Accelerated Ray Casting

Michael Kazhdan

(601.457/657)

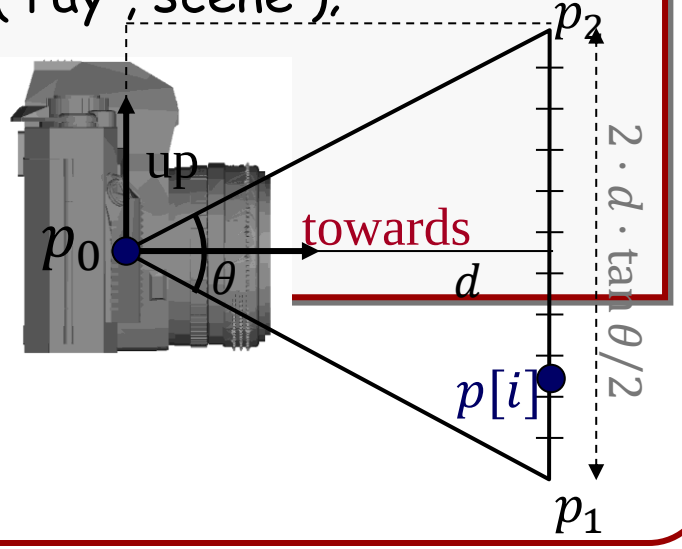
HB Ch. 14.1, 14.2

FvDFH 16.1, 16.2

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;
}
```



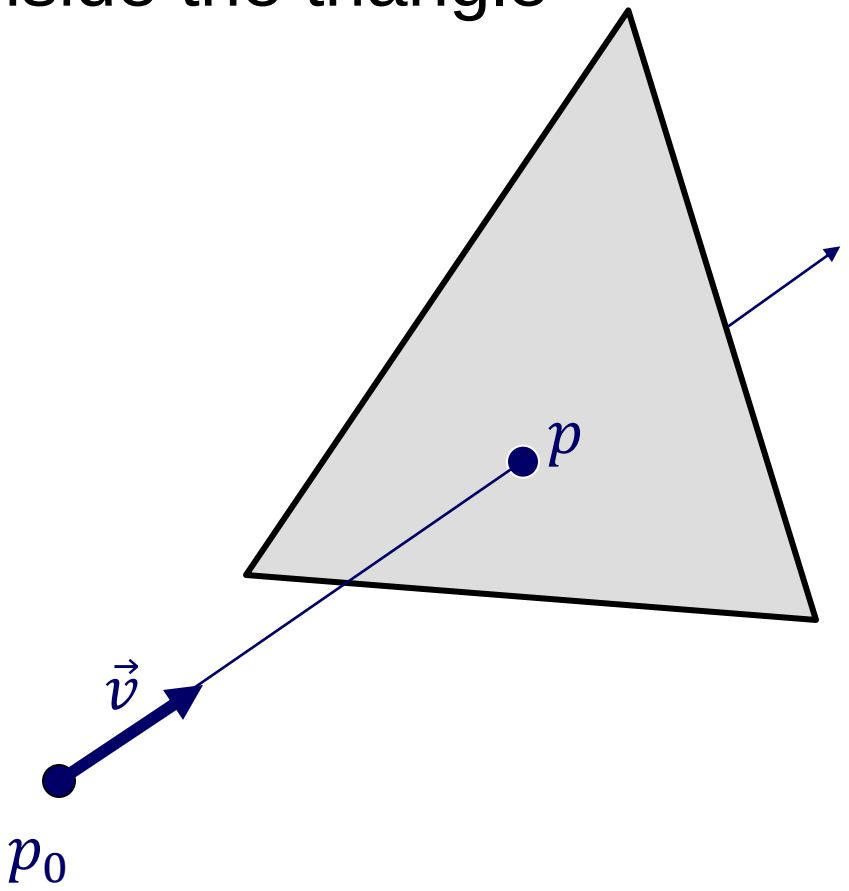
Ray Casting

- 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: $\Phi(p) = \langle p, \vec{n} \rangle - d = 0$

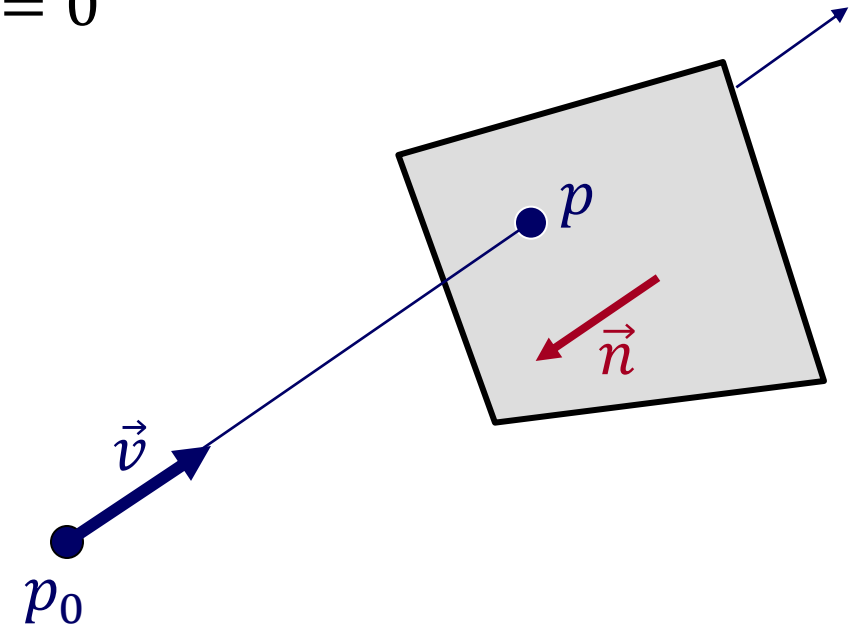
Algebraic Method

Substituting for p, we get:

$$\Phi(t) = \langle p_0 + t \cdot \vec{v}, \vec{n} \rangle - d = 0$$

Solution:

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

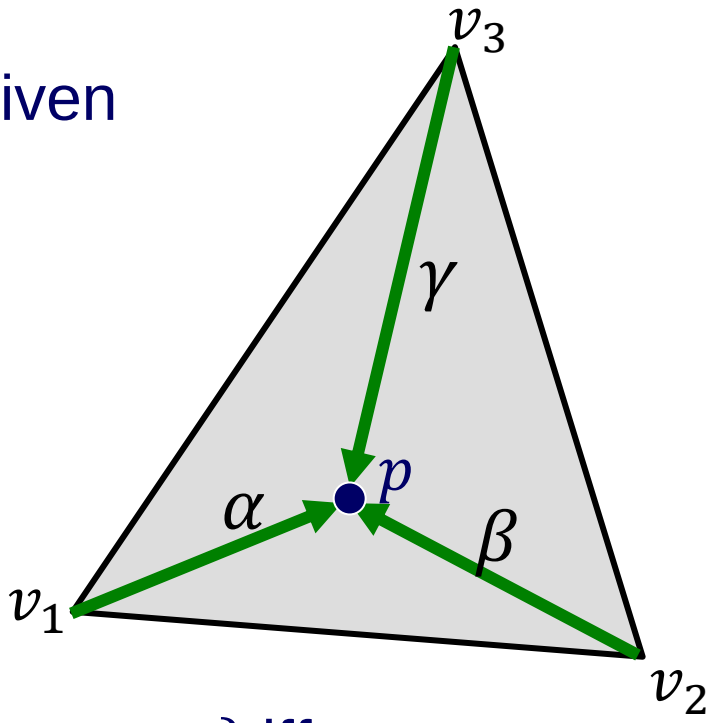


Ray-Triangle Intersection

- Check for point-triangle intersection parametrically

In general, given $p \in \mathbb{R}^3$ and given three points $\{v_1, v_2, v_3\} \subset \mathbb{R}^3$ (in general position) 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 spanned by $\{v_1, v_2, v_3\}$ iff.:

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

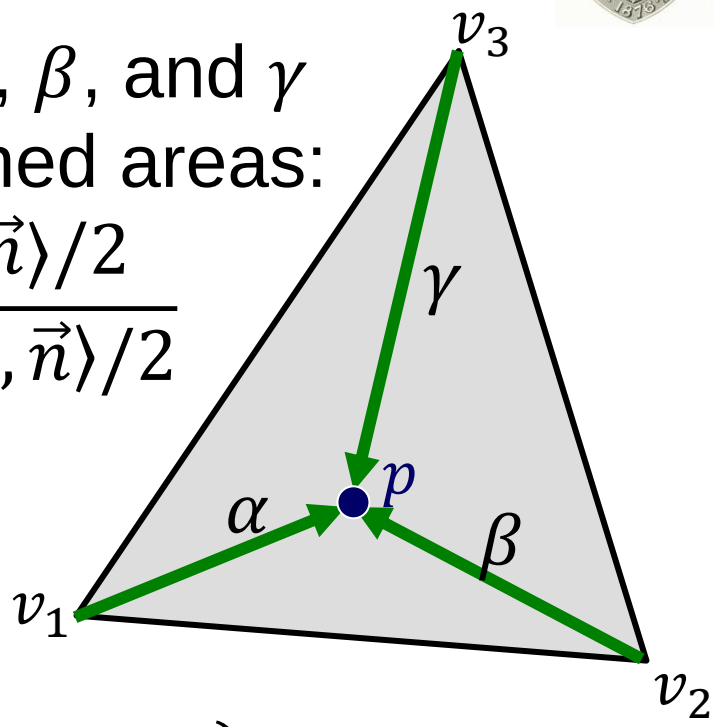
p is inside the triangle with vertices $\{v_1, v_2, v_3\}$ iff.:

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

Ray-Triangle Intersection

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

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



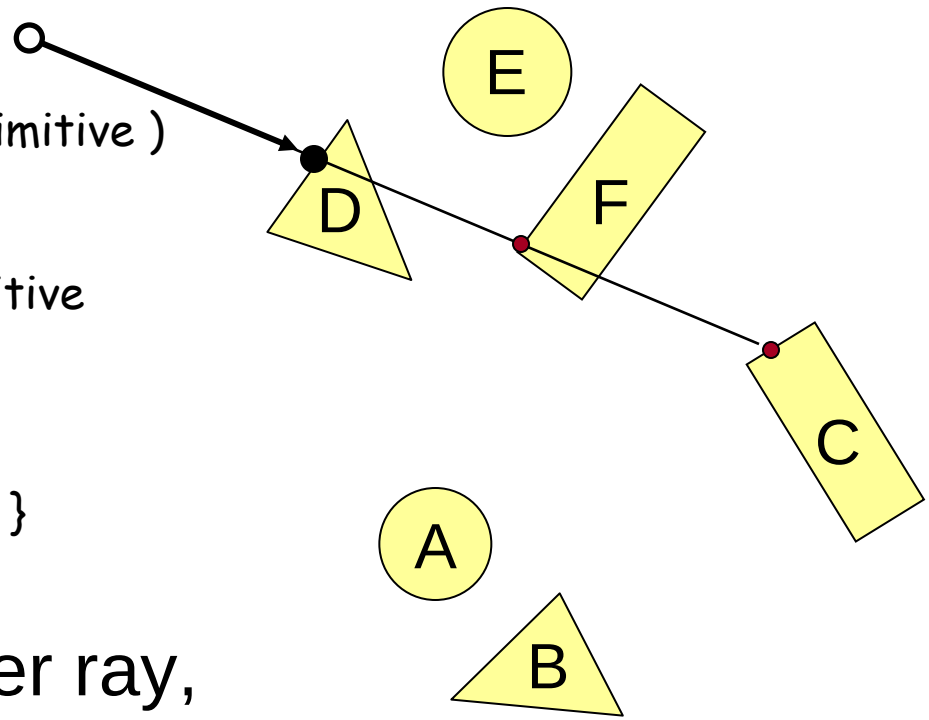
where $\vec{n}$ is a unit vector that is perpendicular to the triangle:

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

Ray-Scene Intersection

A direct (naïve) approach:

```
Intersection FindIntersection( Ray< 3 > ray, Scene scene )
{
    { min_t , min_shape } = { ∞ , NULL }
    for each primitive in scene
    {
        t = Intersect( ray , primitive )
        if( t>0 and t<min_t )
        {
            min_shape = primitive
            min_t = t
        }
    }
    return { min_t , min_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

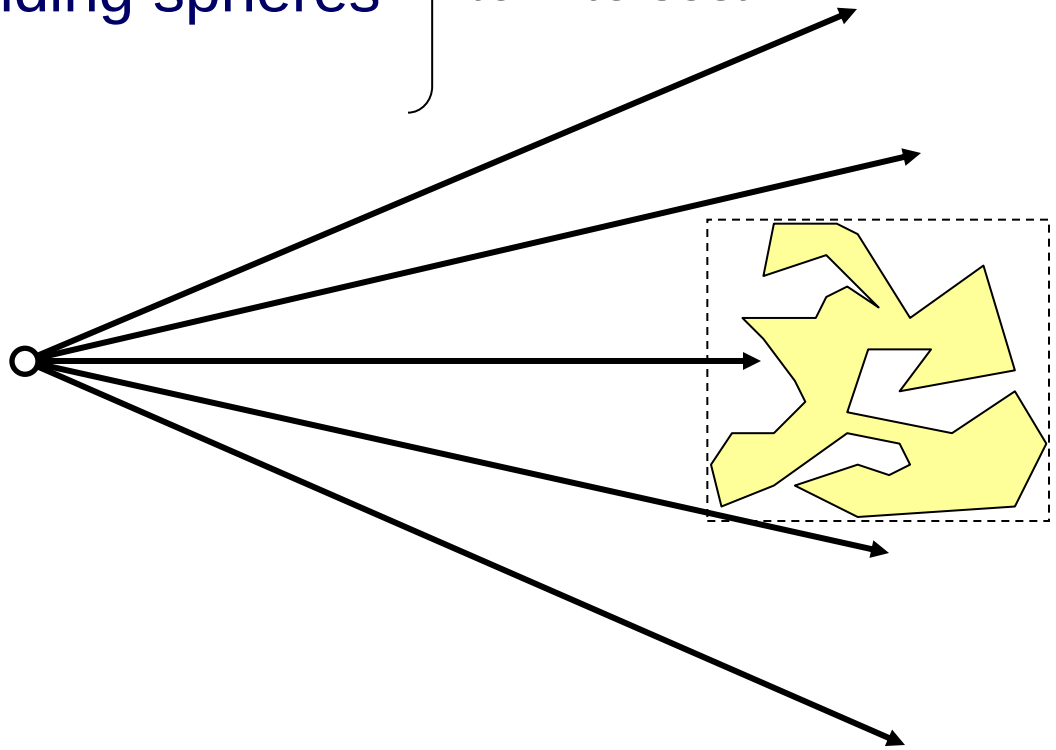
Both data and space partitions accelerate intersections testing by leveraging:

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

Space Partition: Bounding Volume

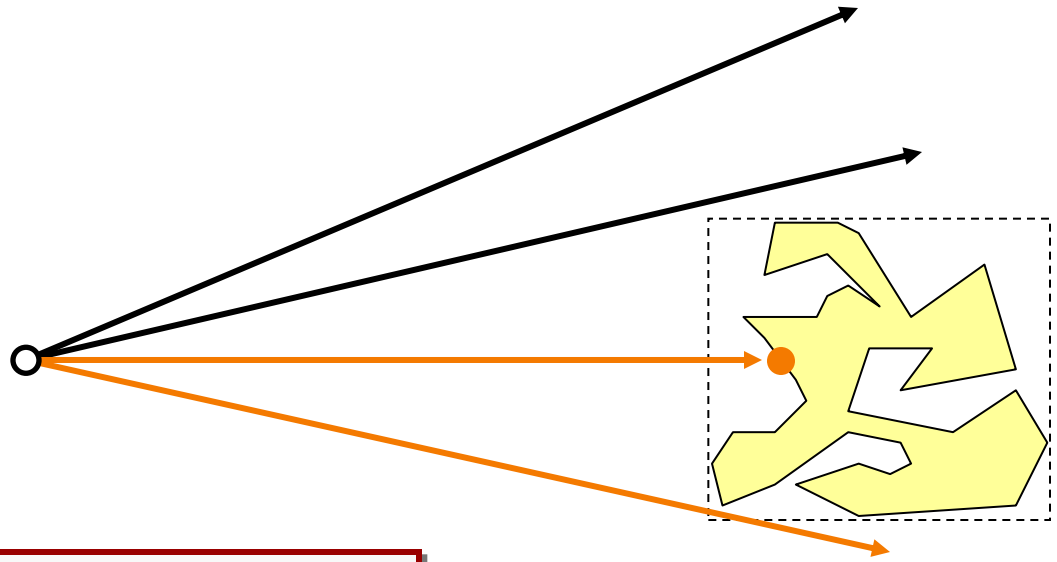
- Check for intersection with the bounding volume:
  - Bounding cubes
  - Bounding boxes
  - Bounding spheres
  - Etc.

Stuff that's easy to intersect



Space Partition: Bounding Volume

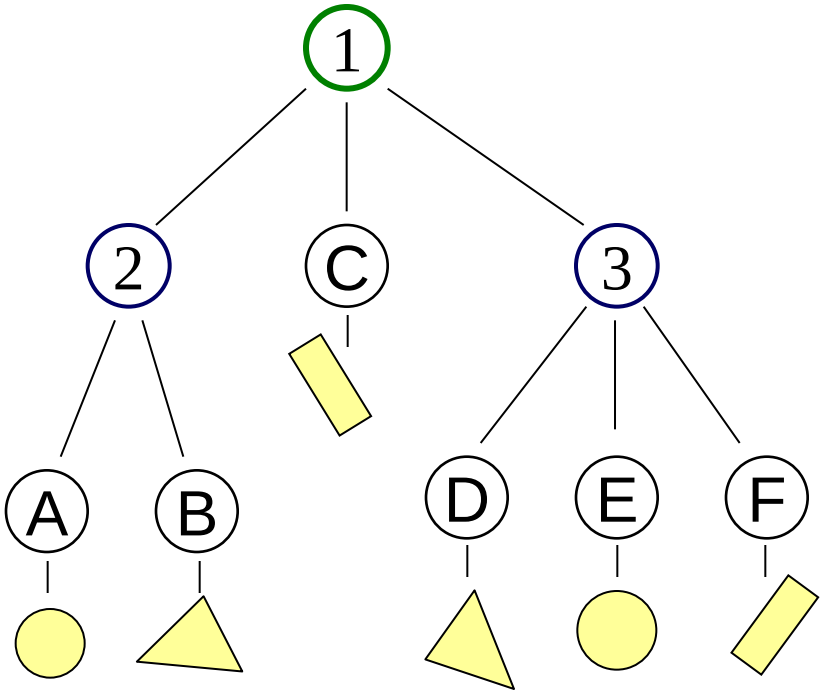
- Check for intersection with the bounding volume
  - If the ray misses the bounding volume, it can't intersect its contents

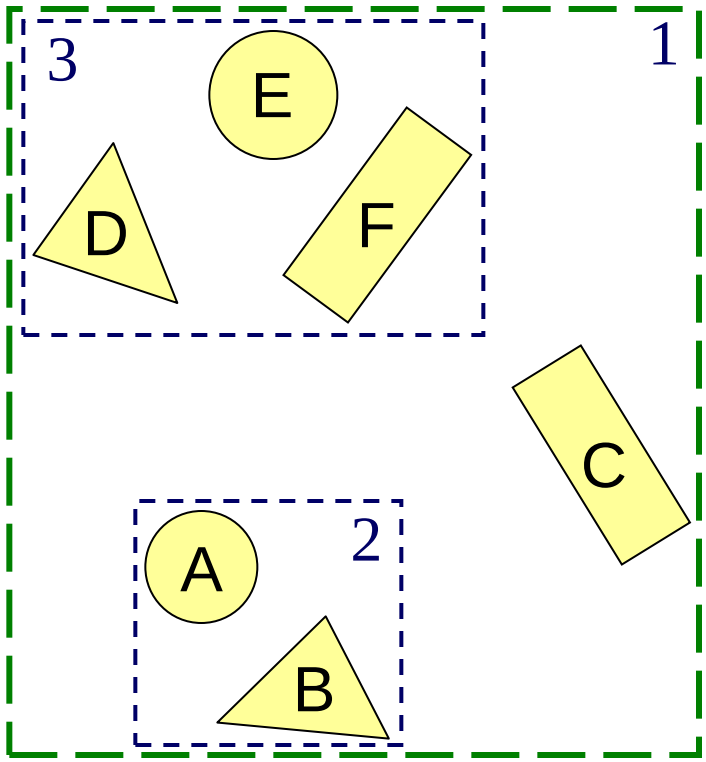


Still need to check for intersections with shape.

Space Partition: BVH

- Build a bounding volume hierarchy (BVH)
  - Each bounding volume stores (and encloses):
    - » Child bounding volumes
    - » A subset of shapes





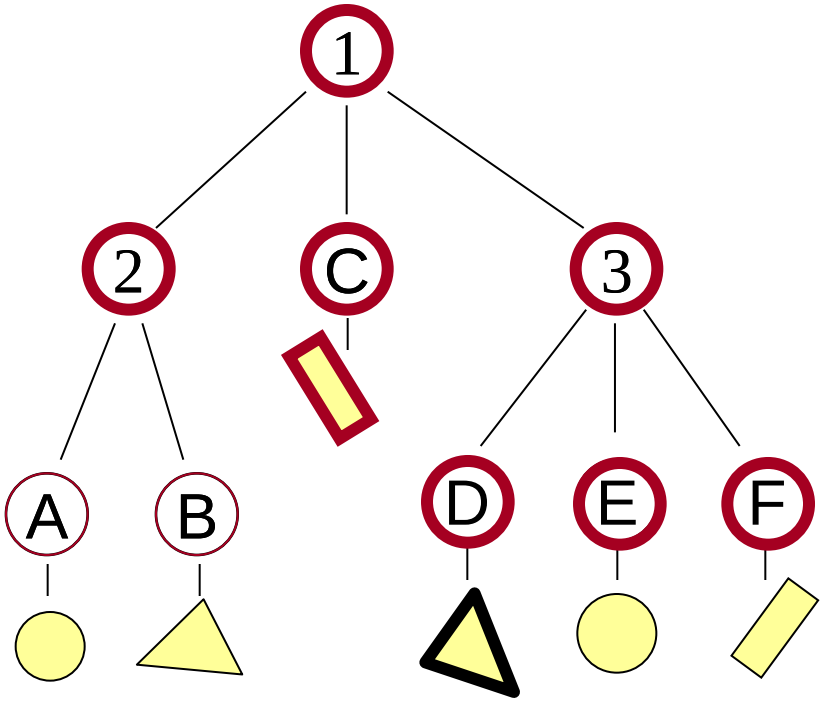
Space Partition: BVH

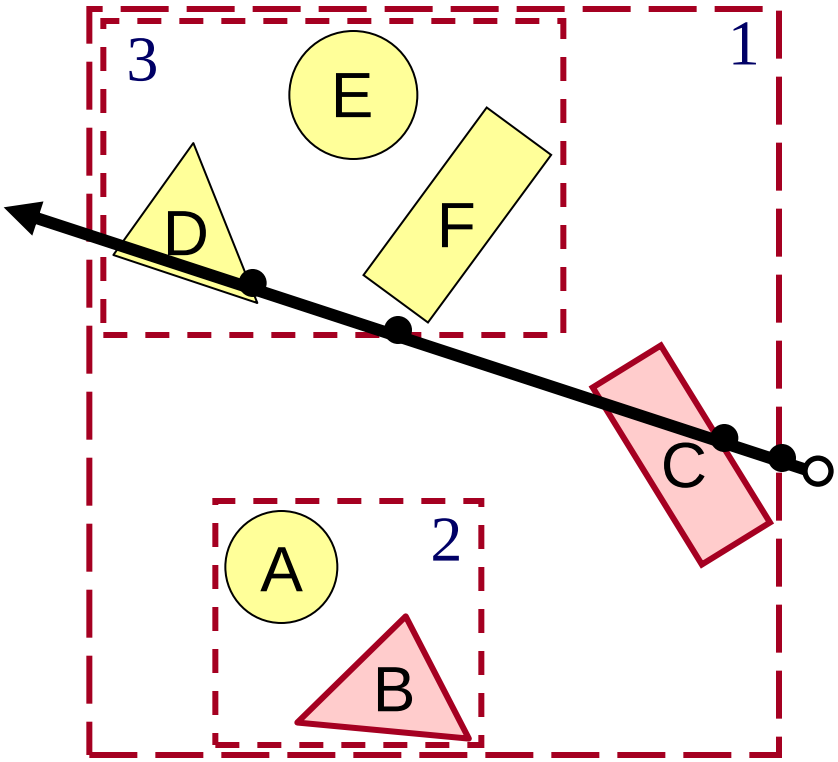
- Grouping:

```
Intersection FindIntersection( Ray< 3 > ray , BoundingBox< 3 > bBox )
{
      { min_t , min_shape } = { ∞ , NULL }

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

      foreach shape in bBox                                  // Test node's shape
      {
            t = Intersect( ray , shape )
            if( t>0 && t<min_t ) { min_t , min_shape } = { t , shape }
      }

      for each child_bBox in bBox                            // Test node's children
      {
            ( t , shape ) = FindIntersection( ray , child_bBox )
            if( t>0 && t<min_t ) { min_t , min_shape } = { t , shape }
      }
      return { min_t , min_shape }
}
```

Space Partition: BVH

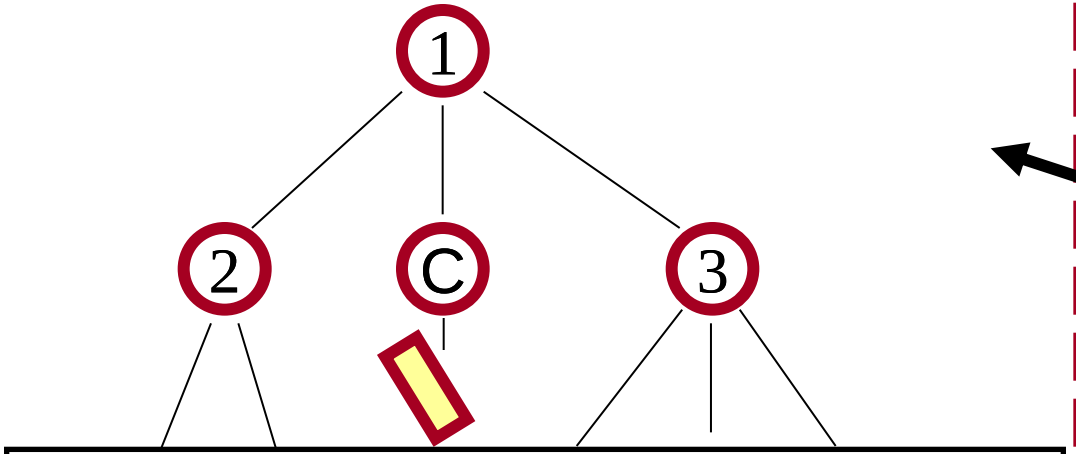
- Grouping:
  Discard groups of primitives that can be (easily) guaranteed to be missed by the ray.

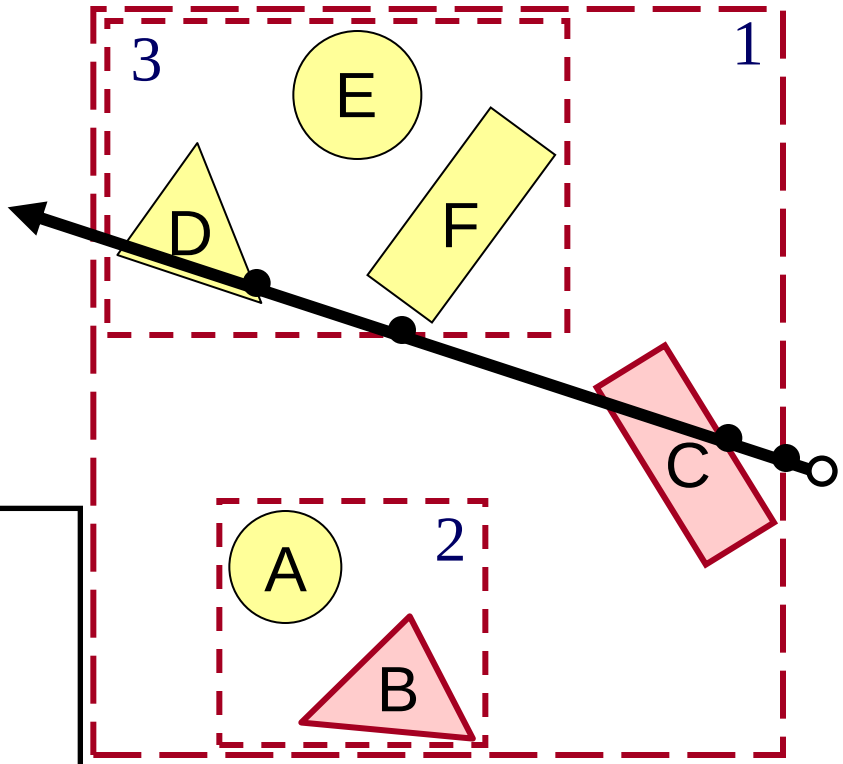




Space Partition: BVH

- Grouping:
  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 groups 1, 2, and 3
- Need to test shapes C, D, E, and F

Space Partition: BVH

- Ordering:

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

      // Process intersections
      for each intersected child bBox
      {

            { t , shape } = FindIntersection( ray , child_bBox )
            if( t>0 && t<min_t )  { min_t , min_shape } = { t , shape }
      }
      return { min_t , min_shape }
}
```

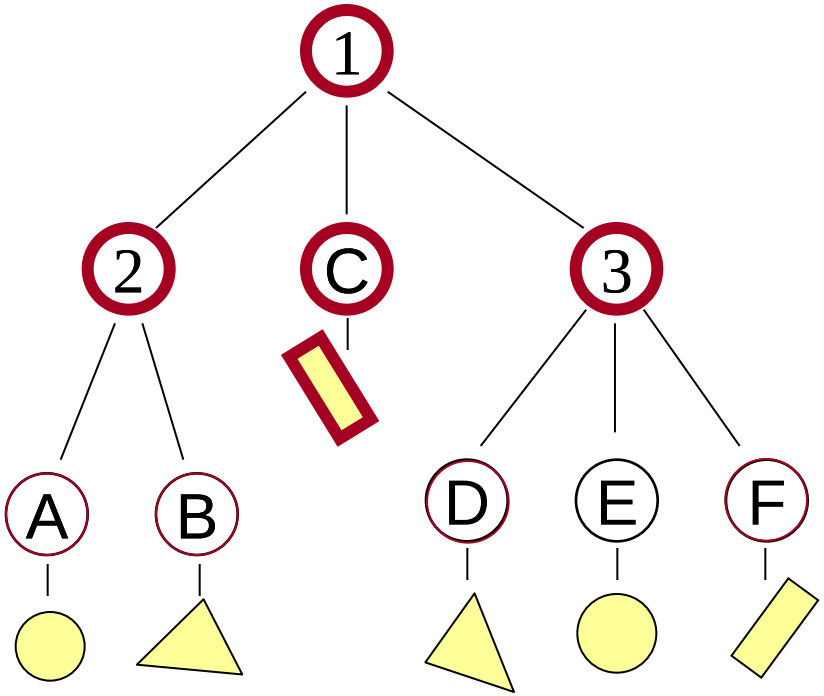
Space Partition: BVH

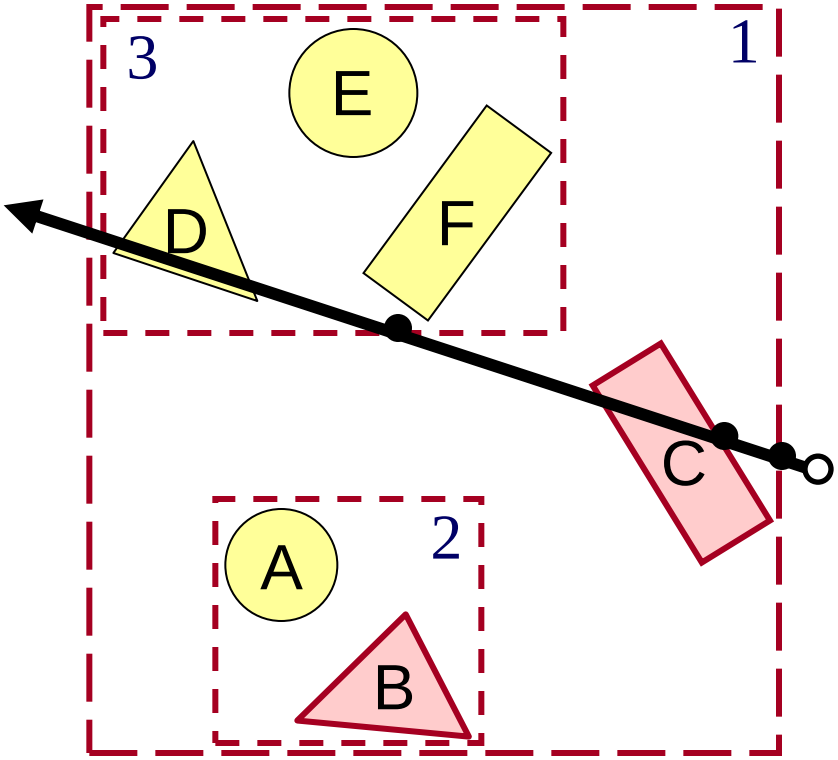
- Ordering:

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

	// Process intersections
	for each intersected child bBox
	{
		if( min_t<bv_t[child_bBox] ) break
		{ t , shape } = FindIntersection( ray , child_bBox )
		if( t>0 && t<min_t ) { min_t , min_shape } = { t , shape }
	}
	return { min_t , min_shape }
}
```

Space Partition: BVH

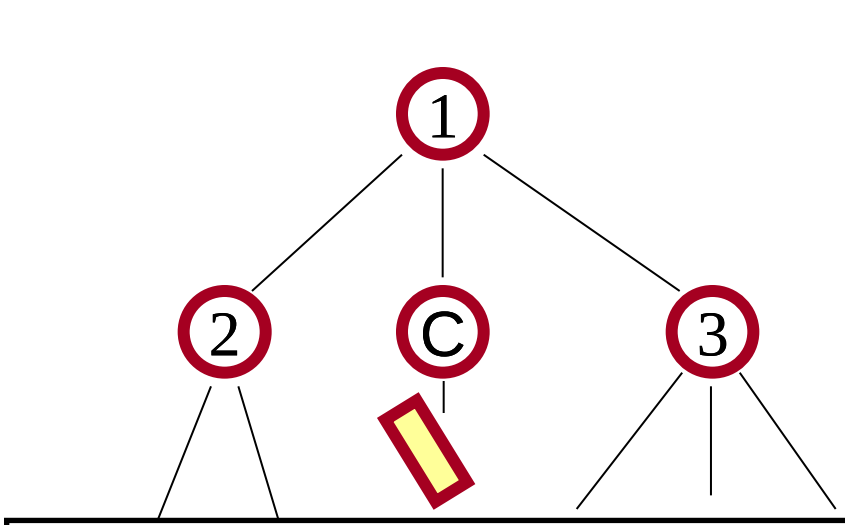
- Ordering:
  Test (likely) nearer intersections first, allowing for early termination if there is a hit.

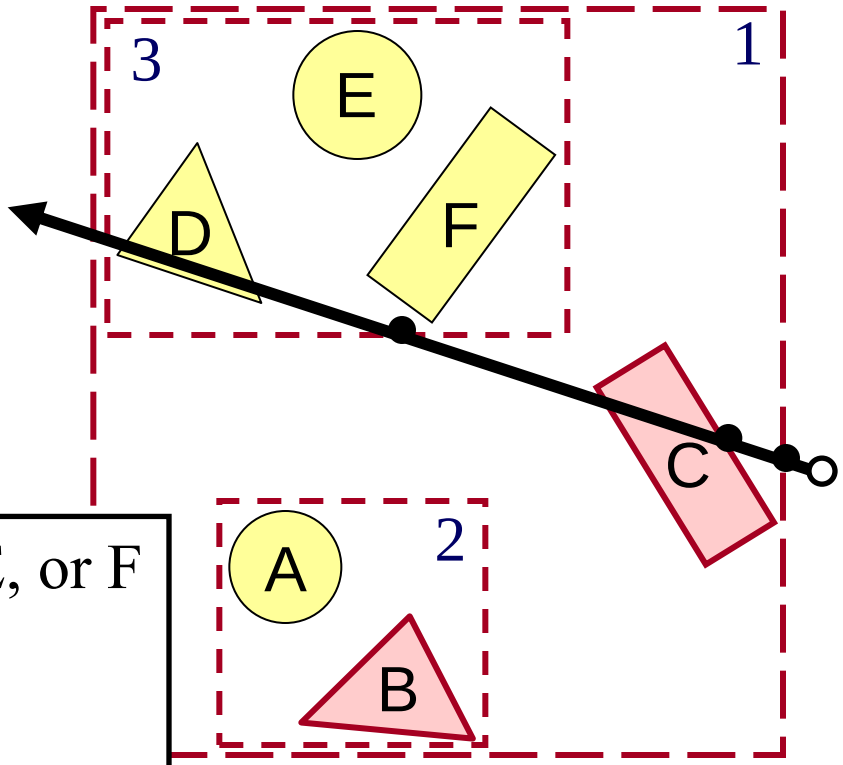




Space Partition: BVH

- Ordering:
  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

Space Partition: BVH

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

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

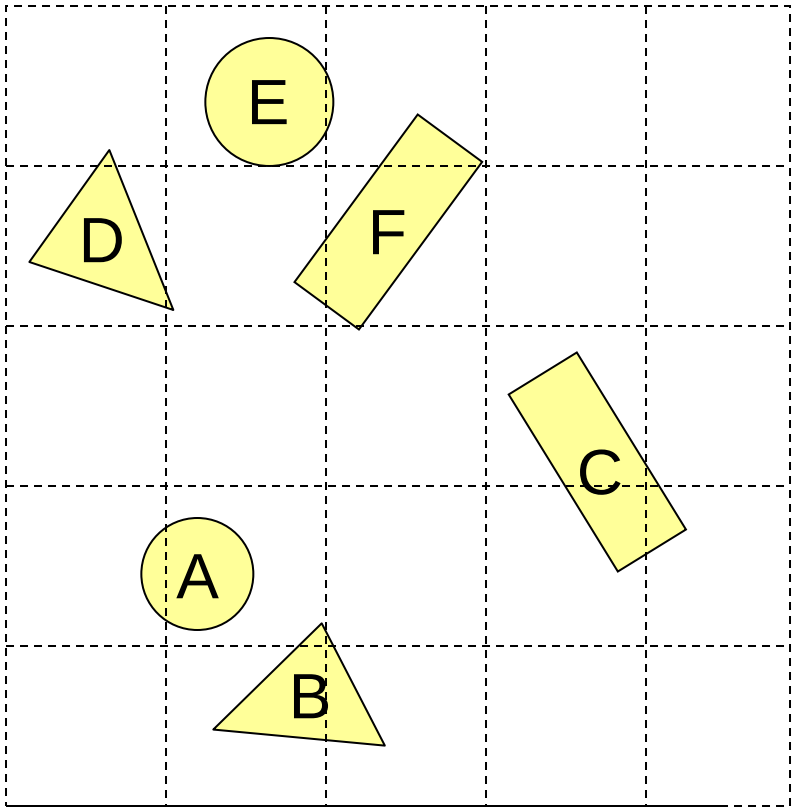
Overview

- 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 (pointers to) intersected primitive with each grid cell

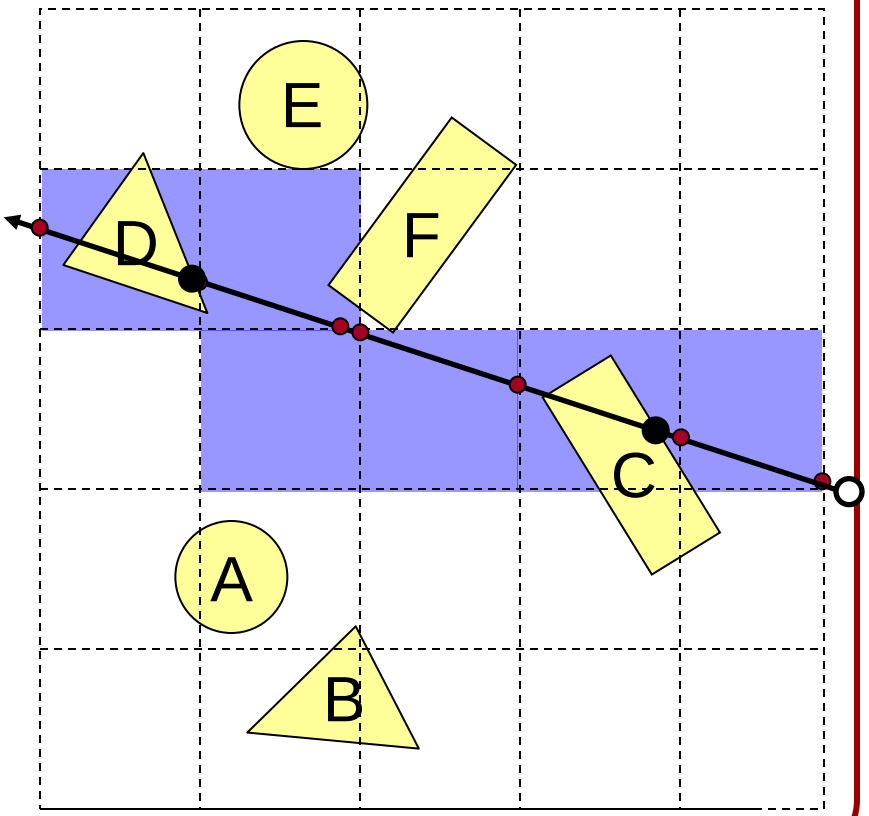
- A primitive may belong to multiple cells
- A cell may have multiple primitives



Space Partitions: Uniform Grid

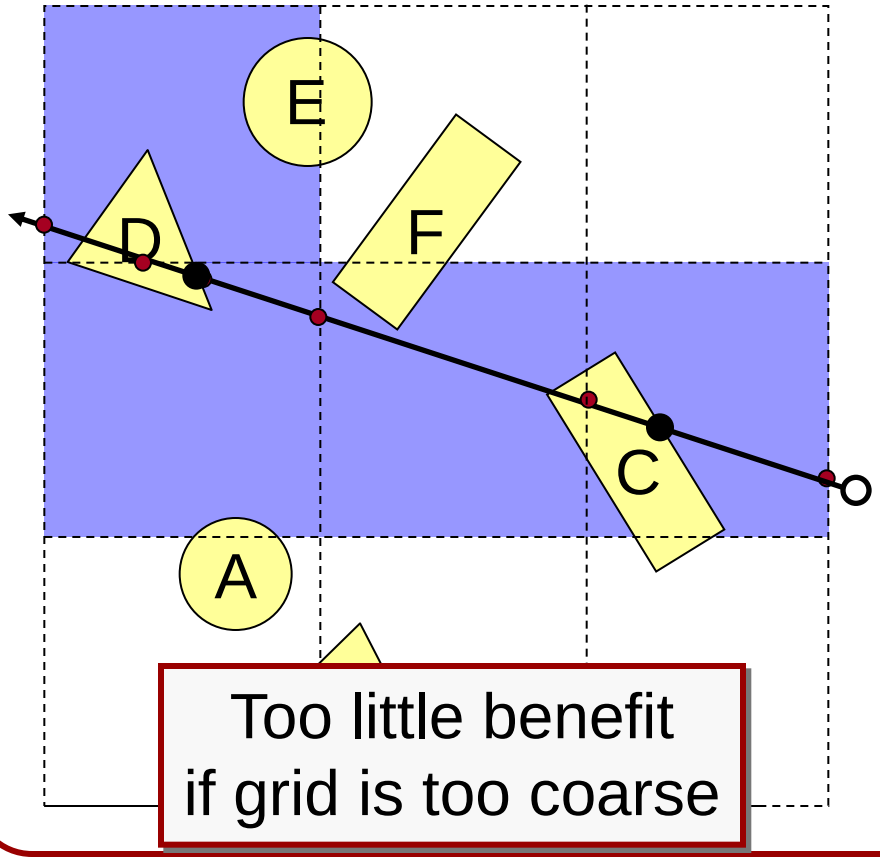
- Trace rays through grid cells
  - Fast
  - Incremental

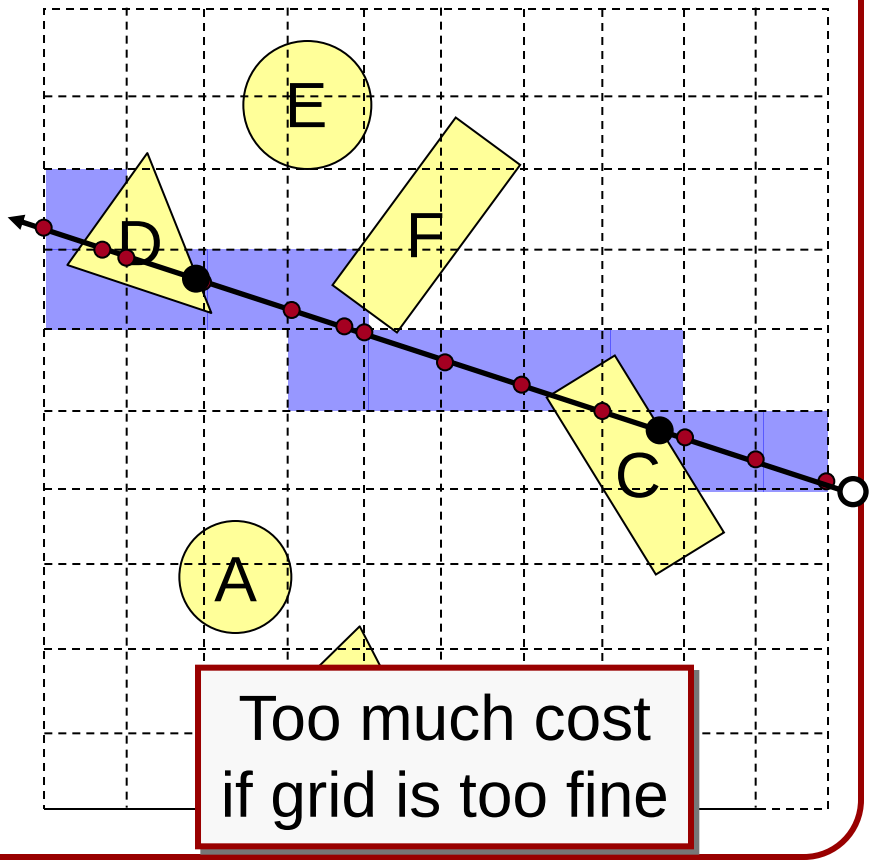
Only check primitives
in intersected grid cells



Space Partitions: Uniform Grid

- Potential problem:
  - How choose suitable grid resolution?



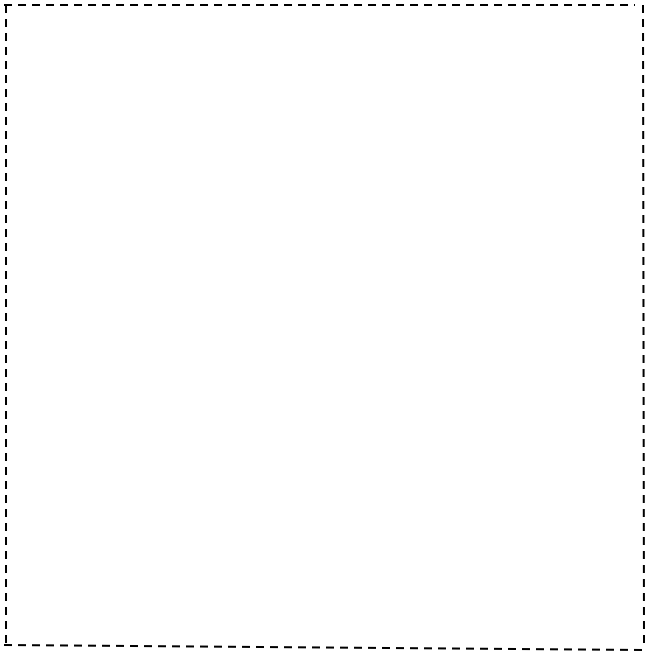


Overview

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

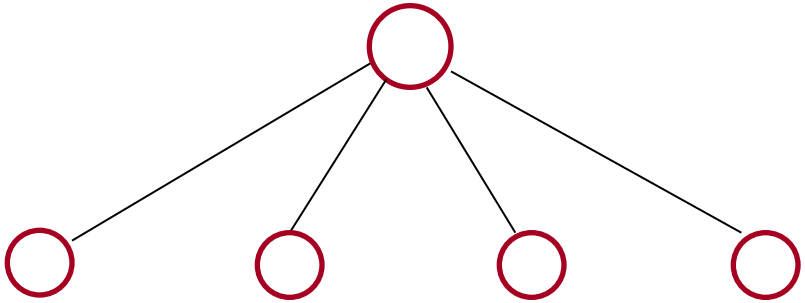
Space Partition: Octree

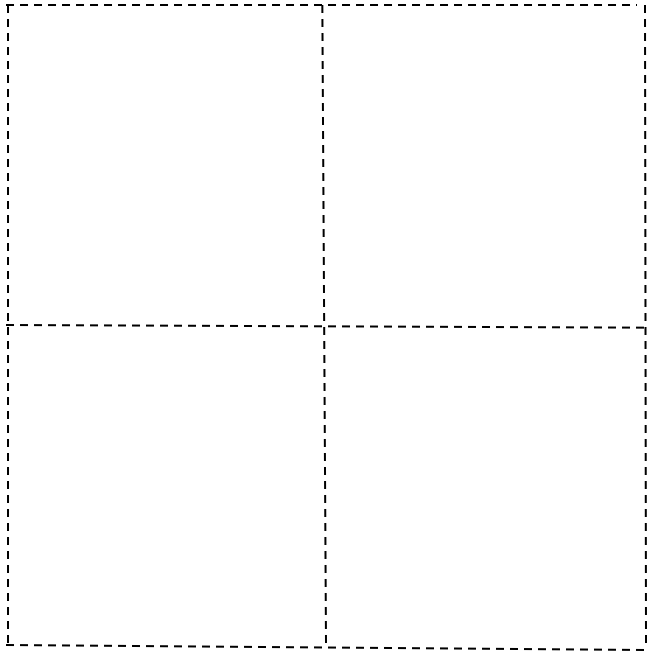
- Think of a voxel grid hierarchically as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions



Space Partition: Octree

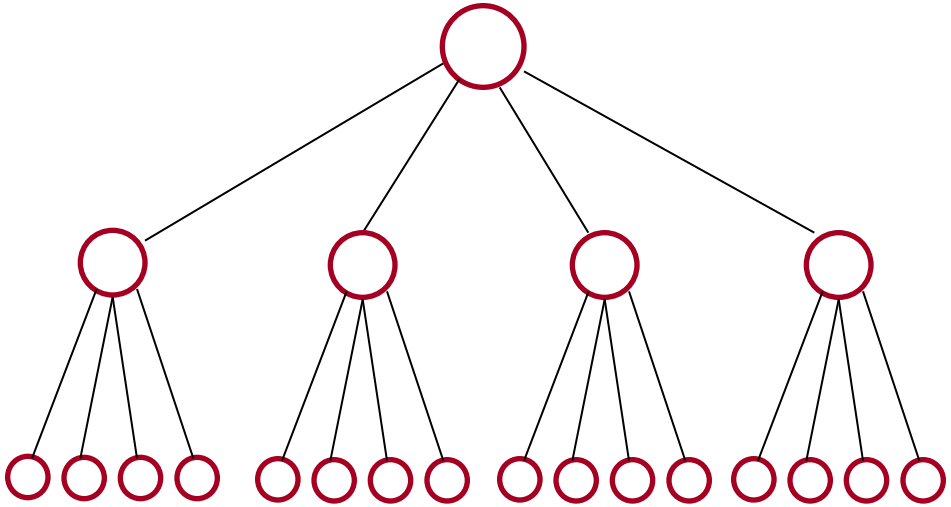
- Think of a voxel grid hierarchically as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions

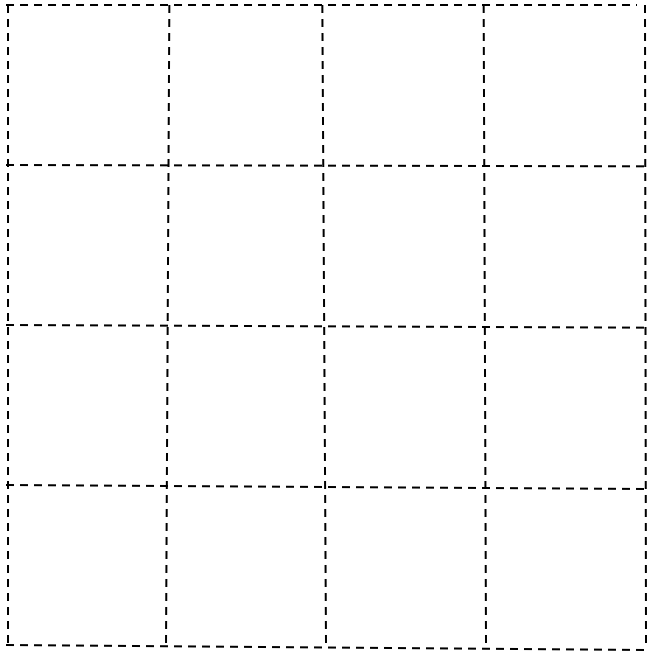




Space Partition: Octree

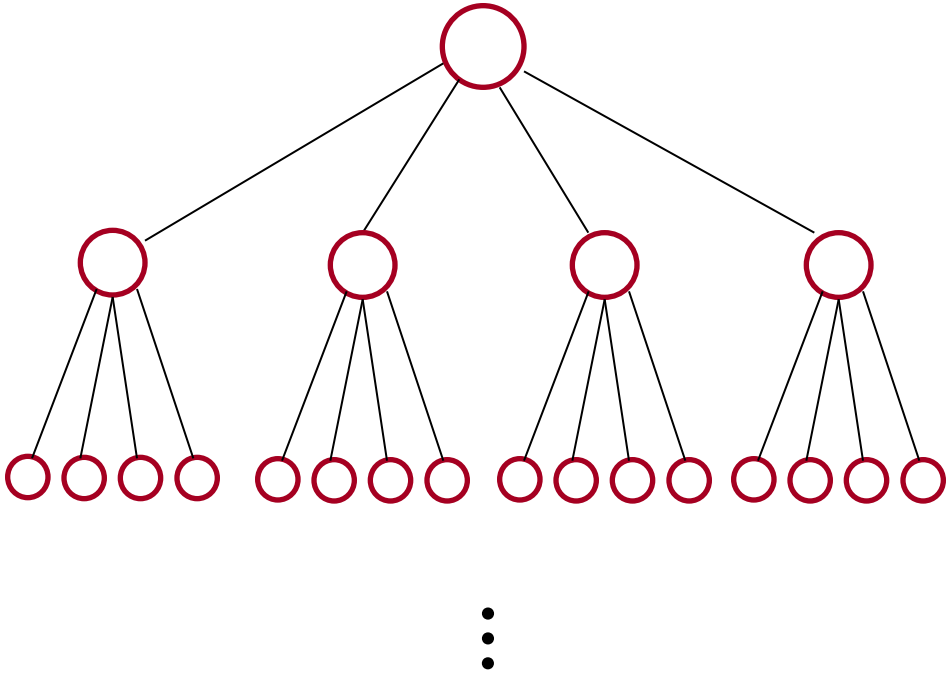
- Think of a voxel grid hierarchically as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions

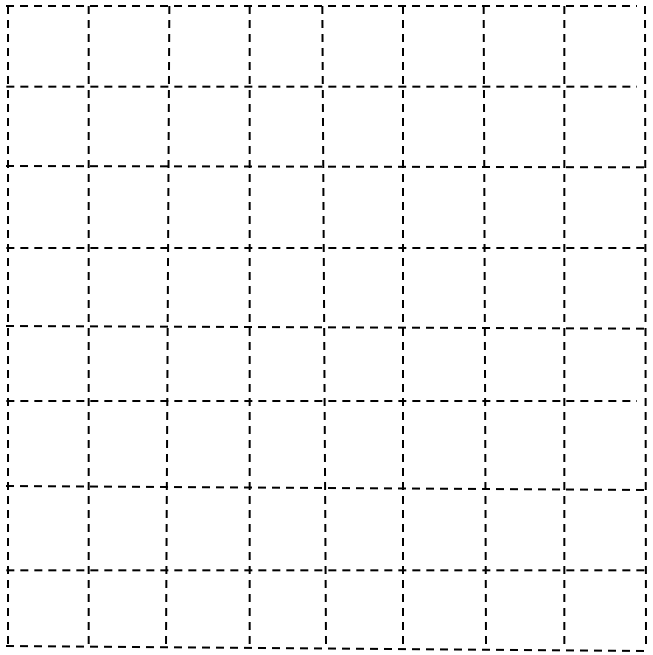




Space Partition: Octree

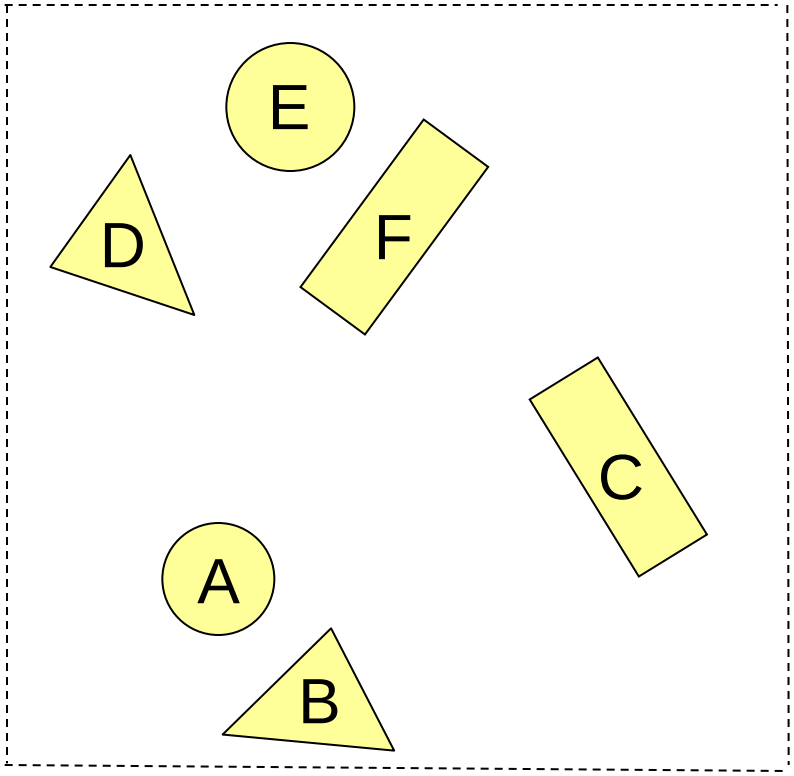
- Think of a voxel grid hierarchically as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions





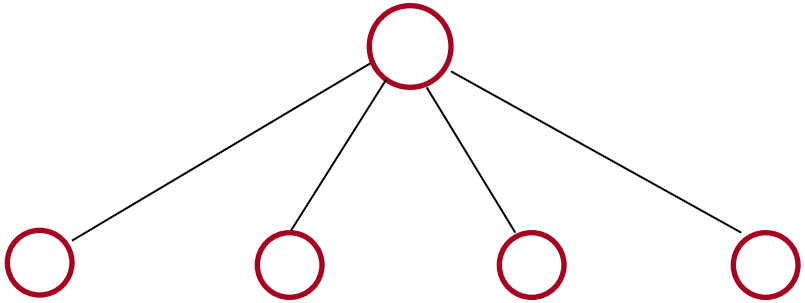
Space Partition: Octree

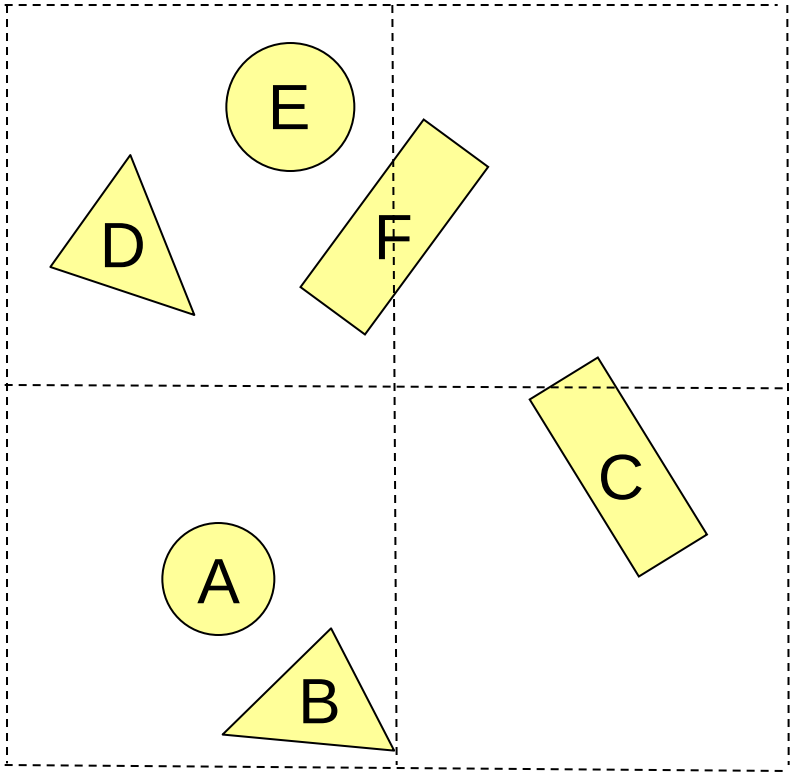
- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).



Space Partition: Octree

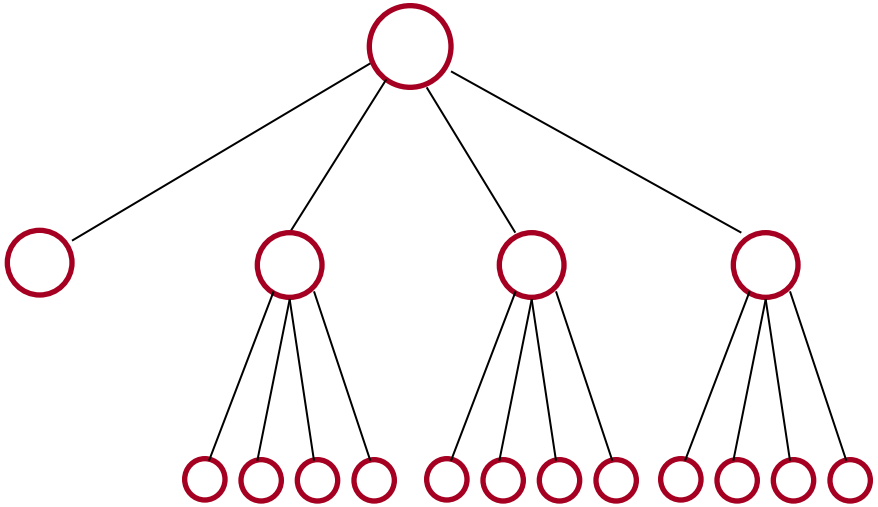
- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).

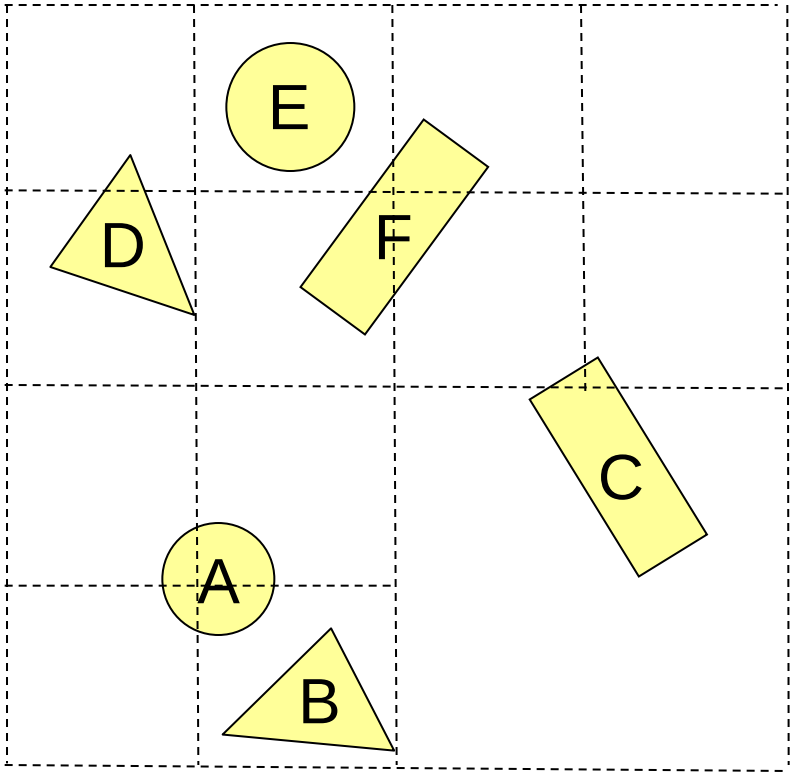




Space Partition: Octree

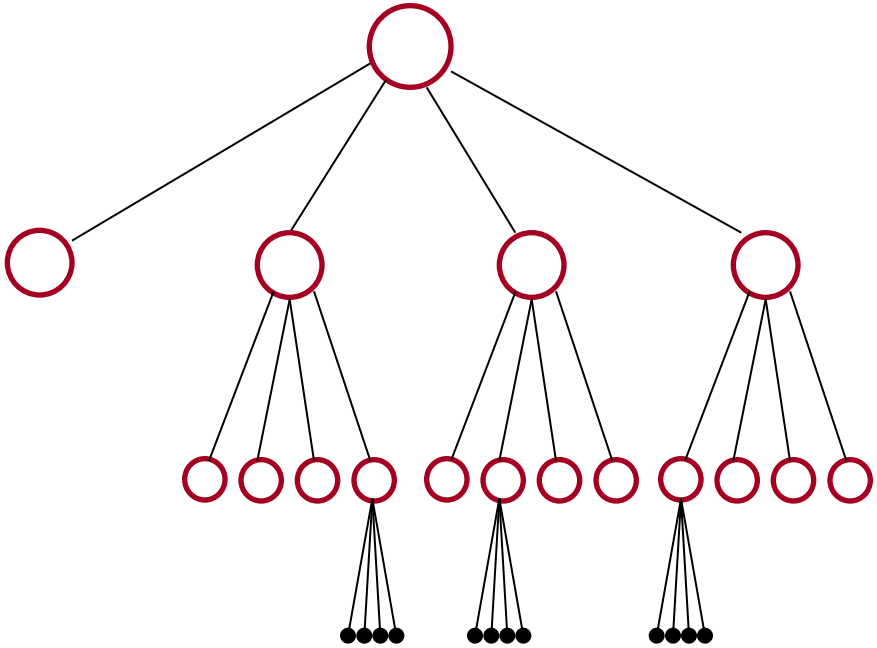
- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).

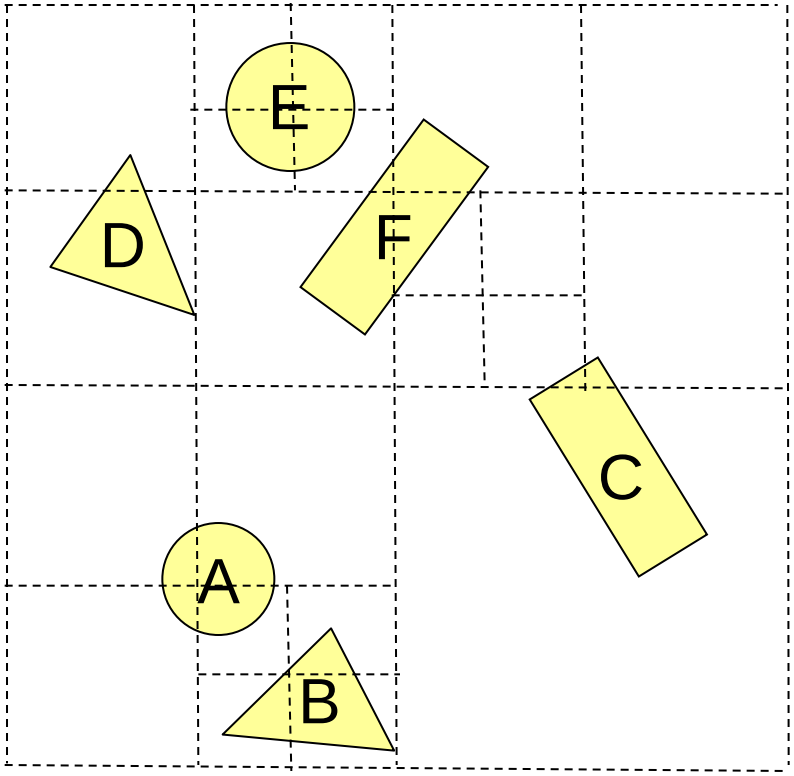




Space Partition: Octree

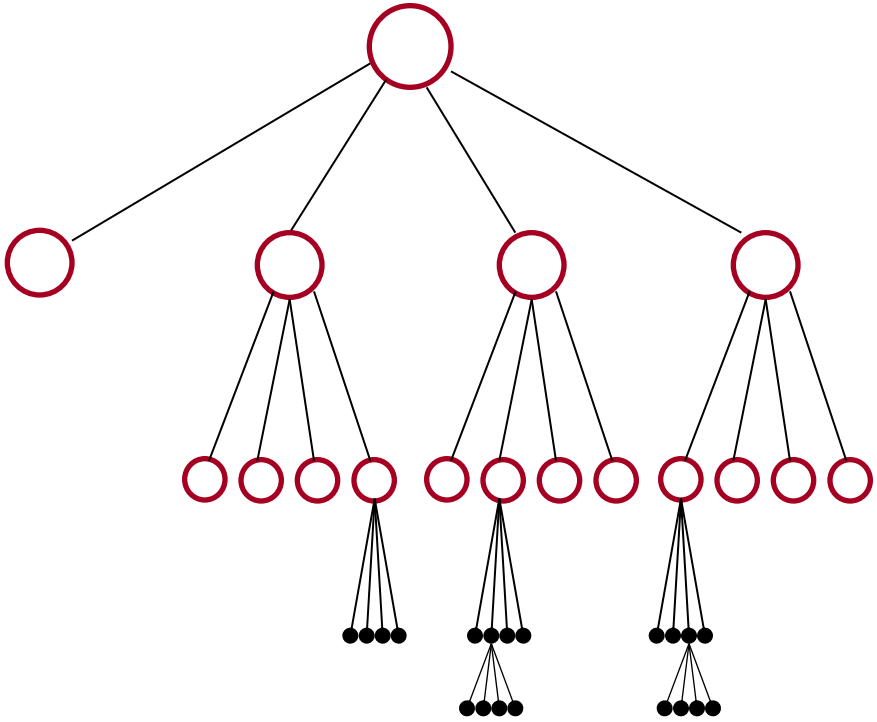
- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).

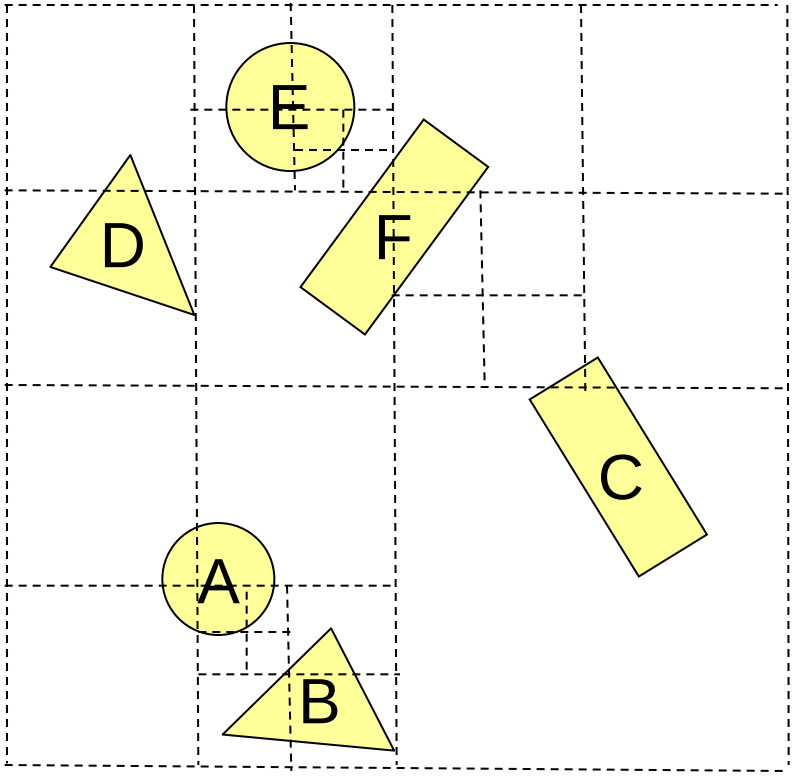




Space Partition: Octree

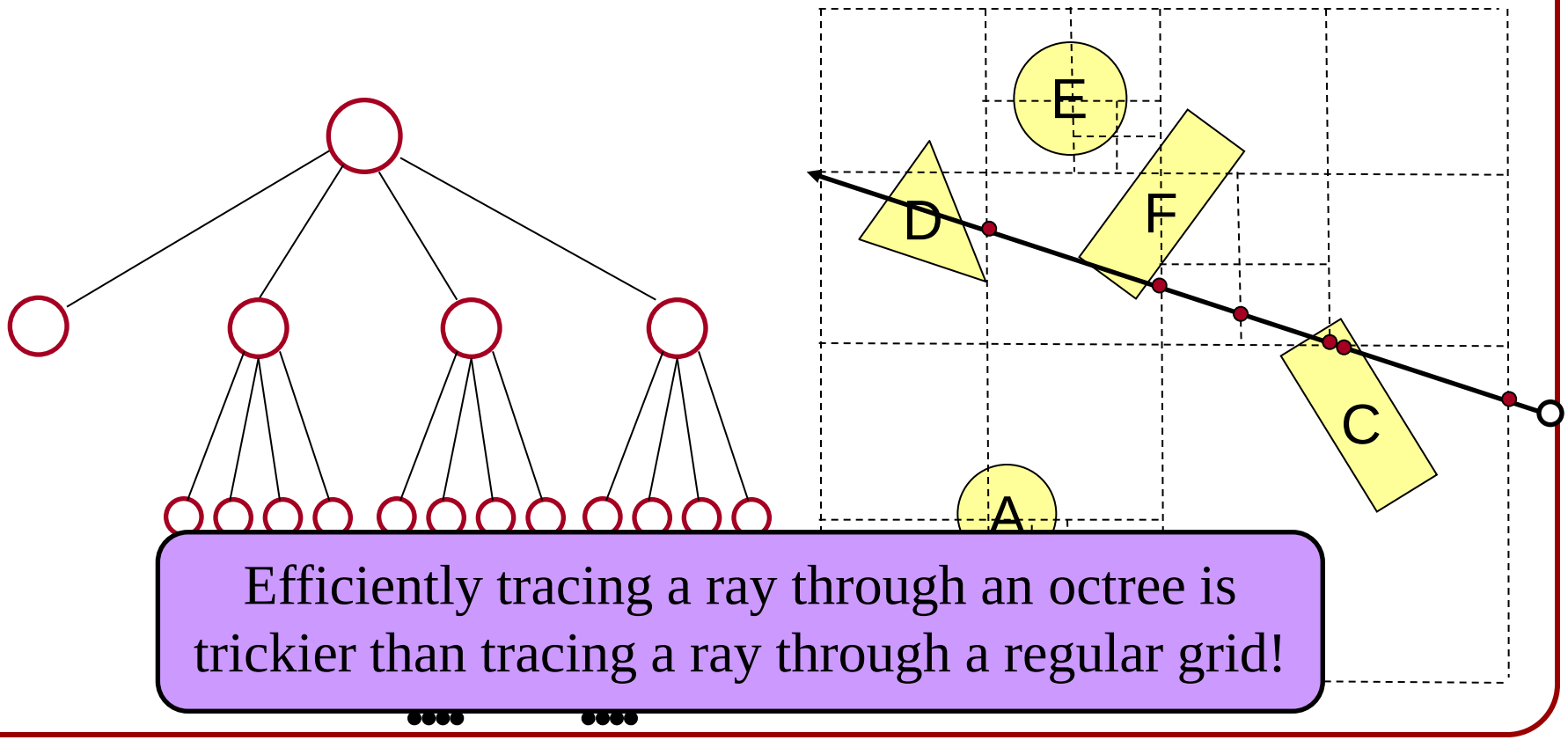
- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).





Space Partition: Octree

- In an octree, we subdivide adaptively (e.g. only refining cells containing more than one shape).



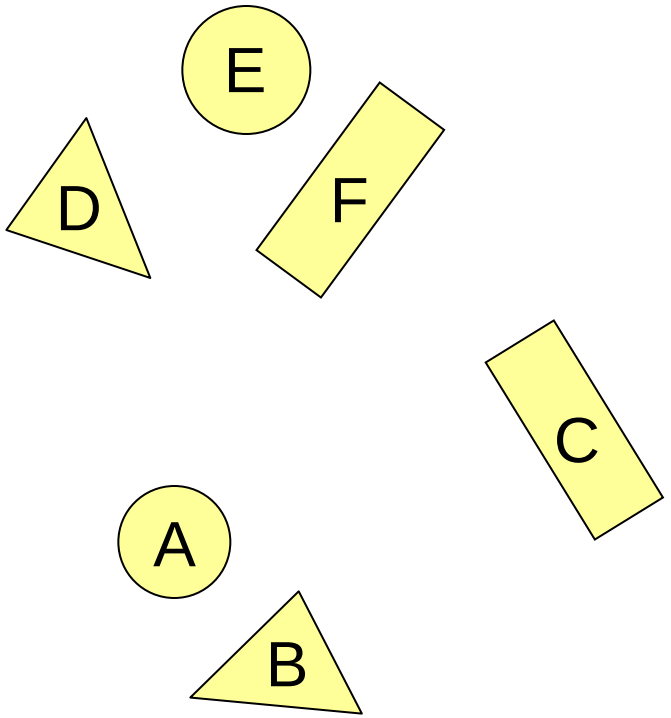
Efficiently tracing a ray through an octree is trickier than tracing a ray through a regular grid!

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

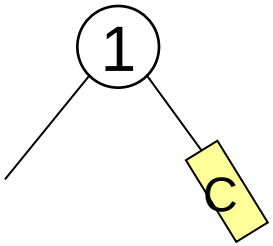
- Alternate between splitting along the x-axis, y-axis, and z-axis.

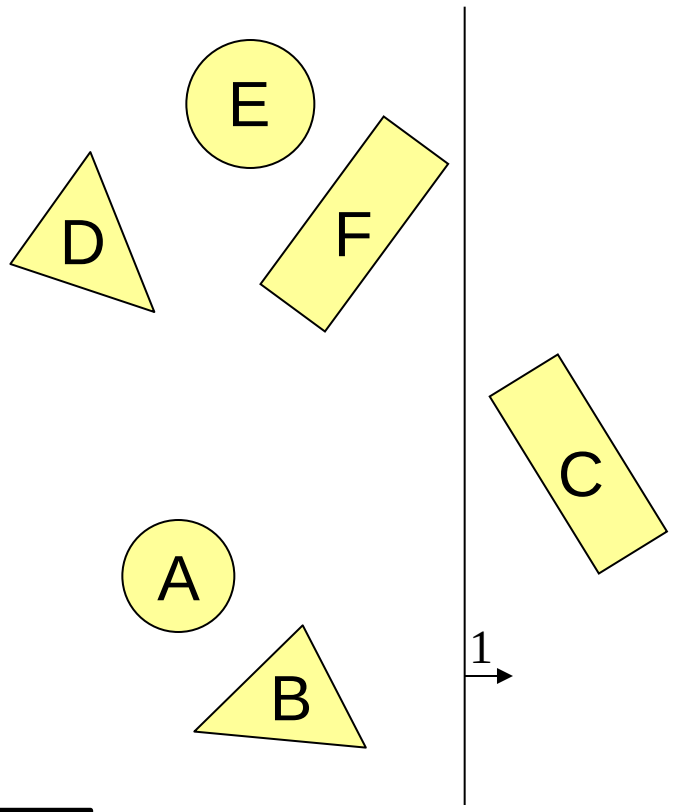


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.

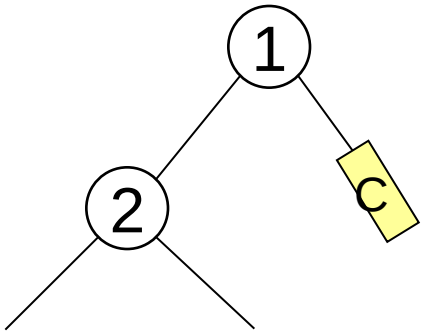


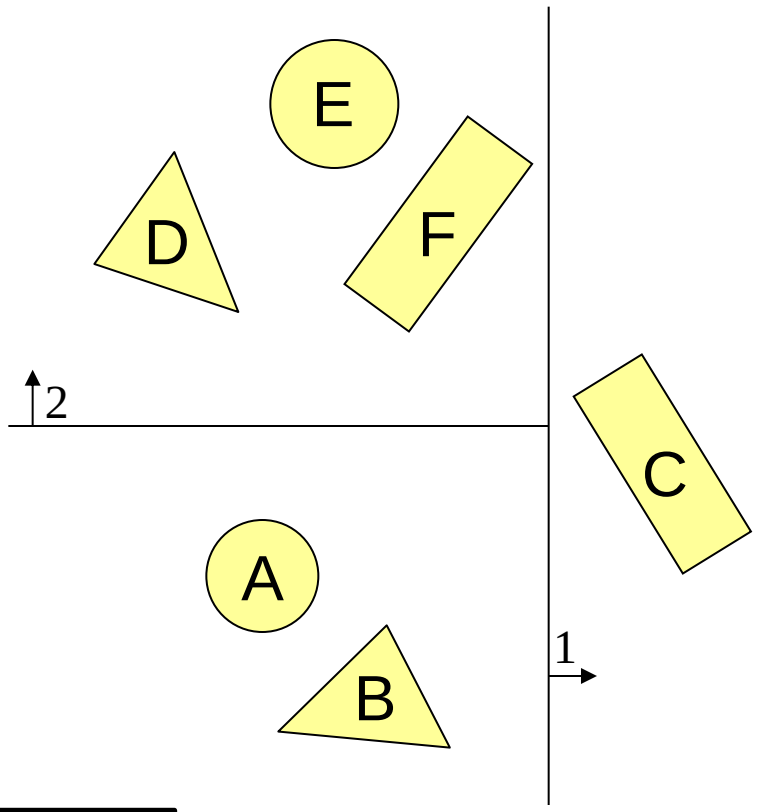


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.

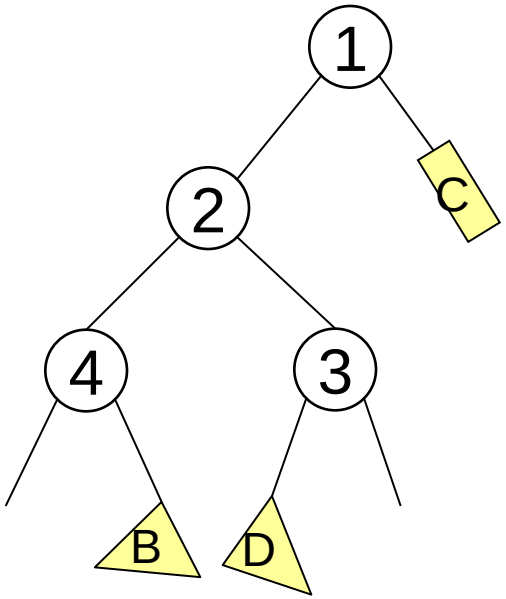


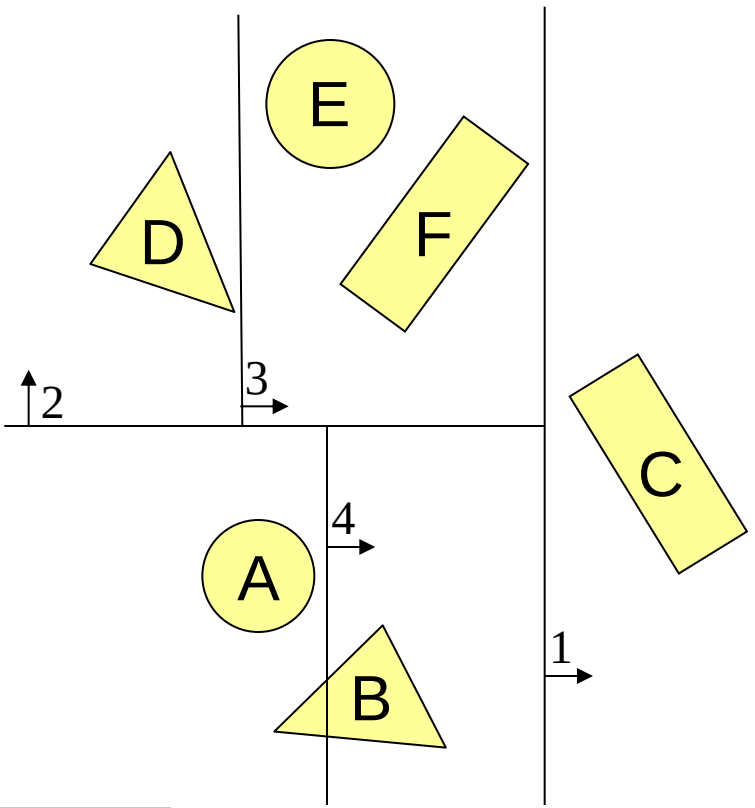


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.

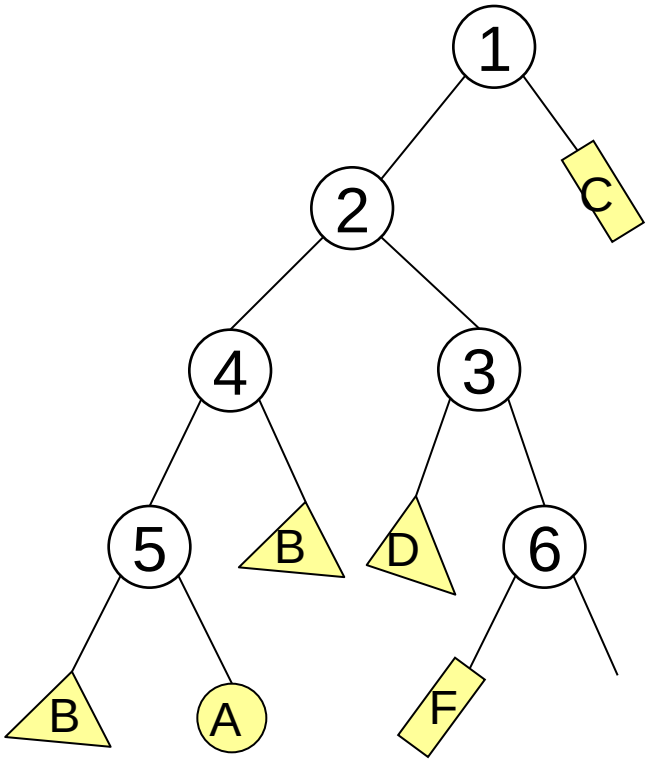


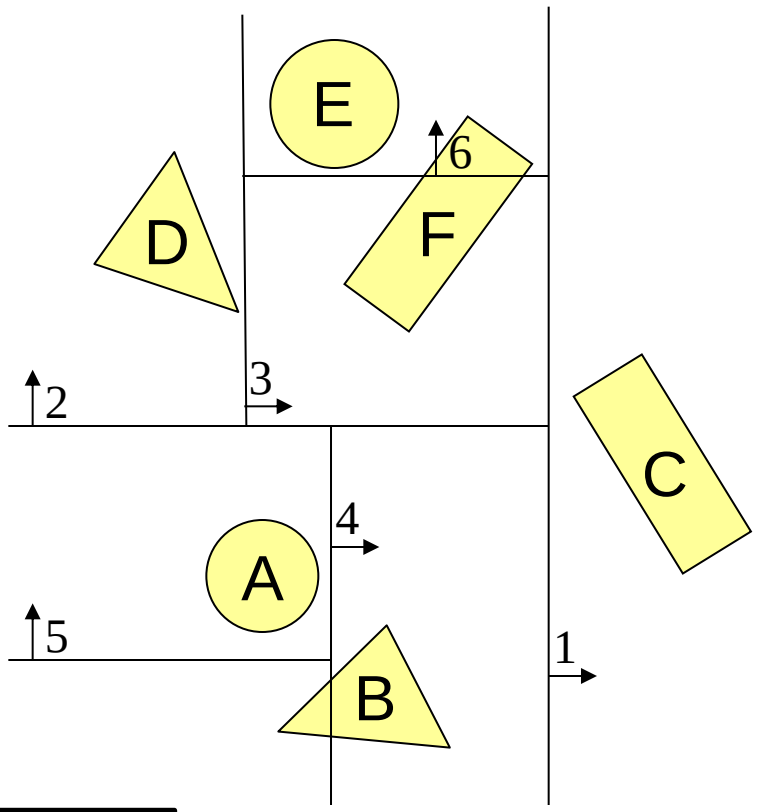


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.

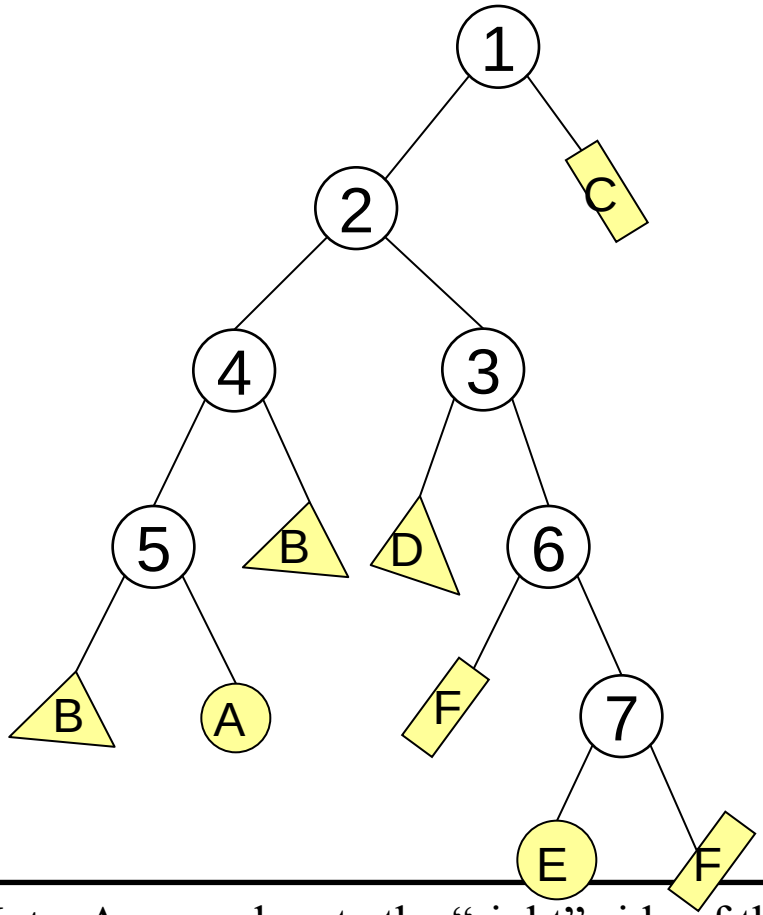


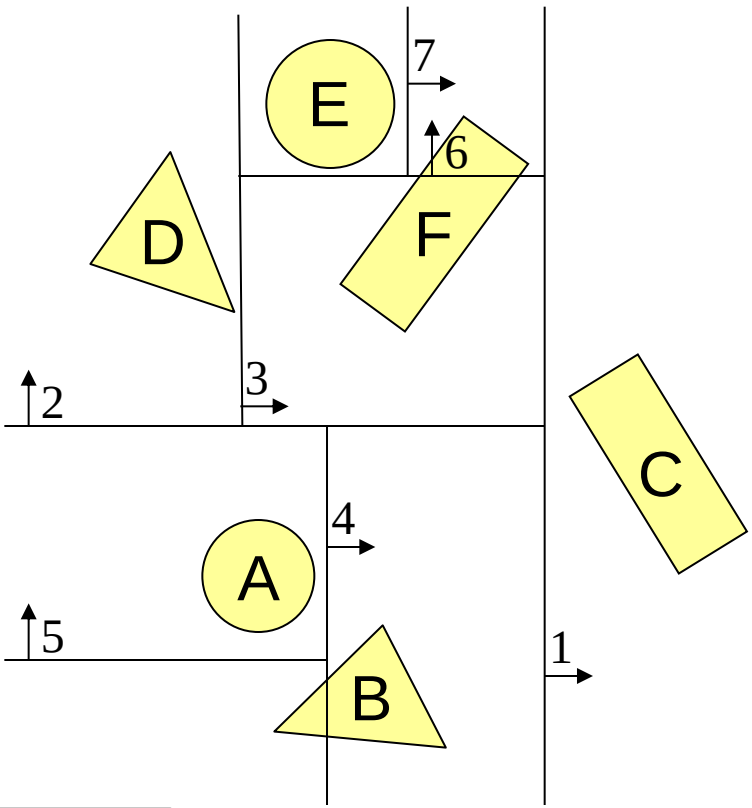


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.

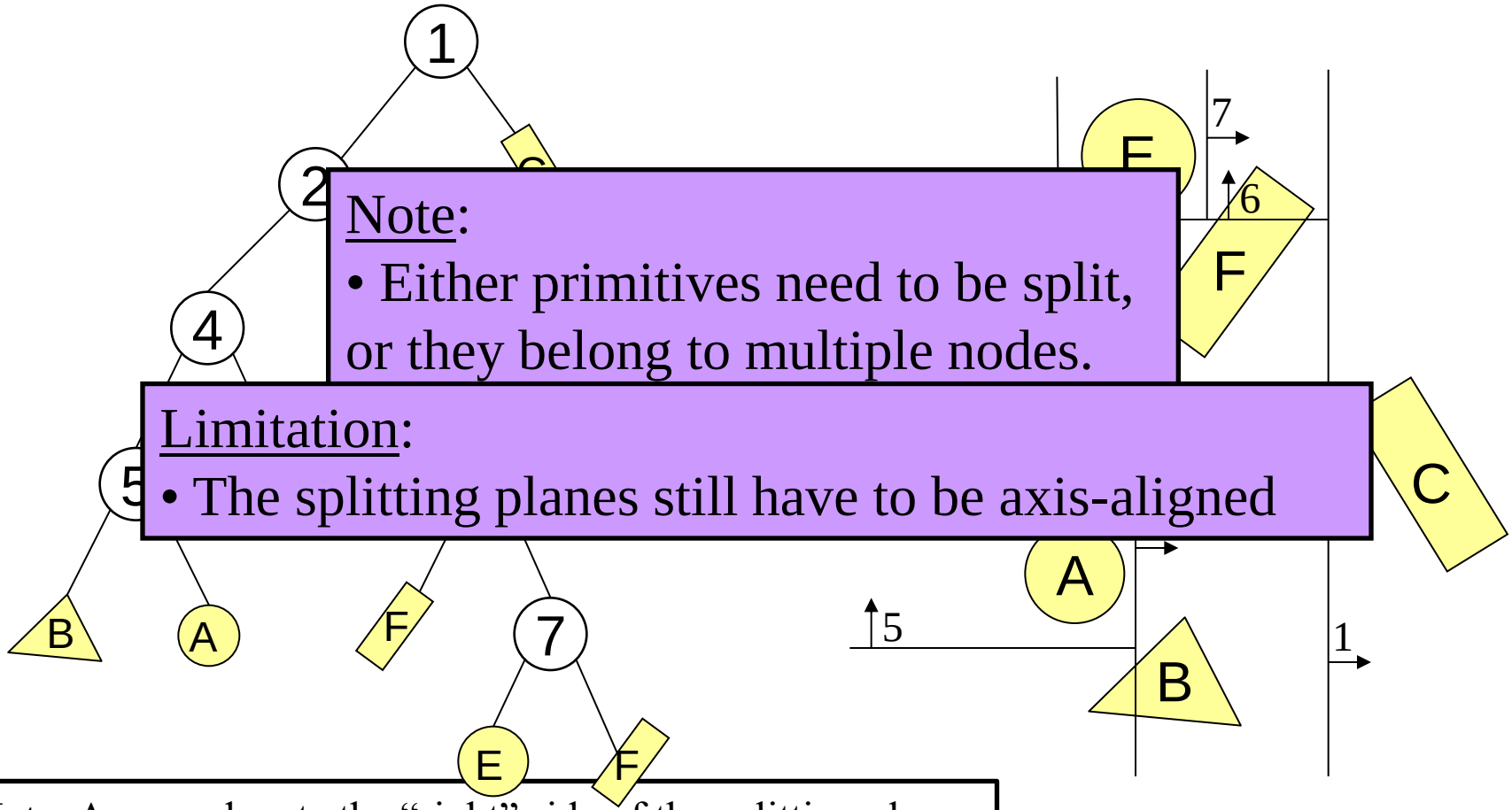




Note: Arrows denote the "right" side of the splitting plane.

Space Partition: k-D Trees

- Alternate between splitting along the x-axis, y-axis, and z-axis.



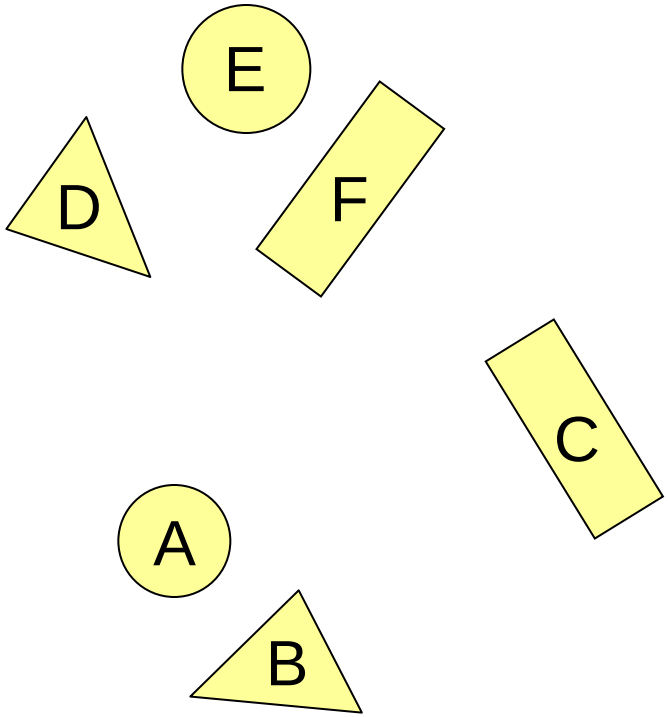
Note:
• Either primitives need to be split, or they belong to multiple nodes.

Limitation:
• The splitting planes still have to be axis-aligned

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

Space Partition: BSP Tree

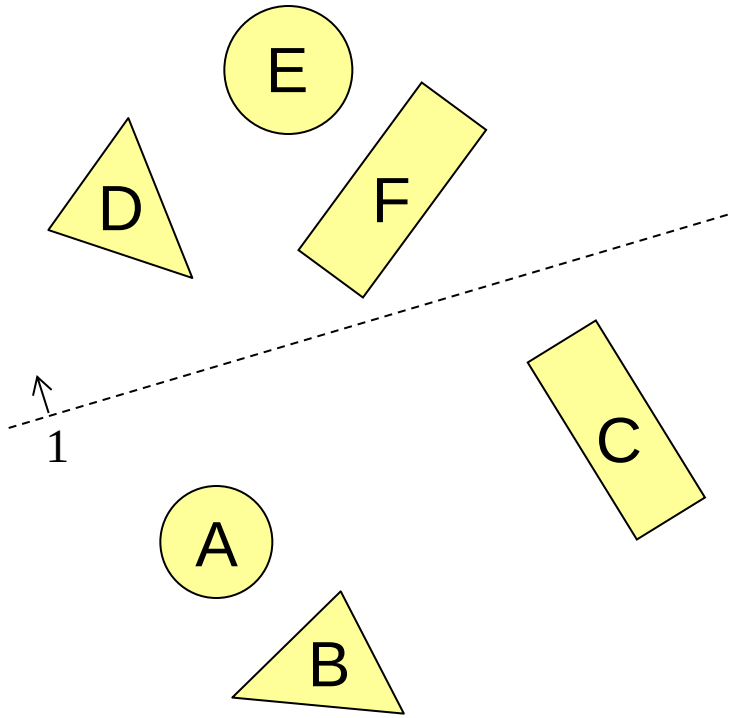
- With a Binary Space Partition (BSP) we recursively partition space by planes



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

Space Partition: BSP Tree

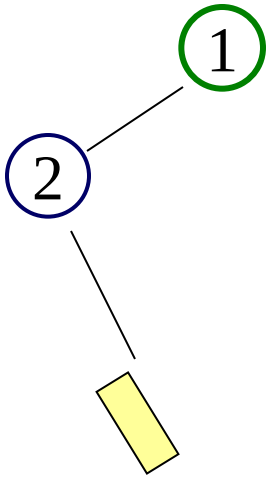
- With a Binary Space Partition (BSP) we recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.

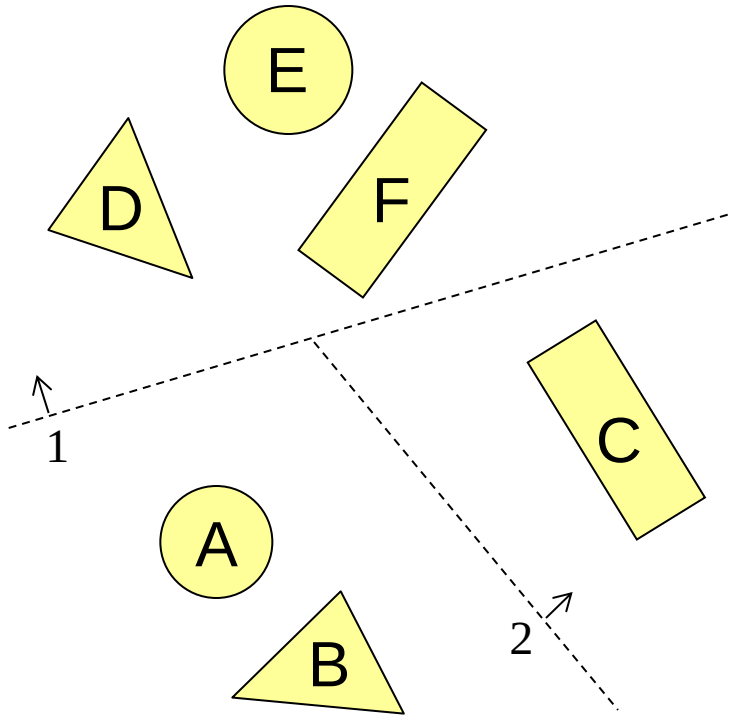


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- With a Binary Space Partition (BSP) we recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.

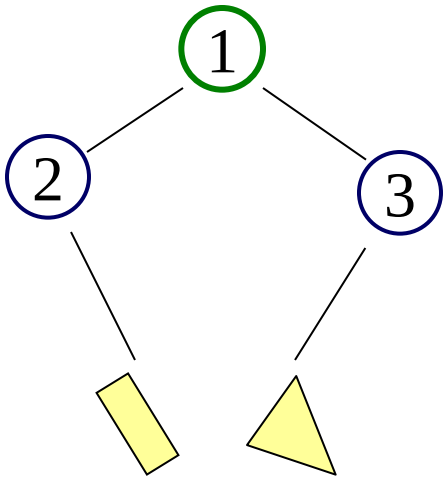


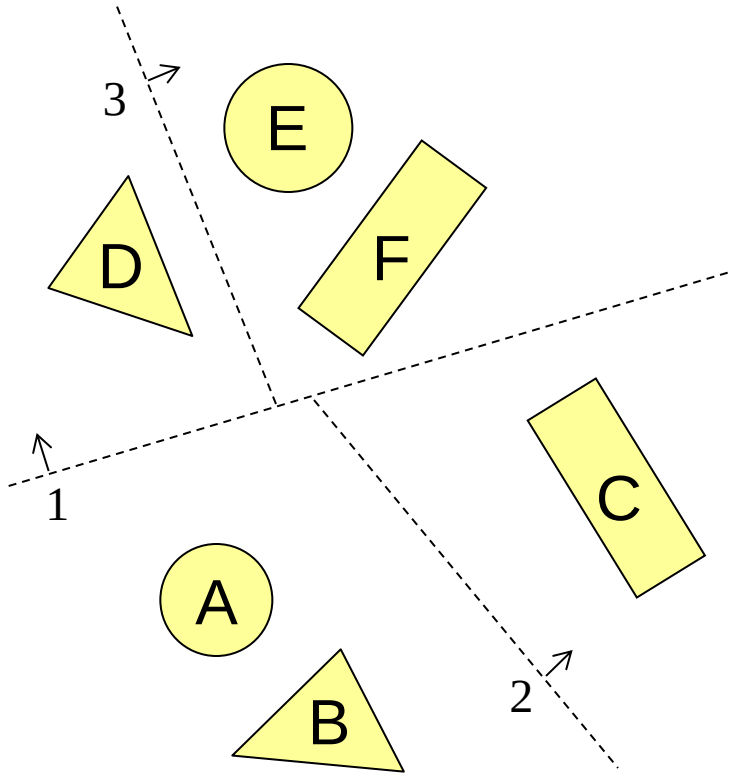


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- With a Binary Space Partition (BSP) we recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.

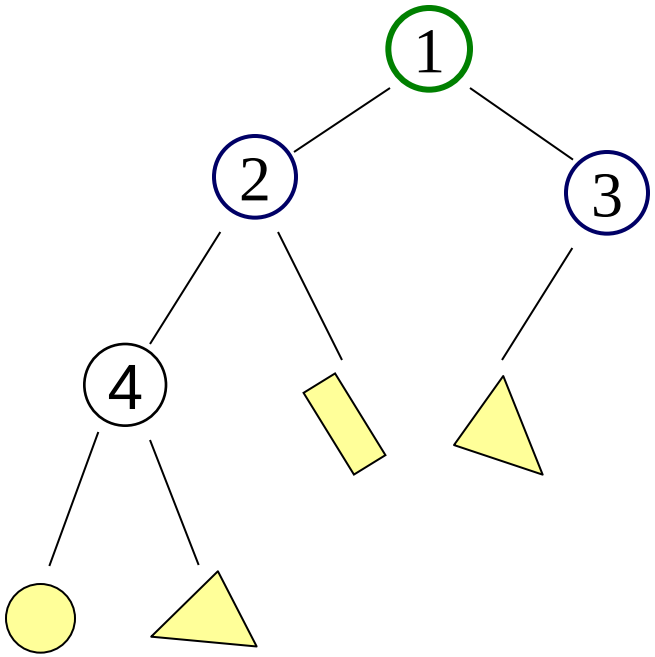


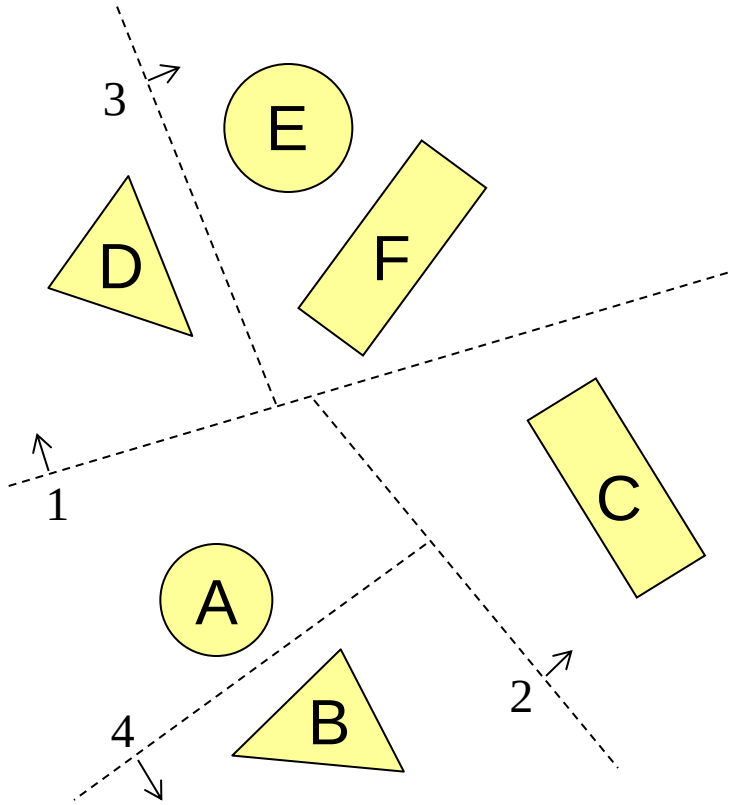


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- With a Binary Space Partition (BSP) we recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.

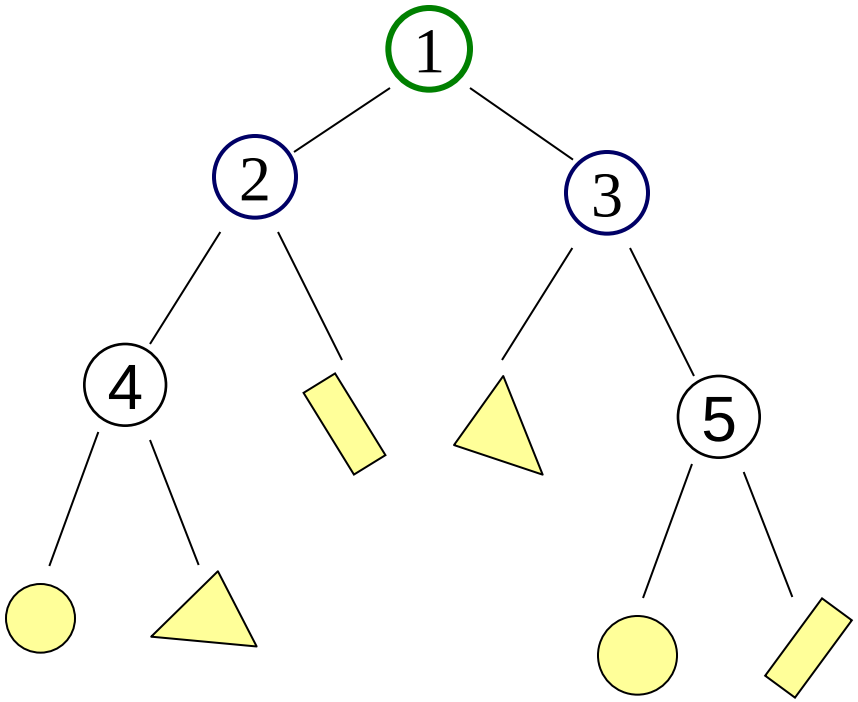


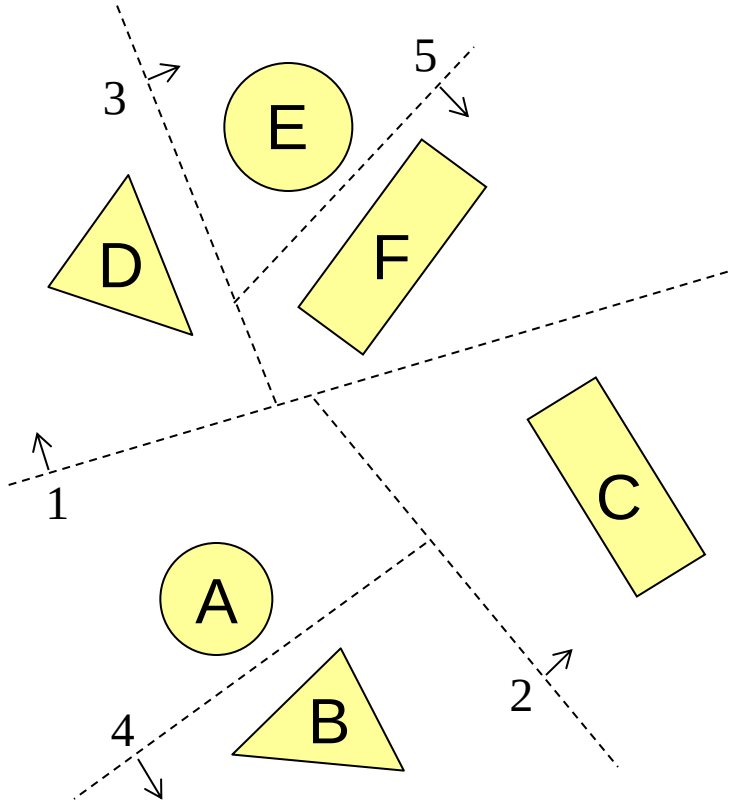


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- With a Binary Space Partition (BSP) we recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.

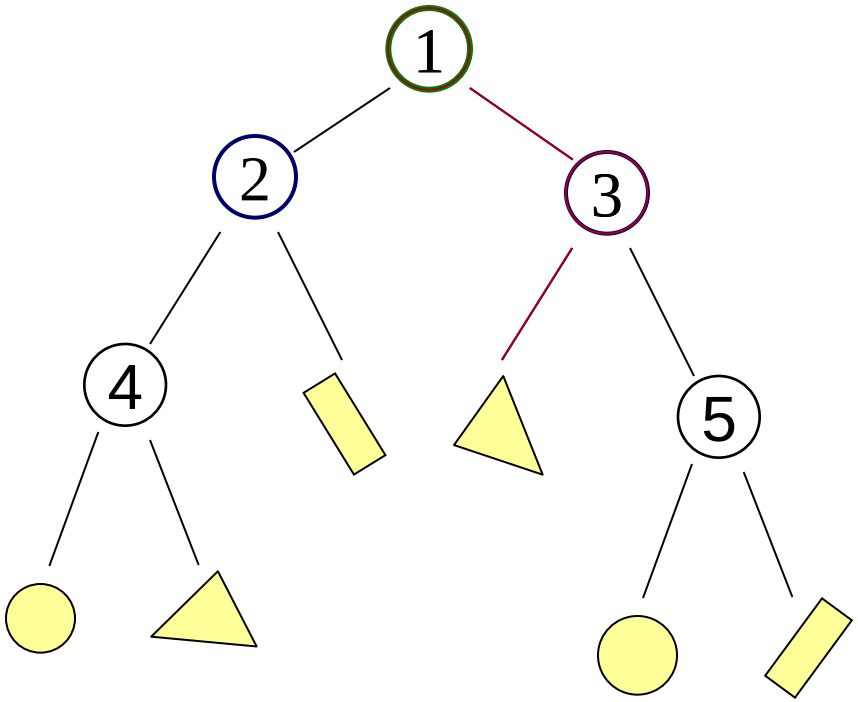


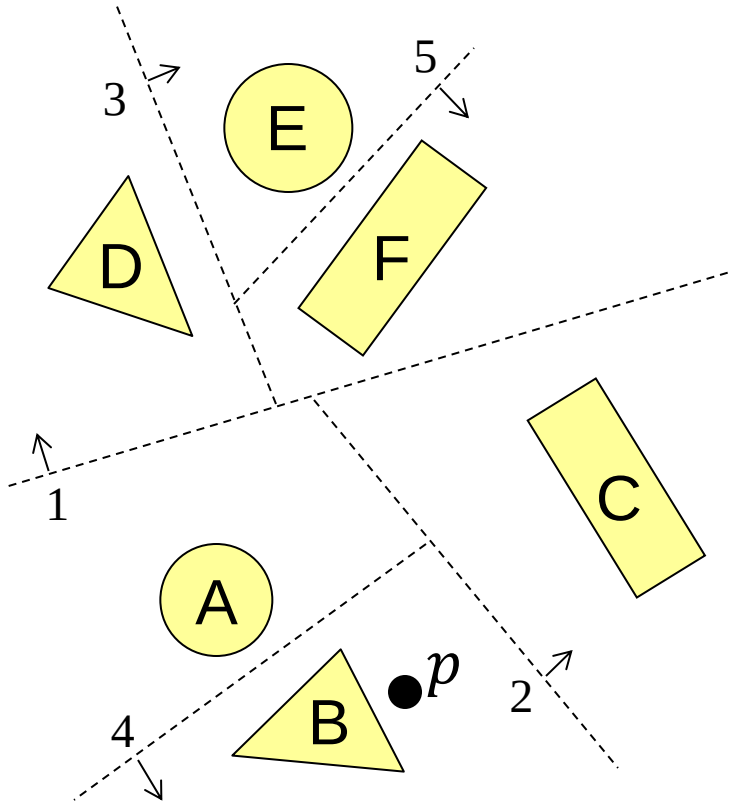


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection

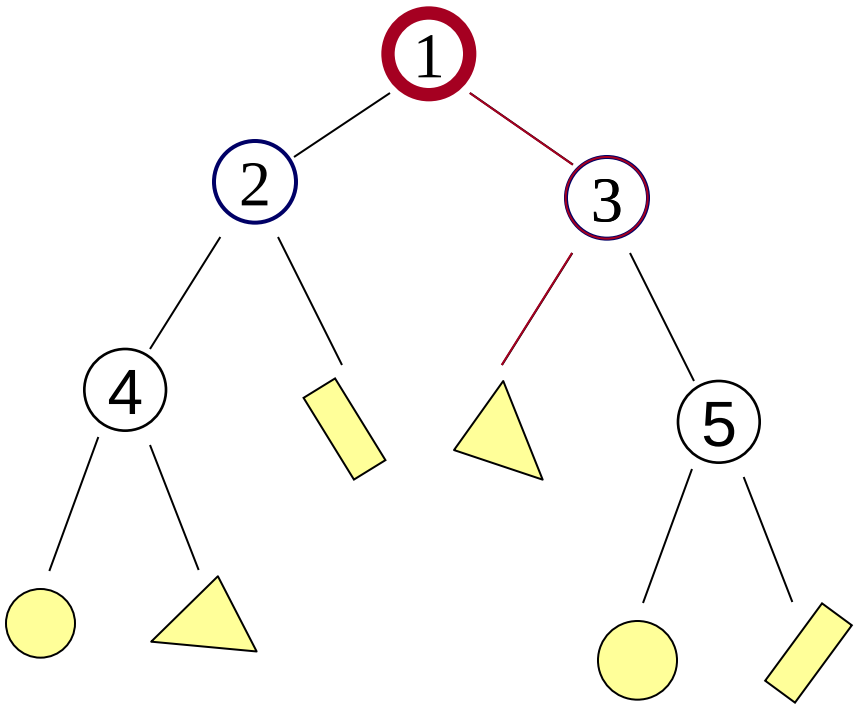


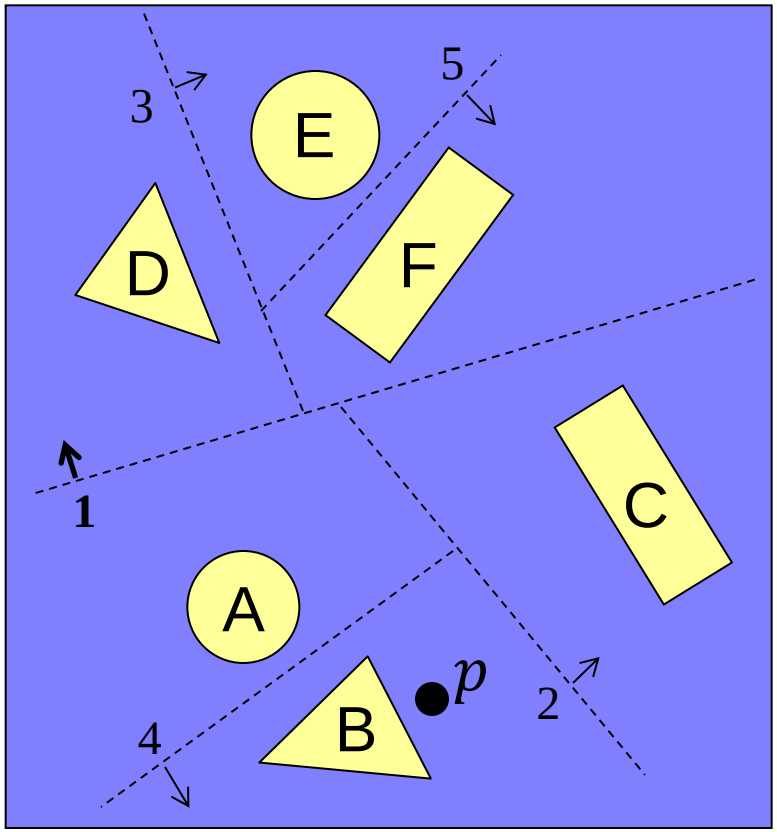


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection
  - Recursively test what side we are on

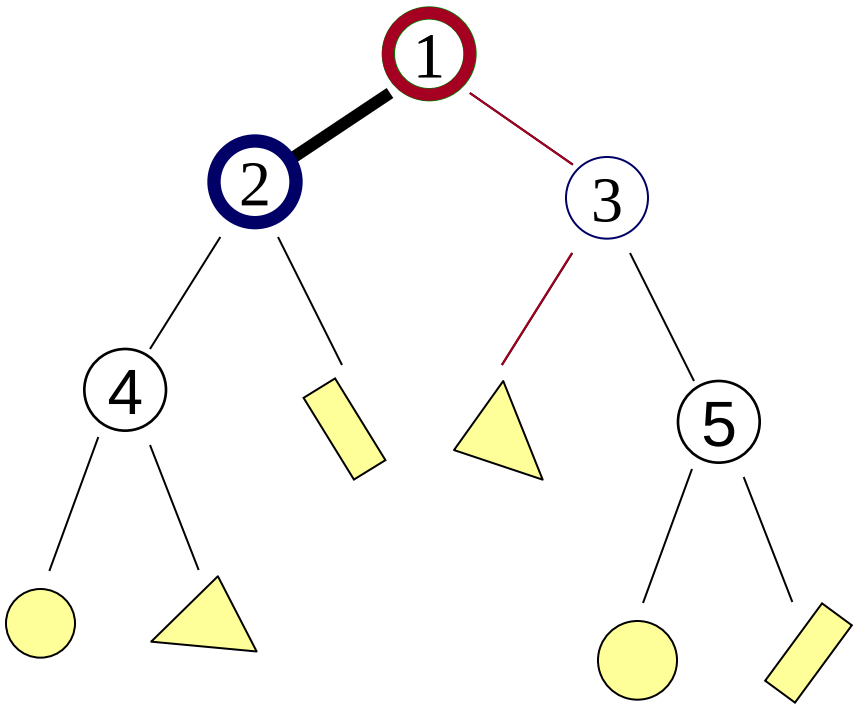


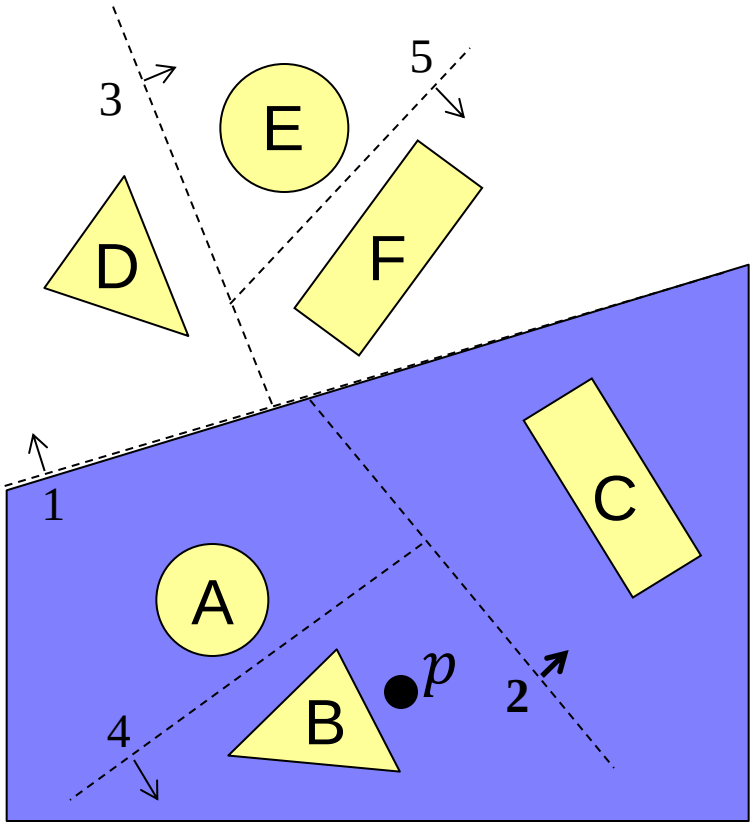


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

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

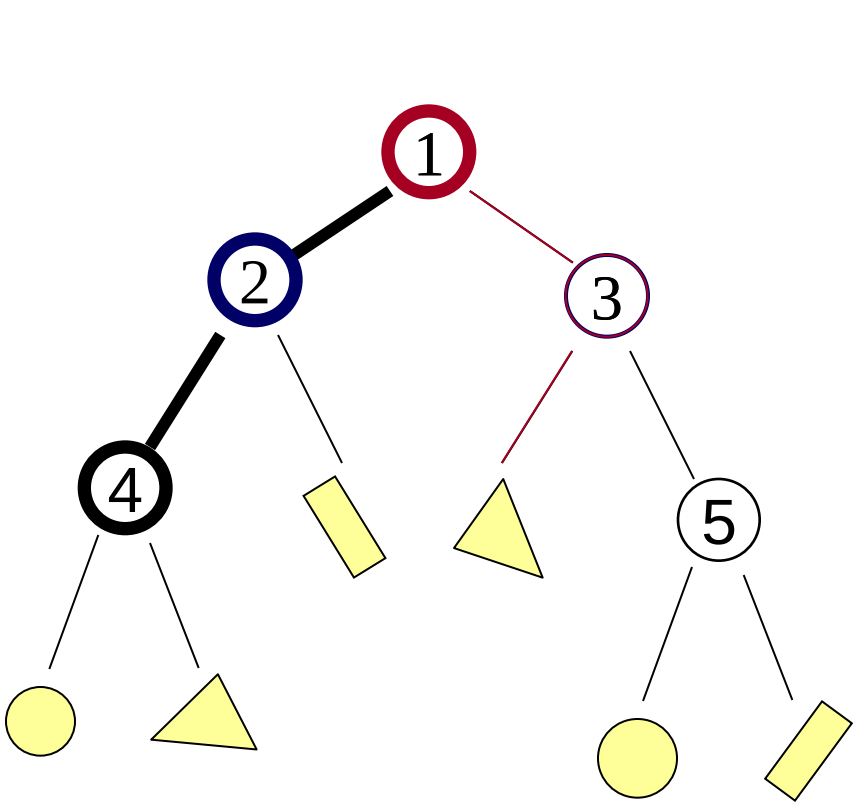


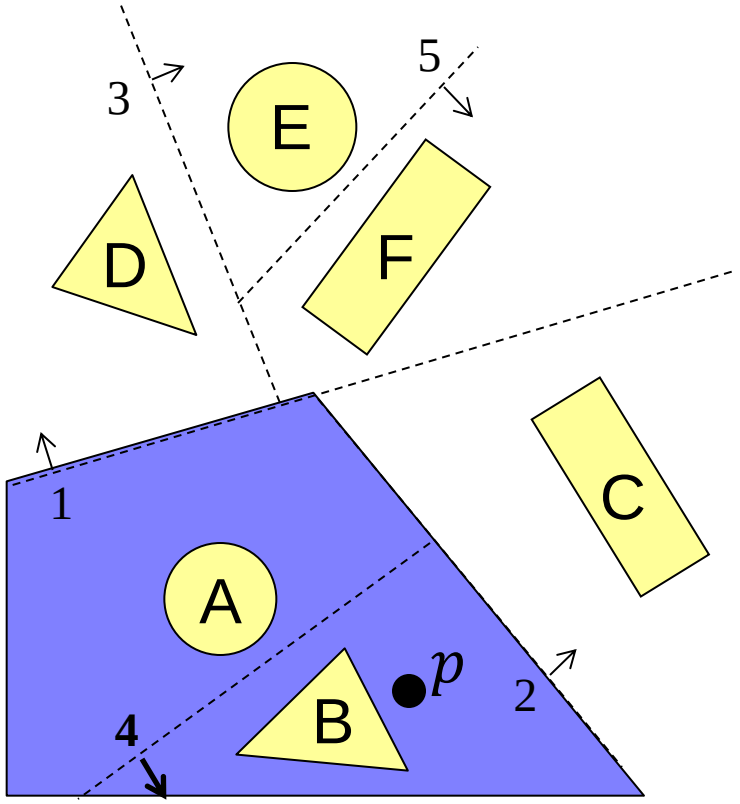


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection
  - Recursively test what side we are on
    - » Left of 2 → 4

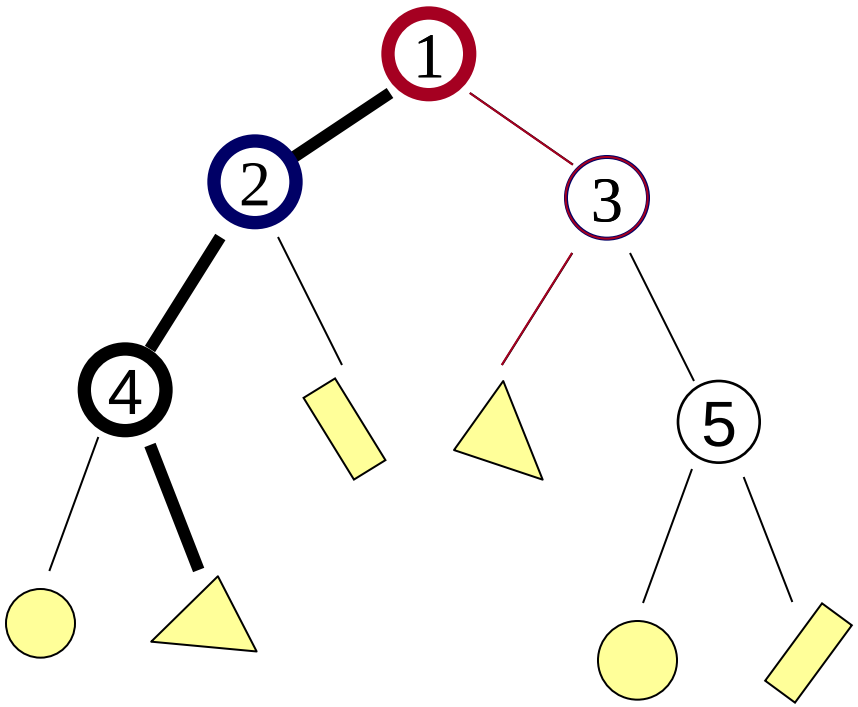


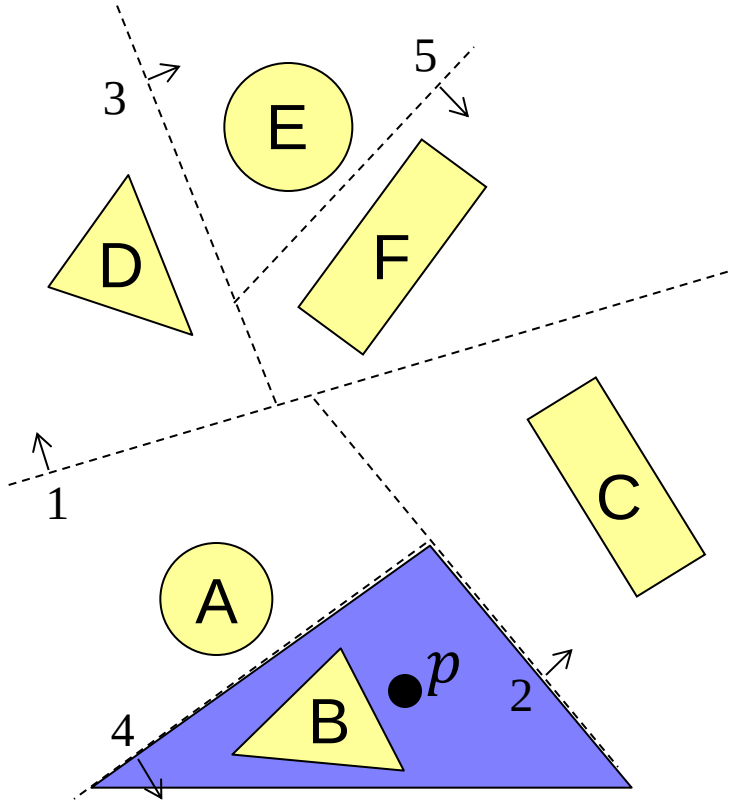


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection
  - Recursively test what side we are on
    - » Right of 4 → Test B

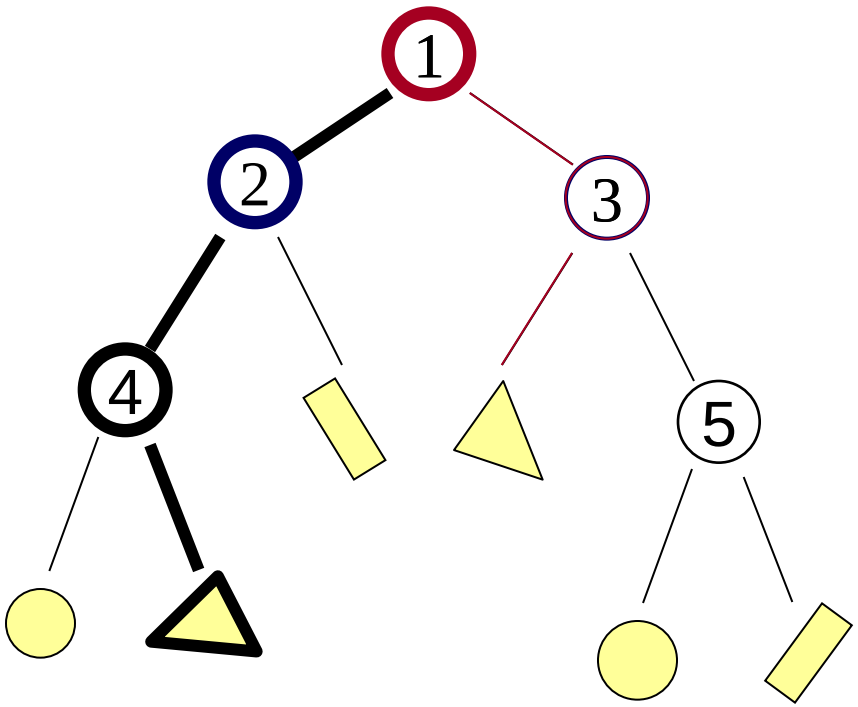


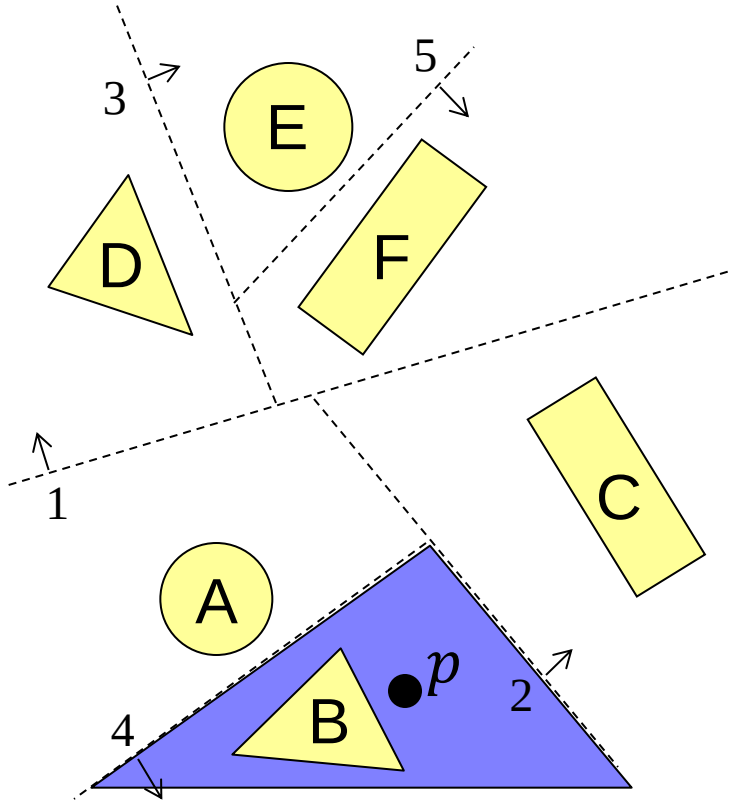


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection
  - Recursively test what side we are on
    - » Missed B. No intersection!

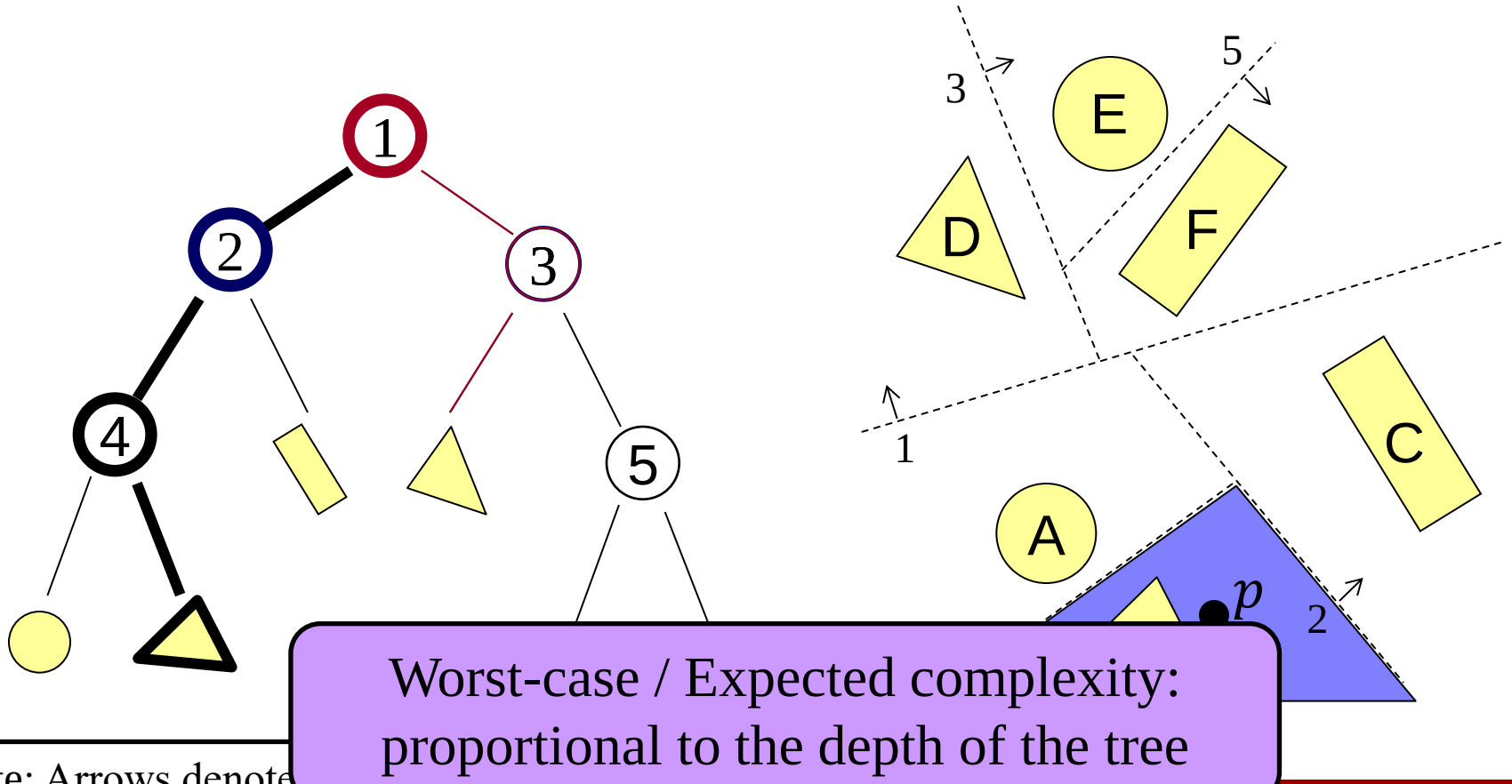




Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Point Intersection
  - 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

Observation:

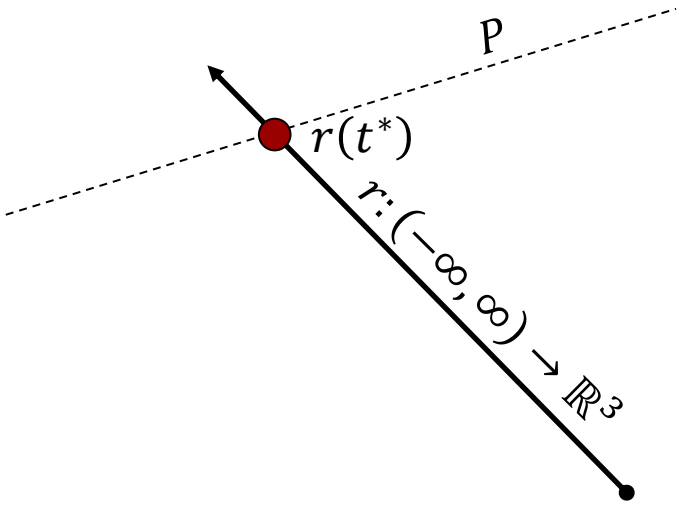
Assume we are given a ray:

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

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

Observation:

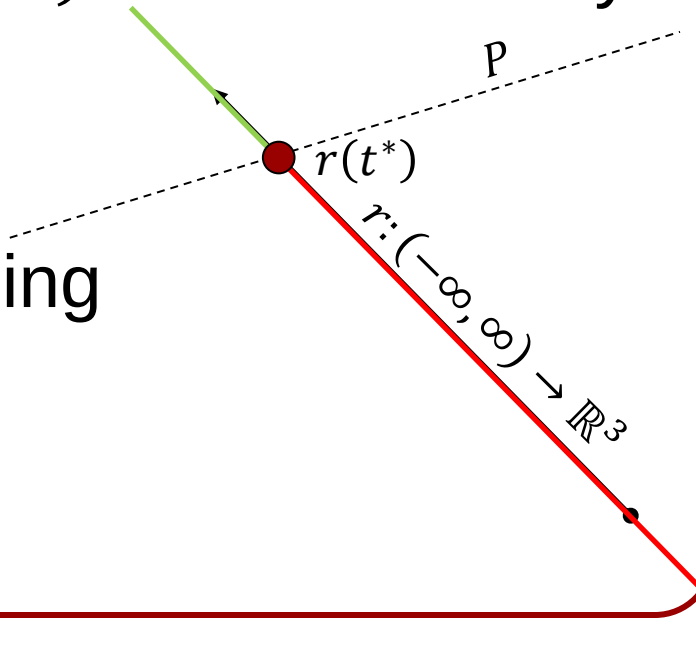
Assume we are given a ray:

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

and a plane P (assuming not parallel).

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

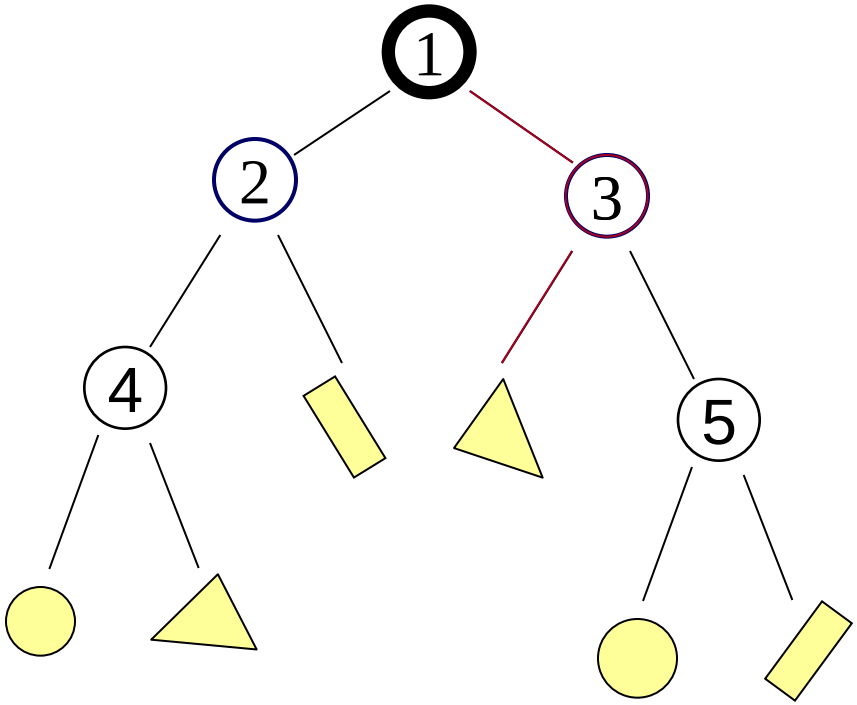
$$r(t^*) \in P.$$

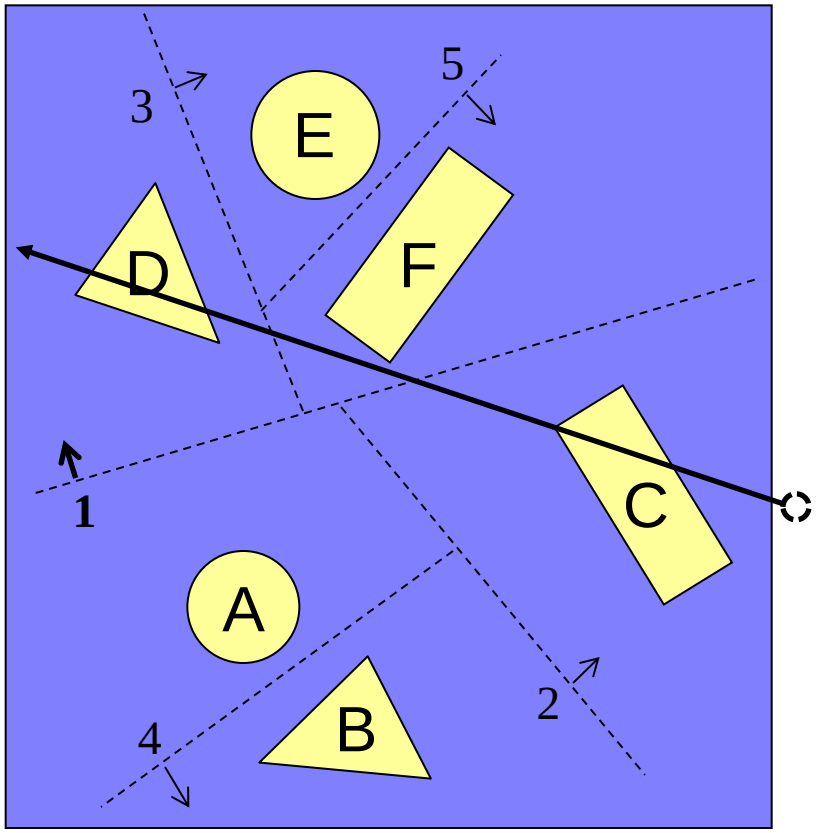


- This partitions the line containing the ray in two parts:
  - Back: $r(t)$ with $t \in (-\infty, t^*)$
  - Front: $r(t)$ with $t \in (t^*, \infty)$

Space Partition: BSP Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

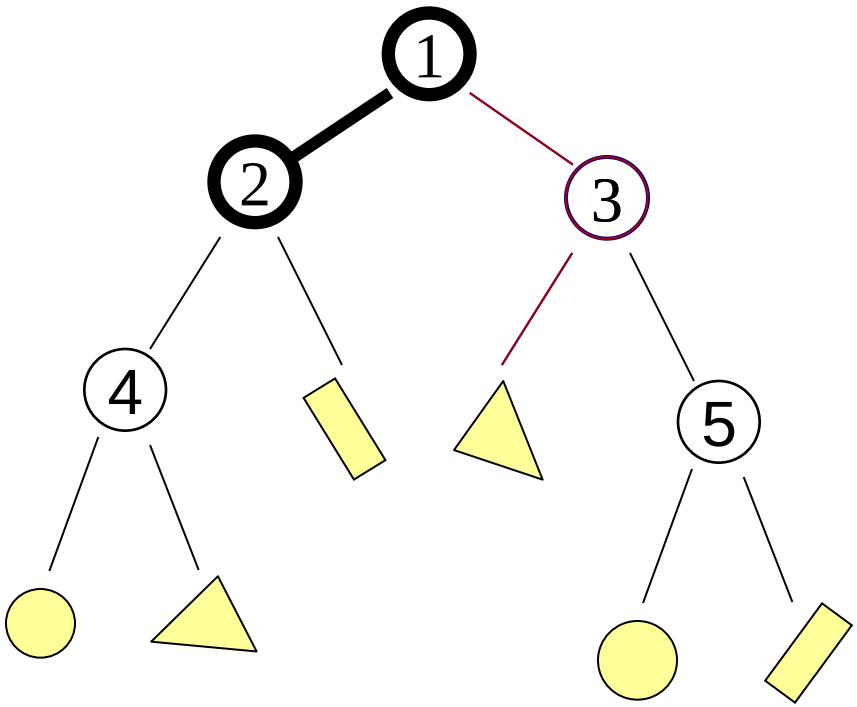


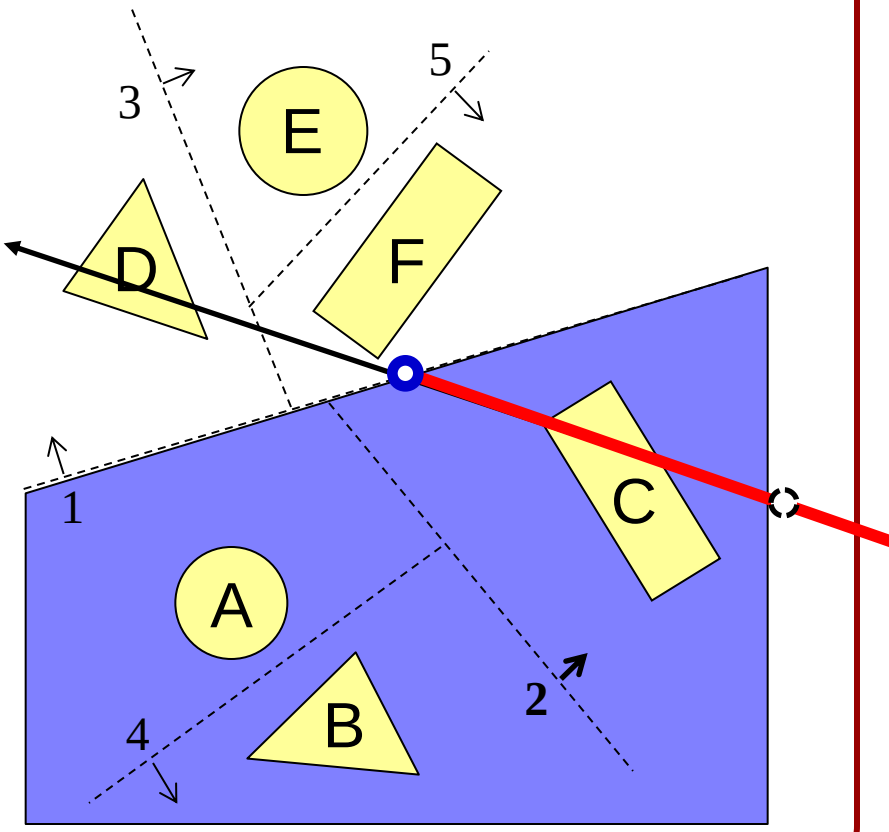


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to the left of 1

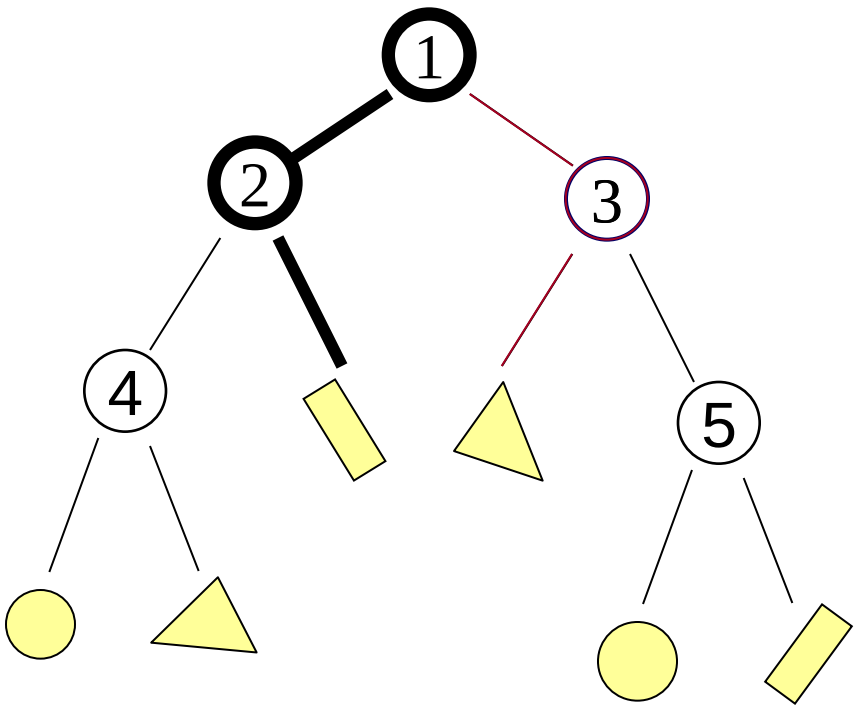


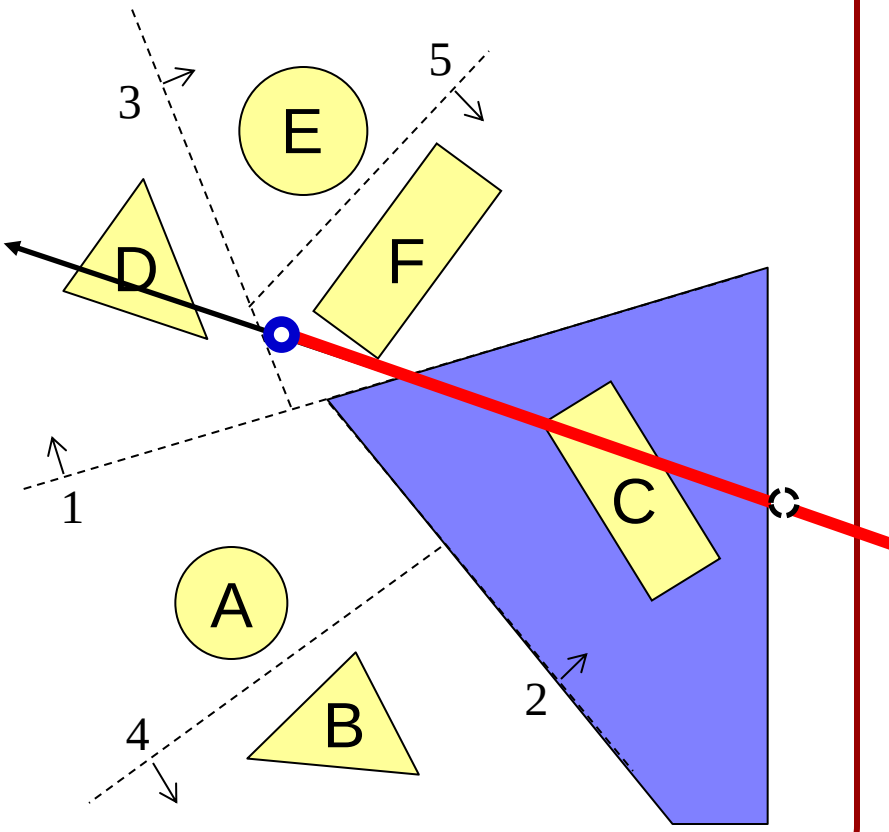


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to the right of 2

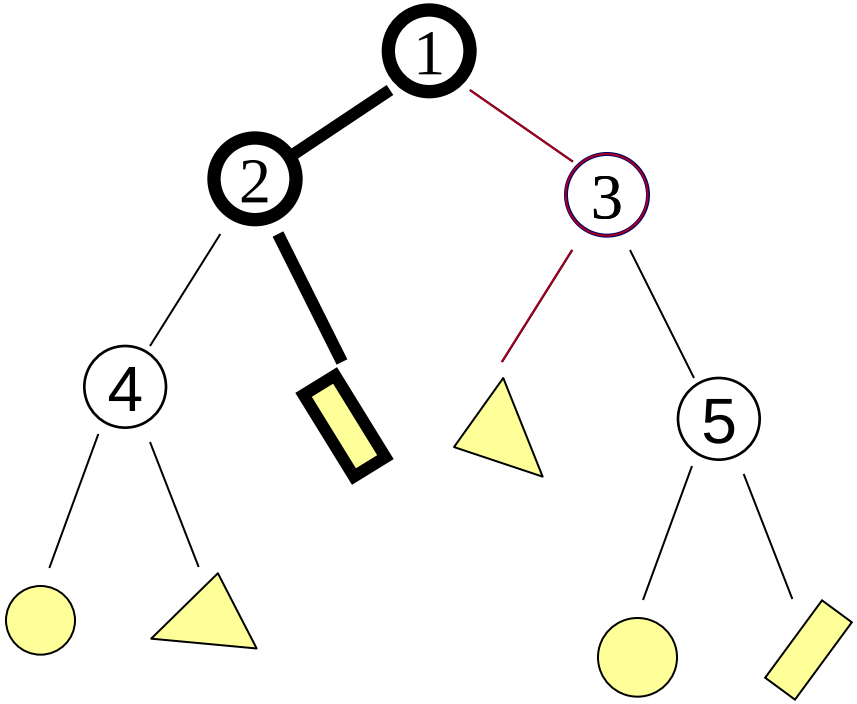


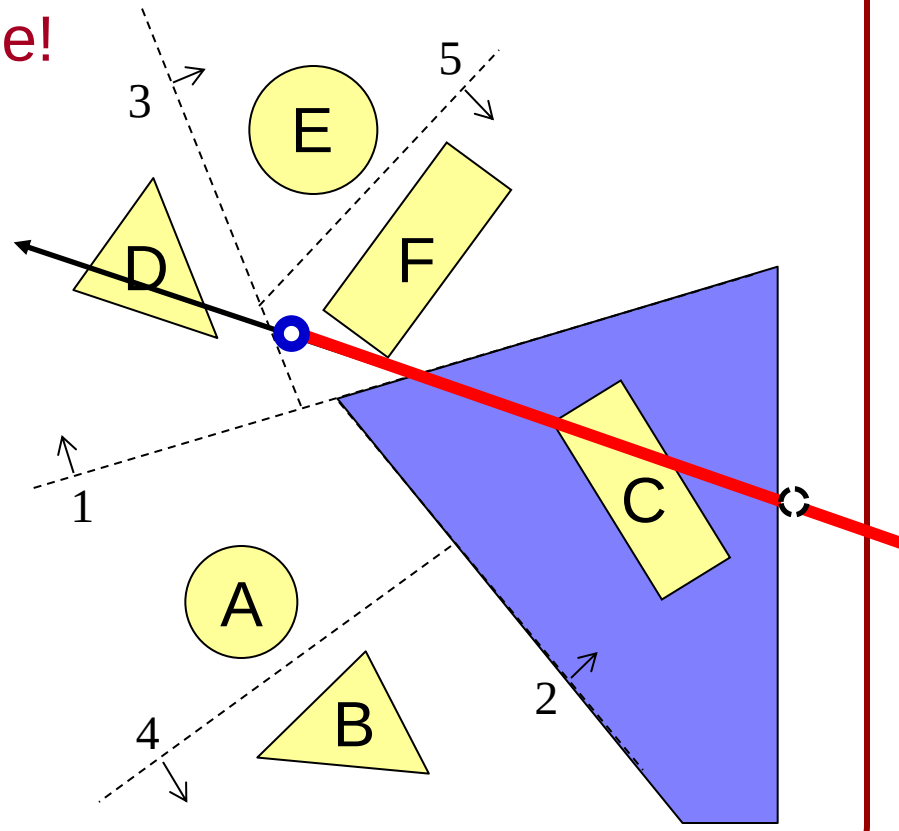


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Intersection with C. Done!

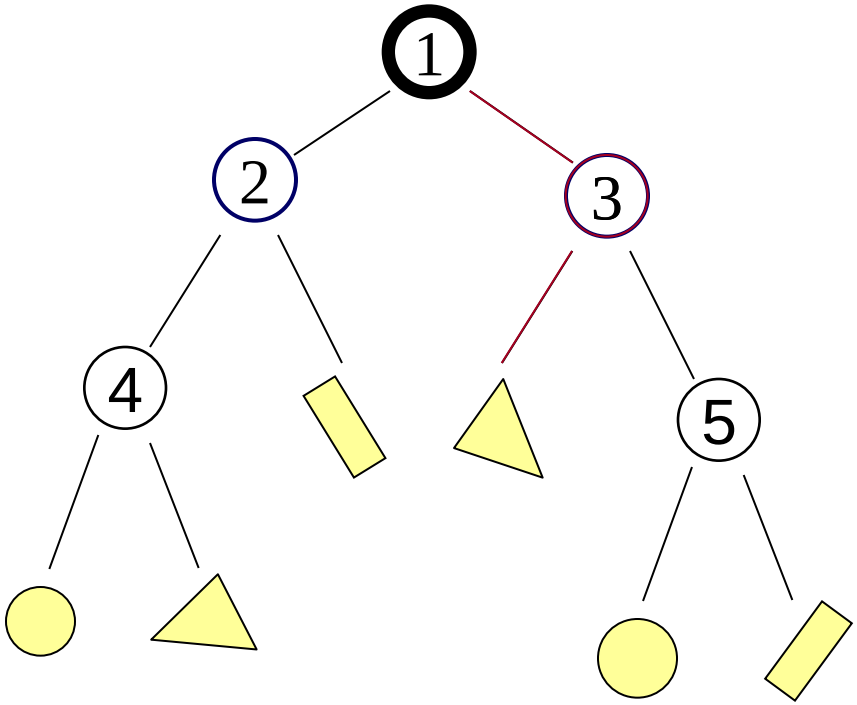


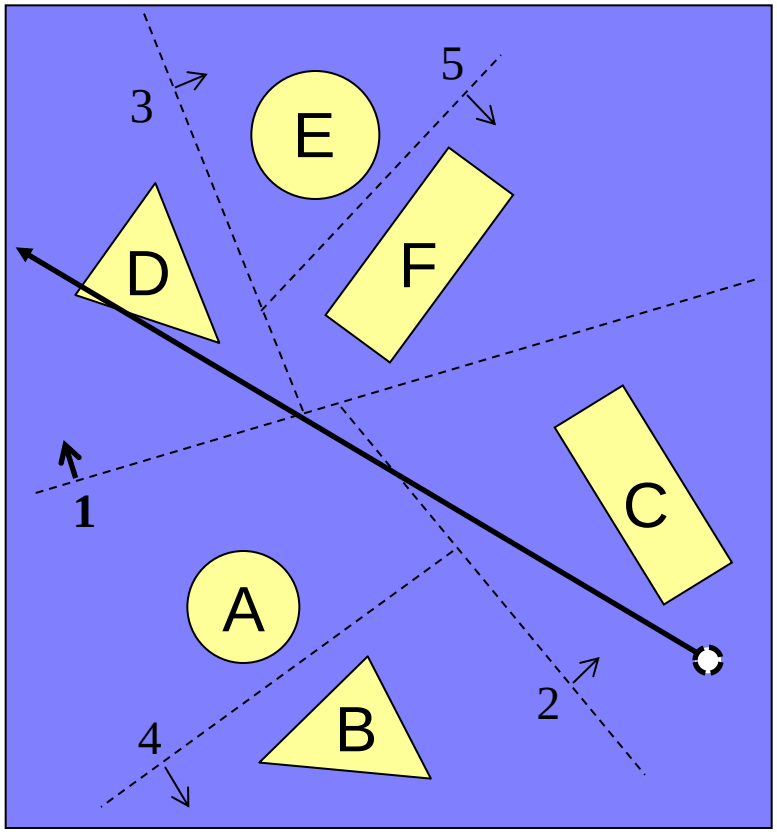


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:

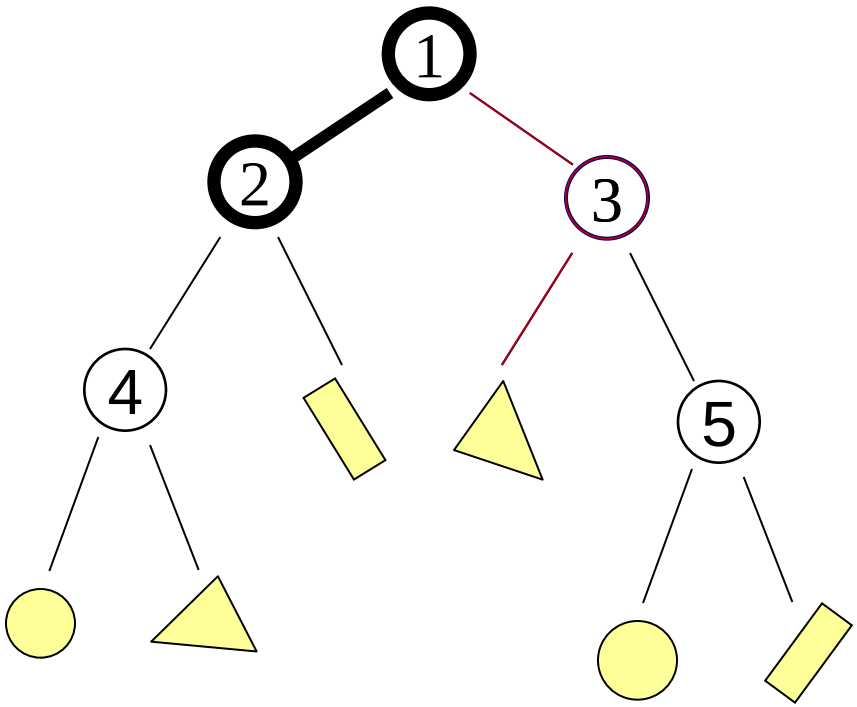


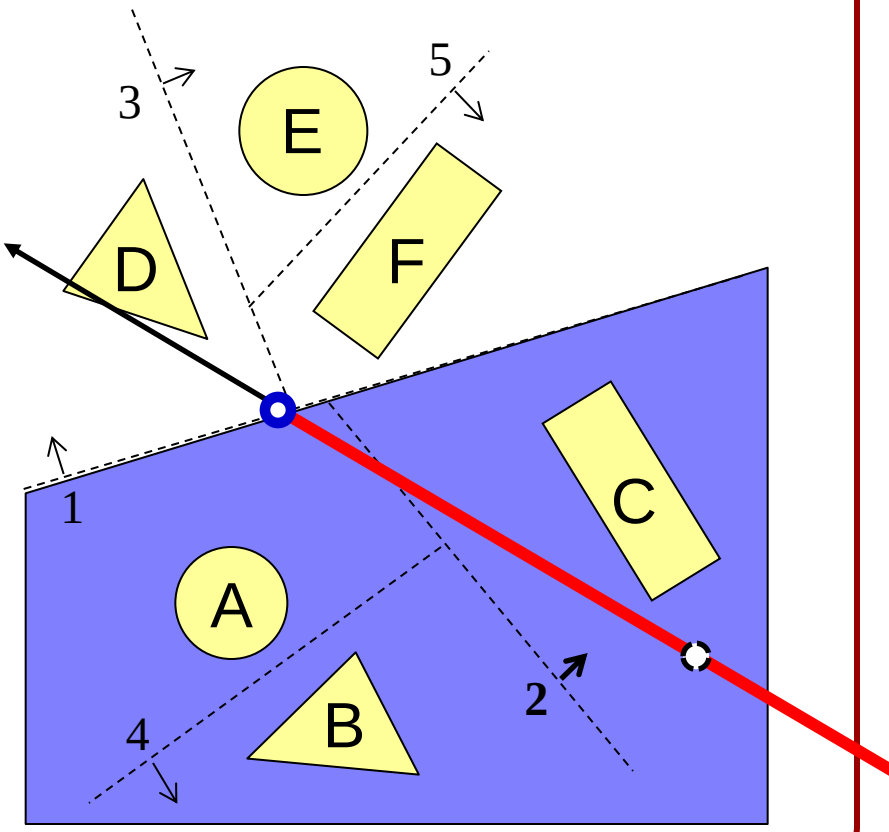


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to the left of 1

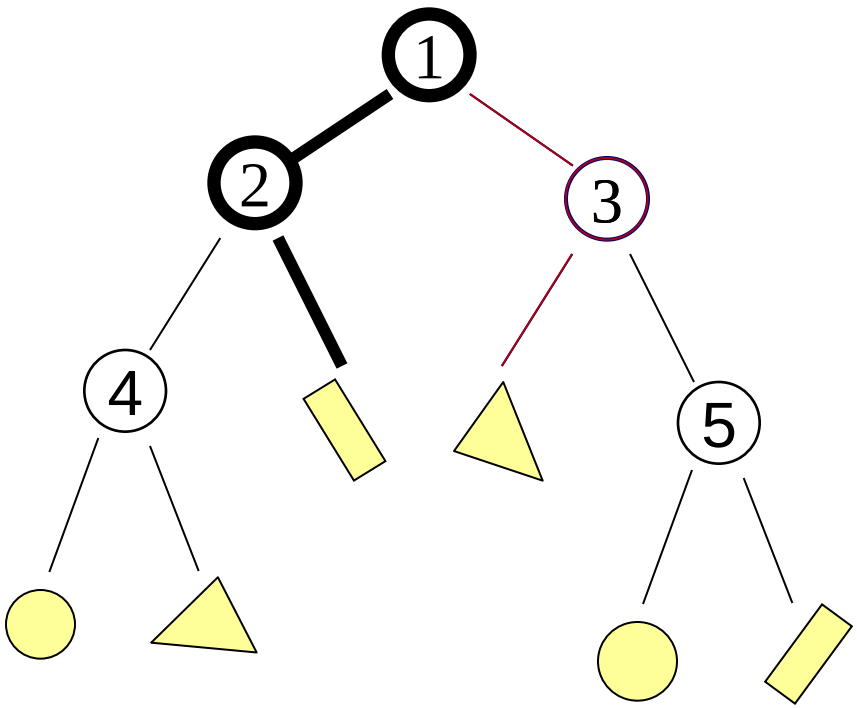


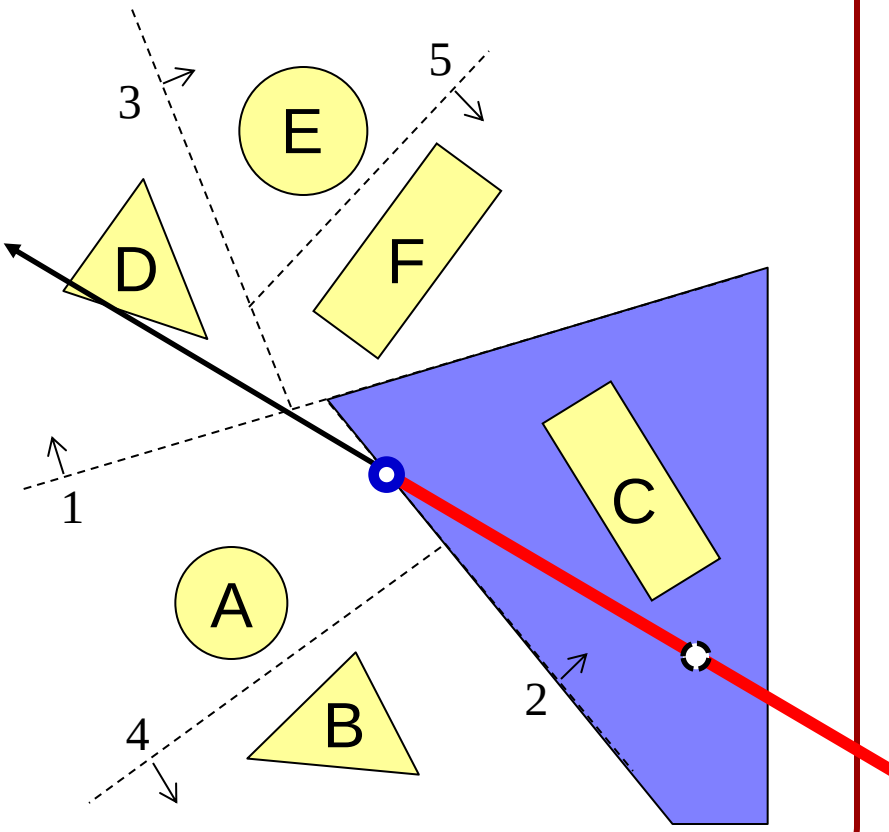


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to the right of 2

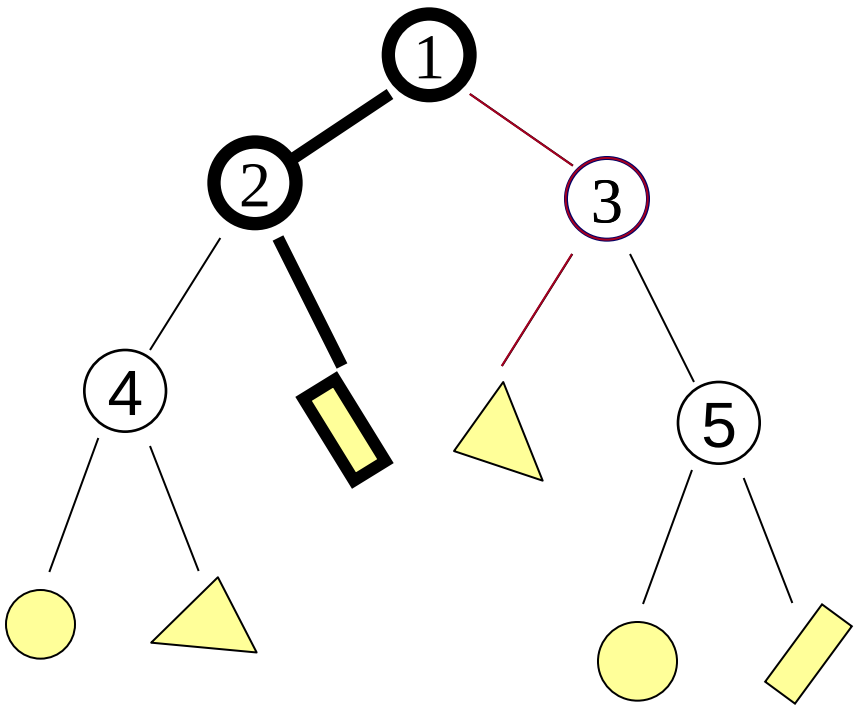


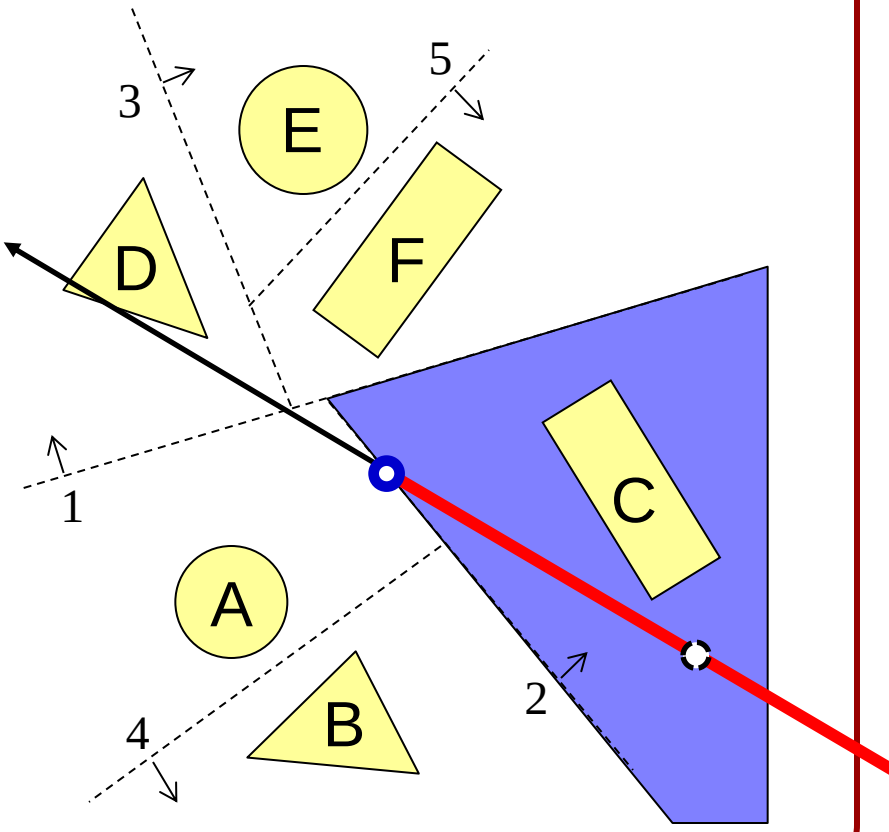


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Missed C. Recurse!

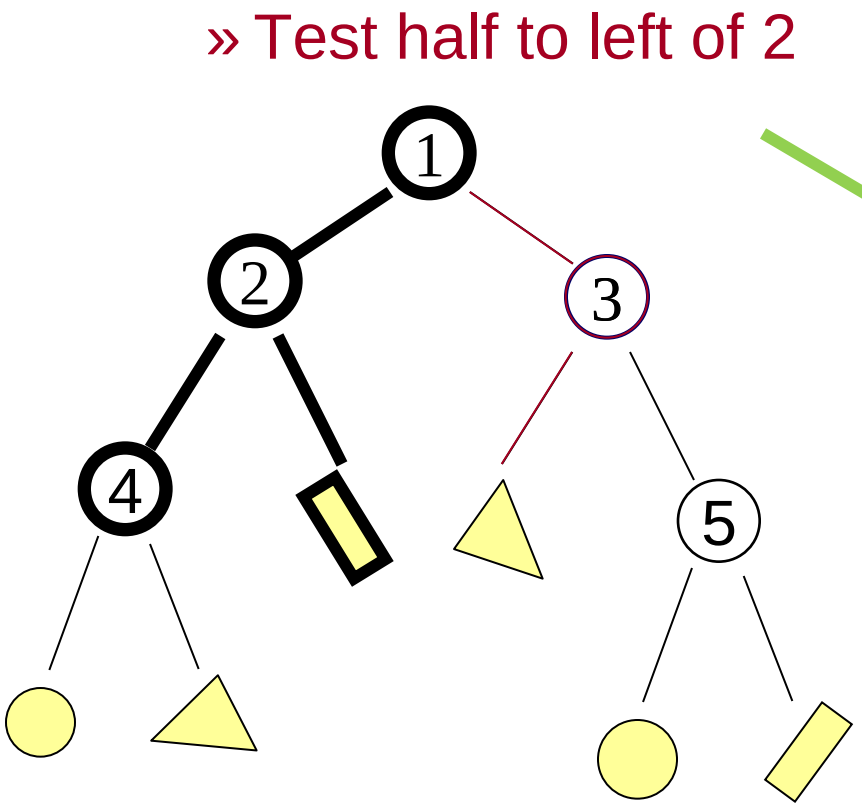


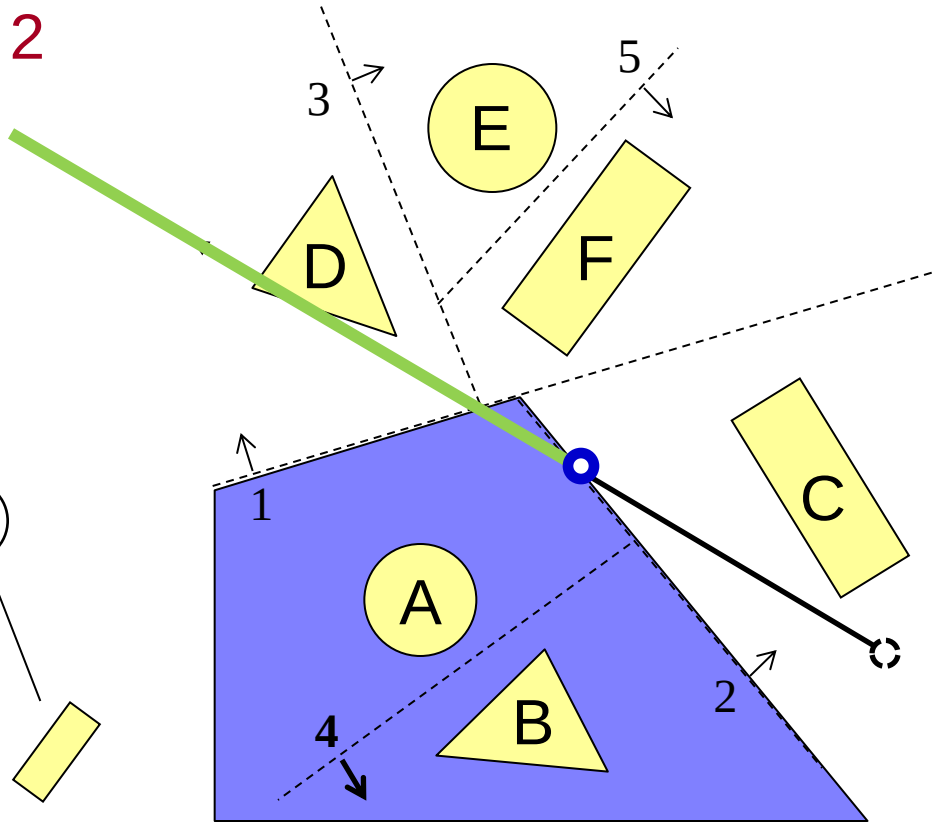


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to left of 2

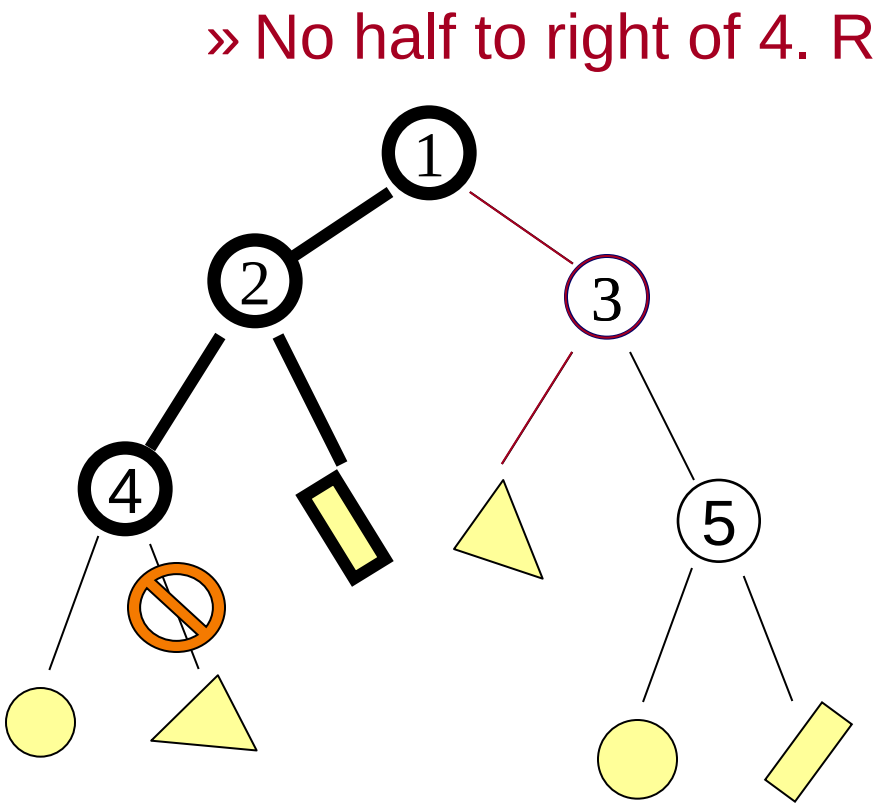


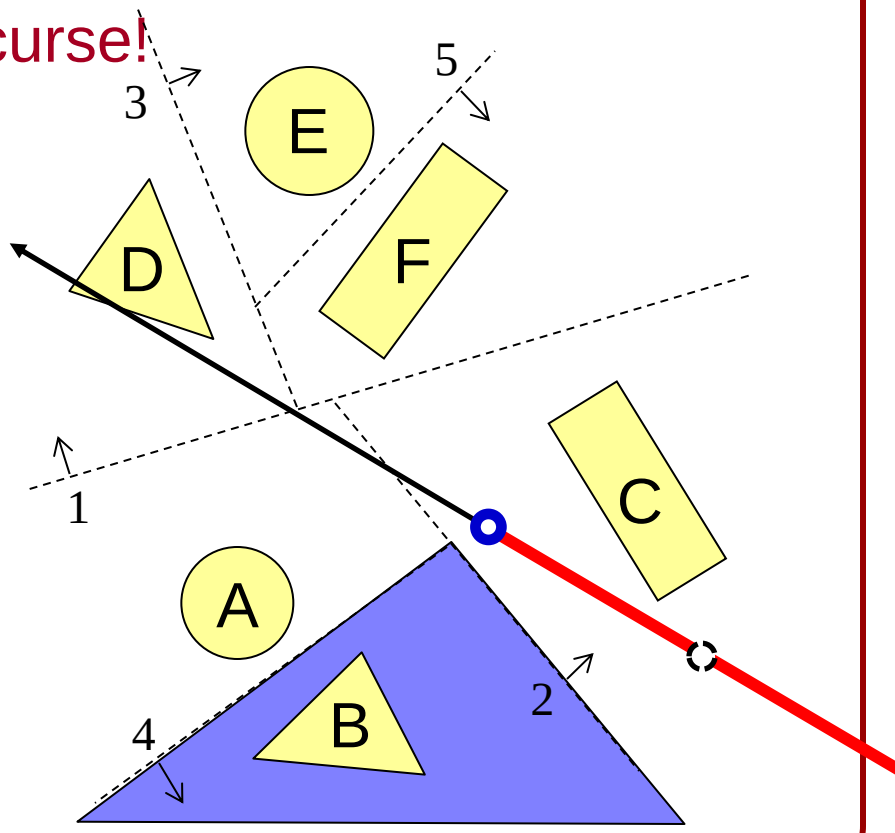


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » No half to right of 4. Recurse!

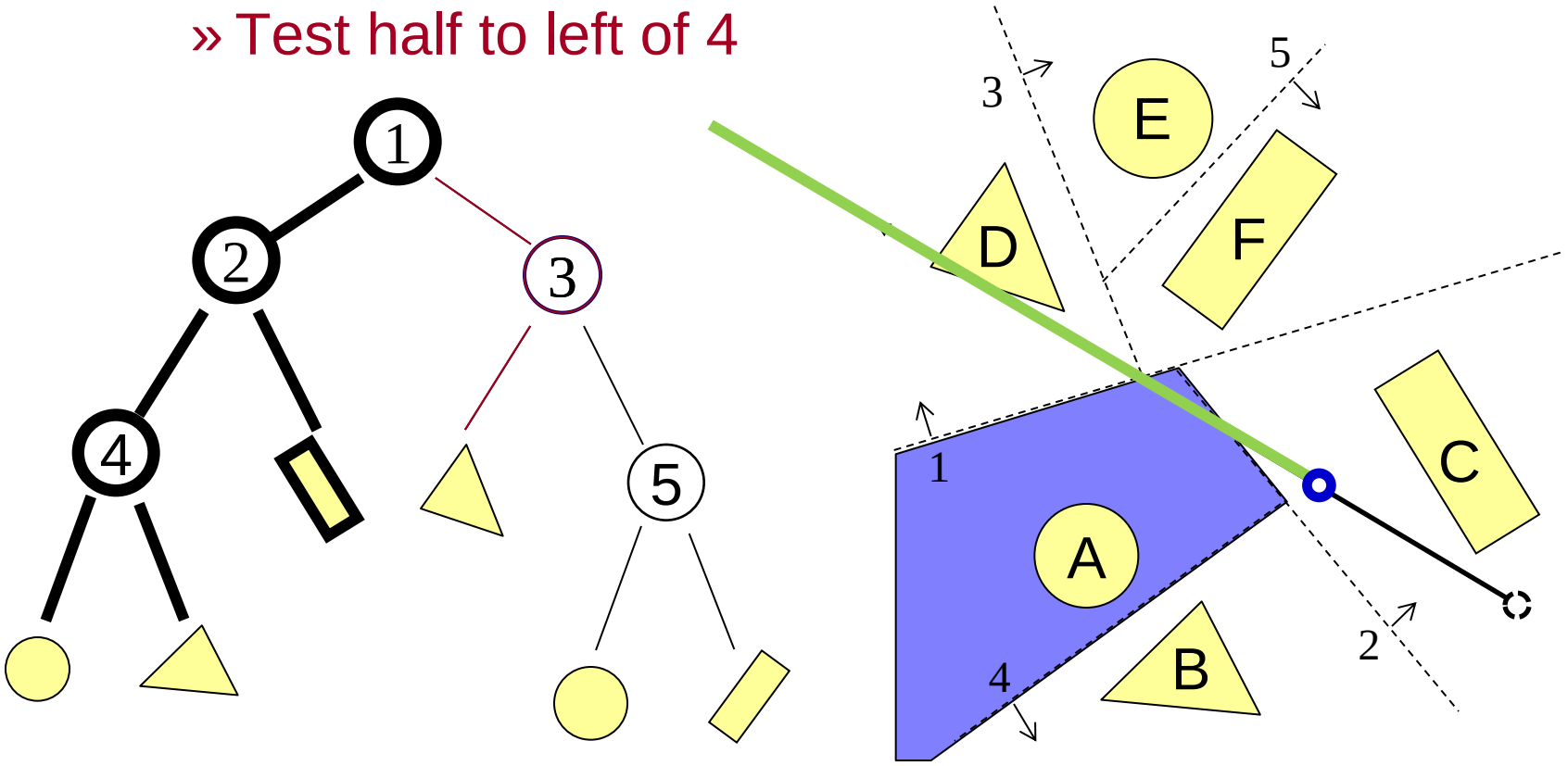




Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

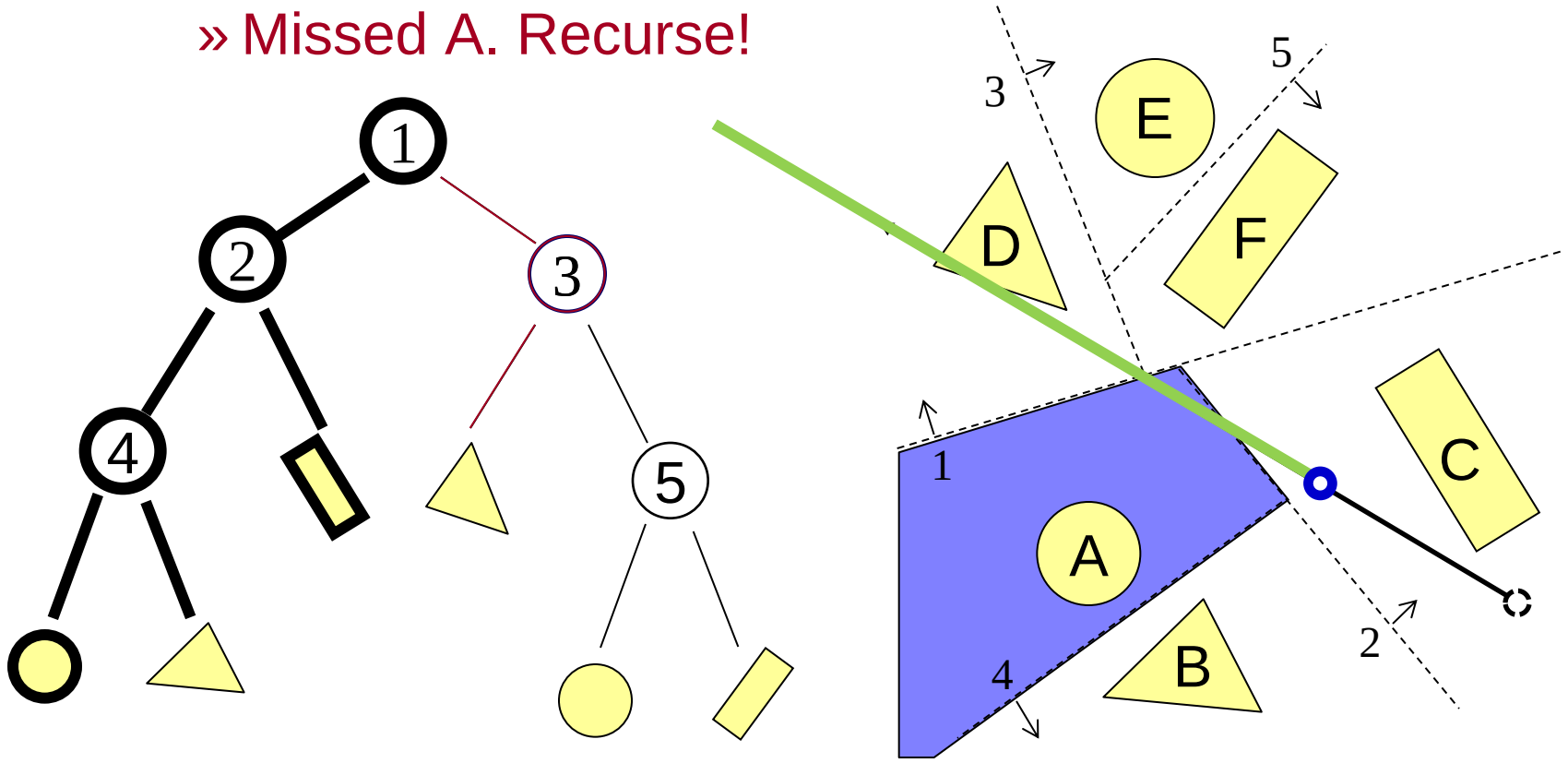
- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to left of 4



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

Space Partition: BSP Tree

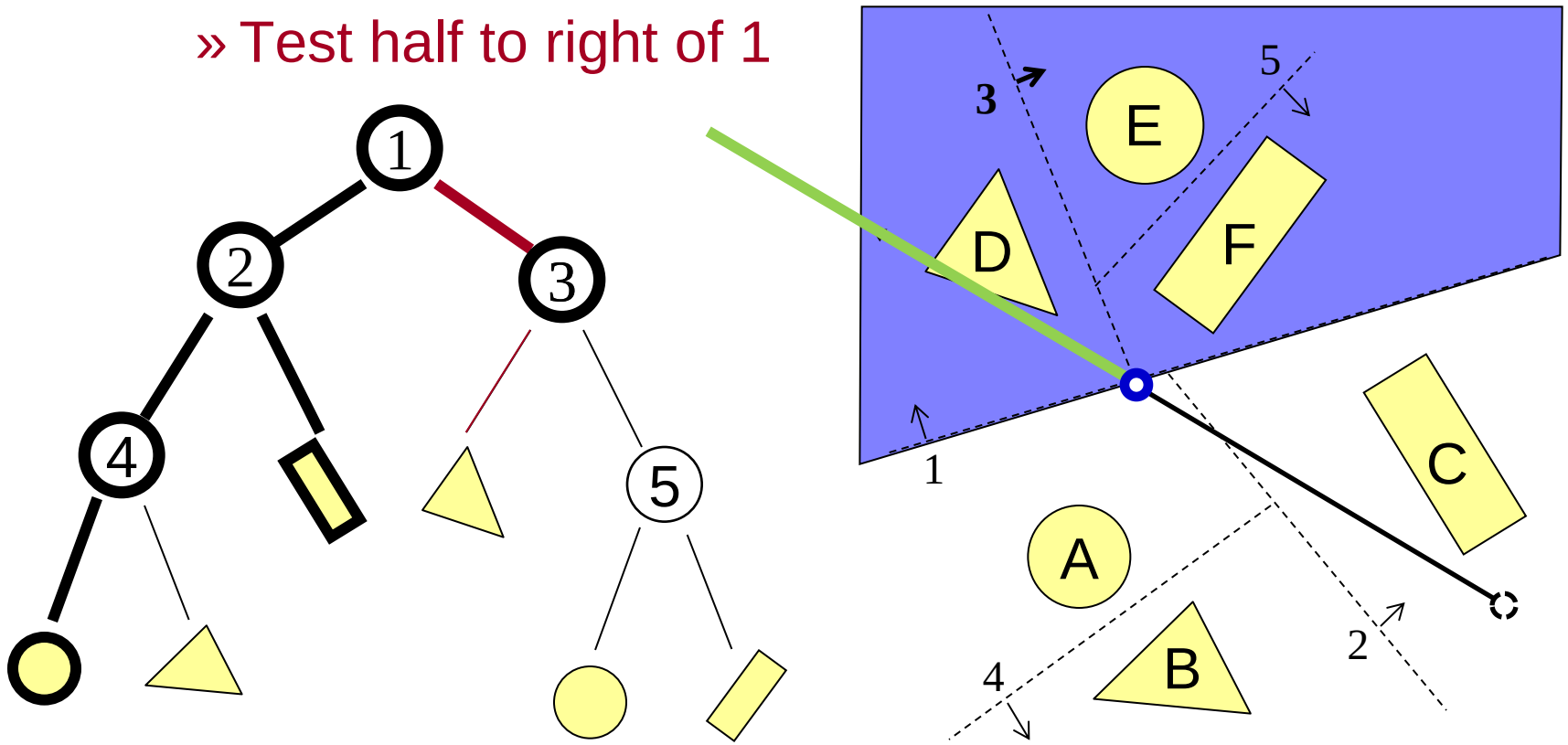
- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Missed A. Recurse!



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

Space Partition: BSP Tree

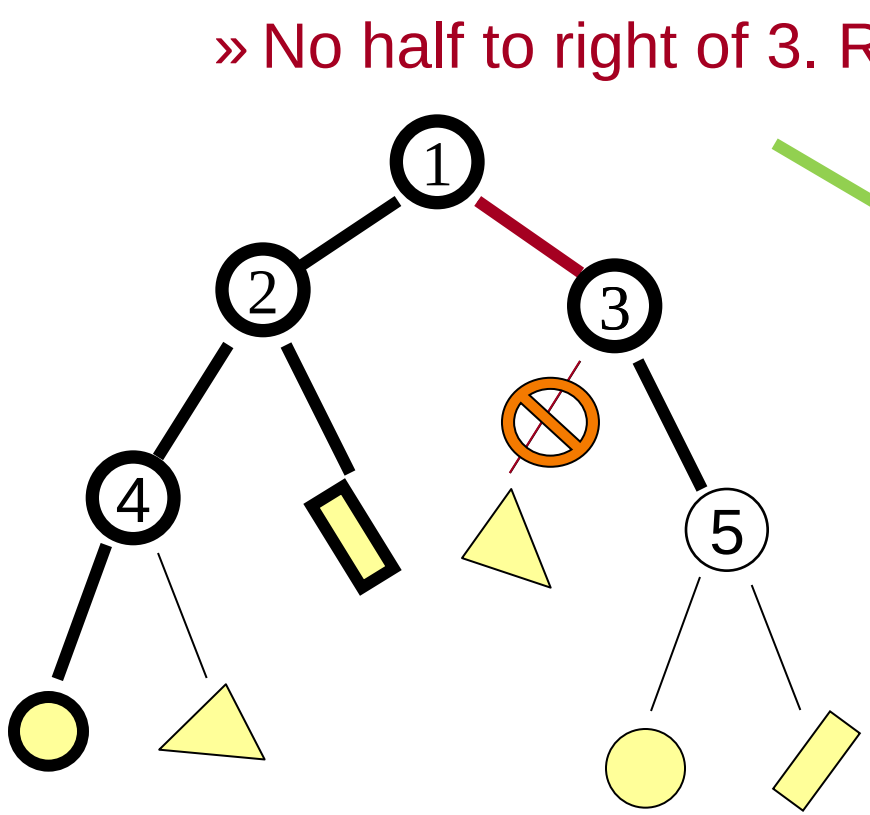
- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to right of 1

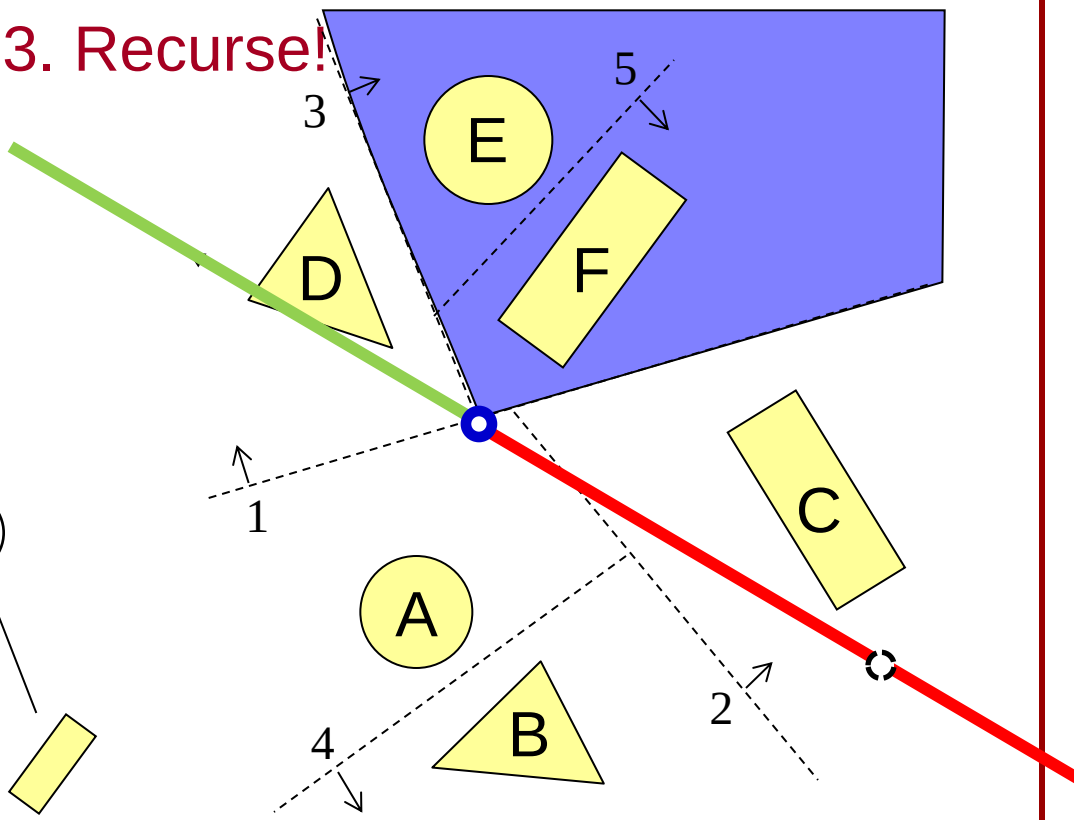


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » No half to right of 3. Recurse!

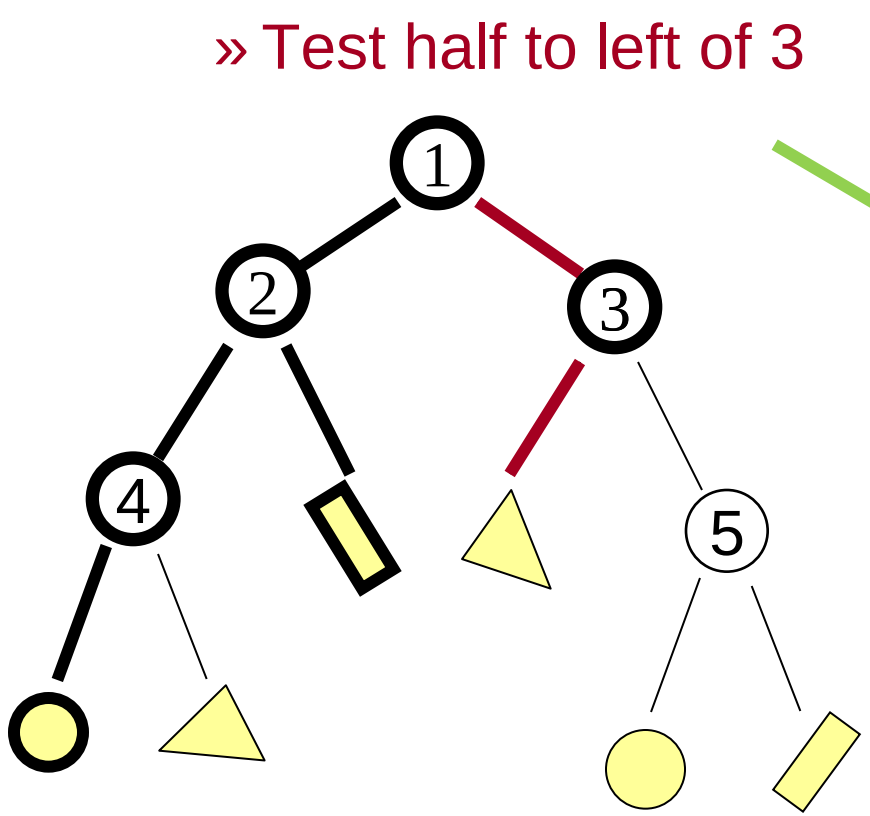


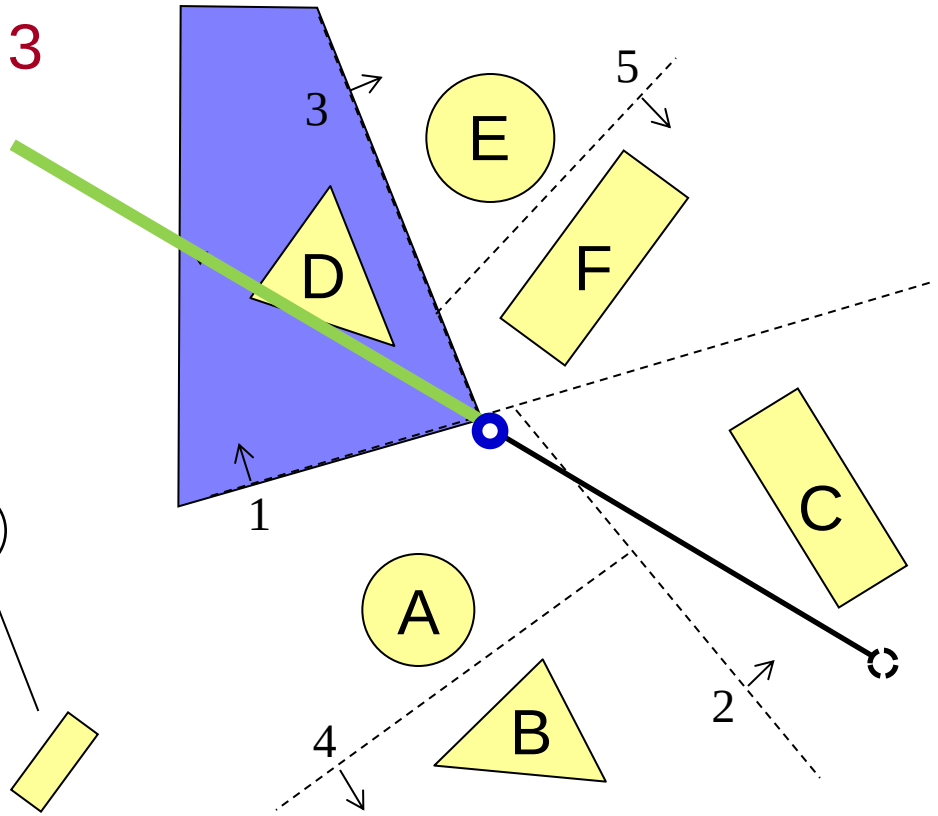


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Test half to left of 3

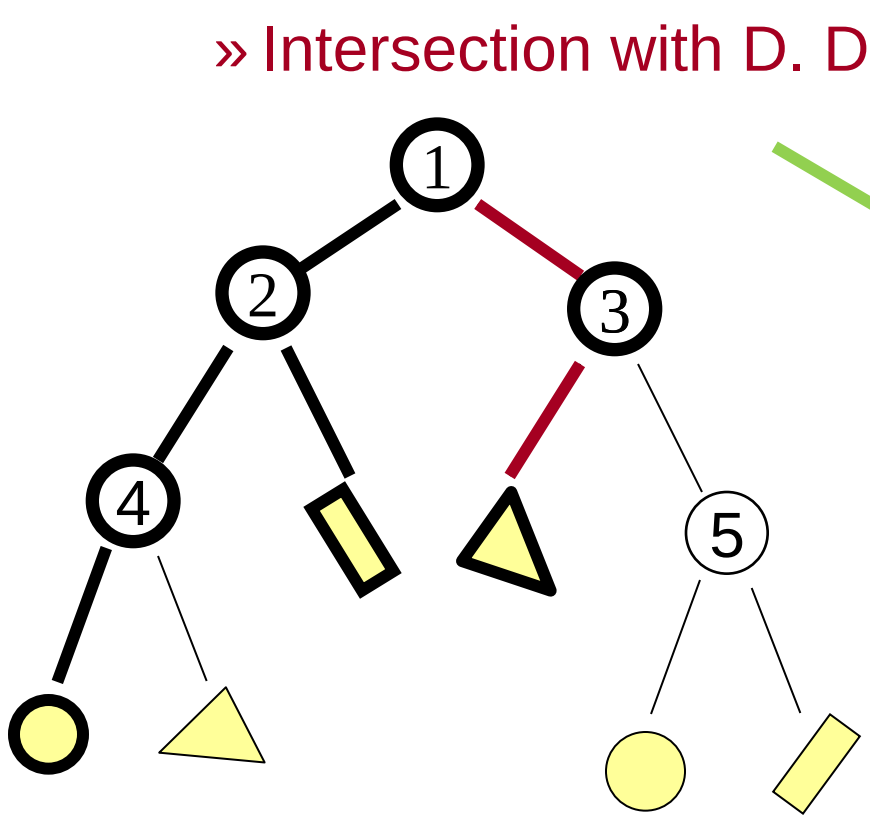


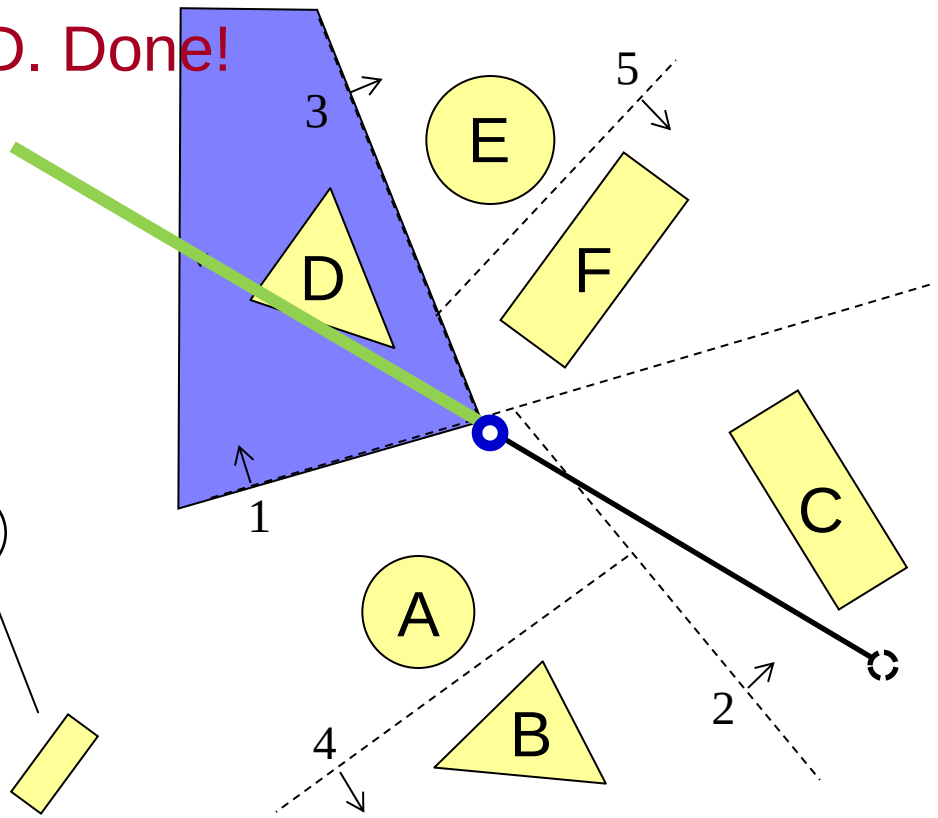


Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test both halves, testing the back part first. Stop once you hit something:
    - » Intersection with D. Done!

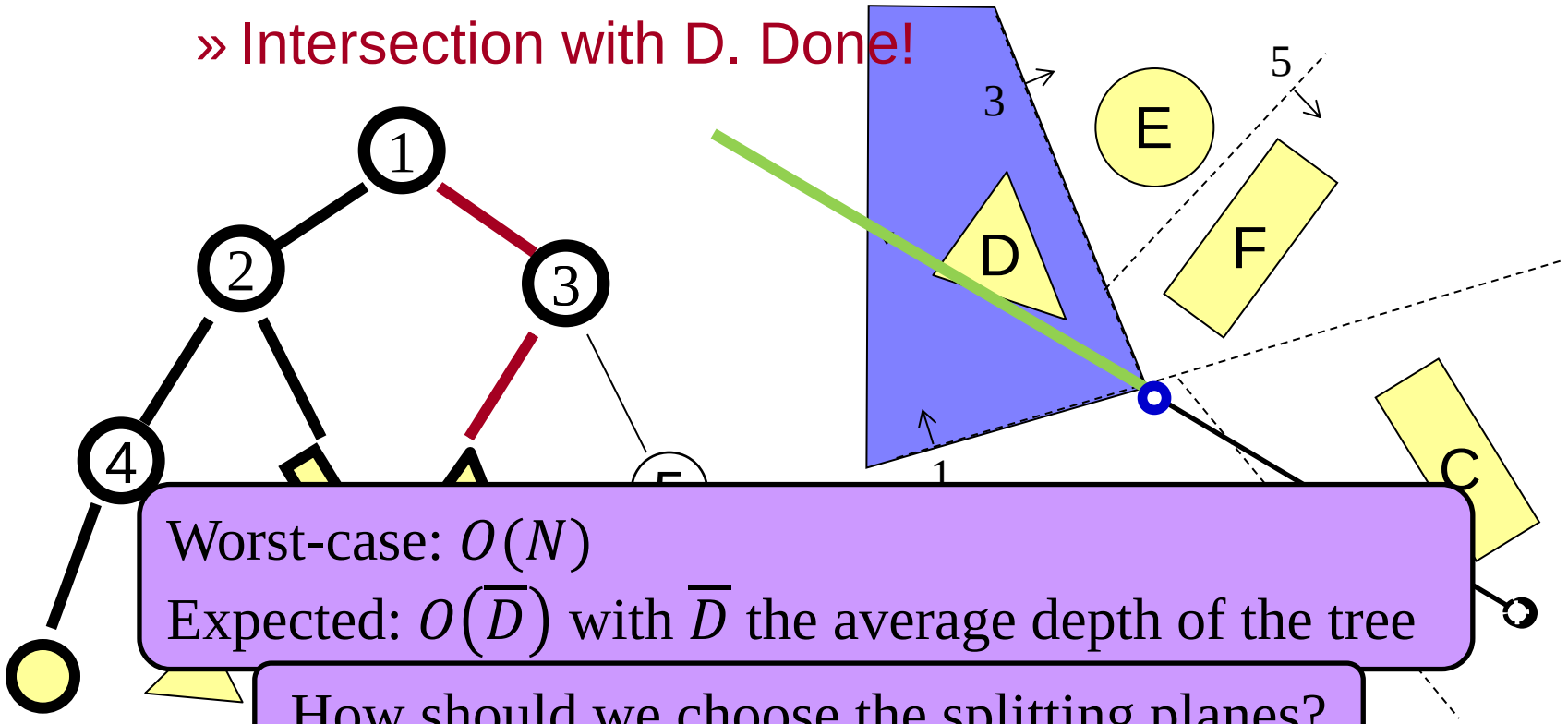




Note: Arrows denote the "right" side of the splitting plane.

Space Partition: BSP Tree

- Example: Ray Intersection 2
  - 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 if the geometry changes/animates