Computer Graphics: 8-Hidden Surface Removal

Prof. Dr. Charles A. Wüthrich,
Fakultät Medien, Medieninformatik
Bauhaus-Universität Weimar
caw AT medien.uni-weimar.de
Depth information

- Depth information is an important clue to our visual system
- It allows us to discern which objects are in front, and which ones are behind
- The challenge is to know which are the closest objects to the viewer
- Basically, it is a 2-dimensional sorting problem!
Introduction on hidden surface removal

- Problem: which elements in a picture are not hidden by other ones?

- Two main classes of algorithms
  - Object precision: based on objects
    
    for each object DO
    compute non-hidden parts
    draw them on screen
  END

- Image precision: based on pixels
  
  for each pixel DO
  search object closest to screen
draw corresp. colour
end

- Trivial algorithm: compare all polygons with each other
Back face culling

- Of all the faces of an object, only the ones facing the viewer need to be rendered to the screen.
- This reduces the number of polygons to be rendered by ca. one half.
- The test to perform is easy if the normals to the polygon are available.
- The scalar product $V \cdot N$ must be negative.

- Note that the test can be easily done by computing the $z$ coordinate of $N$ in screen coordinate space.
- Even easier, looking if the polygon vertices lies clockwise produces the same result.
HSR: Painter’s algorithm

• Ever saw a painter compose a picture?
  – He starts painting the background
  – And proceeds to the foremost

• This is how the painter algorithm works:
  – Sort polygons by decreasing $z_{\text{max}}$
  – Draw polygons from maximum $z$ to minimum $z$
Ever saw a painter compose a picture?
- He starts painting the background
- And proceeds to the foremost

This is how the painter algorithm works:
- Sort polygons by decreasing $z_{\text{max}}$
- Draw polygons from maximum $z$ to minimum $z$

Unfortunately, there are cases when the algorithm does not work
HSR: Painter ‘s algorithm

- And which cases are they?
  - Here $P_2$ partially covers $P_1$, but since $z_{max}$ in $P_2$ is bigger than the one on $P_1$, $P_1$ gets drawn over $P_2$

- The Painter algorithm can be modified to work in all cases
HSR: Painter’s algorithm

- Problems occur when their z domains overlap
- One can store $z_{\text{max}}$ and $z_{\text{min}}$ for each polygon and then compare
- If they overlap, then a number of cases allow to draw $P_1$
  1. $x$ axis proj. do not overlap
  2. $y$ axis proj. do not overlap
  3. xy plane projection does not overlap (use bounding rectangles to overlap)
  4. $P_1$ lies on opposite side of $P_2$ plane WRT viewpoint (replace pt. + VP coords. in plane eq.)
  5. $P_2$ lies on same side of $P_1$ plane WRT viewpoint

If one of these occurs, then polygon $P_1 \& P_2$ can be drawn
HSR: Painter’s algorithm

• If none of the cases is true, then $P_1$ and $P_2$ are swapped and tests 4 and 5 are repeated
  – In this case $P_1$ is drawn in front of $P_2$
• There are still some ambiguous cases remaining:
  – If polygons partially overlap, then one of them must be split by using the plane of the other one
  – Cyclic overlappings, these generate infinite loops. Solution here is to remember when a cycle is done and split (by marking polygons)
Z-Buffering (1)

• Z-buffering is easy to combine with the Scanline Algorithm
• Image Space Algorithm
• Idea: For every pixel on the screen, an additional variable is saved containing the depth value at that pixel
• The buffer of the additional variables is called the Z-buffer
• Whenever a polygon has to be drawn, the depth value of the pixel on the polygon is tested against the content of the Z-buffer to see if the new pixel must be drawn or not
Z-Buffering (2)

- **Algorithm:**
  - Write $+\infty$ in every position of the Z-buffer (max. distance from the screen)
  - Compute for each pixel that is being scan-converted its depth at the z axis.
  - If $z < z_{\text{buf}}$ draw pixel and update the Z-buffer with the depth value of the pixel.

- **Note** that the same algorithm works also for any kind of surfaces, as long as the z value of the surface is computable.
Z-Buffering (3)

• But how do I compute the $Z$-values of the polygon?
  - The computation can be done on the fly, while proceeding in the scanline algorithm
  - Remember the plane equation of a polygon:

\[
Ax + By + Cz + D = 0
\]

\[
\Rightarrow z = \frac{-D - Ax - By}{C}
\]
Z-Buffering (4)

- The scanline algorithm draws horizontally lines \((x, x+1, \ldots)\)
- Suppose you know the z value of the polygon at the point \((x,y)\)
  
  \[ z_1 = P(x,y) \]

  Then you have that by moving to the right with an increment of \(\Delta x\) along the x-axis one obtains

  \[ P(x+\Delta x,y) = z_1 - \left(\frac{A}{C}\right) \Delta x \]

  Since the increment is exactly one on the x-axis we obtain

  \[ P(x+1,y) = z_1 - \left(\frac{A}{C}\right) \]

- This is the increment that has to be added for passing from one pixel to the next to its right
Z-Buffering (5)

• Similarly the increment for computing $z$ while passing from the scanline $y$ to the next scanline can be derived:

$$z_1 = P(x,y)$$

By moving downwards with an increment of $\Delta y$ along the $y$-axis one obtains

$$P(x,y+\Delta y) = z_1 - (B/C) \Delta y$$

Since the increment is exactly one on the $y$ axis we obtain

$$P(x,y+1) = z_1 - (B/C)$$

• This is the increment that has to be added for passing from one scanline to the next one vertically

• Obviously, one has to backtrack the scanline until the left edge of the polygon is reached
Z-Buffering (6)

• Algorithmus:

  Initialize Z-Buffer with $\infty$
  For all Polygons $P$
    For each Pixel in $P$ (obtained by scan conversion)
      Compute $Z_{poly} = P(x,y)$
      $Z_{buffer} = \text{read}_z\_\text{buffer}(x,y)$
      if $Z_{poly} < Z_{buffer}$
        $\text{Draw\_Pixel\_to\_Framebuffer}(x,y,\text{color})$
        $\text{Set\_Z\_Buffer}(x,y,Z_{poly})$
      end if
    end for
  end for
Z-Buffering (7)

- Here a scene rendered with z-buffering
- In the lower pictures, the z-buffer values are rendered
  - white=far
  - black=near
A second class of algorithms uses space partitioning to reduce the complexity. Such algorithms use a divide and conquer strategy to solve the problem. The underlying idea is simple:

- Subdivide the projection plane in smaller regions
- Polygons are sorted to their relevant region
- The problem is recursively subdivided until a simple solution can be found
- The smaller the subdivision region, the less polygons have to be handled, and the easier the decision to be made

Given a polygon, and a region, four cases are possible:
Given a region $R$, and a polygon $P$, 4 cases are possible:
Area Subdivision – Warnock (3)

Given a region R, four cases are possible:

1. all polygons lie outside R
   → Draw R with the background colour

2. Only one polygon intersects or is inside R
   → Draw first background color, then draw the polygon

3. A single polygon covers completely R
   → Draw R in the colour of the polygon

4. More than one polygon intersects R, but one of these polygons covers the whole regions and is in front
   → Draw R in the colour of the surrounding polygon
Area Subdivision – Warnock (4)

- How do I test the last condition?
  - Compute all z coordinates of the planes of the relevant polygons for the region R at the corner points of the region
  - If one of the polygon has all z values at the corners in front of the other polygons relevant to the region, then draw this polygon
- If none of these case occur, then subdivide region further
- Until when?
  - Until the region R is as big as a pixel
  - In this case the colour of the pixel will be set to the colour of the polygon that is in front at the middle point of the pixel (by evaluating z at the centre of the pixel for each polygon)
  - Alternatively, one can subdivide at sub-pixel level and do a mean of the values found at subpixel level
Binary Space Partition trees (1)

- BSP trees are efficient algorithms in the case of a moving viewpoint in a static environment
  - For ex. computer games like flight simulators
- The idea: the polygon planes are used to subdivide the region into two subspaces
  - one corresponding to the front
  - one to the back of the polygon
- Subspaces are recursively subdivided until they contain only one polygon
- This achieves a binary tree with single polygons as leaves, and mid nodes splitting planes
- Given a viewpoint V, correct polygon painting can be done by traversing the tree in an in-order fashion, and drawing polygons as encountered.
- This corresponds to implementing that a polygon will be scan-converted correctly if
  - all polygons on the other side of it from the viewer are scan converted
  - then the polygon itself
  - then the ones on the side of the viewer
How to build the BSP tree

- Choose one polygon, consider its plane and sort remaining polygons in
  - back polygons
  - front polygons
- Decide by substituting in equation
- If a polygon belongs to both, split it into 2 subpolygons
- Redo the splitting on the subspaces obtained
- Continue splitting till each subdivision has only one element
- First we pick one arbitrary polygon, e.g. 3
How to build the BSP tree

• repeat process until one polygon only in subspace
How to render from the BSP tree

- Given tree and viewpoint V, it suffices to render the polygons in the correct order
  - If V is in front space of root polygon
    - display first rear polygons
    - display root polygon
    - display front polygons
    - do it recursively for all subspaces till leaves are reached
  - If V in rear space, display in the order front, root, rear
  - If poly is seen on edge, then either order will be okay
  - Note that V coords. can be substituted in plane eq to decide the front rear question
  - This decision has to be taken at EACH node!!!!
How to render from the BSP tree

Front: rear, root, front
Rear: front, root, rear
How to render from the BSP tree

- The advantage of this method is that the tree is traversed in linear time.
- Once tree is built, it is easy to do visibility from a new point.
- Tree needs no recomputing.
- Algo can be modified to deal with non static scenes.
- Backface culling can be done during rendering time, so that it is done on the fly.
  (front-rear test is all I need to decide backfaces.)
### Relative Performance

<table>
<thead>
<tr>
<th>Algo</th>
<th>100</th>
<th>2500</th>
<th>60000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Painter</td>
<td>1</td>
<td>10</td>
<td>507</td>
</tr>
<tr>
<td>Z-Buffer*</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>Warnock</td>
<td>11</td>
<td>64</td>
<td>307</td>
</tr>
</tbody>
</table>


nach Foley, van Dam, Table 15.3, S. 716
End considerations

- Depth Sort: efficient for few polygons
- Z-Buffer: constant performance, but needs additional buffer
- Warnock: efficient for many polygons
- BSP trees, convenient when viewpoint moves and not the scene
End