Binary Space Partition (BSP) Trees

- Used in visibility calculations
- Building the BSP tree (2D)
  - Start with polygons and label all edges
  - Deal with one edge at a time
  - Extend each edge so that it splits the plane in two, it's normal points in the "outside" direction
  - Place first edge in tree as root
  - Add subsequent edges based on whether they are inside or outside of edges already in the tree. Inside edges go to the right, outside to the left. (opposite of Hill)
  - Edges that span the extension of an edge that is already in the tree are split in two and both are added to the tree.
  - An example should help....
Using BSP trees

- Use BSP trees to draw faces in the right order
- Building tree does not depend on eye location
- Drawing depends on eye location
- Algorithm intuition:
  - Consider any face F in the tree
    - If eye is on outside of F, must draw faces inside of F first, then F, then outside faces. Why?
      - Want F to only obscure faces it is in front of
    - If eye is on the inside of F, must draw faces outside of F first, then F (if we draw it), than inside edges
- This forms a recursive algorithm
BSP Drawing Algorithm

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

Visibility Problem

- Z-Buffer
- Scanline
Z-Buffer

- Scanline algorithm
- Z-buffer algorithm:
  1. Store background colour in buffer
  2. For each polygon, scan convert and …
  3. For each pixel
     - Determine if z-value (depth) is less than stored z-value
     - If so, swap the new colour with the stored colour

Calculating Z

- Start with the equation of a line
  \[ 0 = A x + B y + C z + D \]
- Solve for Z
  \[ Z = (-A x - B y - D) / C \]
- Moving along a scanline, so want z at next value of x
  \[ Z' = (-A (x+1) - b y - D) / C \]
  \[ Z' = z - A/C \]
Calculating Z

- For moving between scanlines, know
  \[ x' = x + \frac{1}{m} \]
- The new left edge of the polygon is \((x + 1/m, y + 1)\), giving
  \[ z' = z - \frac{A/m + B}{C} \]

Z-Buffer Pros and Cons

- Needs large memory to keep Z values
- Can be implemented in hardware
- Can do infinite number of primitives.
- Handles cyclic and penetrating polygons.
- Handles polygon stream in any order throwing away polygons once processed
A-Buffer

- A-buffer

Z-Buffer with anti-aliasing
  - (much more on anti-aliasing later in the course)
- Anti-aliased, area averaged accumulation buffer
- Discrete approximation to a box filter
- Basically, an efficient approach to super sampling
  - For each pixel, build a pixel mask (say an 8x8 grid) to represent all the fragments that intersect with that pixel
  - Determine which polygon fragments are visible in the mask
  - Average colour based on visible area and store result as pixel colour
- Efficient because it uses logical bitwise operators
A-Buffer: Building Pixel Mask

- Build a mask for each polygon fragment that lies below the pixel
- Store 1’s to the right of fragment edge
- Use XOR to combine edges to make mask

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\] \begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
\]

A-Buffer: Building Final Mask

- Once all the masks have been built, must build a composite mask that indicates which portion of each fragment is visible
- Start with an empty mask, add closest fragment to mask
- Traverse fragments in z-order from close to far
- With each fragment, fill areas of the mask that contain the fragment and have not been filled by closer fragments
- Continue until mask is full or all fragments have been used
- Calculate pixel colour from mask:

\[
\text{icolour} = \sum \text{areacolour} \text{tsNumFragmen}
\]

- Can be implemented using efficient bit-wise operations
- Can be used for transparency as well
Illumination

- Coming soon!