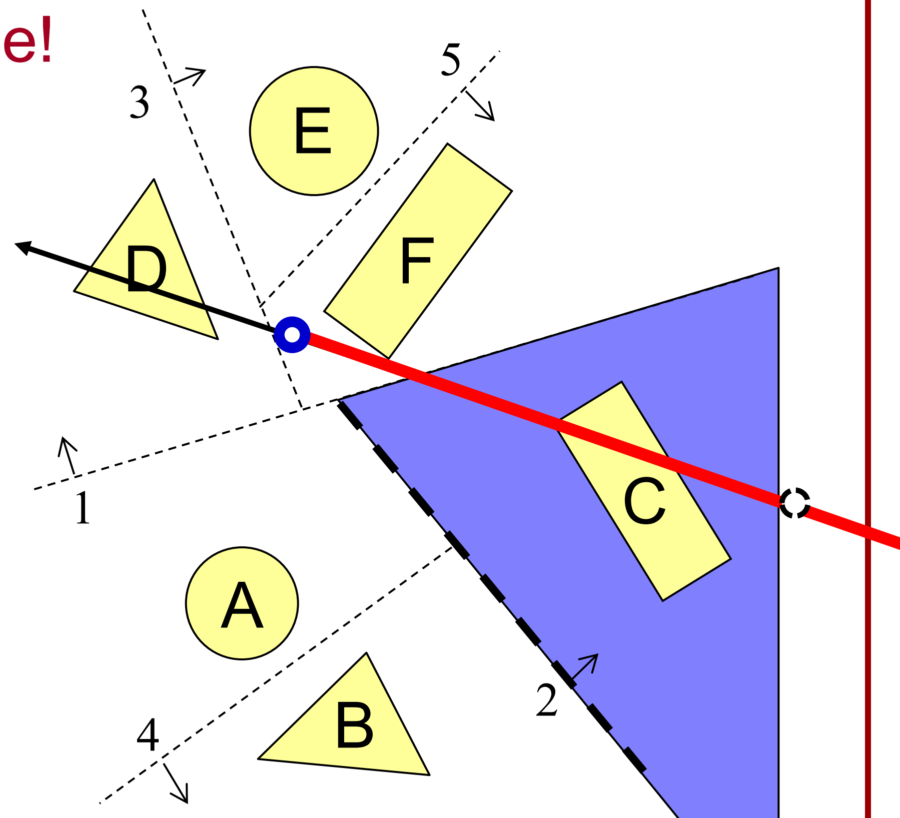
Space-Partition: BSP Tree

Example 2: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Intersection with C. Done!



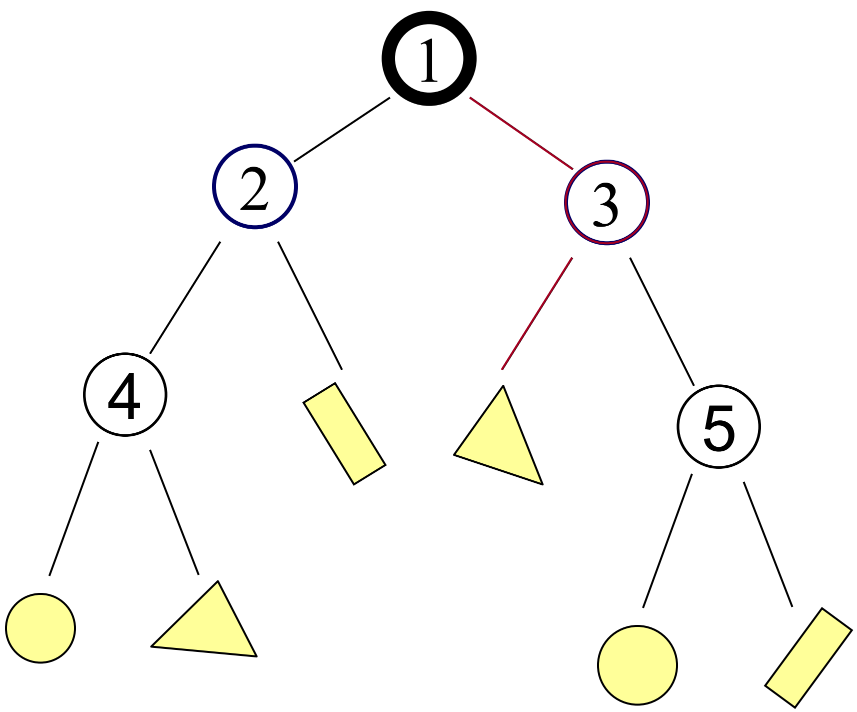


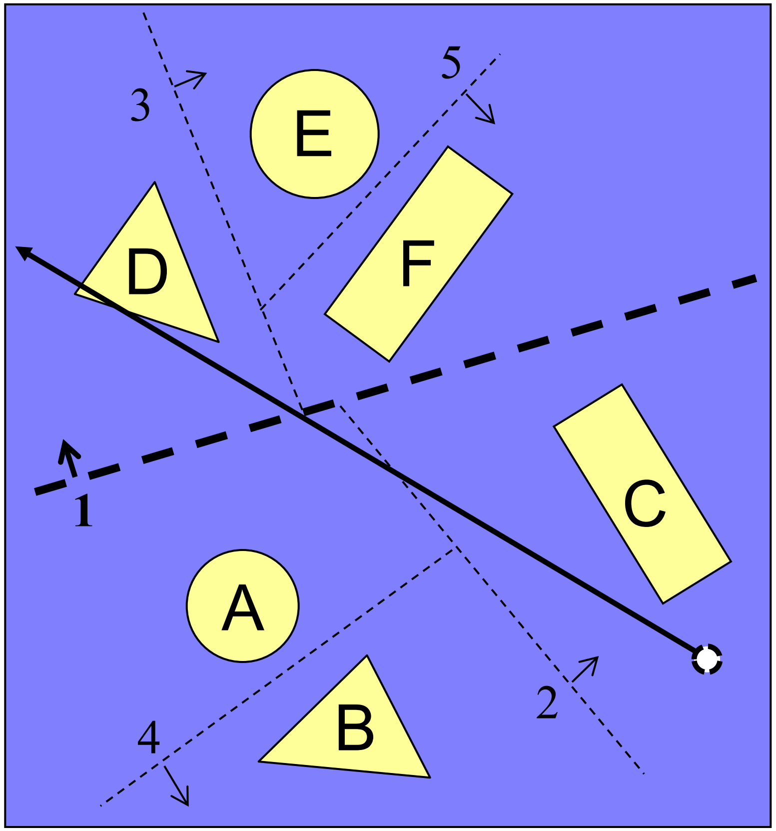
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:





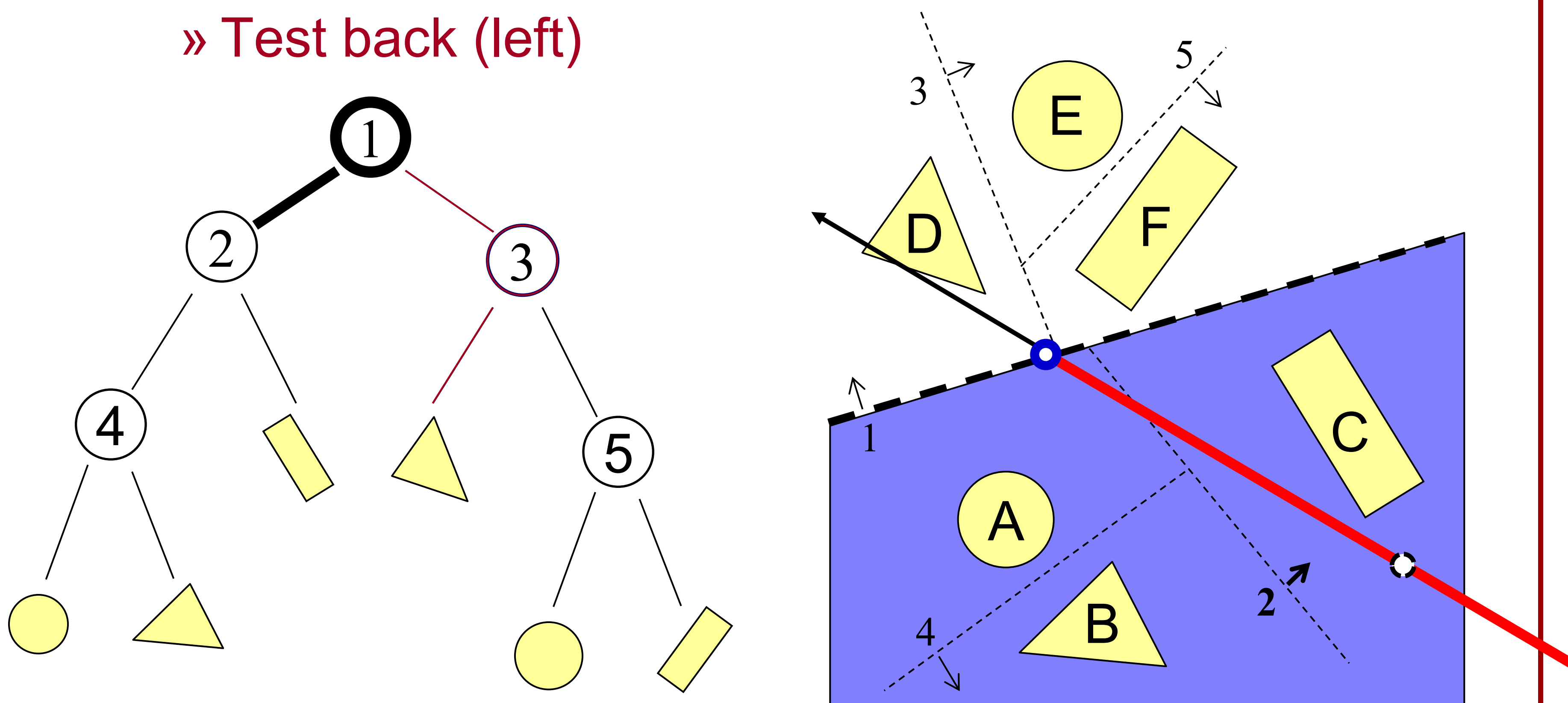
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Test back (left)



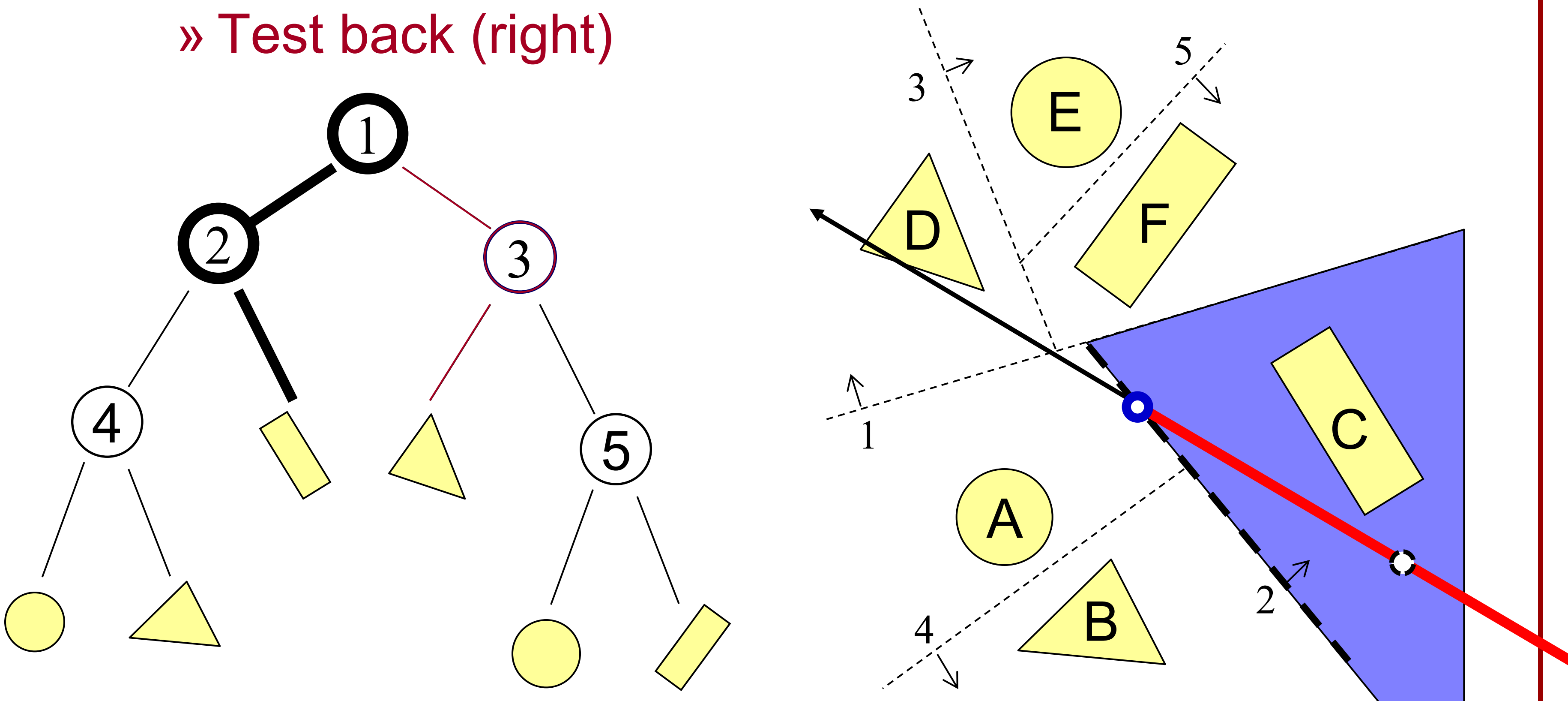
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Test back (right)



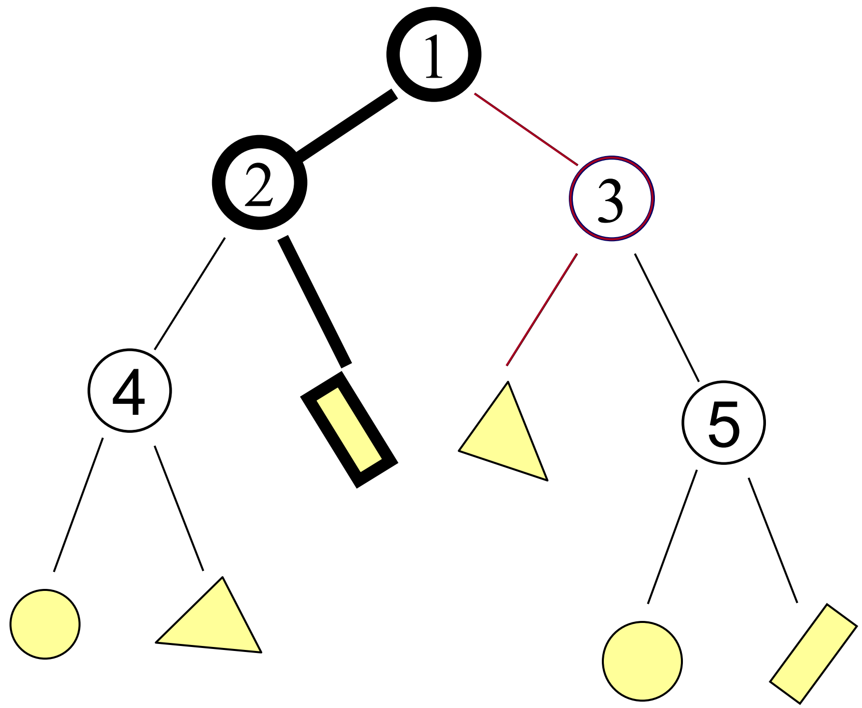
Note: Arrows denote the "right" side of the splitting plane.

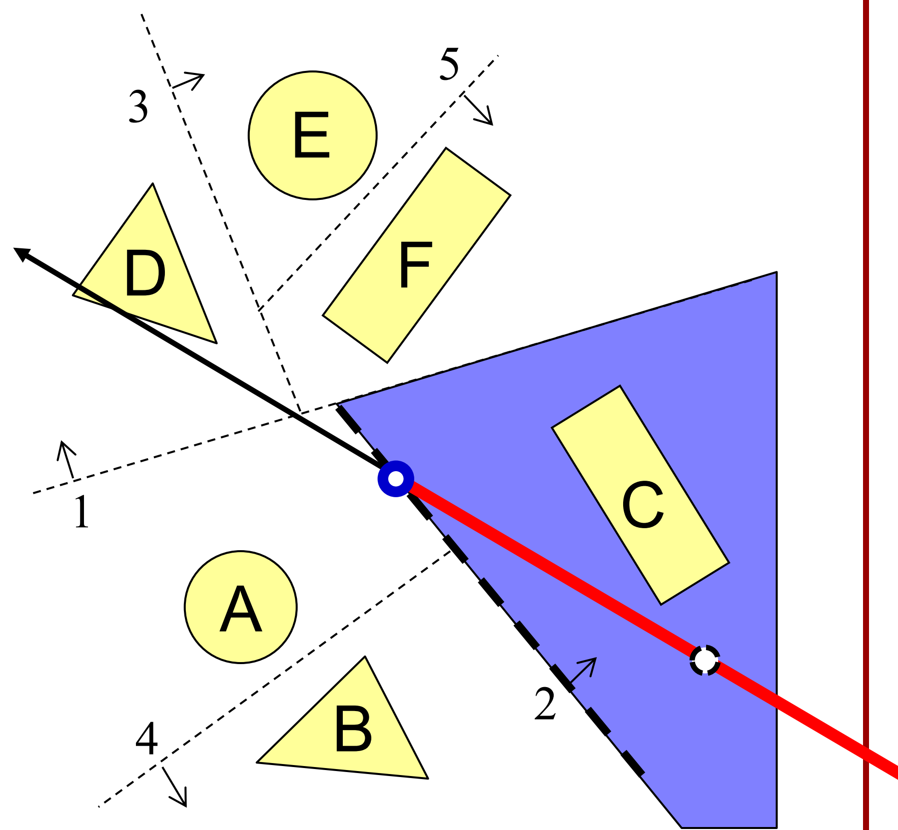
Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Missed C. Recurse!





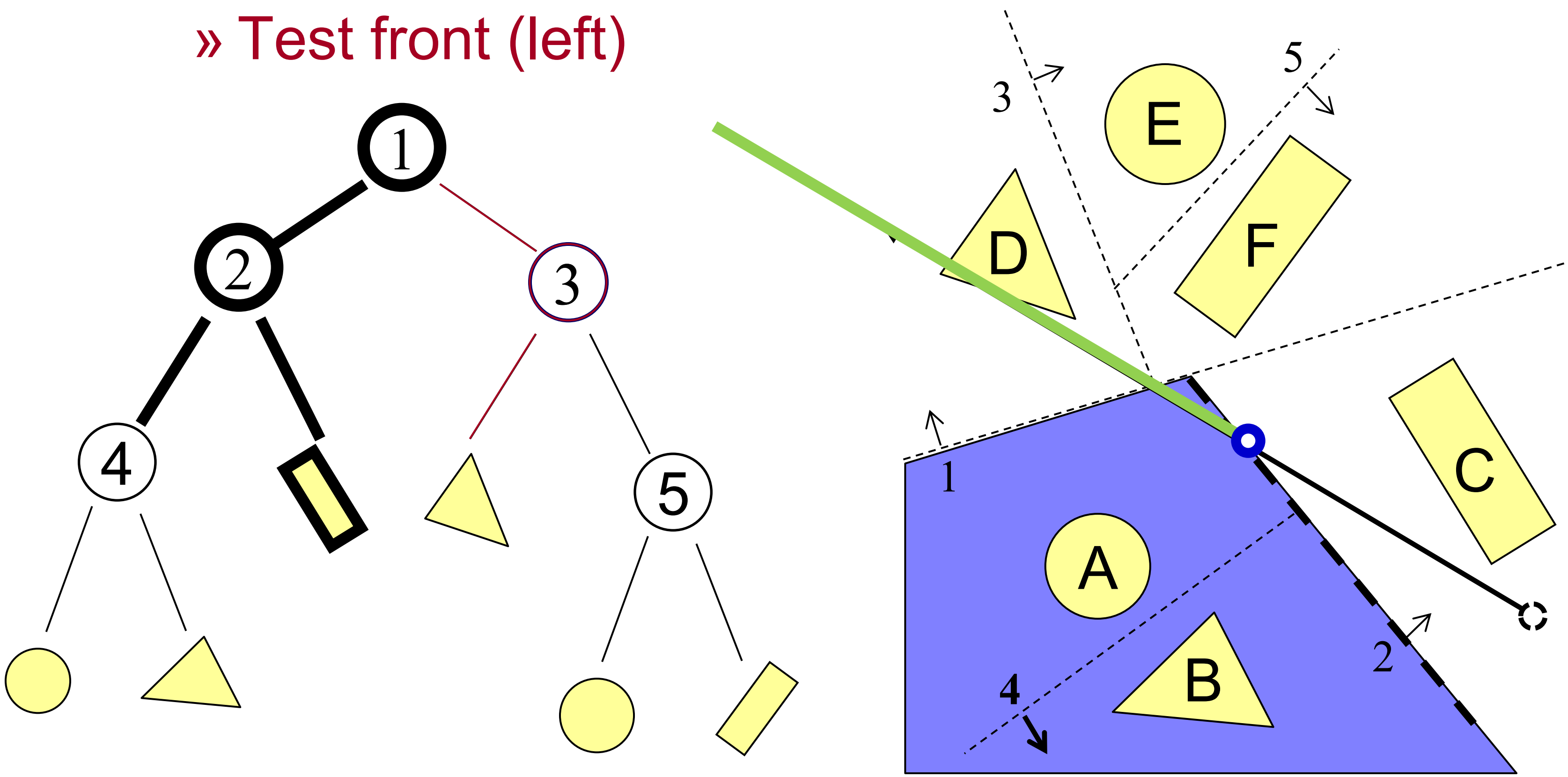
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Test front (left)



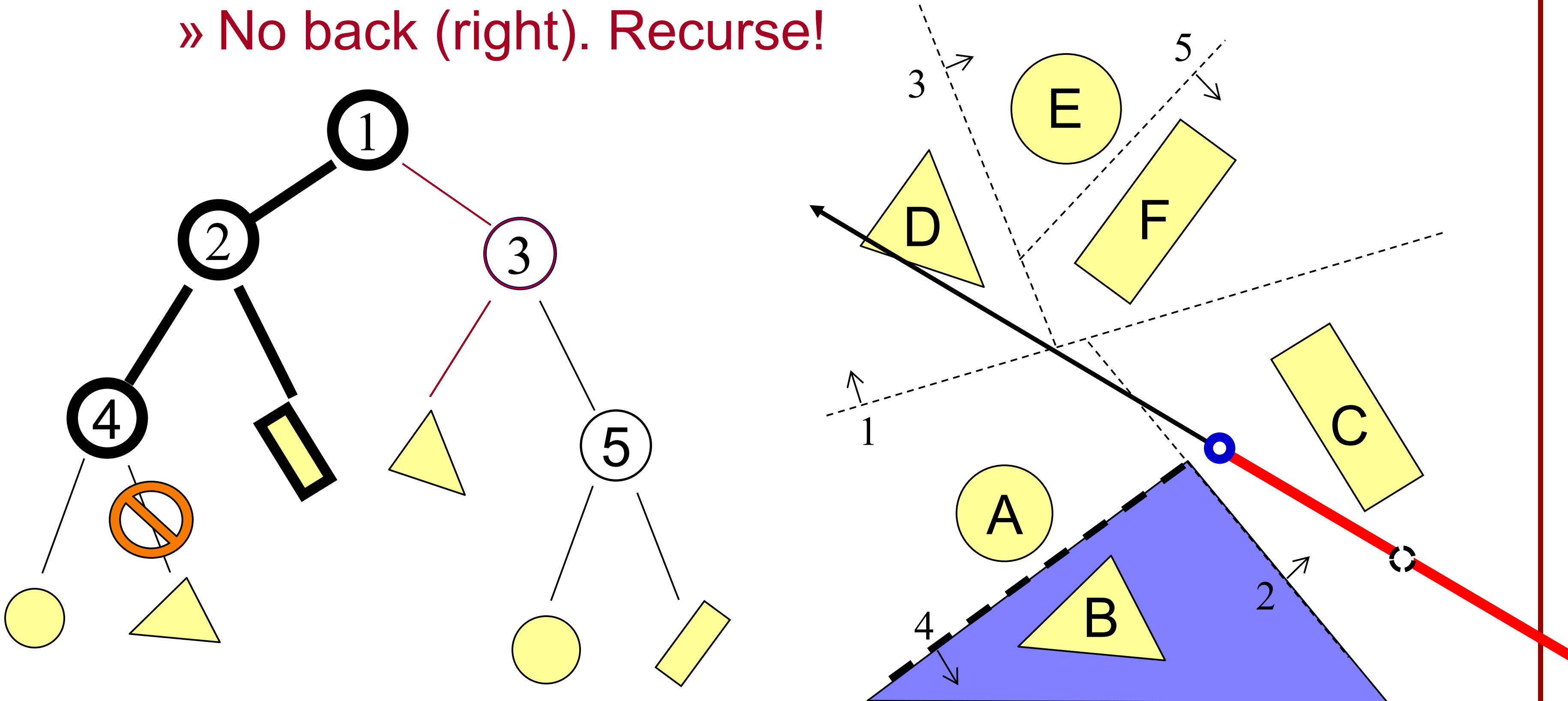
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» No back (right). Recurse!



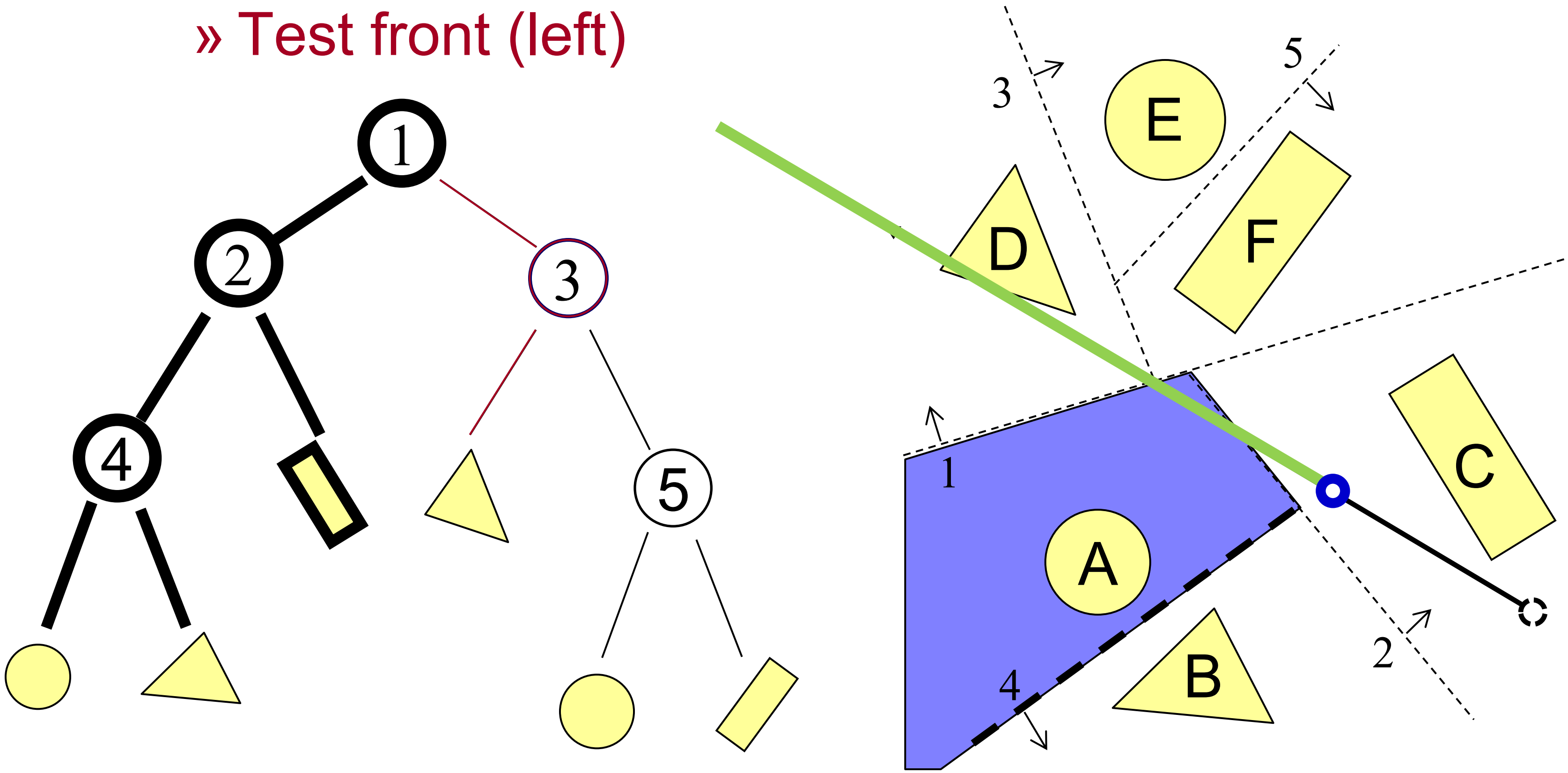
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Test front (left)



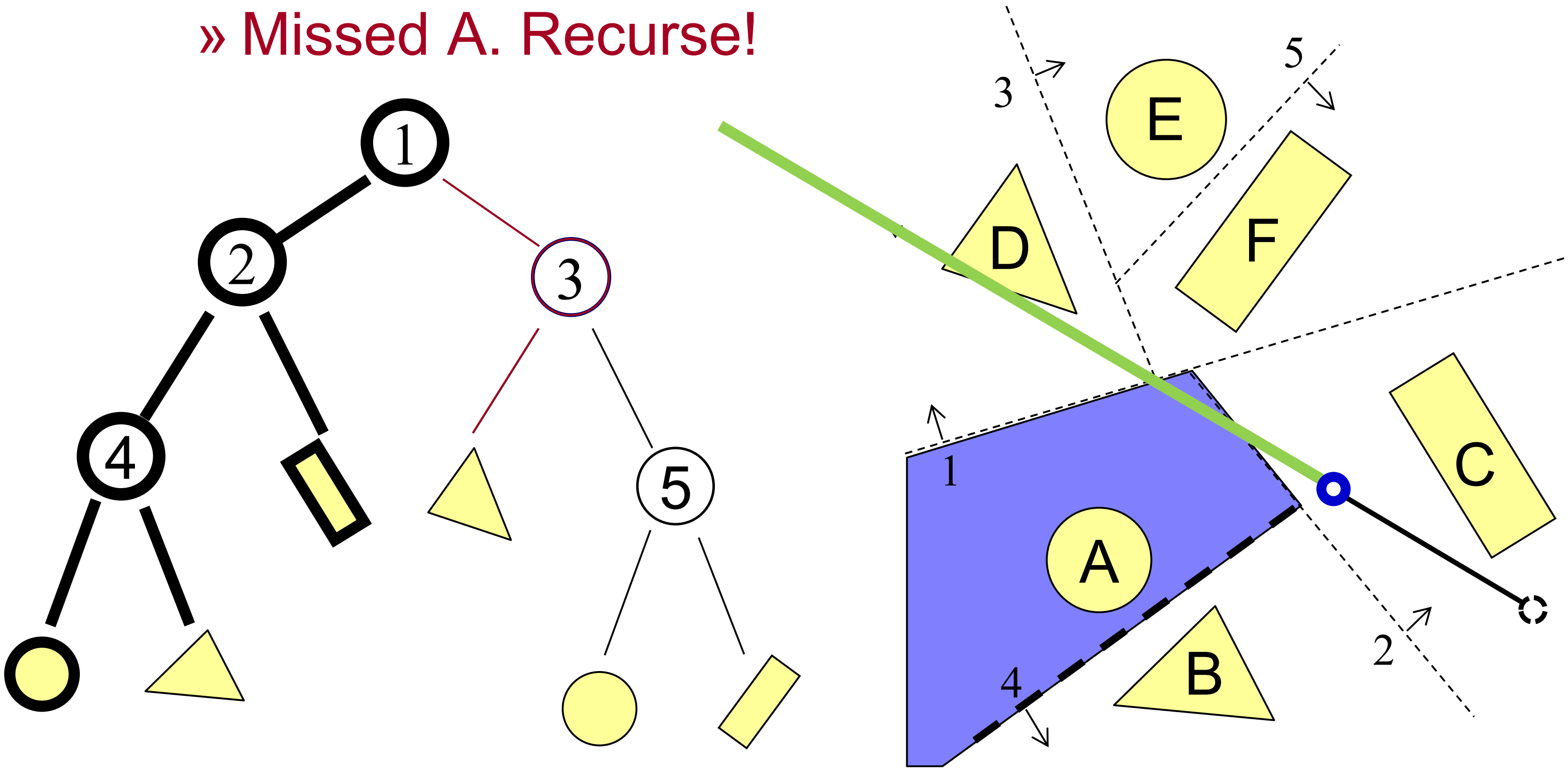
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Missed A. Recurse!

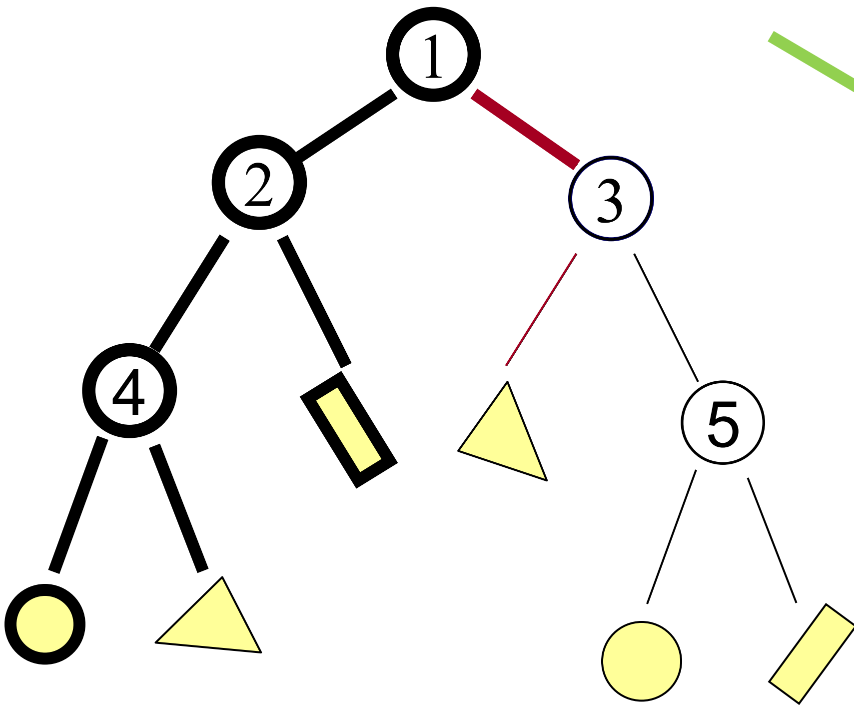


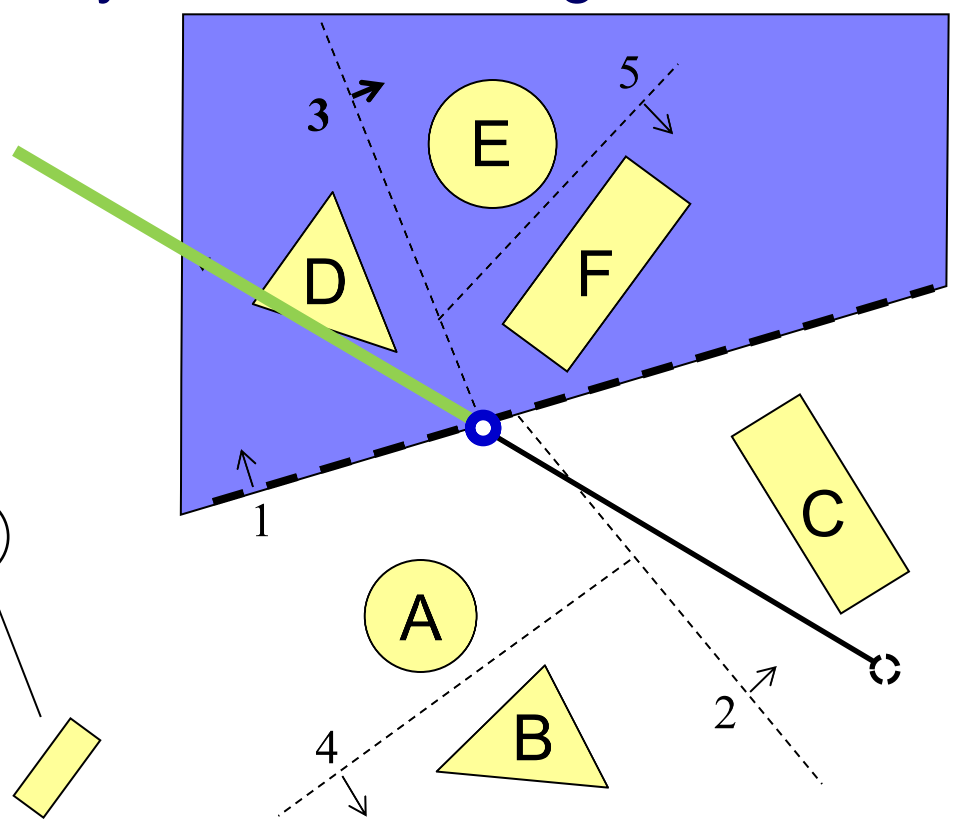
Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

- Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
  - » Test front (right)





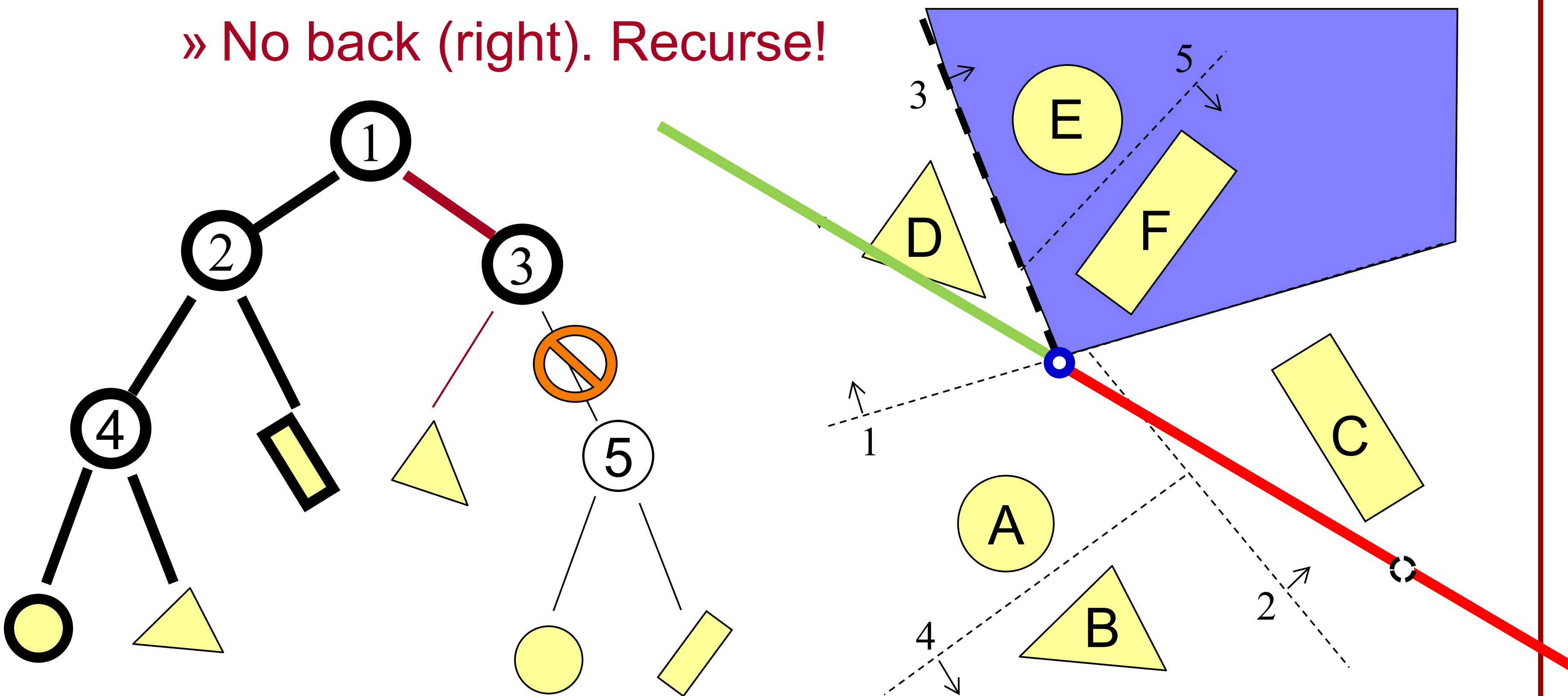
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» No back (right). Recurse!

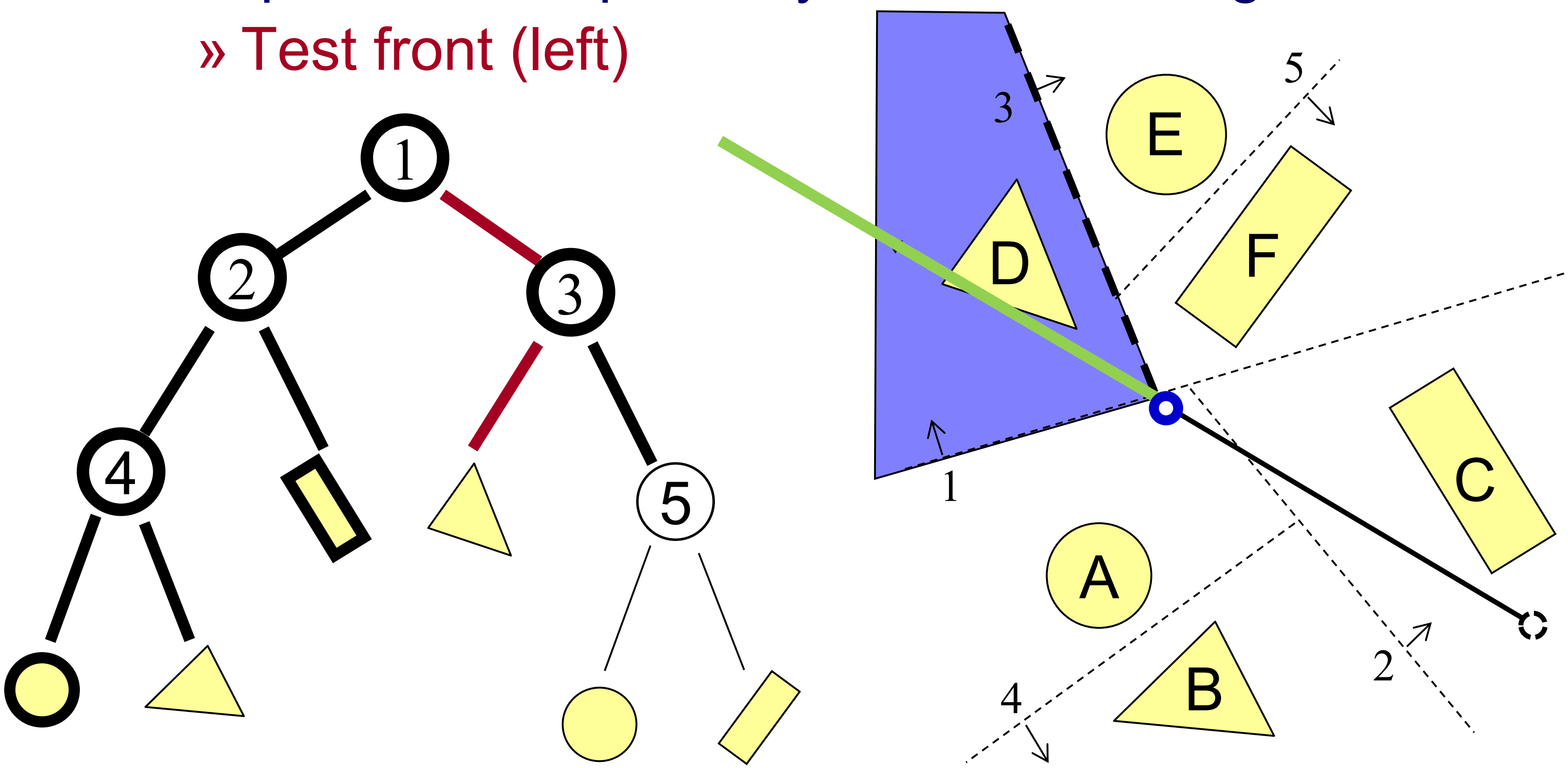


Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

- Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
  - » Test front (left)

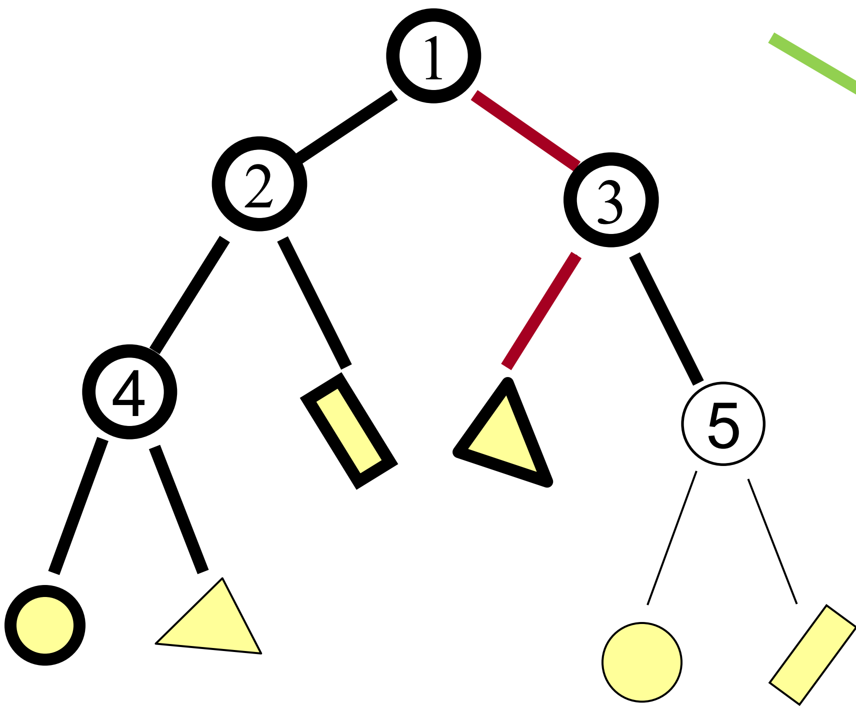


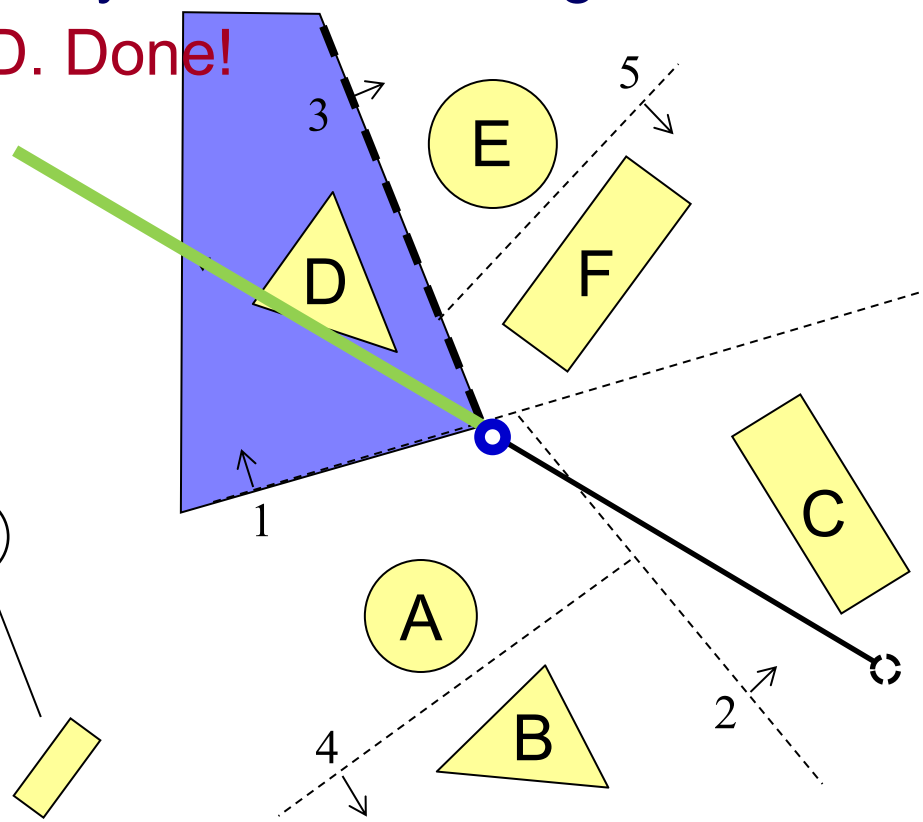
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

- Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
  - » Intersection with D. Done!





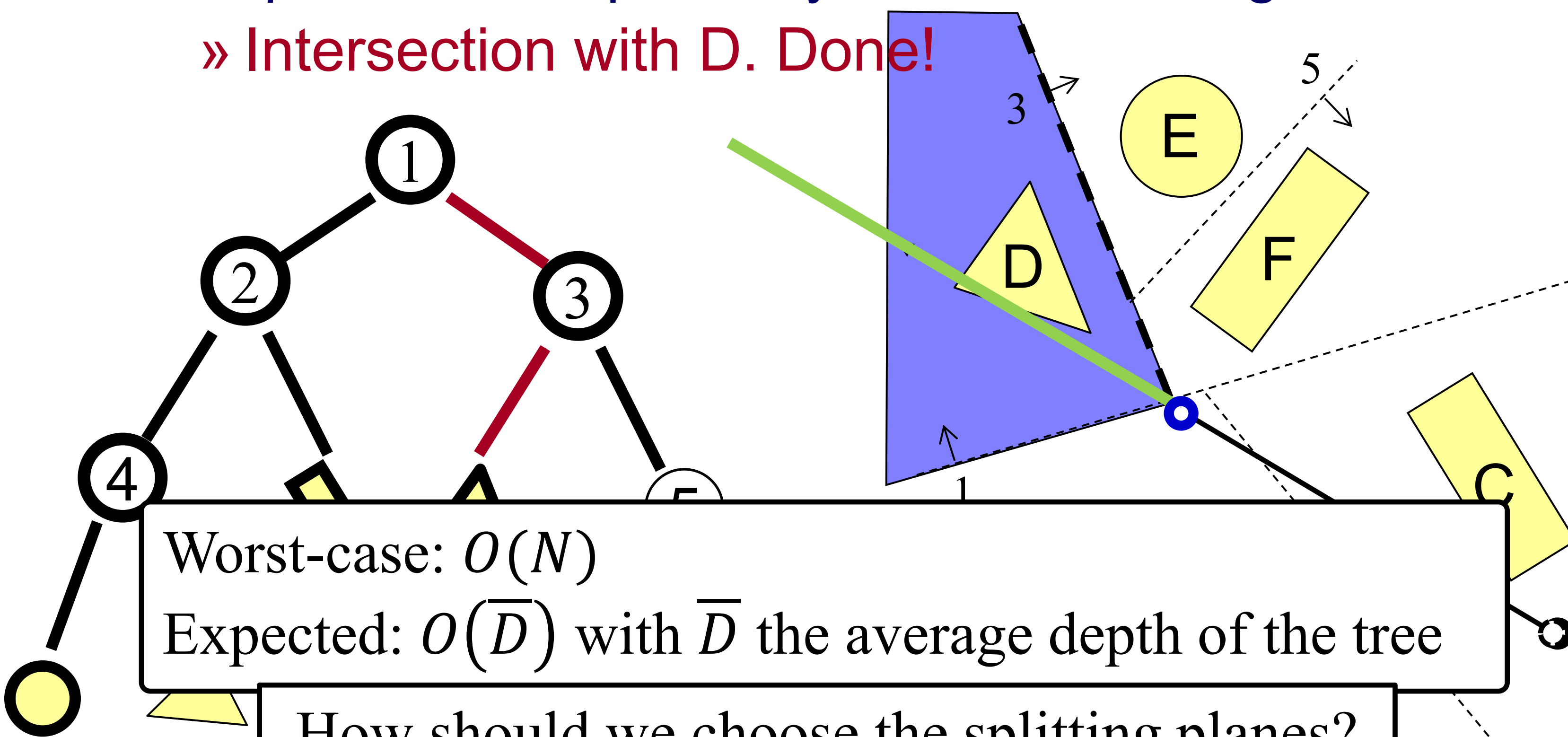
Note: Arrows denote the “right” side of the splitting plane.

Space-Partition: BSP Tree

Example 3: Ray intersection query

Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

» Intersection with D. Done!



Worst-case: $O(N)$

Expected: $O(\overline{D})$ with $\overline{D}$ the average depth of the tree

How should we choose the splitting planes?

Note: Arrows denote the "right" side of the splitting plane.

Space-Partition: BSP Tree

```
Intersection RayTreeIntersect( Ray ray< 3 > ,  Node node )
{
	if ( Node is a leaf ) return intersection of closest primitive in cell, or NULL if none
	else
	{
		// Find splitting plane and near and far children
		near_child = child of node that contains the start (r(−∞))
		far_child = other child of node

		// Recurse down near child first
		isect = RayTreeIntersect( ray , near_child )
		if( isect ) return isect    // If there's a hit, we are done

		// If there is no hit, test the far child
		return RayTreeIntersect( ray , far_child )
	}
}
```

Acceleration Techniques

- Data-Partitions
  - » Bounding volume hierarchy (BVH)
- Space-Partitions
  - » Uniform (voxel) grid
  - » Octree
  - » Binary space partition (BSP) tree

Note:

- All are independent of the viewer position
- All need to be adapted to changing geometry