Visible Surface Determination

Dr Nicolas Holzschuch
University of Cape Town

Map of the lecture: basic algorithms

- Image space *vs*. object space
- Height fields
  - floating horizon
  - contouring
- Backface removal
- Depth-sort algorithms
- BSP-trees

Map of the lecture: advanced algorithms

- Area-subdivision
- Scan-line algorithm
- Z-buffer
- Relative costs and best choice

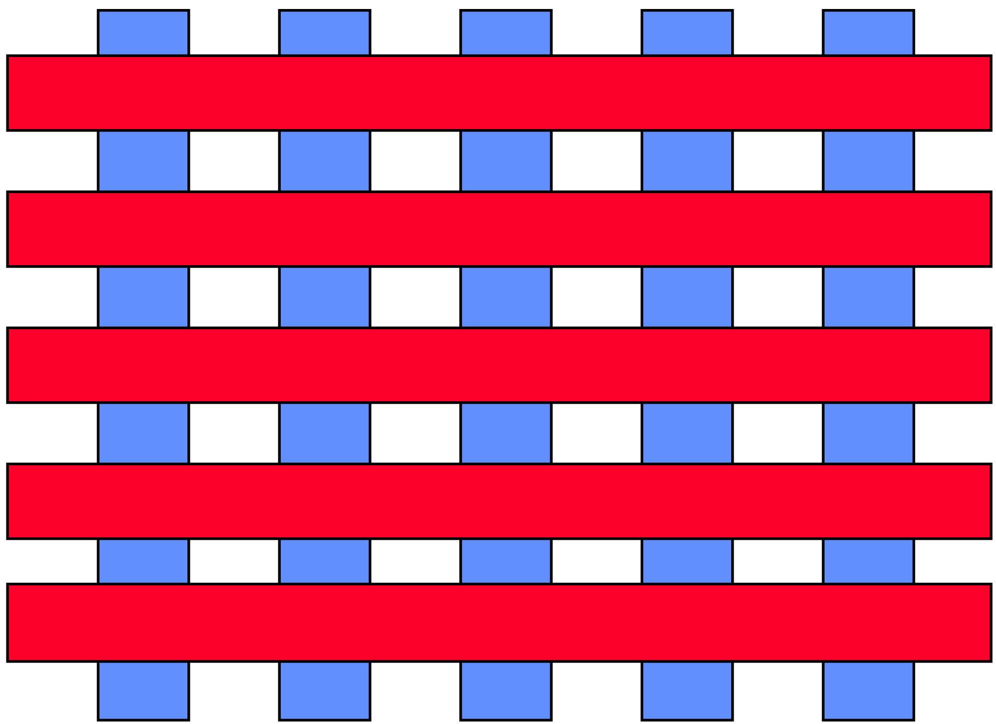
Visible Surface Determination

- Or is it hidden surface removal?
- Complexity: at best equal to sorting
  - $O(n \log n)$
- Solving can be done in *image space* or in *object space*
  - image space: limited precision, $O(np)$ (p = number of pixels, 10^6)
  - object space: infinite precision, $O(n^2)$

Solving in Object space

- May be required, but has a worst case:

n objects



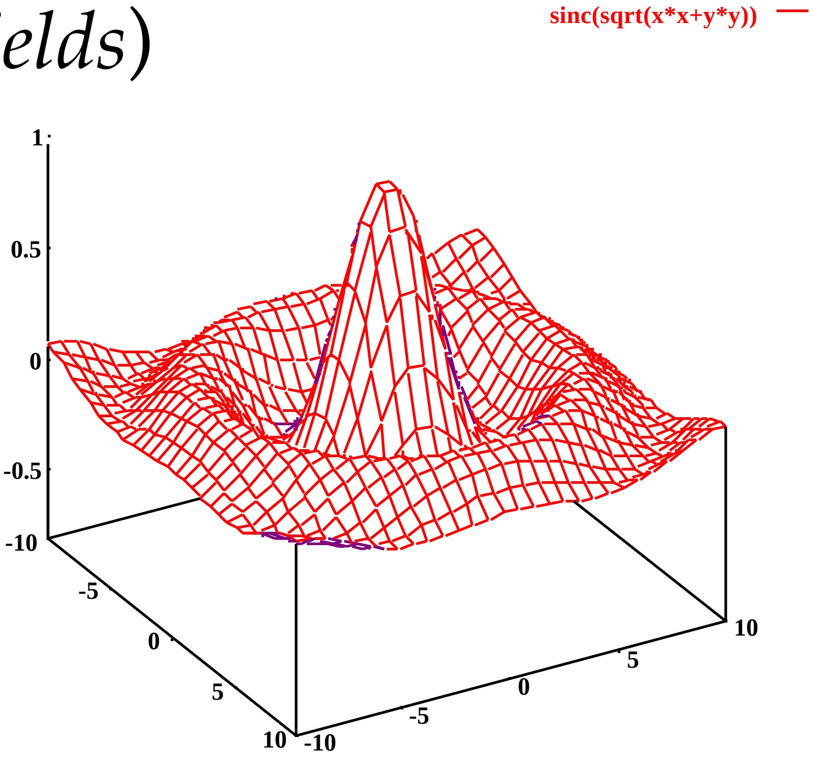
n^2 after visibility

Solving in Image Space

- Allows for using coherence:
  - object visible at one pixel is likely to be also visible at neighbouring pixels
- faster computation on the average
- worst case still O(np)

Floating Horizon

- For single valued functions of two variables (*height fields*)
  - $h = f(x,y)$

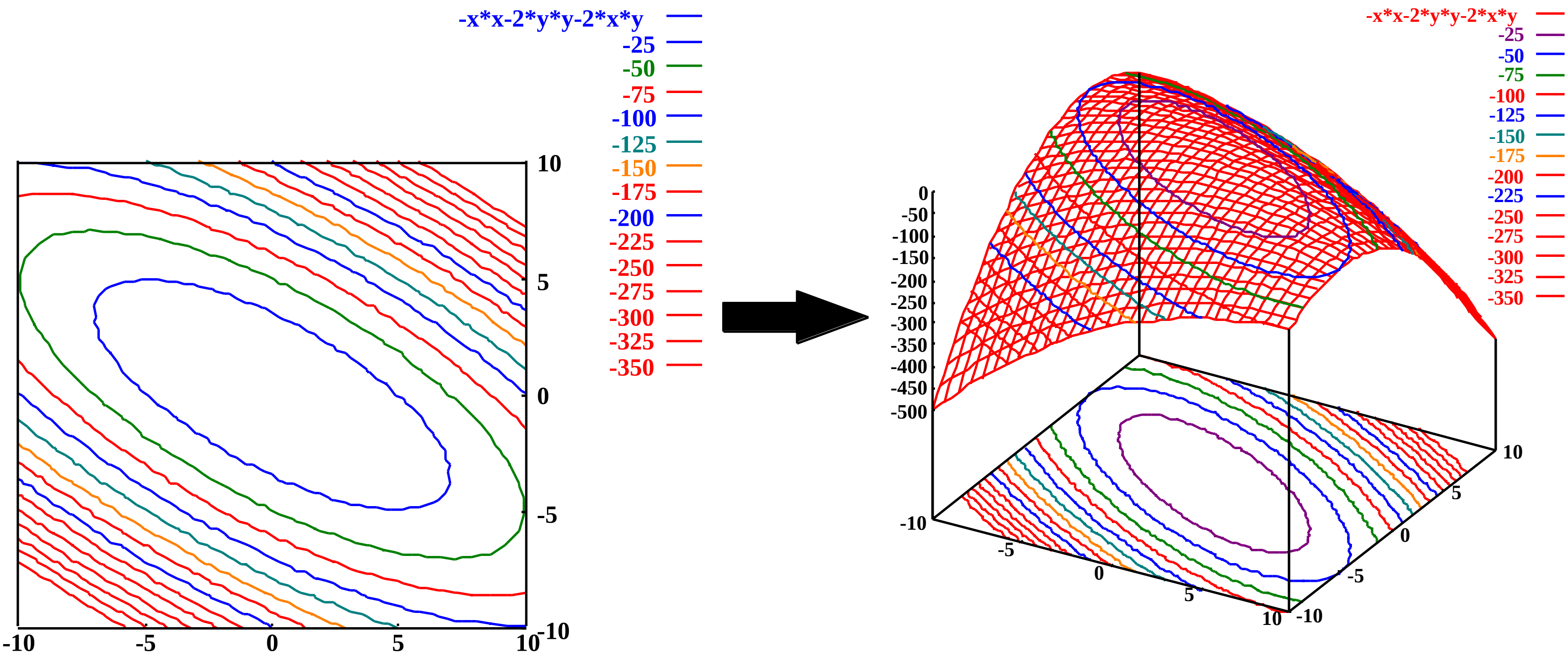


Floating Horizon (2)

- Do the *v*-lines one at a time
  - For each value of *u* on the *v*-line
  - Keep track of highest and lowest *h*-value (= horizons)
  - clip the curve segment against the two horizons
- Image-space: use an array
- Object-space: use a list

Contouring

- Another way of showing height fields:

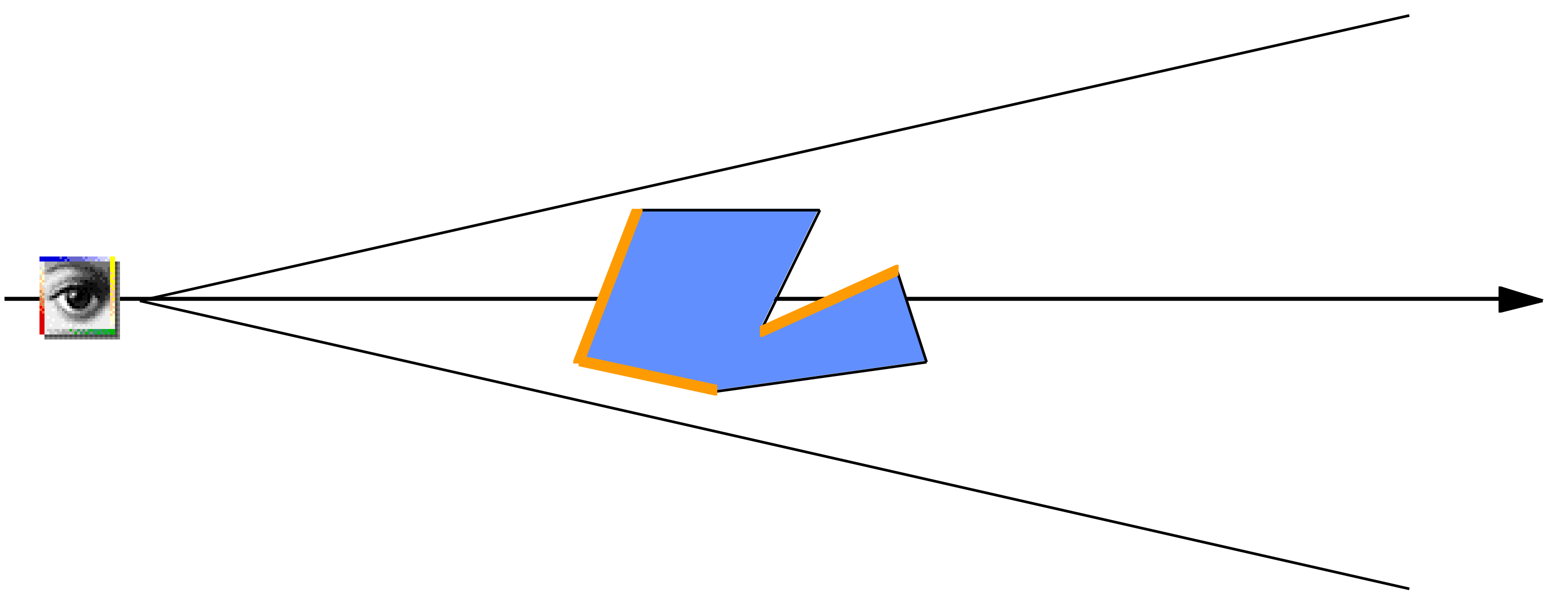


Visible Surfaces

- Some simple ideas:
  - Back-face elimination
  - Depth-Sort algorithms (painter)
- Some more complex ideas:
  - BSP-trees
  - Area subdivision
  - Scanline
  - Z-buffer

Back Face Culling

- Eliminate all polygons facing away from the eye:



Back Face Culling Algorithm

- If the eye isn't in front of the polygon, don't display it.
- Dot product:
  - (Vertex-ViewPoint)*normal
  - > 0 : keep it
  - > 0 : eliminate it

Back Face Culling: summary

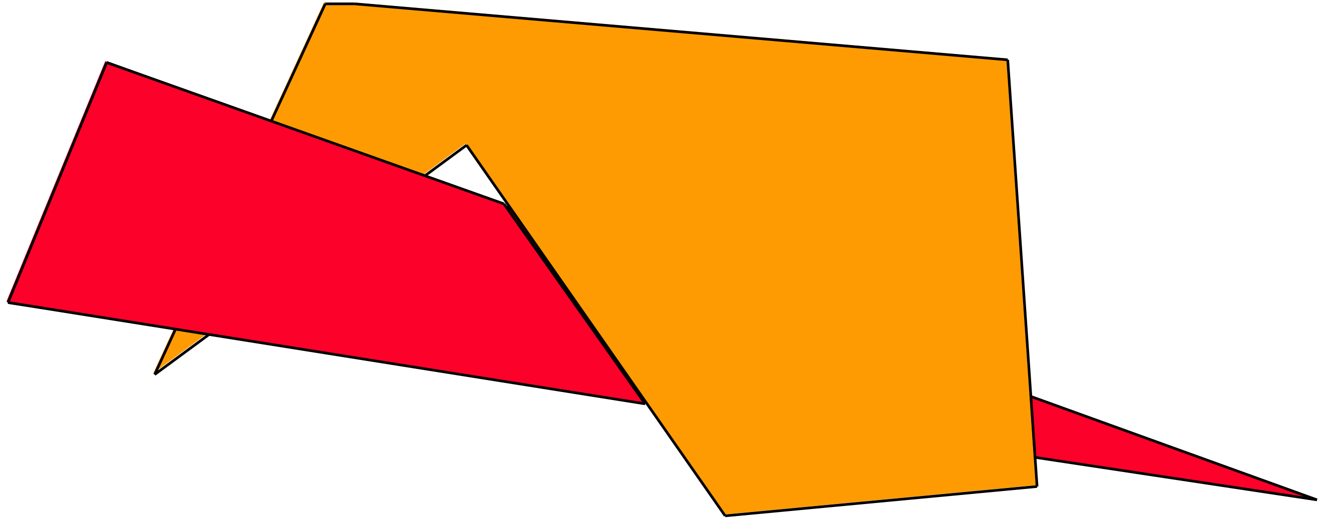
- On the average, 50 % speedup
- Small cost
- Preliminary step for all other algorithms
- Back Face Culling is sufficient if there is one convex object (cube, pyramid)

Depth sort algorithms

- Draw polygons that are far away first, then polygons closer to the eye
- Closer polygons will hide far away polygons
- Like a painter drawing the horizon first, then the paysage, then the front scene

Depth Sort: basic algorithm

- Need to sort polygons according to distance to the eye
- The order is incomplete
- There may be problems:



Depth Sort: complete algorithm

- Sort all polygons according to their farthest z coordinate
- If two polygons have overlapping z-range:
  - test if their extents are separate: no problem
  - test if one is fully behind the other
  - test if their projections are separate
  - if all fails, we have to split one of the polygon

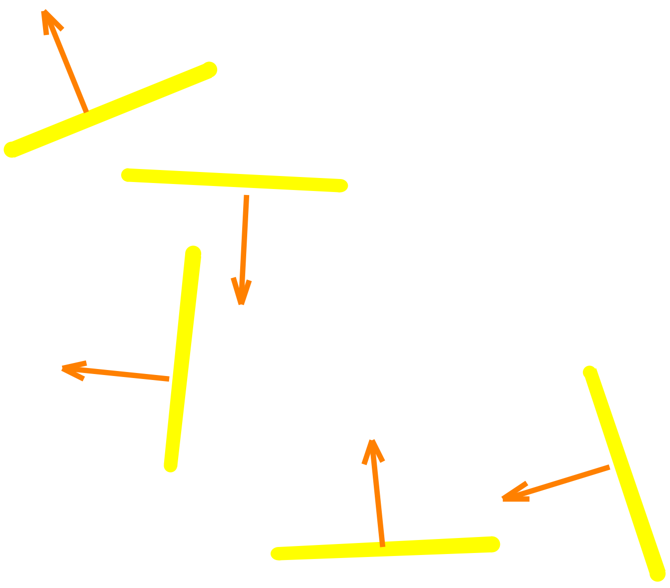
Depth sort: discussion

- Most intuitive algorithm
- Memory cost:
  - uses only screen space for display: O(p)
  - sorting necessary: O($n \log n$)
- Time cost:
  - you have to draw the entire scene
  - only valid for simple scenes

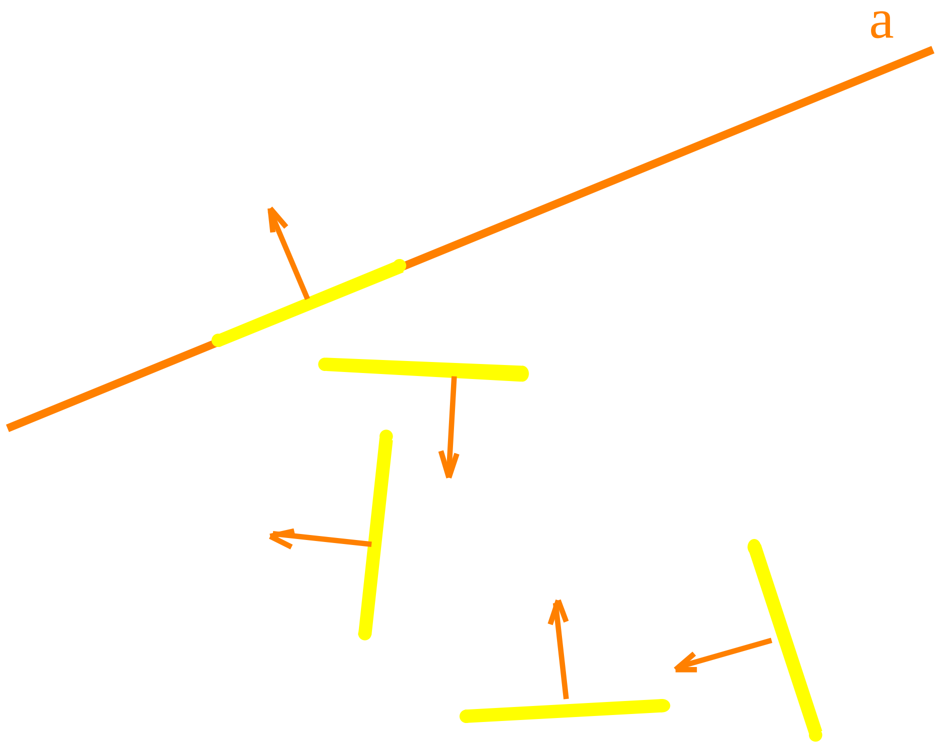
BSP-Trees: description

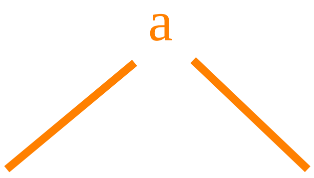
- Build a 3D BSP-Tree for the whole scene
  - splitting polygons if cut by the node plane
- Display the polygons according to their position in the tree:
  - first, polygons behind current node
  - then, current node
  - then, polygons in front of current node

Building a BSP-Tree (1)

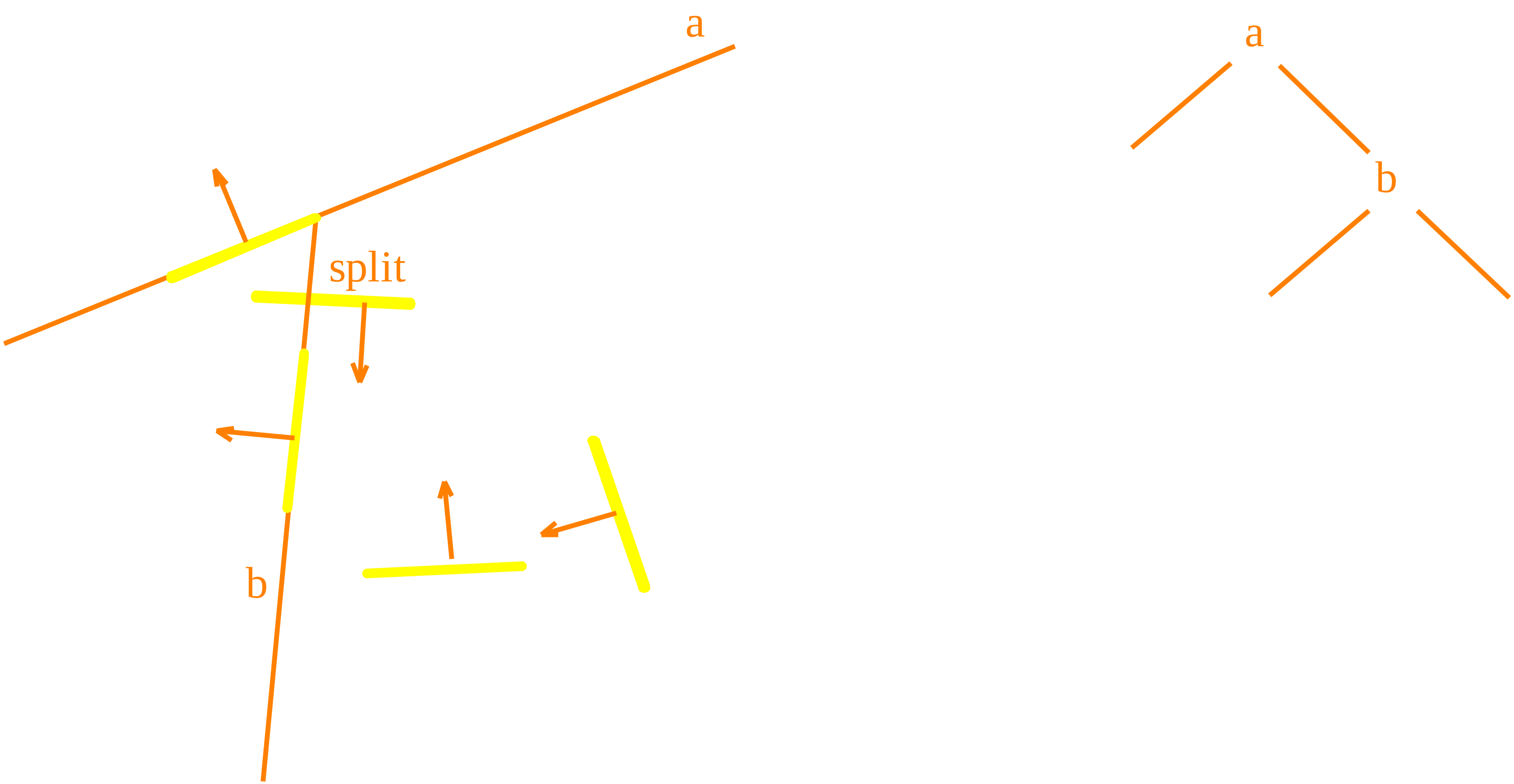


Building a BSP-Tree (2)

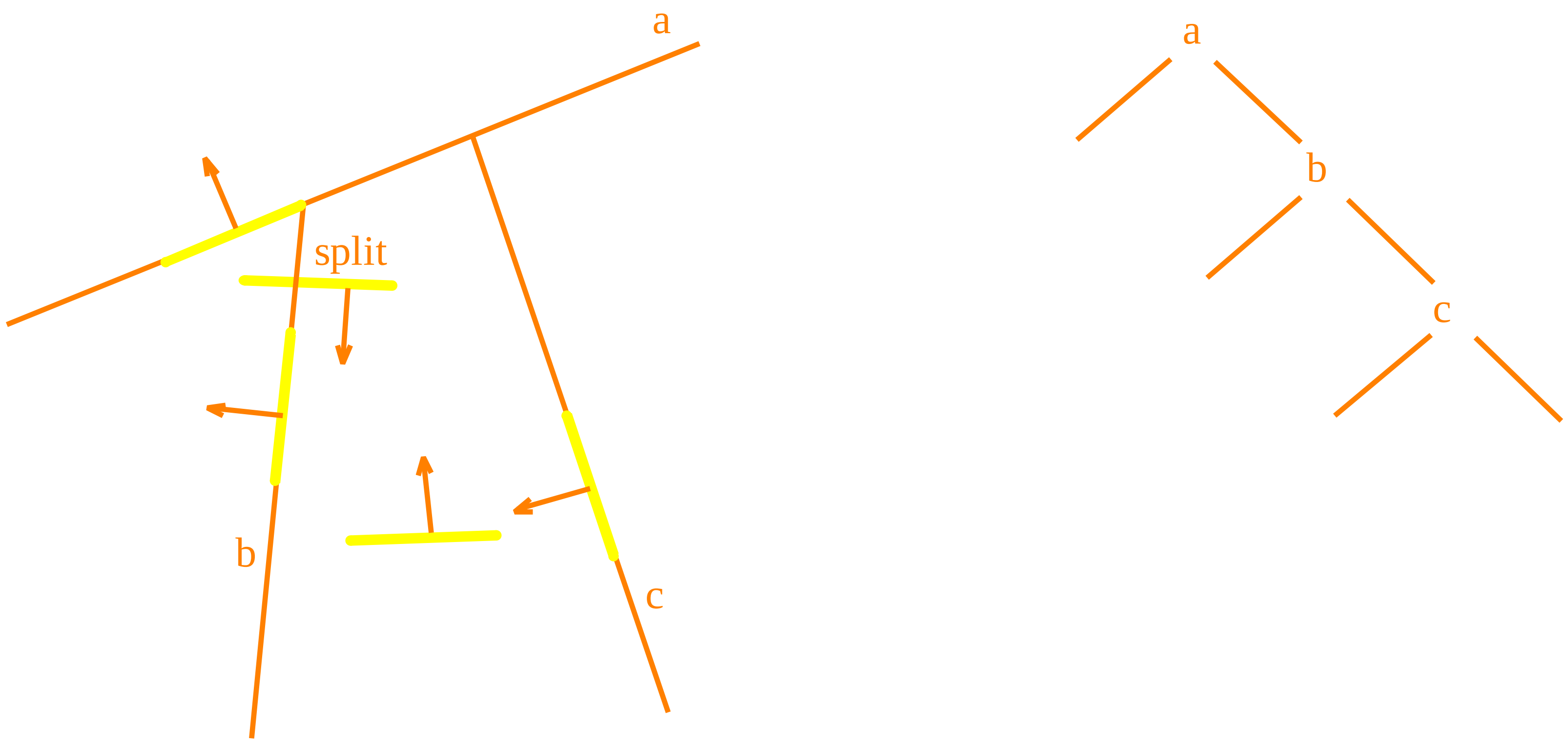




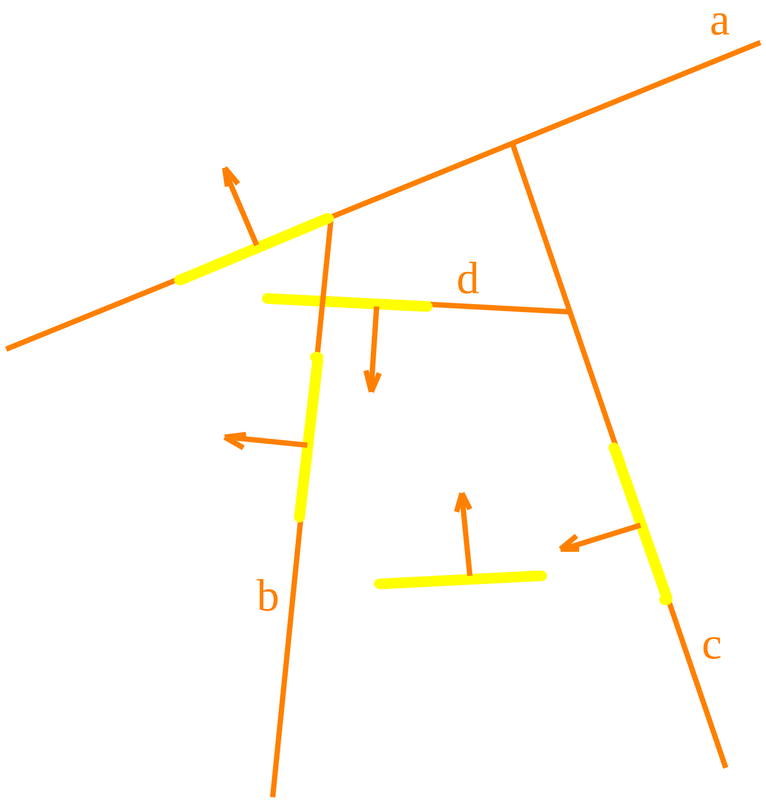
Building a BSP-Tree (3)

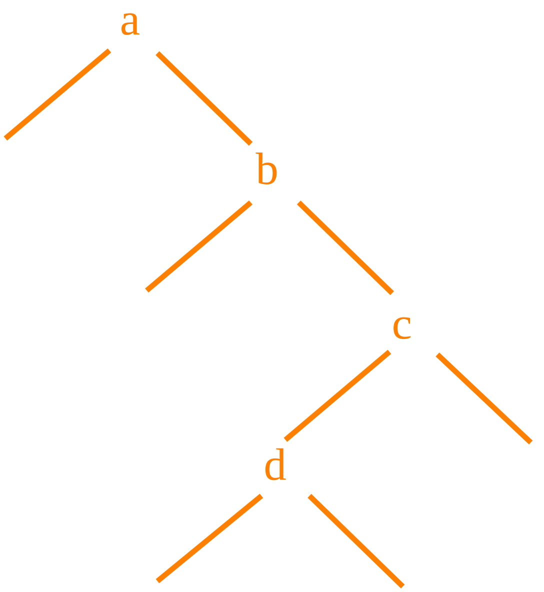


Building a BSP-Tree (4)

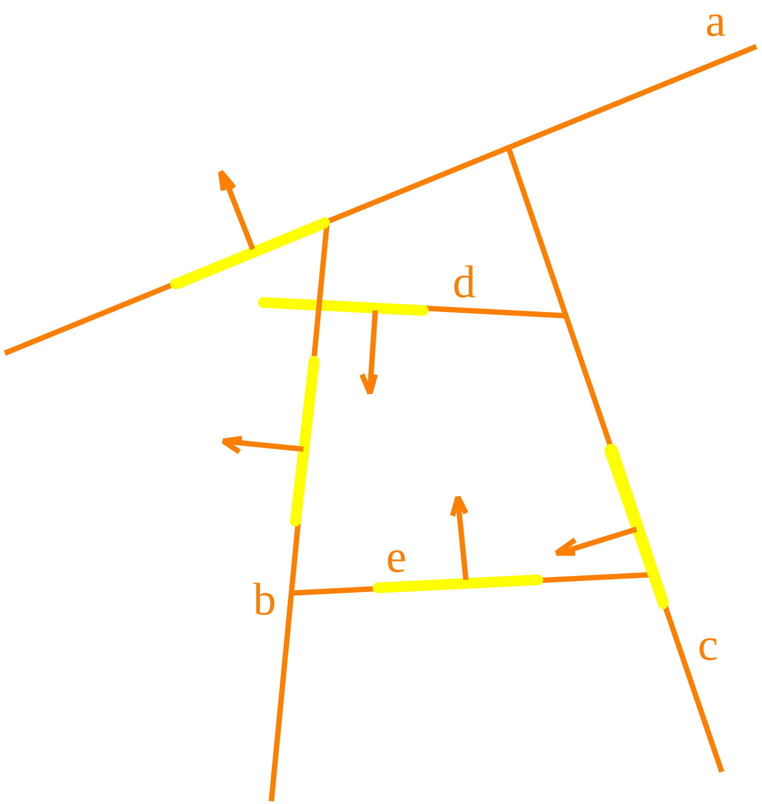


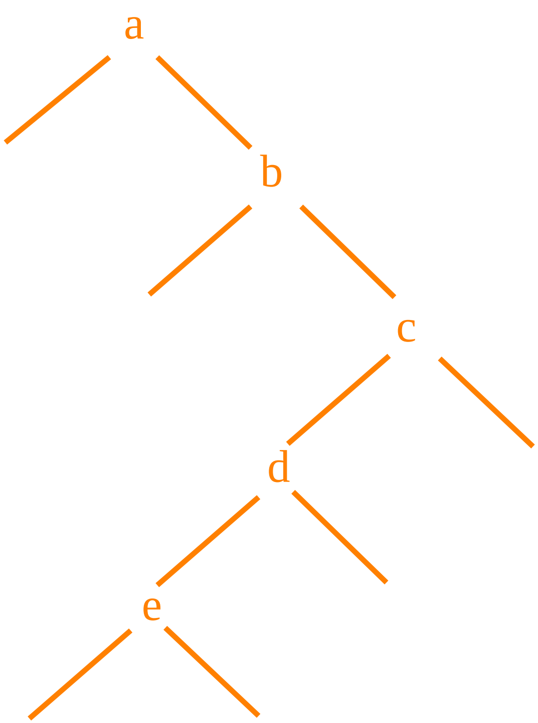
Building a BSP-Tree (5)



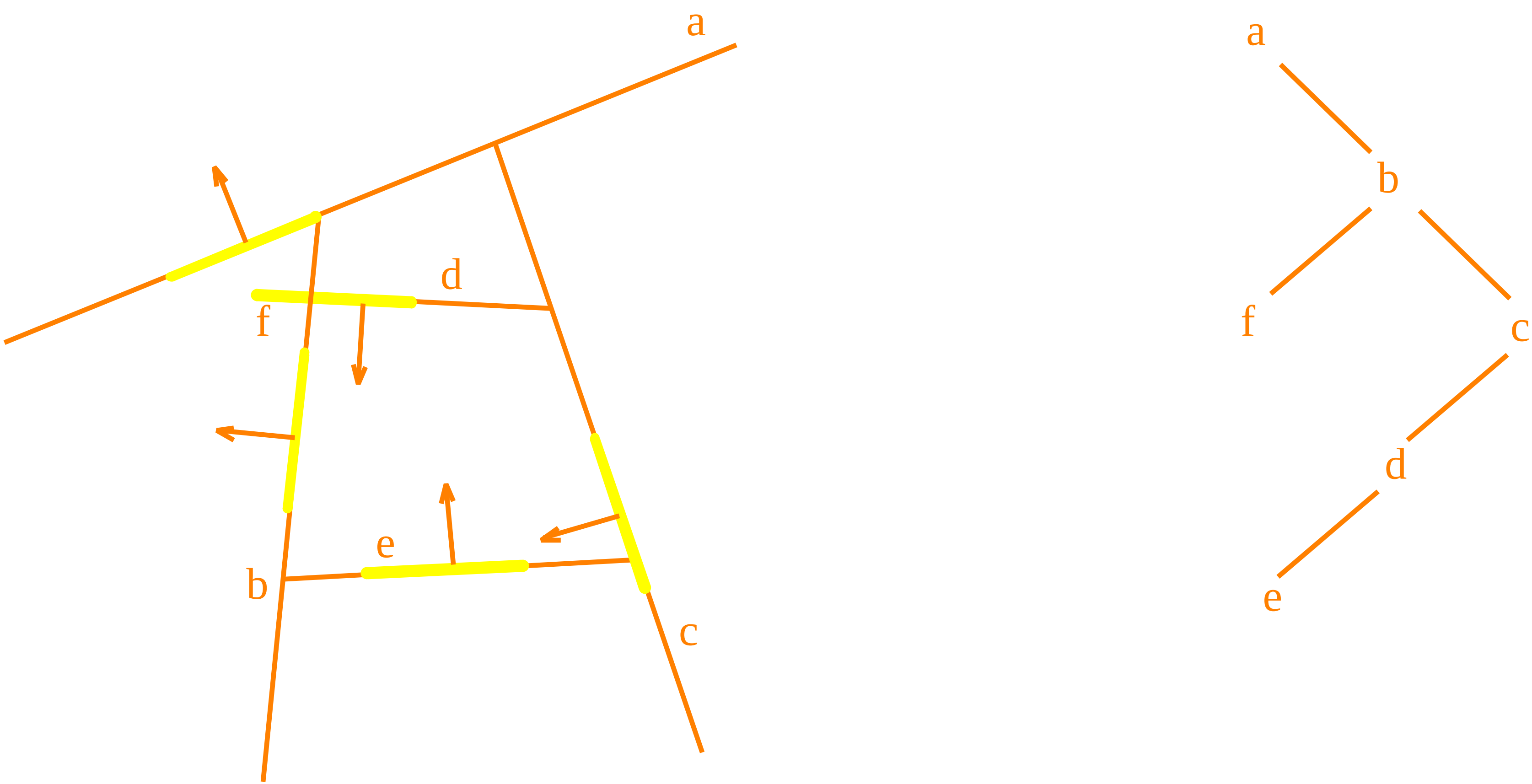


Building a BSP-Tree (6)

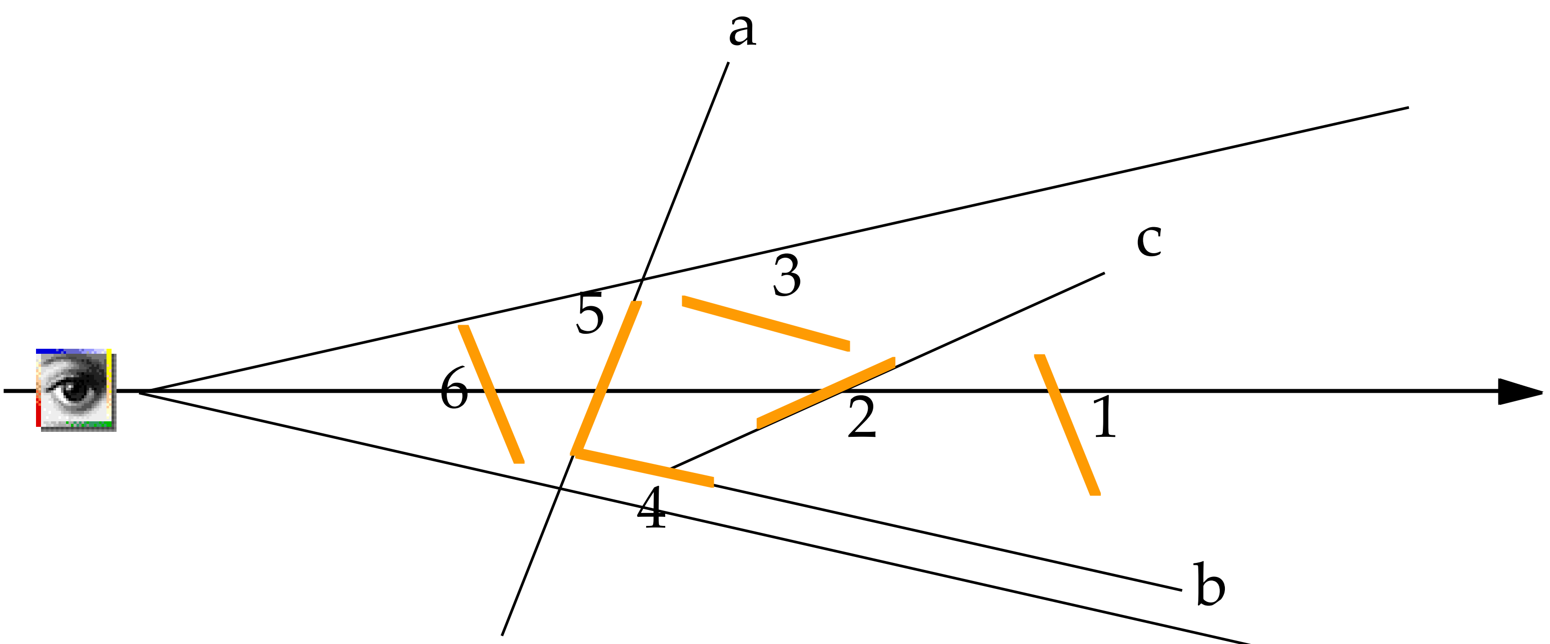




Building a BSP-Tree (7)



Using a BSP-Tree for visibility



BSP Tree *vs* standard depth-sort

- BSP-Tree will induce more polygon cuts
- But there is no “special case” for display
- BSP-Tree:
  - bigger pre-treatment
  - small time per request: move the viewpoint
- Painter:
  - no pre-treatment
  - bigger time per request

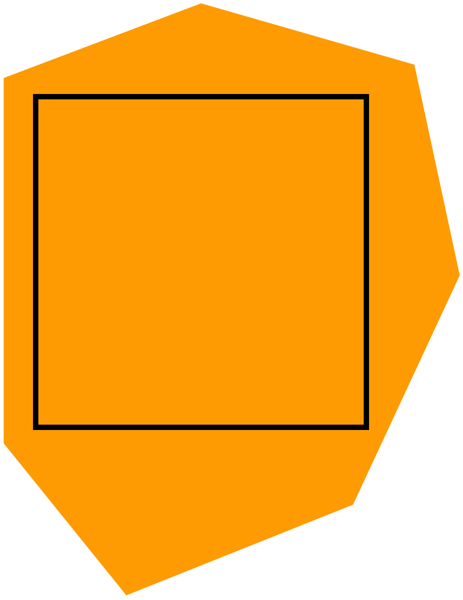
BSP-tree: discussion

- Useful as a pre-treatment for object space solving
- Memory cost:
  - lots of additional polygons
- Time costs:
  - building the BSP-tree
  - you still have to display the entire scene
- Front-to-Back BSP trees: Doom

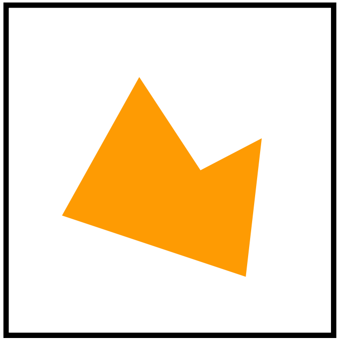
Area Subdivision (aka Warnock)

- Uses spatial coherence
- Divides the screen in small areas
- For each area, considers only polygons intersecting with the area
- If visibility is not clear, subdivide
- Stop subdivision when you reach pixel size

Polygons / area of interest

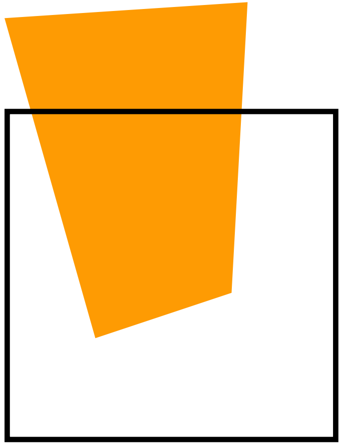


Surrounding

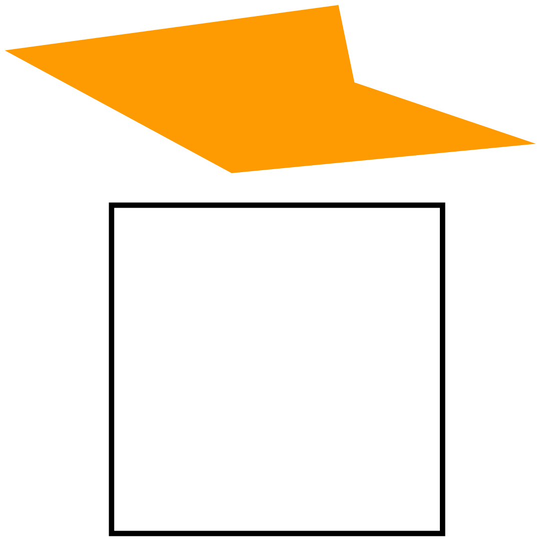


Contained

Polygons / area of interest



Intersecting

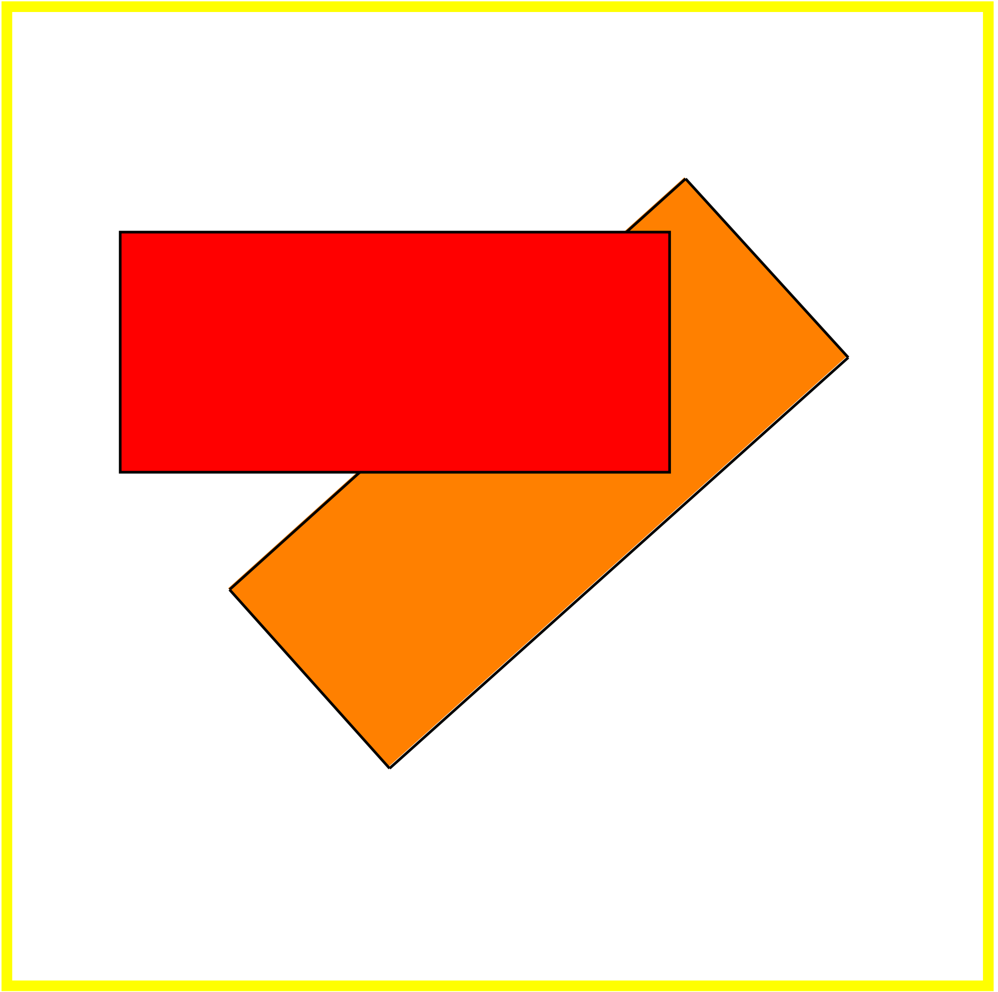


Disjoint

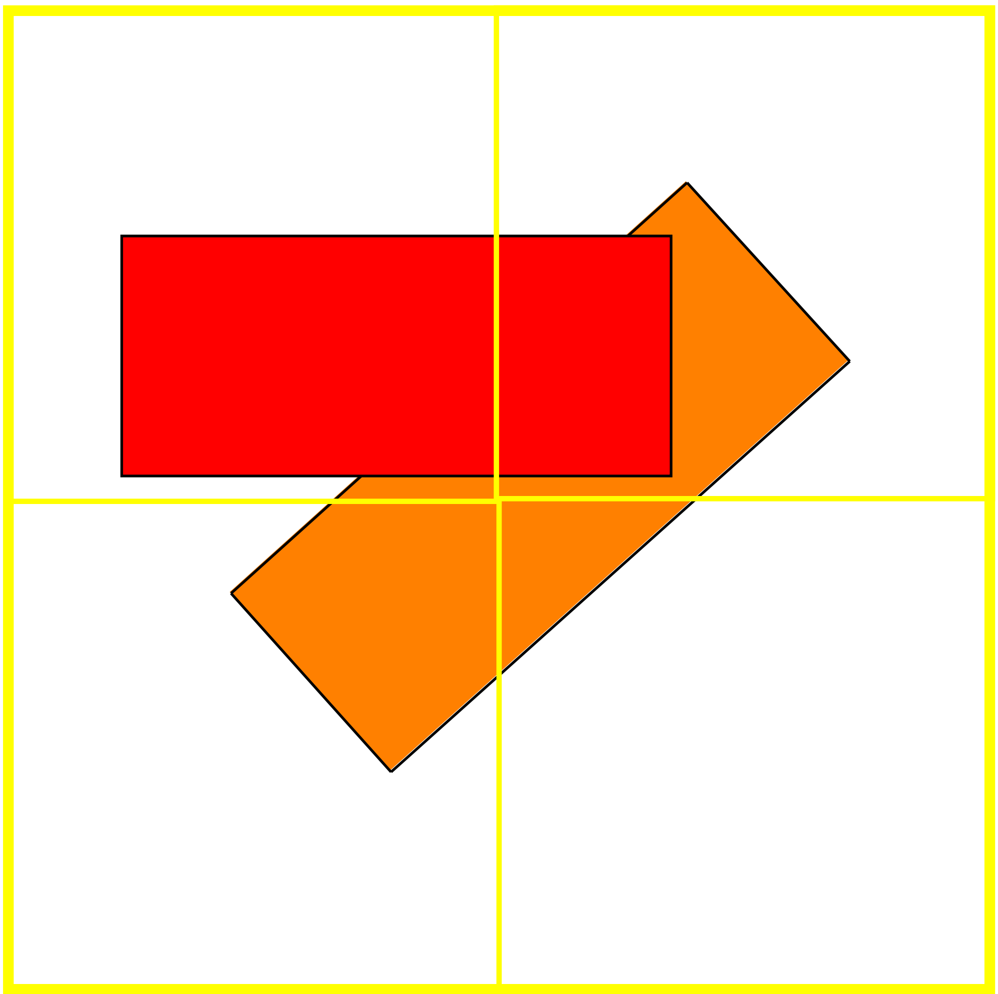
When is visibility clear?

- All polygons are disjoint from the area
- Only one intersecting or contained polygon in the area
- One surrounding polygon in front of all the others

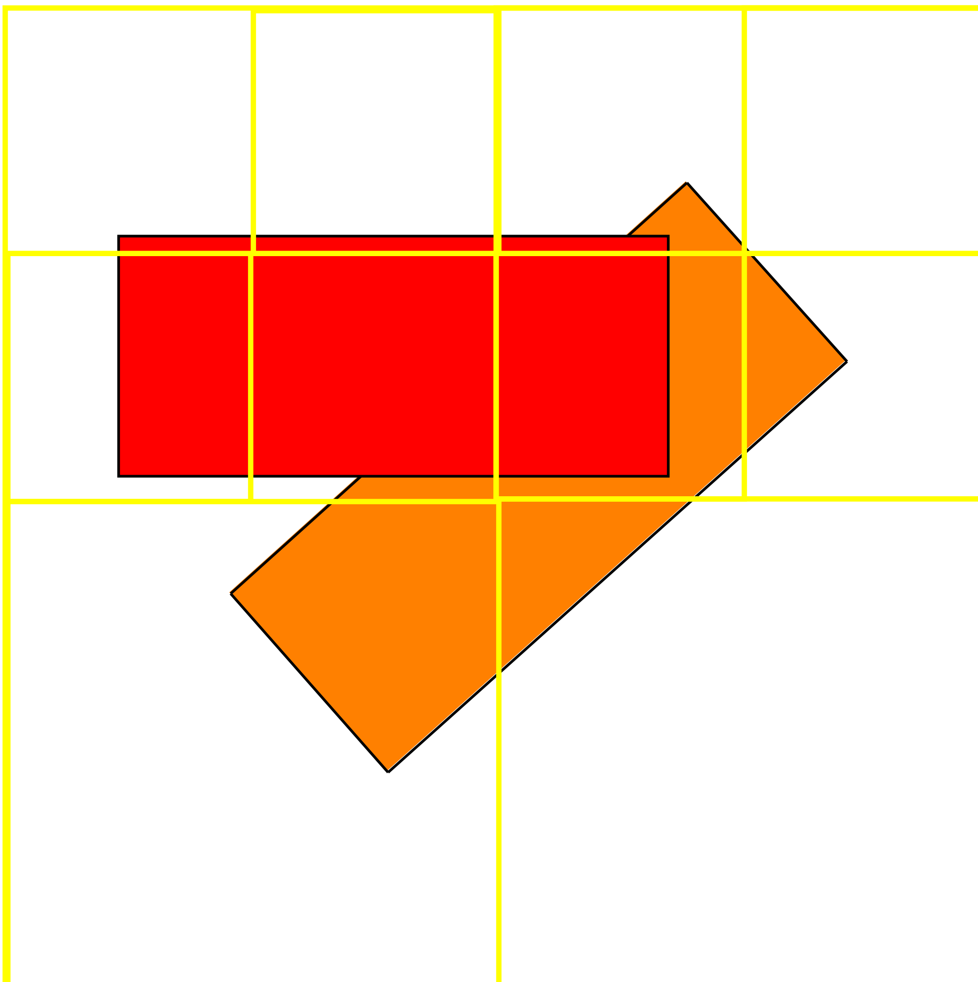
Warnock algorithm: example (1)



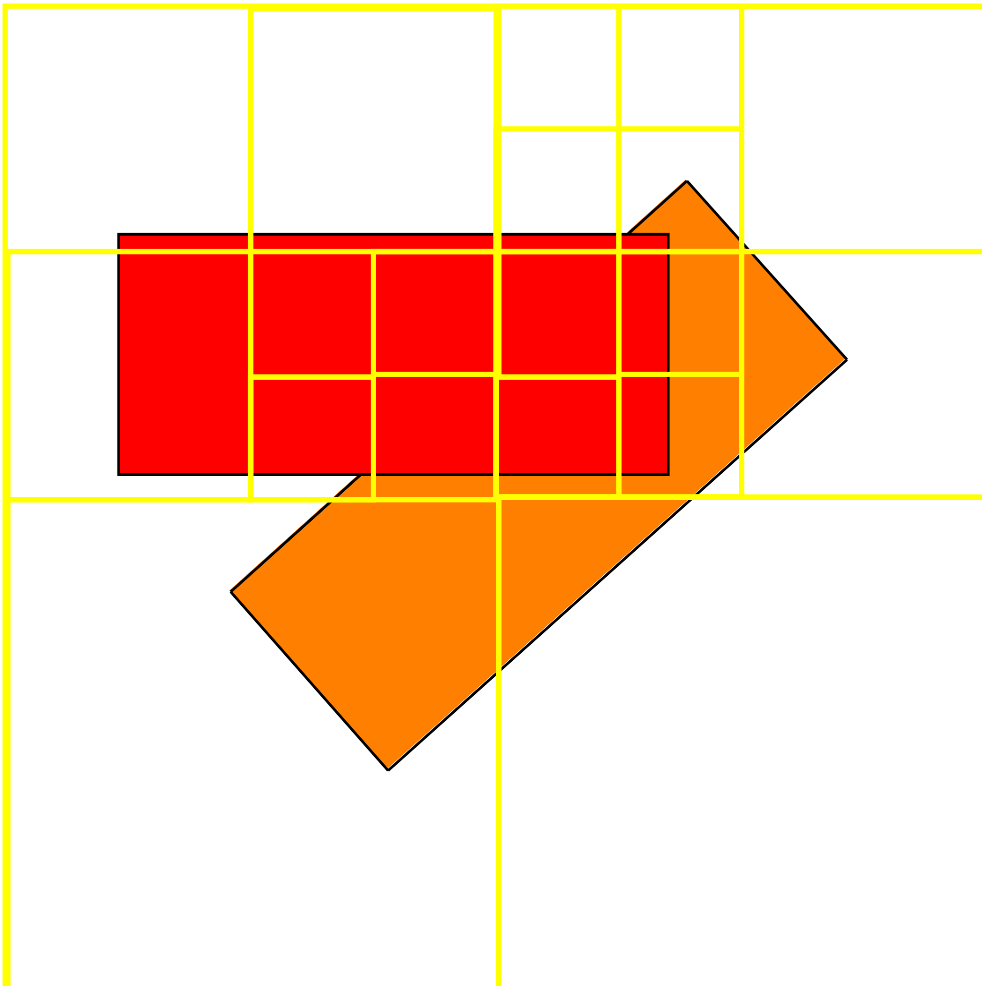
Warnock Algorithm: example (2)



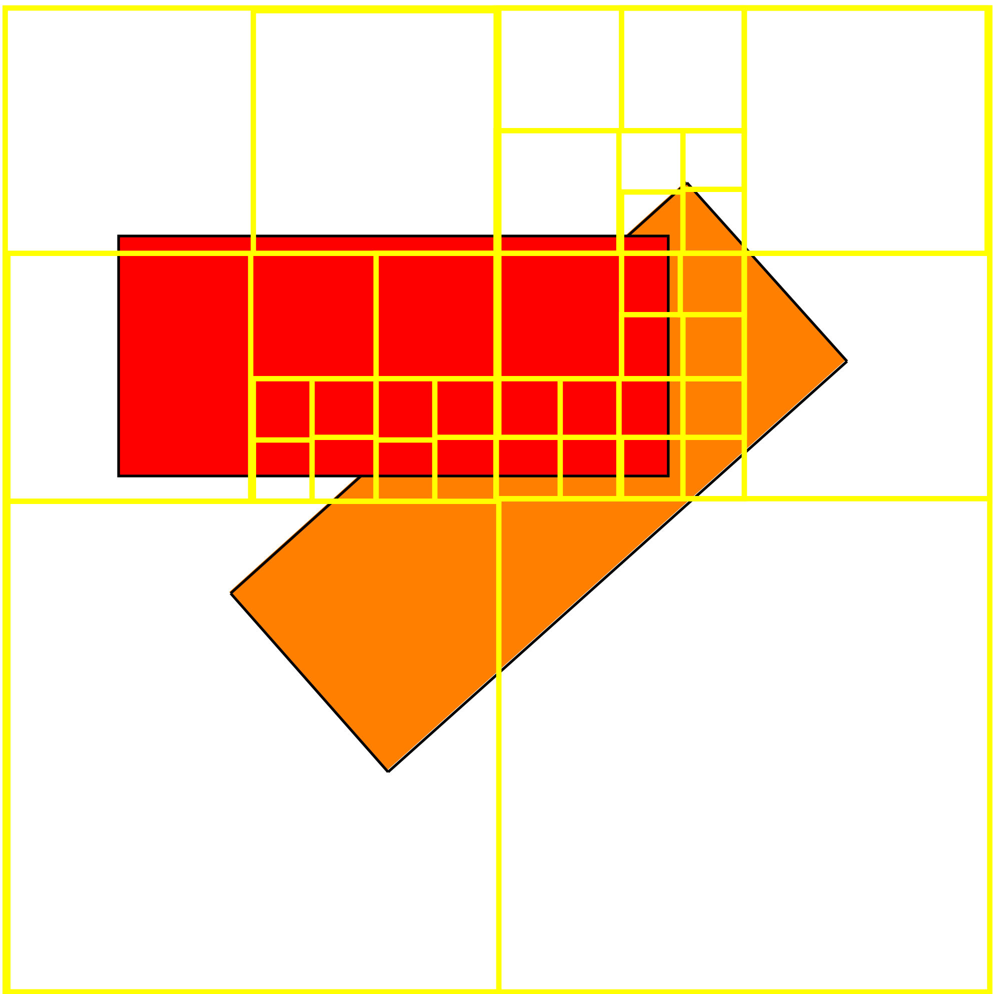
Warnock Algorithm: example (3)



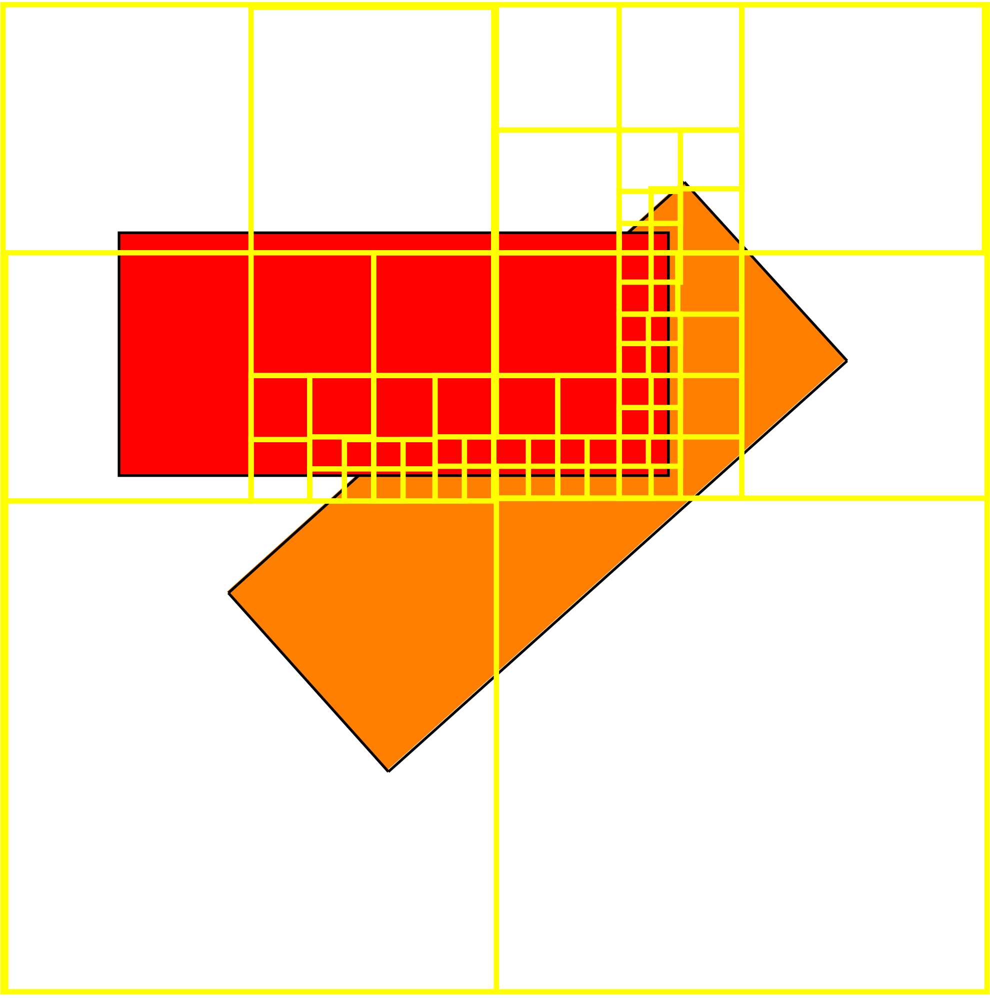
Warnock Algorithm: example (4)



Warnock Algorithm: example (5)



Warnock Algorithm: example (6)



Warnock algorithm: discussion

- Uses spatial coherence
- Useful with many large polygons
- Memory costs can be large
- Easy implementation: recursive calls to function

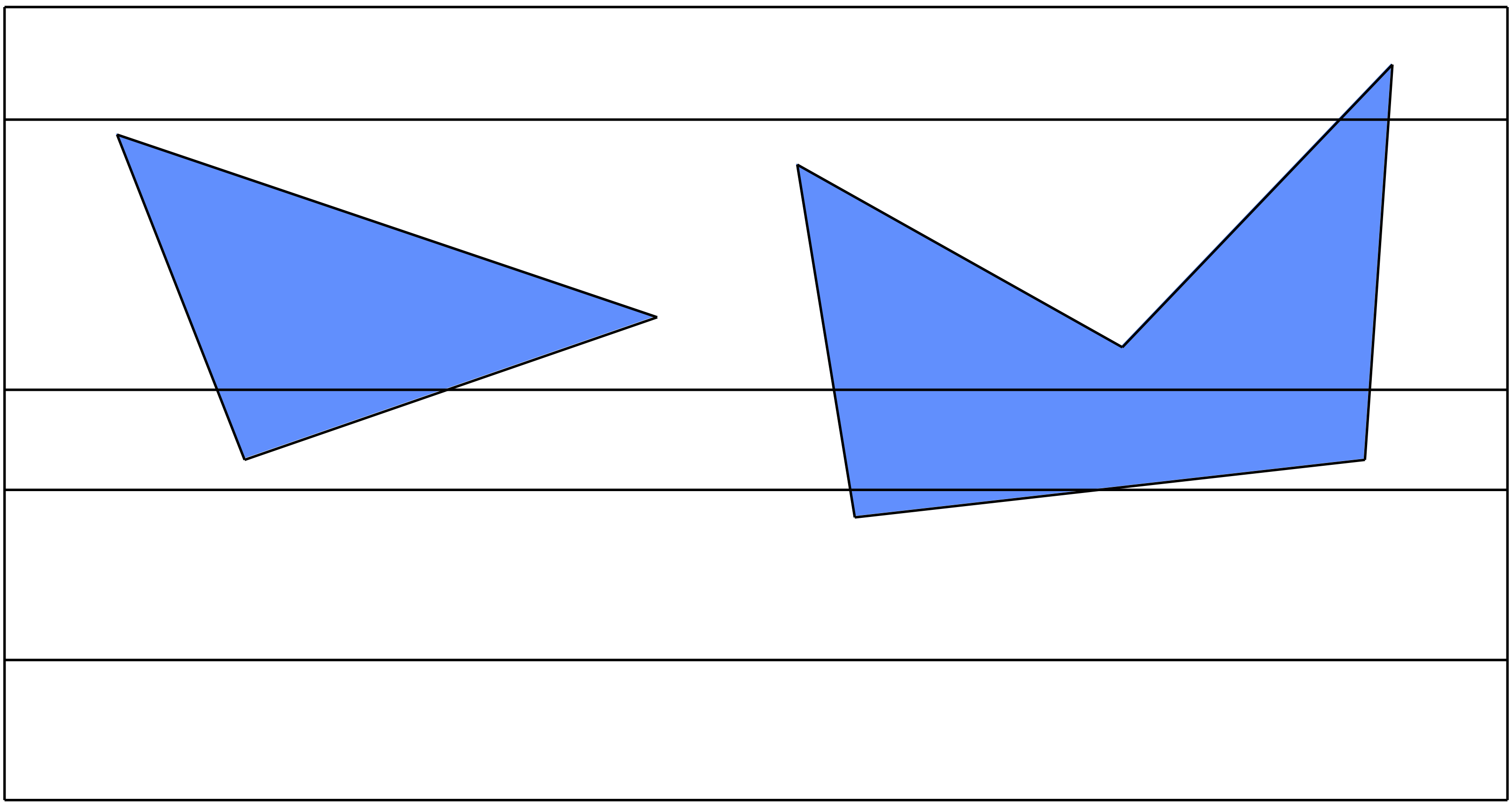
Scan Line algorithm

- Operate scan line by scan line
- For each scan line, find polygons in front
- Display the scan line

Scan Line Algorithm

- Sort all polygon edges:
  - into buckets, by smaller y coordinate
  - in each bucket, by slope
- Walk along the scan line:
  - if edge is encountered, polygon is *in*.
  - if only one polygon *in* at a time: no problem.

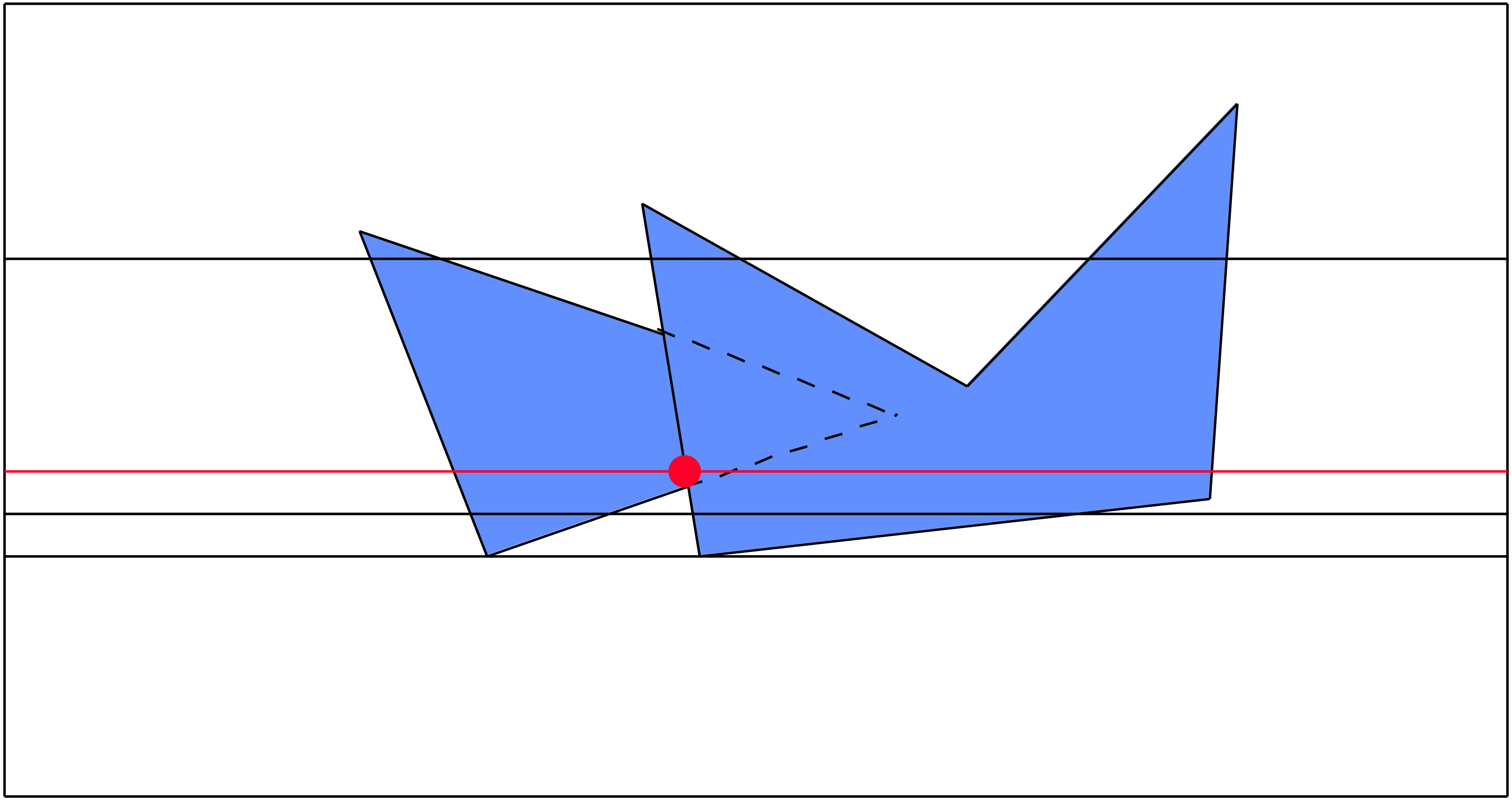
Scan Line Algorithm



Scan Line Algorithm

- If two polygons are *in* at a time:
  - when encountering the starting edge of the second polygon, find which is in front.
  - display only the polygon in front.

Scan Line Algorithm



Scan line: discussion

- Low memory cost
- Uses scan-line coherence
  - but not vertical coherence
- Has several side advantages:
  - filling the polygons
  - reflections
  - texture mapping
- Renderman (Toy Story) = only scan line

Z-Buffer

- Have one array, the size of the screen;
- Store maximal z value at this pixel
- Initially, all points at minus infinity
- Update the points that fall inside the projection of each polygon

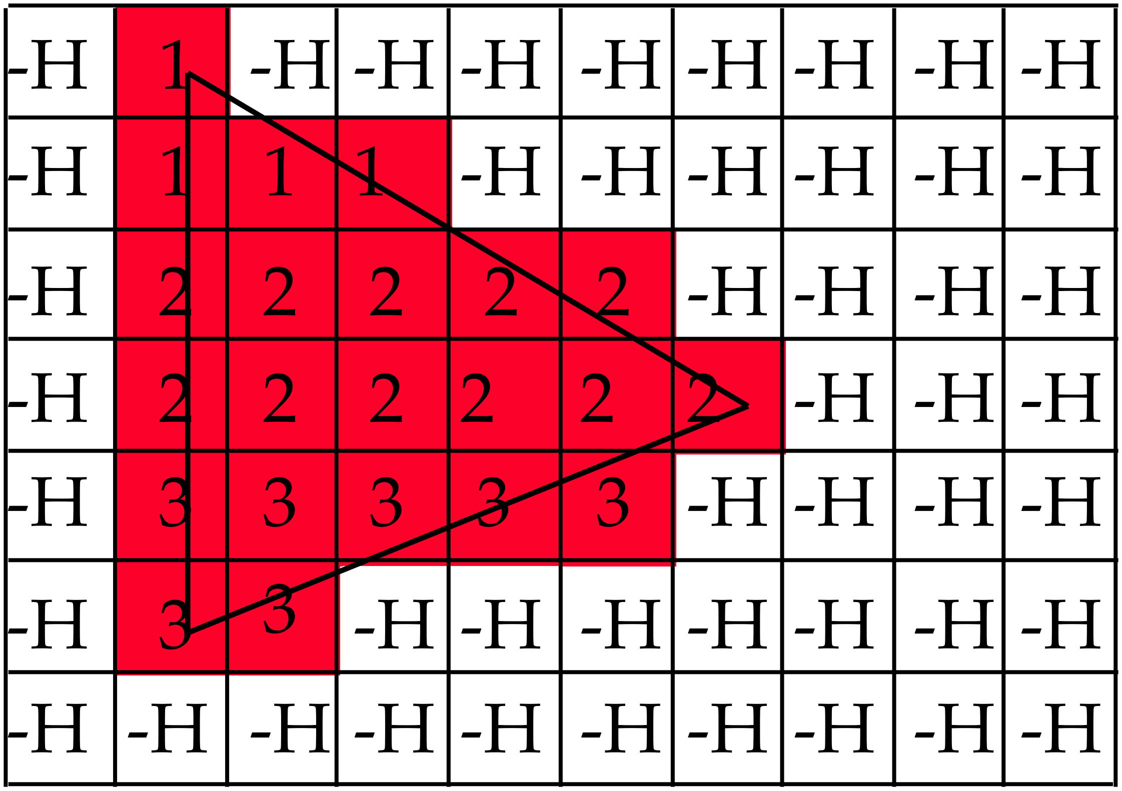
Z-buffer algorithm

- For each polygon:
  - For each pixel in polygon's projection
    - compute z-value at this pixel
    - if z-value is in front of current max z-value
      - change maximal z-value
      - write pixel on the screen using polygon color

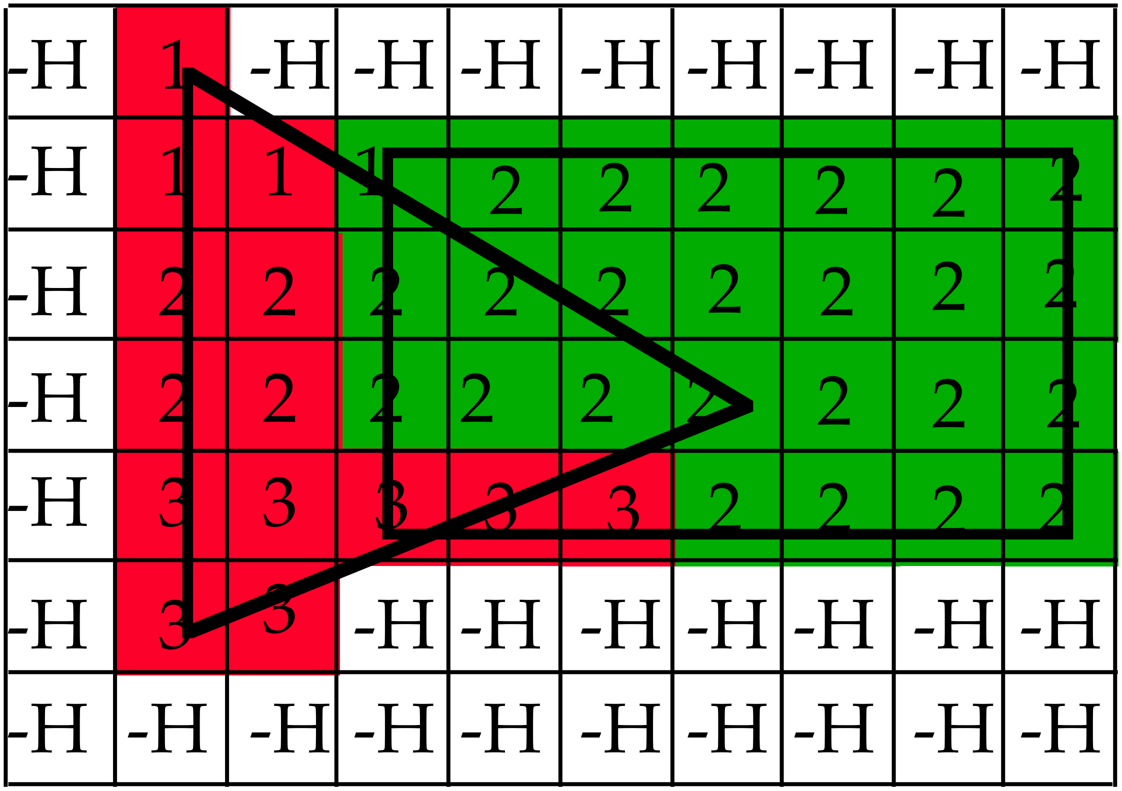
Z-buffer algorithm

-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H
-H	-H	-H	-H	-H	-H	-H	-H	-H	-H

Z-buffer algorithm



Z-buffer algorithm



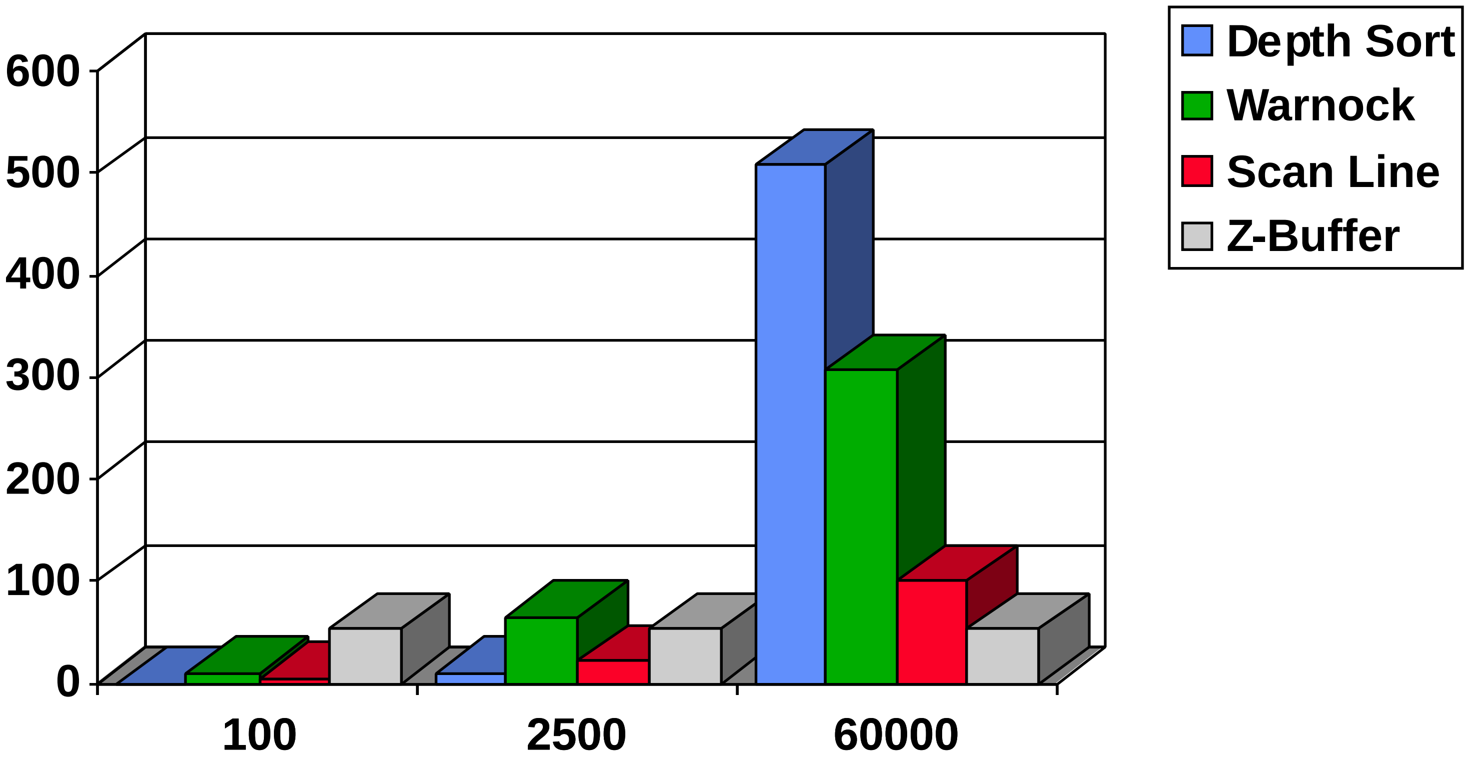
Z-Buffer: discussion

- Pros:
  - simple to implement
  - operates at image precision
    - faster
- Cons:
  - memory cost
  - operates at image precision
    - aliasing
    - artefacts

Z-Buffer

- How many bits of information?
  - limited by memory costs
    - 8 bits, 1024x1280: 1.25 Mb
    - 16 bits, 1024x1280: 2.5 Mb
  - needed for separation of objects that are close to each other:
    - 8 bits, minimal distance is 0.4 % (4mm at 1m)
    - 16 bits, minimal distance is 0.001 % (1mm at 1km)
    - what happens below this distance?

Relative time costs



Which algorithm?

- Depends on expected scene complexity
- Z-Buffer for complex scene:
  - hardware implementation
  - no pre-sorting required
  - but memory?
- Scan-line is low cost choice:
  - lowest memory costs
  - can be done during display