The A-buffer

• For transparent surfaces and filter based anti-aliasing:

• Algorithm (1): filling buffer
  – at each pixel, maintain a pointer to a list of polygons sorted by depth.
  – when filling a pixel:
    • if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    • if polygon is opaque and only partially covers pixel, insert into list, but don’t remove farther polygons

The A-buffer

• Algorithm (2): rendering pixels
  – at each pixel, traverse buffer using brightness values in polygons to fill.
  – values are used for either for calculations involving transparency or for filtering for aliasing

Scan line algorithm

• Assume (for a moment) that polygons do not intersect one another (in 3D).
• Observation: on any given scan line, the visible polygon can change only at an edge.
• Algorithm:
  – fill all polygons simultaneously at each scan line, have all edges that cross scan line in AEL
  – keep record of current depth at current pixel
  – use current depth to decide which polygon to use for a span when a new edge is encountered

Scan line algorithm

(non-intersecting in 3D)

Depth checks to make sure that correct one is filled
Scan line algorithm

- To deal with penetrating polygons, split them up

```
Scan line algorithm
• Advantages:
  – potentially fewer quantization errors (typically more
    bits available for depth because a separate frame-size
    buffer is not needed)
  – filter anti-aliasing can be made to work somewhat
• Disadvantages:
  – invisible polygons clog AEL and ET (can get
    expensive for complex scenes)
```

Depth sorting

- Logically like painter’s algorithm, except that problem cases are
  handled properly (painters++).

- First, sort polygons in order of decreasing depth.
  - (based on closest point)

- Render in sorted order as described next slide.

- Each polygon encountered is checked using several tests to see if it is
  safe to paint (otherwise fix).

- Structure cases so that cheap tests are done first, and hope that
  expensive tests do not occur too often.

```
Depth sorting
• For surface S with greatest depth
  – if no depth overlaps (z extent do not intersect) with other surfaces, then
    render (like painter’s algorithm), and remove surface from list
    • test for depth overlaps by considering the z bounds (extents).
  – if a depth overlap is found, test for problem overlap in image plane
    • see next slides
  – if S, S’ overlap in depth and in image plane, swap and try again
  – if S, S’ have been swapped already, split one across plane of other (like
    clipping) and reinsert
```
Depth sorting

• Testing image plane problem overlaps (test get increasingly expensive):
  – xy bounding boxes do not intersect
  – or S is behind the plane of S’
  – or S’ is in front of the plane of S
  – or S and S’ do not intersect

See figures

Depth sorting (2D illustrations)

No depth overlap between furthest object and rest of list—paint it.

Depth sorting (2D illustrations)

Overlap in depth, but no overlap in image plane.

This can determined by studying bounding boxes.

Depth sorting (2D illustrations)

It is safe to paint the furthest, but figuring this out requires observing that the near one is all on the same side of the plane (dotted line) of the furthest.

In other words: the near polygon is in front of the plane of the far polygon.
It is safe to paint the furthest, but figuring this out requires observing that it is all on the other side of the plane (dotted line) of the nearest plane.

In other words: the far polygon is behind the plane of the near polygon.

The furthest (as defined by the closest point) obscures the “nearest”. It is safe to paint the “nearest”, but figuring this out requires reversing the nearest and furthest, and then reapplying one of the previous tests.

If the preceding tests fail, we have to split the far polygon (line in the drawing) with the plane of the near polygon (basically a clip operation), and put the pieces into the list, and carry on.

- Advantages:
  - filter anti-aliasing works fine (with good over-painting rules)
  - no depth quantization error
  - works well if not too much depth overlap (rarely get to expensive cases)

- Disadvantages:
  - gets expensive with lots of depth overlap (over-renders)
BSP - trees

- Construct a tree that gives a rendering order

- Tree recursively splits 3D world into cells, each of which contain at most one piece of polygon.

- Constructing tree:
  - choose polygon (arbitrary)
  - split its cell using plane on which polygon lies
  - continue until each cell contains only one polygon

BSP - trees

2D version for illustration
BSP - trees

• Rendering tree:
  – recursive descent
  – render back, node polygon, front
    • back/front is determined by what side of the plane the camera is on
• Disadvantages:
  – many small pieces of polygon (more splits than depth sort!)
  – over rendering (does not work well for complex scenes with lots of depth overlap)
• Advantages:
  – one tree works for all focal points (good for cases when scene is static)
  – filter anti-aliasing works fine, as does transparency
  – data structure is worth knowing about
• Comment
  – expensive to get approximately optimal tree, but for many applications this can be "off-line" in a pre-processing step.