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 \(i,j\) { for all pixels in \(P\) { if \(Z_{\text{pixel}} < Depth[i,j]\) { Image[i,j] = C_{\text{pixel}} Depth[i,j] = Z_{\text{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->root)
    DrawPoly(BSPtree->front)
    } 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
}