Spatial Data Structures

1. Hierarchical Bounding Volumes
2. Grids
3. Octrees
4. BSP Trees

Speeding Up Ray Tracing

- Trace fewer rays
  - most relevant in recursive ray tracing
- Speed up each ray-surface intersection test
  - optimize ray-triangle, ray-sphere intersection code
  - compile with optimizer
- Do fewer ray-surface intersection tests
  - subsequent hits on the same object often hit the same polygon.
  - shadow object caching
    - » When a shadow ray hits an object, remember that object and check it first against the next shadow ray heading toward that light.
    - » If it hits, you know that shadow applies; it doesn't matter if some other shadow source is closer to the object than the light source.
- For more info

  Chapter by Arvo & Kirk in the book *Introduction to Ray Tracing*

Spatial Data Structures

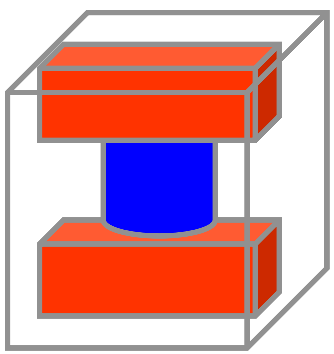
- Data structures for efficiently storing geometric information
- They are useful for
  - Collision detection (will the spaceships collide?)
  - Location queries (which is the nearest post office?)
  - Chemical simulations (which protein will this drug molecule interact with?)
  - Rendering (is this aircraft carrier on-screen?), and more
- Good data structures can give speed up ray tracing by 10x, 100x, or more

Spatial Data Structures

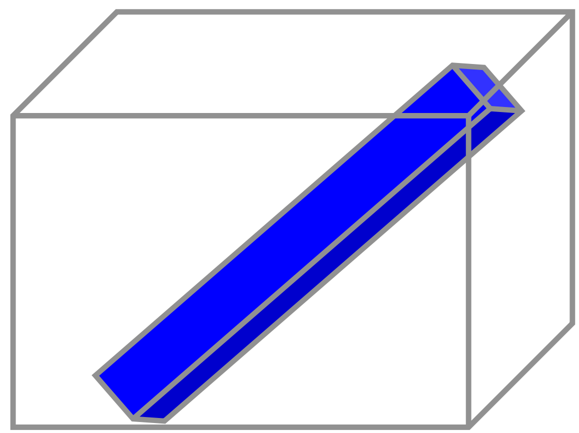
- We'll look at
  - Hierarchical bounding volumes
  - Grids
  - Octrees
  - BSP trees

Bounding Volumes

- Simple notion: wrap things that are hard to check for ray intersection in things that are easy to check.
  - Example: wrap a complicated polygonal mesh in a box
  - Ray can't hit the real object unless it hits the box
  - Adds some overhead, but generally pays for itself.
- Most common bounding volume types: sphere and box
  - box can be axis-aligned or not
- You want a snug fit!



Good!



Bad!

Hierarchical Bounding Volumes (HBV's)

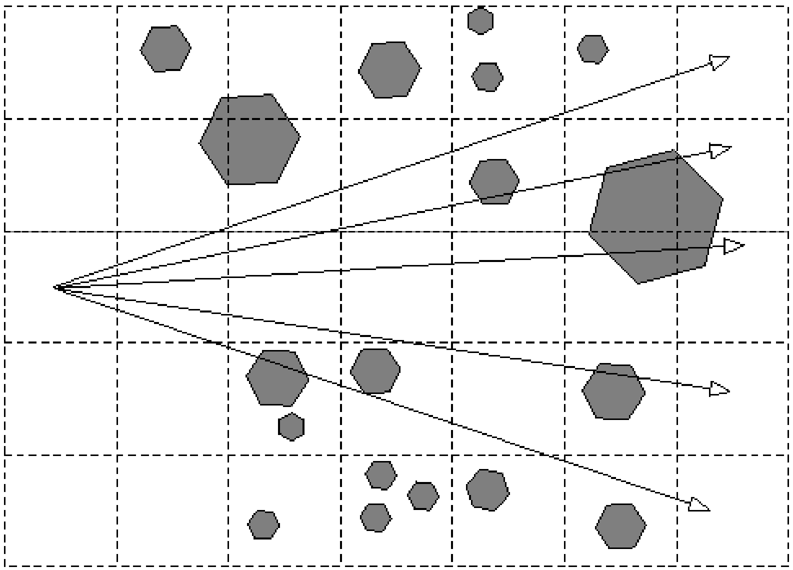
- Tree data structure:
  - List of bounding volumes (BV's), e.g. spheres, boxes
  - Each BV can contain a list of sub-volumes
  - E.g., Human figure:
    - » torso bounding-box (BB) contains arm BB, which contains finger BB, etc.
- Intersection testing: recursively descend tree

```
intersect(BV)
    if ray misses BV, return MISS
    closest = infinity
    for each subvolume stored in BV
        if ray intersects subvolume, and closer than closest
            update closest
    return closest
```

- Works well if you use good (appropriate) bounding volumes
- If your BVs are objects, you can have multiple classes and pick the best for each enclosed object!

Grids

- Data structure: a 3-D array of cells (voxels) that tile space
  - Each cell points to list of all surfaces intersecting that cell



- Intersection testing:
  - Start tracing at cell where ray begins
  - Step from cell to cell, searching for the first intersection point
  - At each cell, test for intersection with all surfaces pointed to by that cell
  - If there is an intersection, return the closest one

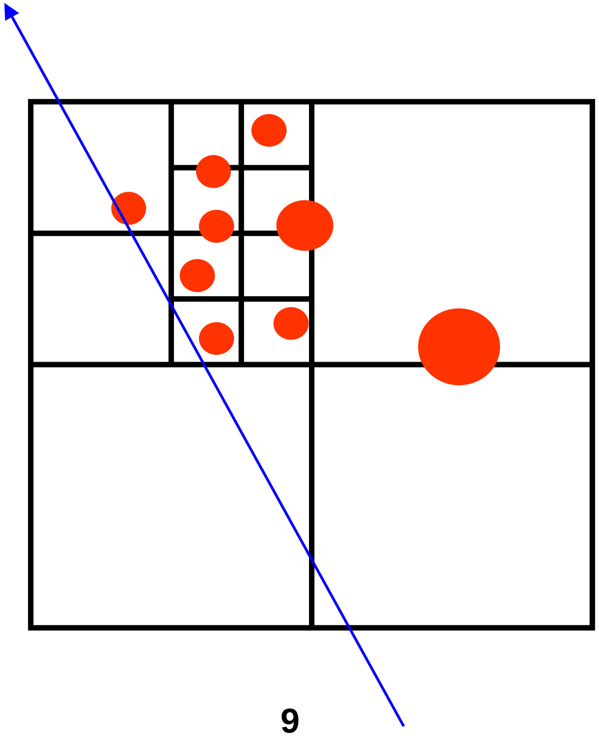
More on Grids

- Be Careful! The fact that a ray passes through a cell and hits an object doesn't mean the ray hit that object in *that* cell
- Optimization: cache intersection point and ray id in "mailbox" associated with each object

- Grids are a poor choice when the world is nonhomogeneous (clumpy)
  - e.g. a teapot in a stadium: many polygons clustered in a small space
- How many cells to use?
  - too few $\Rightarrow$ many objects per cell $\Rightarrow$ slow
  - too many $\Rightarrow$ many empty cells to step through $\Rightarrow$ slow
- Grids work well when you can arrange that each cell lists a few (ten, say) objects
- Better strategy for some scenes: *nested grids*

Octrees

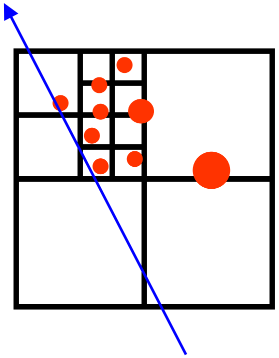
- Quadtree is the 2-D generalization of binary tree
  - node (cell) is a square
  - recursively split into four equal sub-squares
  - stop when leaves get "simple enough"



Octrees

- Octree is the 3-D generalization of quadtree
  - node (cell) is a cube, recursively split into eight equal sub-cubes
  - for ray tracing:
    - » stop splitting when the number of objects intersecting the cell gets "small enough" or the tree depth exceeds a limit
    - » internal nodes store pointers to children, leaves store list of surfaces
  - more expensive to traverse than a grid
  - but an octree adapts to nonhomogeneous, clumpy scenes better

```
trace(cell, ray) {                    // returns object hit or NONE
    if cell is leaf, return closest (objects_in_cell(cell))
    for child cells pierced by ray, in order        // 1 to 4 of these
        obj = trace(child, ray)
        if obj!=NONE return obj
    return NONE
}
```



Which Data Structure is Best for Ray Tracing?

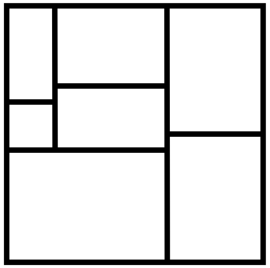
- Grids are easy to implement, but they're memory hogs (and slow) for nonhomogeneous scenes, i.e. most scenes
- Octrees are pretty good, but not as fast as grids for some scenes
- Nested grids seem to be the fastest on static scenes

- If scene is dynamic, the cost of regenerating or updating the data structure may become an issue
- In such cases, hierarchical bounding volumes may be best
- Hierarchical bounding volumes easy to implement if your model is naturally hierarchical (e.g. human), otherwise not

- For other visibility algorithms:
  - BSP trees useful for Painter's algorithm...

k-d Trees

- Relax the rules for quadtrees and octrees:

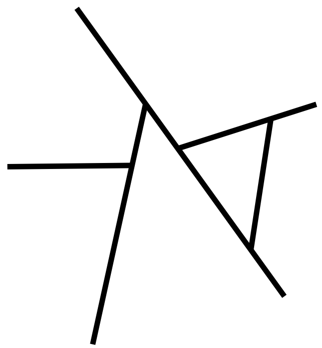


- first variant: *k-dimensional (k-d) tree*
  - don't always split at midpoint
  - split only one dimension at a time (i.e. x or y or z)
  - useful for clustering and choosing colormaps for color image quantization

BSP Trees

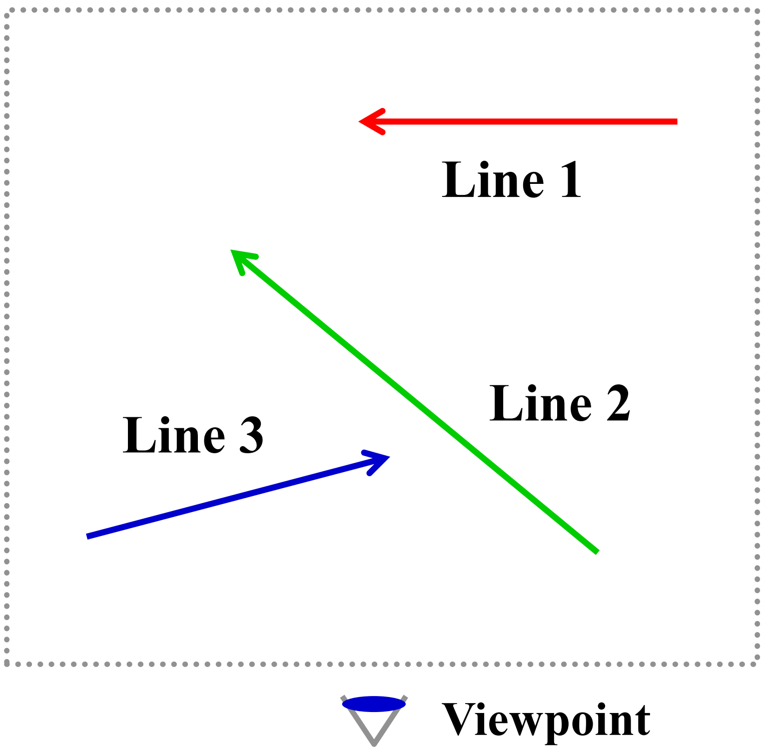
- Relax the rules for quadtrees and octrees:

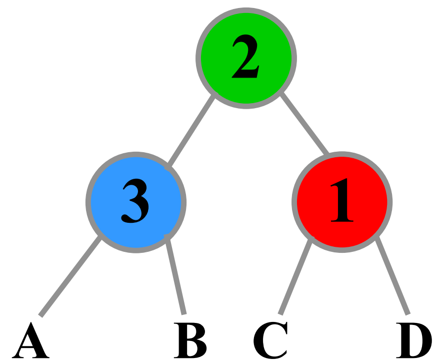
- second variant: *binary space partitioning (BSP) tree*
  - – permit splits with any line
  - – in general, split *k* dimensional space with *k*-1 dimensional hyperplane
    - » 2-D space split with lines (most of our examples)
    - » 3-D space split with planes
    - » each node corresponds to a (potentially unbounded) convex polyhedron
  - – For lots of info, see http://reality.sgi.com/bspfaq/
  - – useful for Painter's algorithm



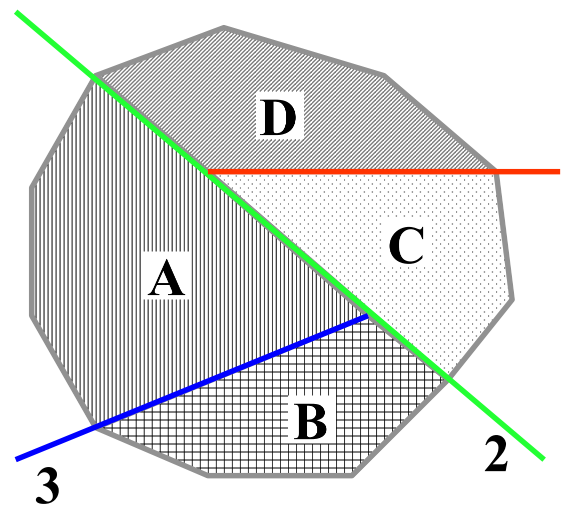
Building a BSP Tree

- **Let's look at simple example with 3 line segments**
- **Arrowheads are to show left and right sides of lines.**
- **Using line 1 or 2 as root is easy.**
- **(examples from http://www.geocities.com/SiliconValley/2151/bsp.html)**





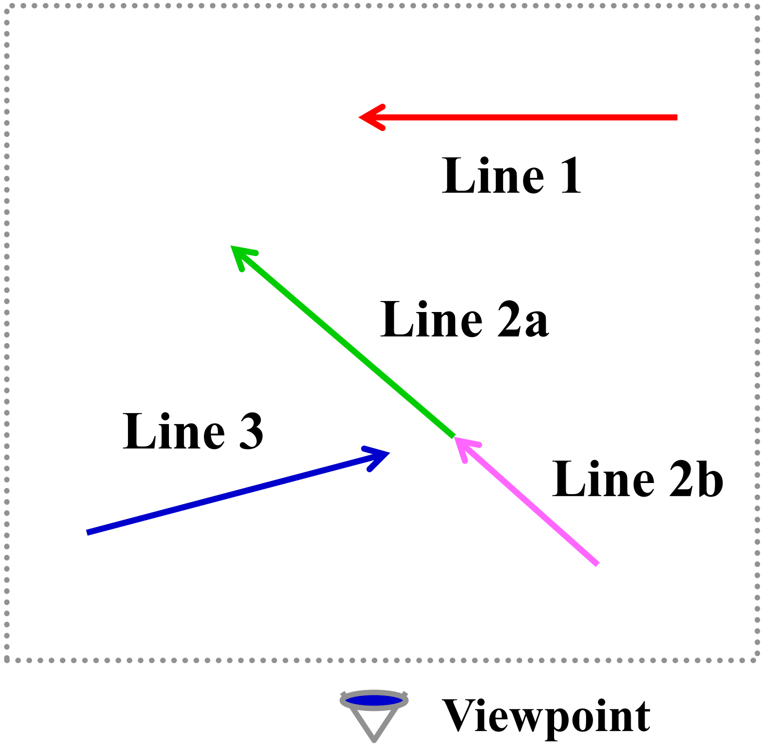
a BSP tree using 2 as root

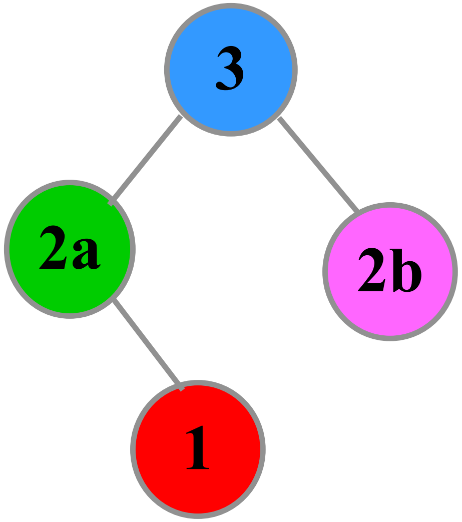


the subdivision of space it implies

Building the Tree 2

- Using line 3 for the root requires a split





Building a Good Tree - the tricky part

- **A naïve partitioning of *n* polygons will yield *O(n³)* polygons!**

- **Algorithms exist to find partitionings that produce *O(n²)*.**
  - **For example, try all remaining polygons and add the one which causes the fewest splits (I think this works ;)**
  - **Fewer splits -> larger polygons -> better polygon fill efficiency**

- **Also, we want a balanced tree.**
  - **More important for ray casting than scan conversion.**

- **These goals conflict.**

- *note: in the examples we've shown, the geometric objects being stored are planar, and we split using the planes of these objects, but that needn't be so – could theoretically split with any plane*

Uses for Binary Space Partitioning (BSP) Trees

- Painter's algorithm rendering
  - good for
    - » static 3-D scenes with moving viewpoint (flight simulators)
    - » architectural scenes with a small number of polygons (DOOM)
    - » if you don't have z-buffer hardware
  - Add a few monsters and such after the environment is drawn
- Ray tracing
- Solid modeling with polyhedra

- History:
  - BSP trees first used by Naylor, Fuchs, et al. for Painter's algorithm ~1980
  - theoreticians scoffed at their worst-case performance
  - considered unpromising
  - revived by John Carmack, author of Quake, and the PC game community
    - » out of necessity: no z-buffer hardware for PC's at the time

Painter's Algorithm with BSP trees

- Build the tree
  - Involves splitting some polygons
  - Slow, but done only once for static scene

- Correct traversal lets you draw in back-to-front or front-to-back order for any viewpoint
  - Order is view-dependent
  - Precompute tree once
  - Do the "sort" on the fly

Drawing a BSP Tree

- **Each polygon has a set of coefficients:**

  Ax + By + Cz + D
- **Plug the coordinates of the viewpoint in and see:**

  >0 : front side

  <0 : back facing

  =0 : on plane of polygon
- **Back-to-front draw: inorder traversal, do farther child first**
- **Front-to-back draw: inorder traversal, do near child first**

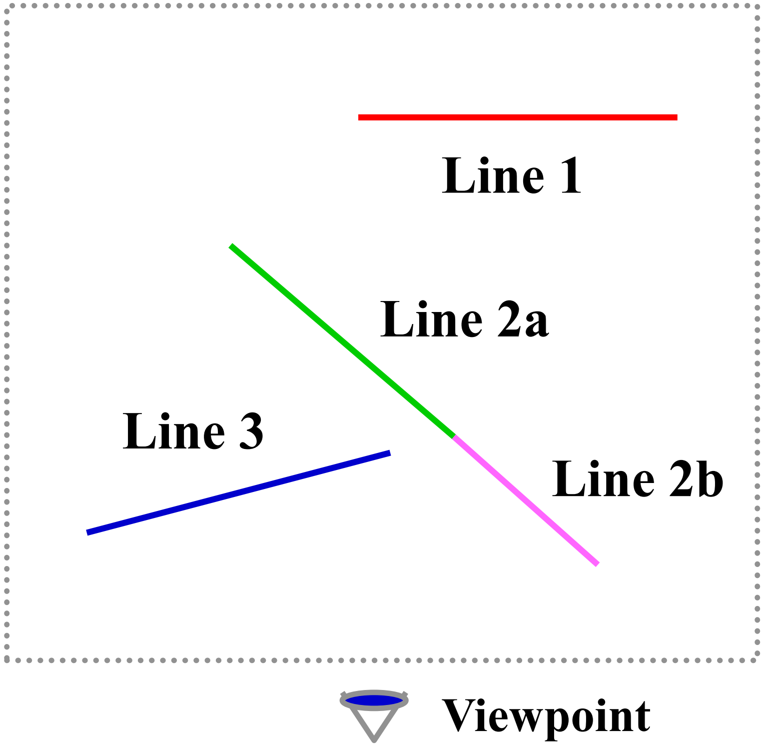
```
front_to_back(tree, viewpt) {
  if (tree == null)  return;
  if (positive_side_of(root(tree), viewpt)) {
    front_to_back(positive_branch(tree, viewpt);
    display_polygon(root(tree));
    front_to_back(negative_branch(tree, viewpt);
  }
  else { …draw negative branch first…}
}
```

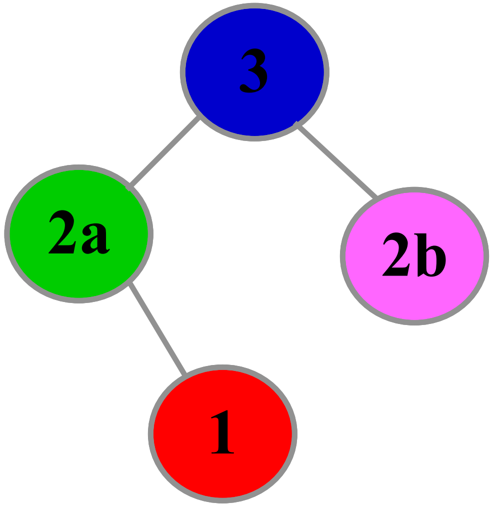
Drawing Back to Front

- **Use Painter's Algorithm for hidden surface removal**

Steps:

- ***Draw objects on far side of line 3***
  - ***Draw objects on far side of line 2a***
    - **Draw line 1**
  - **Draw line 2a**
- **Draw line 3**
- ***Draw objects on near side of line 3***
  - **Draw line 2b**





Further Speedups

- Do backface culling with same sign test

- Draw front to back, and…
  - Keep track of partially filled spans
  - Only render parts that fall into spans that are still open
  - Quit when the image is filled

- Clip the BSP tree against the portions of space that you can see!
  - Called *portals*
  - Initial view volume is entire viewing frustum
  - When you look through a doorway, intersect current volume with "beam" defined by doorway
  - Skip a BSP node if it doesn't intersect the current view volume
  - Much faster than clipping every polygon

Clipping BSP Trees

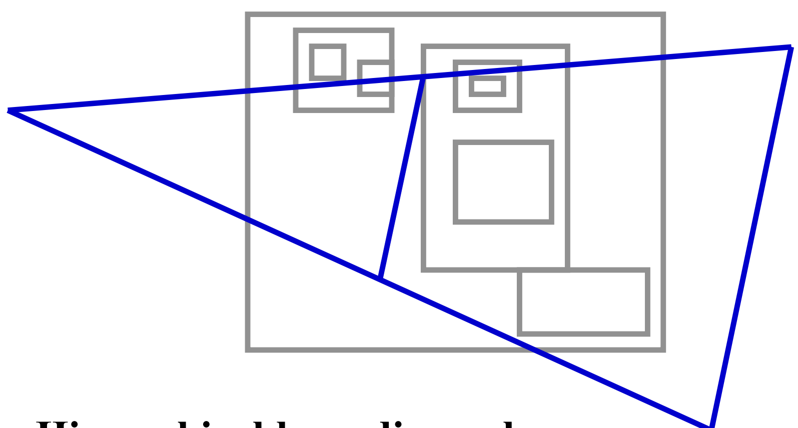
- **Suppose you have all n polygons in a BSP tree, and it's time to clip them for rendering.**

- **Clip the tree to the view frustum!**
  - **This is an intersection operation between the tree of polygons and a BSP tree representing the frustum**
  - **An O(log n) operation, while clipping all n polygons is O(n)**

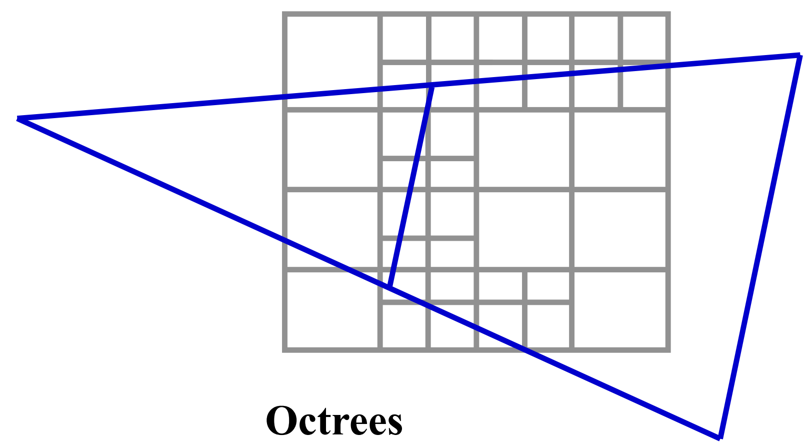
- **Algorithm is a bit involved, but straightforward**
  - **merge the polygon tree into the frustum tree**
  - **large parts of the polygon tree lie on known sides of the splits in the frustum tree, and thus need never be traversed**

Clipping Using Spatial Data Structures

- The data structures we used to accelerate ray tracing will work here too!
- In each case, the goal is to accept or reject whole sets of polygons.
- The O(n) task becomes O(log n)
- Scene must be (mostly) fixed, to amortize cost of building the data structure
  - terrain fly-through
  - gaming
- Off-screen stuff can swap out!



Hierarchical bounding volumes



Octrees