Binary Space Partitioned Trees

John E. Laird

Motivation

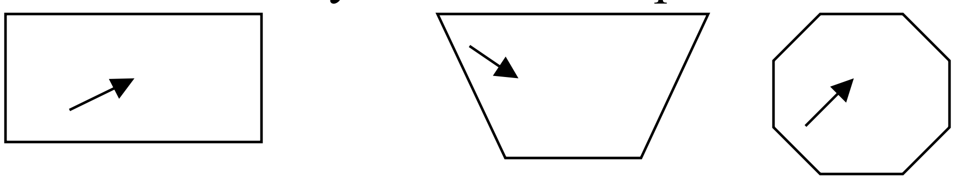
- Want to find fast, correct method for ordering polygons in the Painters algorithm
  - Avoid the five checks of painters algorithm
  - Preprocessing to determine the split planes
- Create a binary tree that partitions space.
  - Can use it to find ordering for drawing polygons.
  - Will be << n^2 for rendering
- Technique used in Doom, Quake, Descent, ...

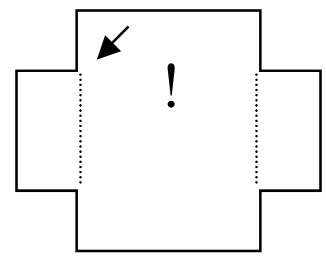
Assumptions

- Examples will be 2D but this generalizes to 3D
- Works best for static information
  - Good for map structures and even monster structure
  - Gets tricky if topography can change a lot
- Can require significant space at runtime
  - Must be managed efficiently to avoid cache problems

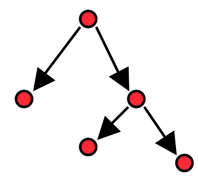
General Idea

- Recursively divide space into pairs of regions
  - Stop when regions are "atomic"
    - Doesn't matter which order walls are drawn no matter where you are in the space: convex





  - Builds up a binary tree



- When rendering, traverse tree depth-first, always first rendering region that you are not in
  - This does the right thing!

BSP Tree Dividing Issues

- Want to maintain a balanced tree if possible
- Want to minimize splits of existing walls
  - If divider crosses wall, wall must be split into two walls
- Keep dividers orthogonal to principle axes
  - Simplifies math with splits being more likely to be integer values.

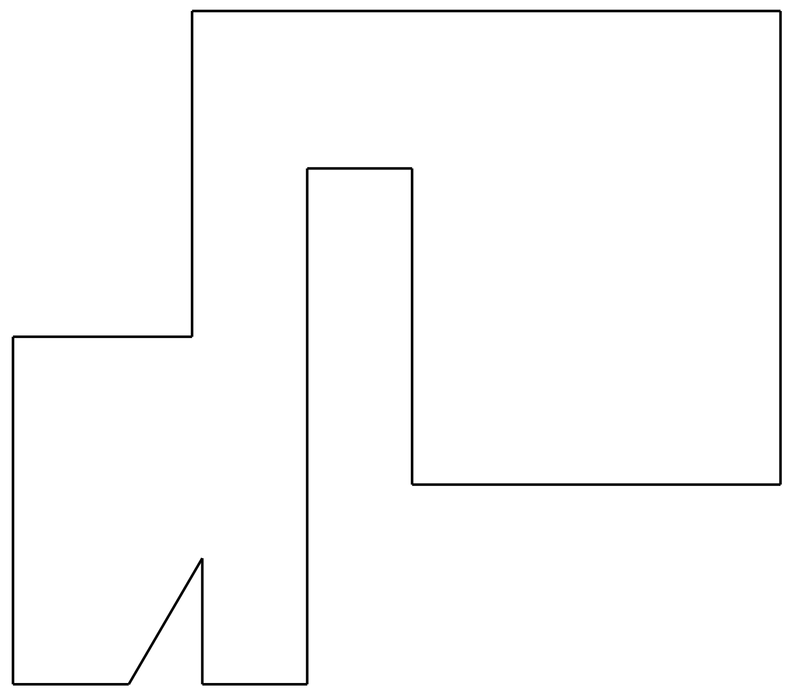
Picking a Divider: Key Question

- Pick on coincident with a wall
  - Less likely to split walls
- Pick 1% of existing walls, but at least 10
  - Evaluate based on simple calculation and pick best

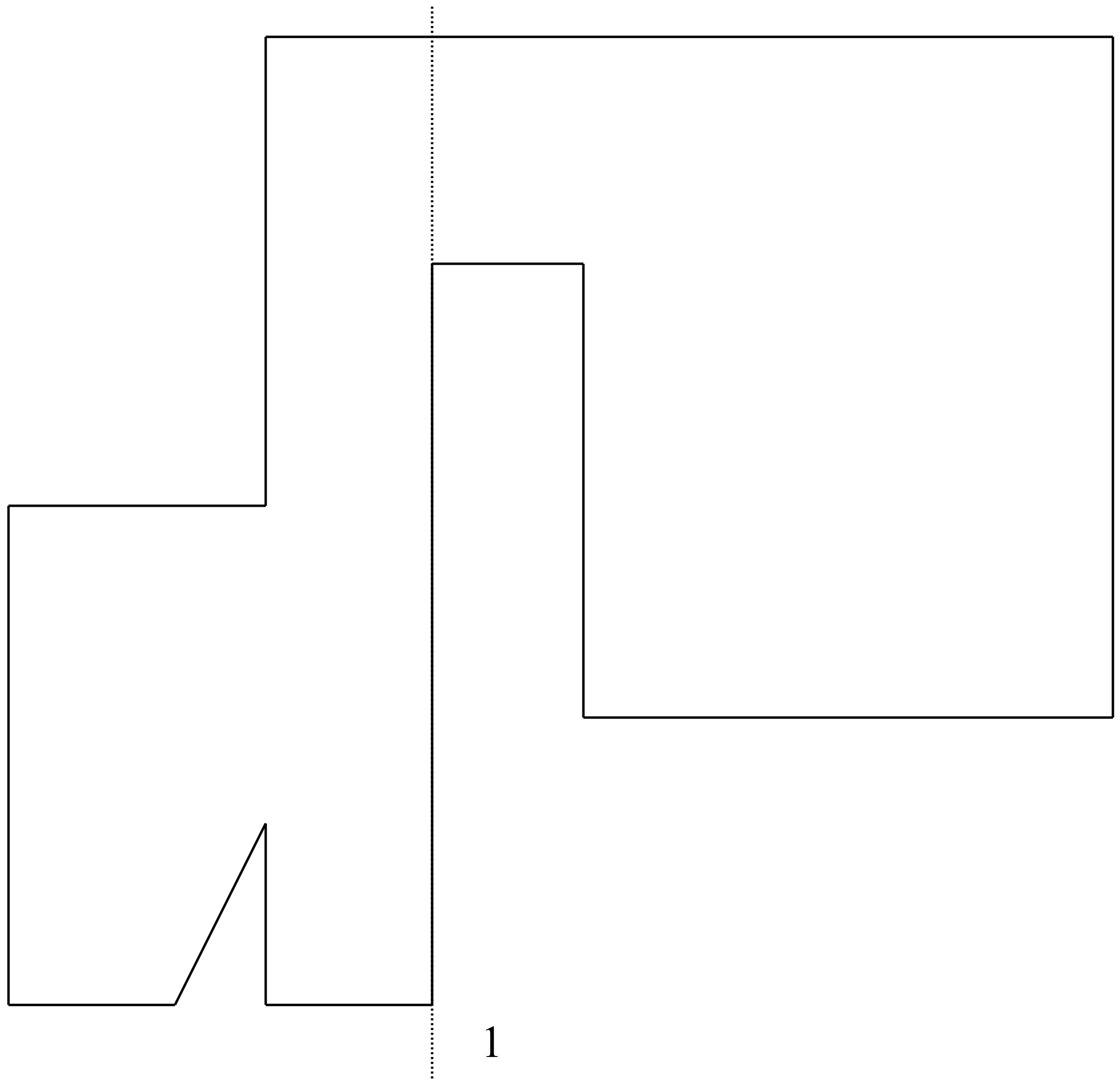
unbalanced walls +

15 * # splits +

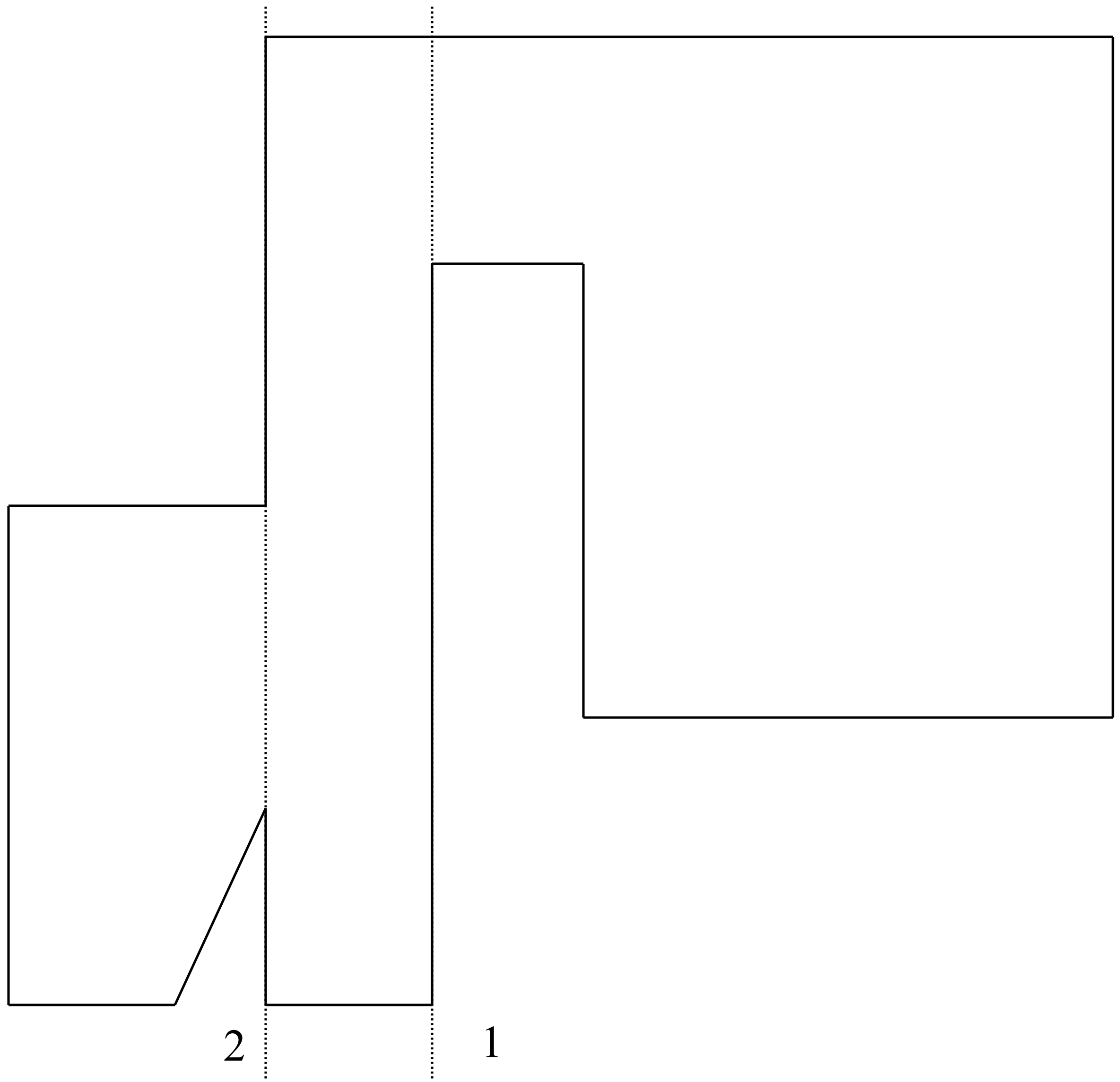
5 if not on principle axis



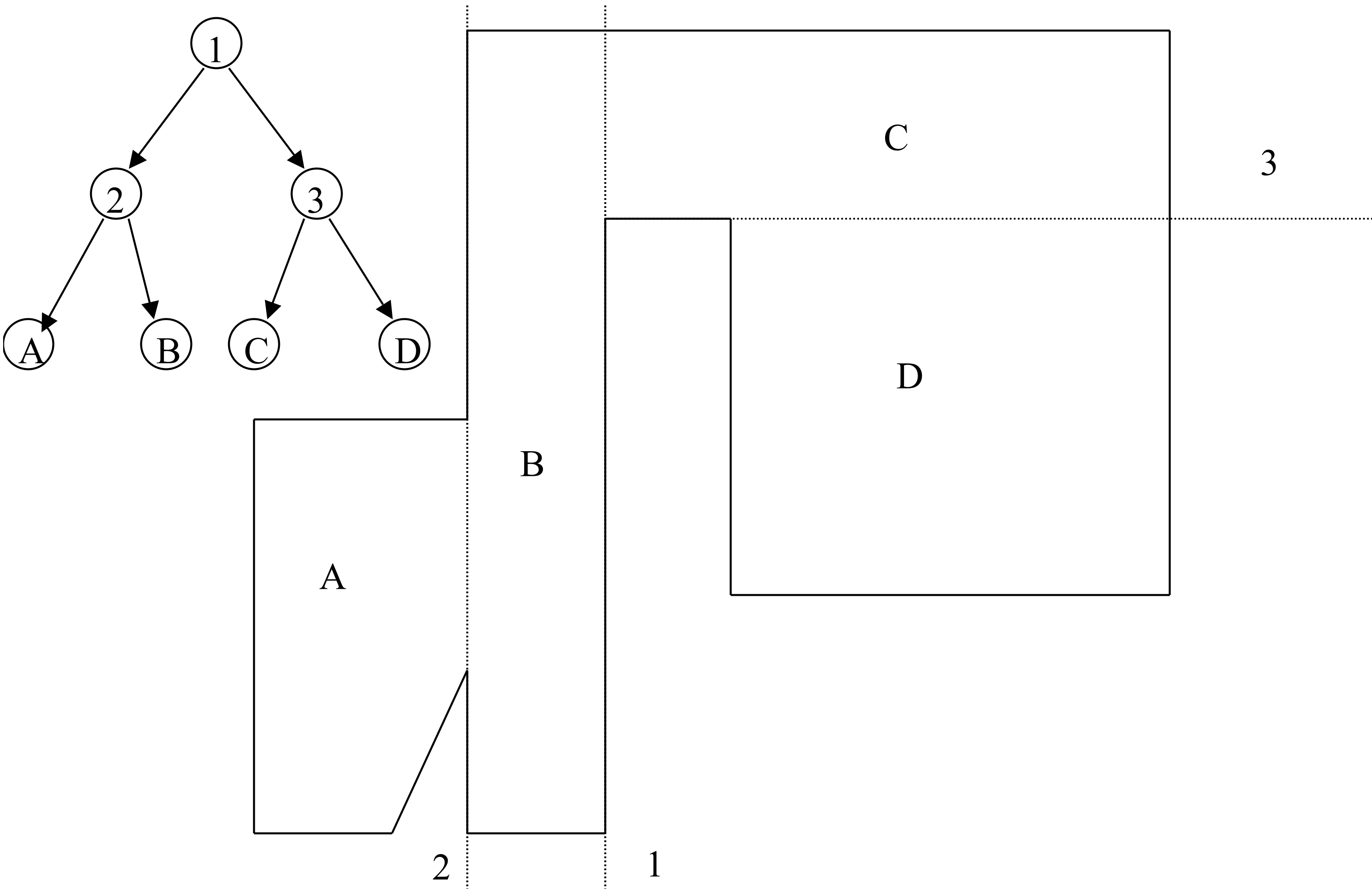
Example: Step 1



Example: Step 2



Example: Step 3



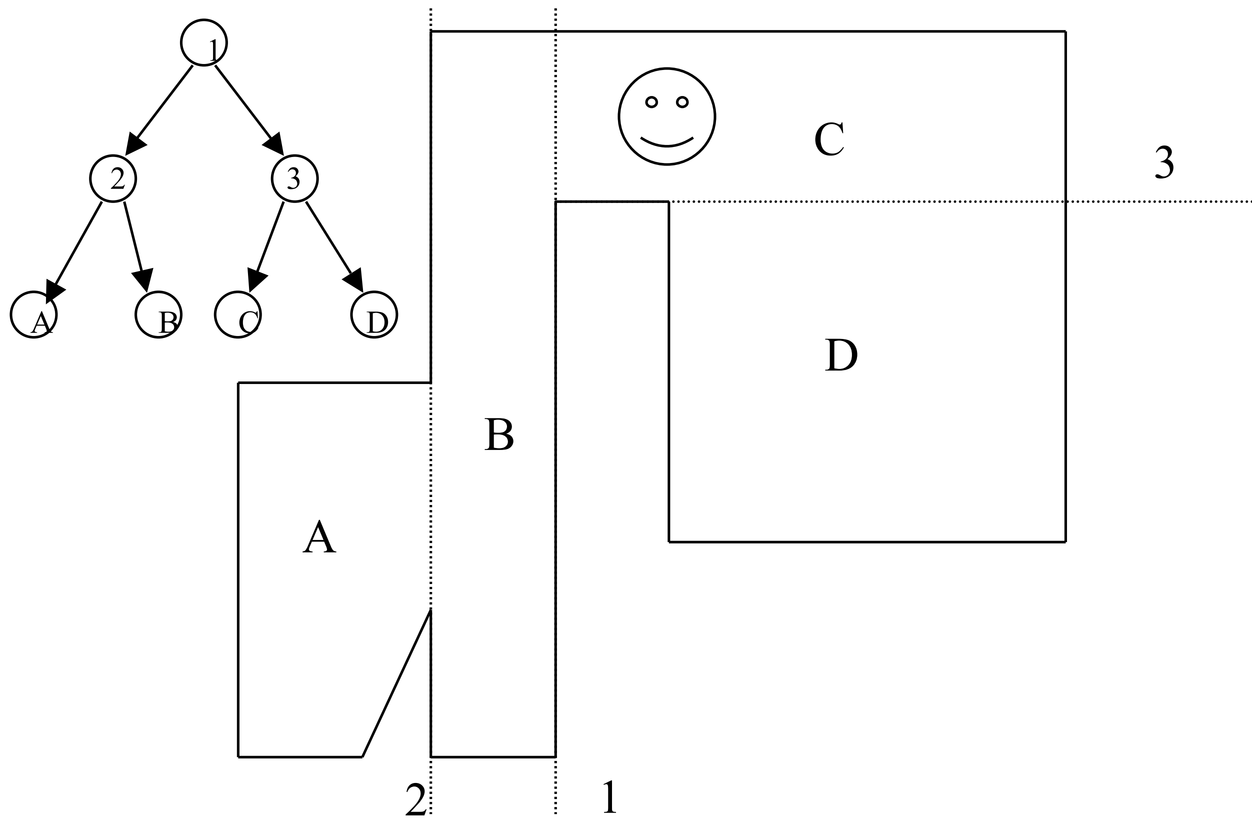
Rendering

- To start with, all we care about ordering of rendering
- Not going to worry about line of sight or orientation of viewer

- Depth-first traversal, always visiting nodes on opposite side of divisor from current node.
  - Render space when atomic

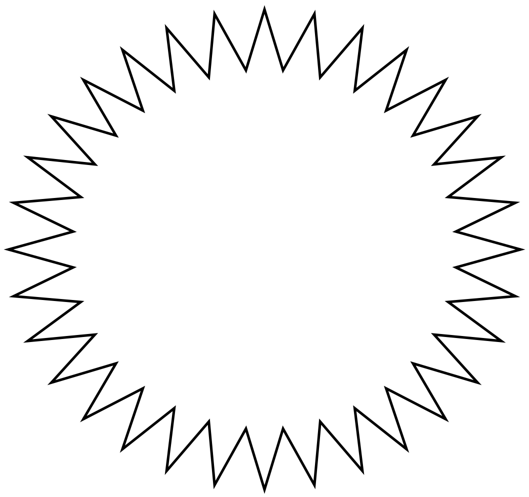
Rendering

- Go to node 2 (because C is right of divider 1)
- Go to A (because C is right of 2)
- Render A
- Render B
- Go to 3
- Go to D
- Render D
- Render C



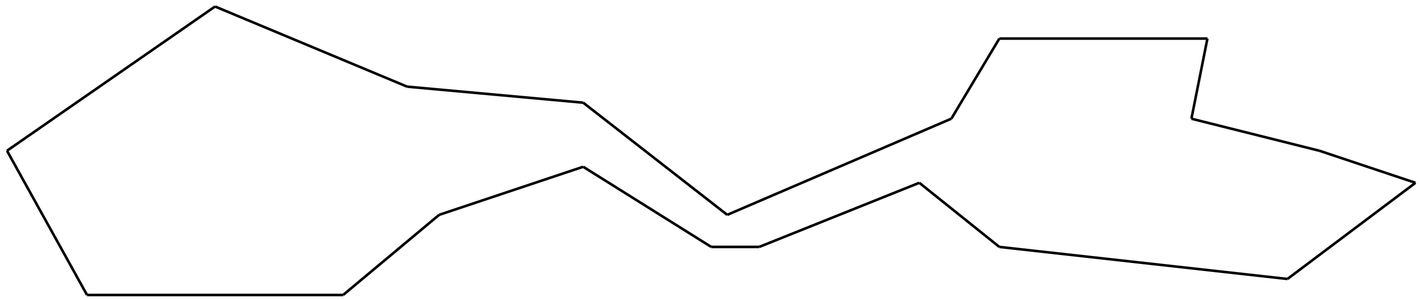
Observations

- Will work very well with walls that are on x, y axes.
  - Might be worthwhile to have as basis for room dividers
  - Other angles can be used to fill in outside of rooms.
- Depth will be related to log of # of concave areas



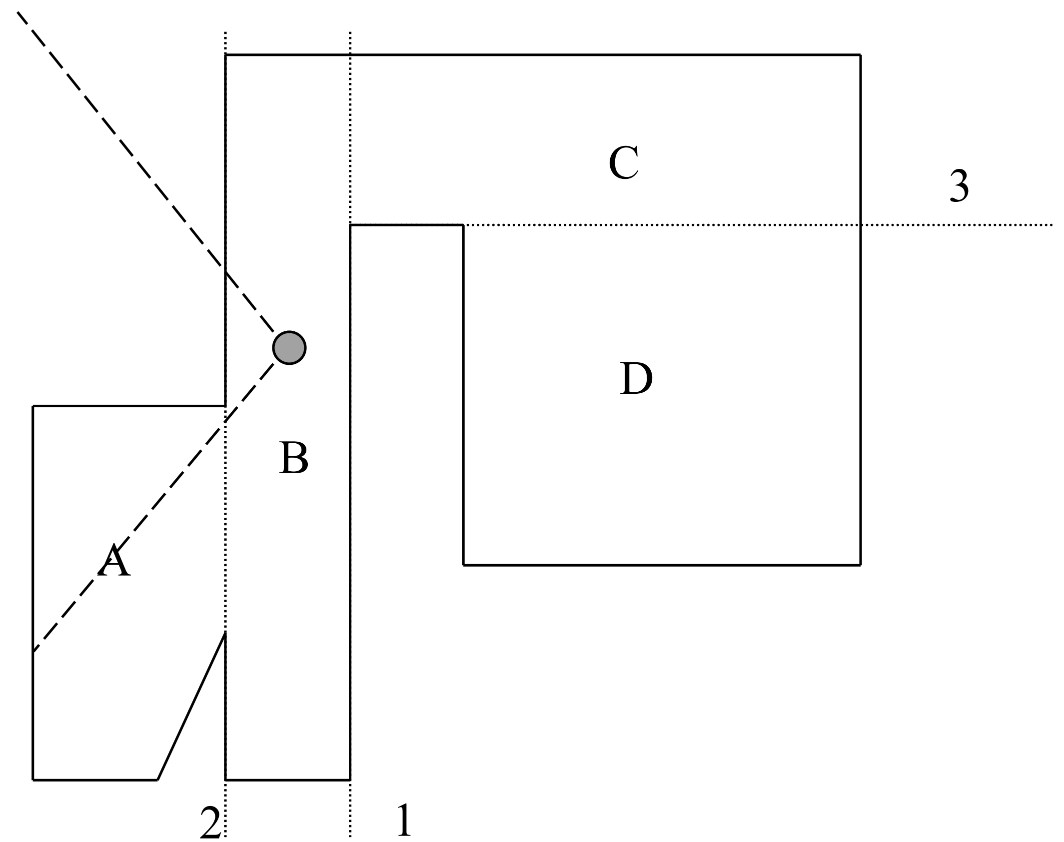
Inverted Painters: Front-to-Back

- Problem with Back-to-Front is lots of "over-draw"
  - Set same pixel over and over
  - Expensive because of lighting and texture calculations
- Front-to-back can avoid this
  - First draw front rooms first
    - Keep track of which pixels are filled in
  - Only draw pixels in back rooms that haven't been filled in
  - Stop completely when all pixels are filled in
    - Dynamically cuts off processing of rooms far away.



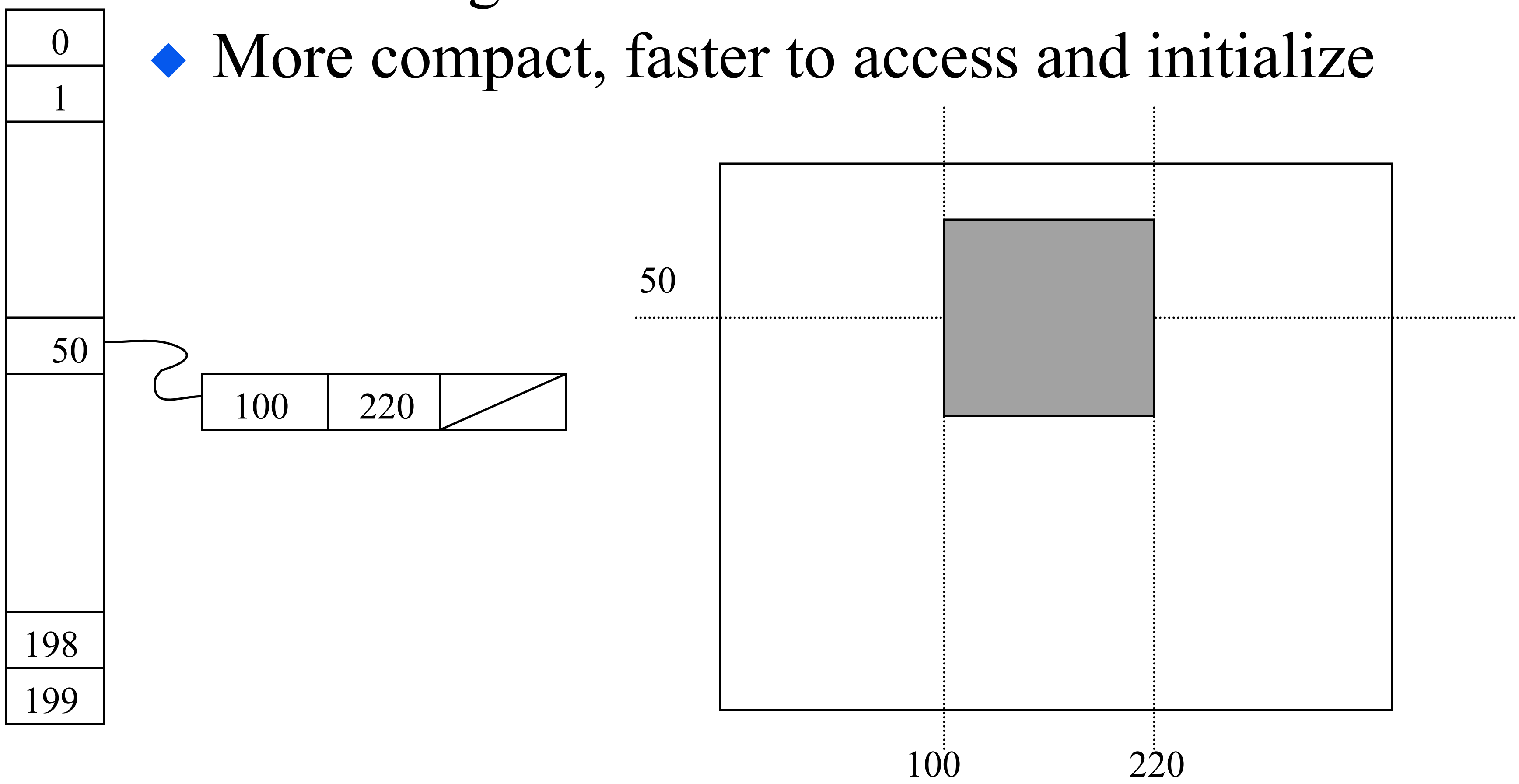
Front-to-Back: Field of View

- Don't traverse a node if field of view completely on other side of divider.



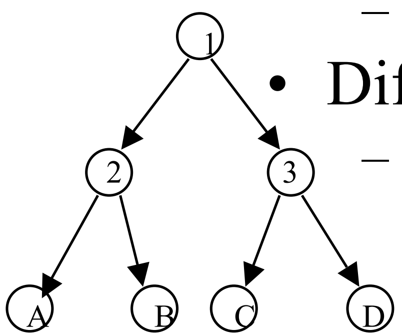
Front-to-back Data Structure

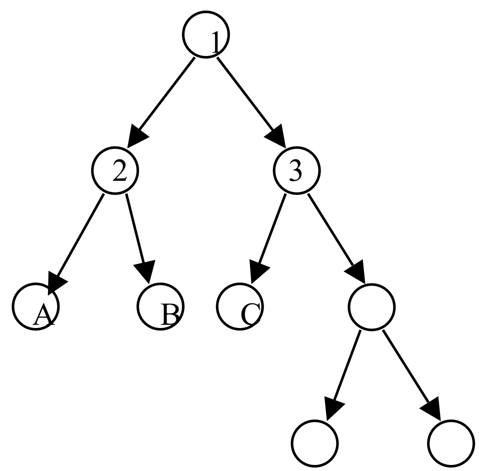
- To hold data on filled in pixels: use linked list
- Holds ranged of filled in horizontal lines
- More compact, faster to access and initialize



Dynamic Modification of BSP

- Extremely expensive to dynamically recalculate BSP if topology of game can arbitrarily change
- Can have pre-stored variants and swap in as world changes
  - Blow holes in walls - open doors
    - Add subtree
  - Different atomic regions
    - Swap in





3D Objects in BSP Trees

- Same idea, but render “outside” of object, not “inside”.
- Can just drop in to existing BSP tree at the bottom as a child of the atomic region it is in
- As 3D object moves, it changes where it is in BSP tree

Conclusion

- Even with Z-buffers, BSP Trees are an important tool for rendering static structures
- With front-to-back rendering, can eliminate overdraw and greatly reduce polygons considered.