Visibility

• image space algorithms:
  – operate on pixels or scan-lines
  – visibility resolved to the precision of the display
  – e.g.: Z-buffer
• object space algorithms
  – explicitly compute visible portions of polygons
  – painter’s algorithm: depth-sorting, BSP trees

Z-buffer

store \((r, g, b, z)\) for each pixel

• typically 8+8+8+24 bits, can be more
  for all \(i, j\) {
    \(\text{Depth}[i, j] = \text{MAX\_DEPTH}\)
    \(\text{Image}[i, j] = \text{BACKGROUND\_COLOUR}\)
  }
  for all polygons \(P\) {
    for all pixels in \(P\) {
      if \(Z\_pixel < \text{Depth}[i, j]\) {
        \(\text{Image}[i, j] = C\_pixel\)
        \(\text{Depth}[i, j] = Z\_pixel\)
      }
    }
  }

• hardware support in graphics cards
• poor for high-depth-complexity scenes
  – need to render all polygons, even if most are invisible
  – "jaggies": pixel staircase along edges

The A-Buffer

• antialiased, area-averaged accumulation buffer
  – z-buffer: one visible surface per pixel
  – A-buffer: linked list of surfaces
  – data for each surface includes
    • RGB, Z, area-coverage percentage, ...

BSP trees

Binary Space Partitions

• object-space method
• produces a back-to-front ordering
• build the BSP tree once
• traverse the BSP in a view-dependent fashion

BSP trees (example)
**Building a BSP tree**

BSPtree *BSPmaketree(polygon list) {
    choose a polygon as the tree root
    for all other polygons {
        if polygon is in front, add to front list
        if polygon is behind, add to behind list
        else split polygon and add one part to each list
    }
    BSPtree = BSPcombinetree(BSPmaketree(front list),
                           root, BSPmaketree(behind list) )
}

**Using a BSP tree**

* producing a back-to-front ordering

DrawTree(BSPtree) {
    if (eye is in front of root) {
        DrawTree(BSPtree->behind)
        DrawPoly(BSPtree->front)
        DrawPoly(BSPtree->root)
    } else {
        DrawTree(BSPtree->front)
        DrawPoly(BSPtree->root)
        DrawTree(BSPtree->behind)
    }
}

**Ray Tracing**

- cast a ray through each pixel
- requires efficient intersection tests
  - walk along ray until first intersection

Ray tracing:

for each pixel on screen {
    determine ray from eye through pixel
    colour = raytrace(ray)
    set pixel to colour
}

colour raytrace(ray) {
    find closest intersection of ray with an object
    reflect_colour = raytrace(reflected_ray)
    refract_colour = raytrace(refracted_ray)
    local_colour = lighting_computation()
    return k1*reflect_colour + k2*refract_colour
           + k3*local_colour
}