

Computer Graphics

Lecture 8
Hidden Surface Removal

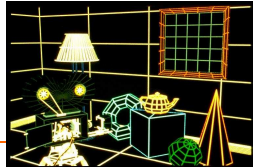
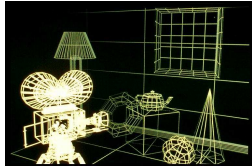
Taku Komura

Only rendering visible surfaces

- We cannot see every surface in scene
- We don't want to waste computational resources rendering primitives which don't contribute to the final image
 - Drawing polygonal faces on screen consumes CPU cycles
 - e.g. Illumination

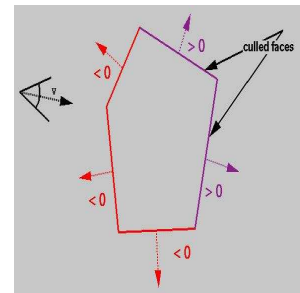
Visibility of primitives

- A scene primitive can be invisible for 3 reasons:
 - Primitive lies outside field of view (last lecture)
 - Primitive is *back-facing*
 - Primitive is occluded by one or more objects nearer the viewer (hidden surface removal)



Back face culling.

- We do not draw polygons facing the other direction
- Test z component of surface normals. If negative – cull, since normal points away from viewer.
- Or if $N \cdot V > 0$ we are viewing the back face so polygon is obscured.



Hidden surface removal algorithms.

Definitions:

- Object space techniques: applied before vertices are mapped to pixels
 - Painter's algorithm, BSP trees, portal culling
- Image space techniques: applied while the vertices are rasterized
 - Z-buffering

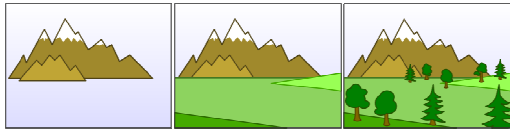
Correct Visibility

- A correct rendering requires correct visibility calculations
- when multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)



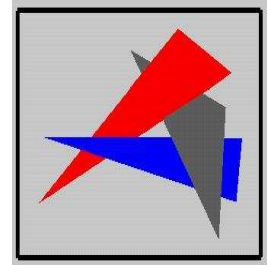
Painters algorithm (object space).

- Draw surfaces in back to front order – nearer polygons “paint” over farther ones.
- Need to decide the order to draw – far objects first



Painters algorithm (object space).

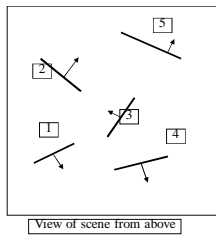
- Key issue is order determination.
- Doesn't always work – see image at right.



BSP (Binary Space Partitioning) Tree.

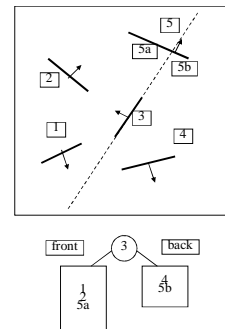
•One of class of “list-priority” algorithms – returns ordered list of polygon fragments for specified view point (static pre-processing stage).

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.



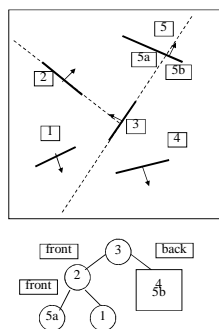
BSP Tree.

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.



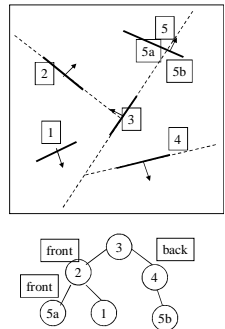
BSP Tree.

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.



BSP Tree.

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.



Displaying a BSP tree.

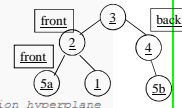
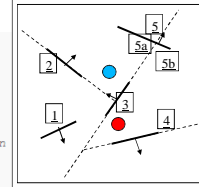
- Once we have the regions – need priority list
- BSP tree can be traversed to yield a correct priority list for an arbitrary viewpoint.
- Start at root polygon.
 - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
 - If viewer is in back half-space, draw polygons in front of root first, then the root polygon, then polygons behind.
 - If polygon is on edge – either can be used.
 - Recursively descend the tree.
- If eye is in rear half-space for a polygon can back face cull.
- Always drawing the opposite side of the viewer first

In what order will the faces be drawn?

```

traverse_tree(bsp_tree* tree, point eye)
{
    location = tree->find_location(eye);
    if (tree->empty())
        return;
    if (location > 0) // if eye in front of location
    {
        traverse_tree(tree->back, eye);
        display(tree->polygon_list);
        traverse_tree(tree->front, eye);
    }
    else if (location < 0) // eye behind location
    {
        traverse_tree(tree->front, eye);
        display(tree->polygon_list);
        traverse_tree(tree->back, eye);
    }
    else // eye coincidental with partition hyperplane
    {
        traverse_tree(tree->front, eye);
        traverse_tree(tree->back, eye);
    }
}

```



BSP Tree.

- A lot of computation required at start.
 - Try to split polygons along good dividing plane
 - Intersecting polygon splitting may be costly
 - Cheap to check visibility once tree is set up.
 - Can be used to generate correct visibility for arbitrary views.
- ⇒ Efficient when objects don't change very often in the scene.

BSP performance measure

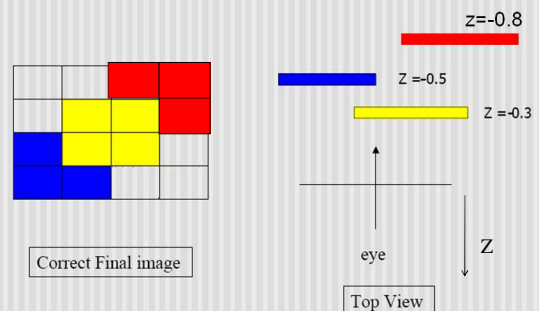
- Tree construction and traversal (object-space ordering algorithm – good for relatively few static primitives, precise)
- Front-to-back traversal is more efficient
 - Record which region has been filled in already
 - Terminate when all regions of the screen is filled in
- S. Chen and D. Gordon. "Front-to-Back Display of BSP Trees." IEEE Computer Graphics & Algorithms, pp 79–85. September 1991.

Z-buffering : (image space)

Basic Z-buffer idea:

- rasterize every input polygon
- For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)
- Track depth values of closest polygon (largest z) so far
- Paint the pixel with the color of the polygon whose z value is the closest to the eye.

Z buffer example



Z buffer example

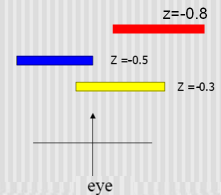
Step 1: Initialize the depth buffer

-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the program draws blue polygon first – the order does not affect the final result any way).

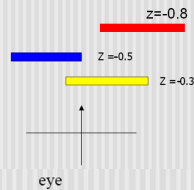
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0



Z buffer example

Step 3: Draw the yellow polygon

-1.0	-1.0	-1.0	-1.0
-1.0	-0.3	-0.3	-1.0
-0.3	-0.3	-0.3	-1.0
-0.3	-0.3	-1.0	-1.0

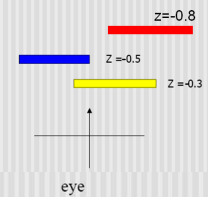


If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated

Z buffer example

Step 4: Draw the red polygon

-1.0	-1.0	-0.8	-0.8
-1.0	-0.3	-0.3	-0.8
-0.3	-0.3	-0.3	-1.0
-0.3	-0.3	-1.0	-1.0



If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated
z-buffer drawback: wastes resources by rendering a face and then drawing over it

Implementation.

- Initialise frame buffer to background colour.
- Initialise depth buffer to $z = \text{min value for far clipping plane}$
- For each triangle
 - Calculate value for z for each pixel inside
 - Update both frame and depth buffer

Why is Z-buffering so popular ?

Advantage

- Simple to implement in hardware.
 - Memory for z-buffer is now not expensive
- Diversity of primitives – not just polygons.
- Unlimited scene complexity
- No need to sort the objects
- Don't need to calculate object-object intersections.

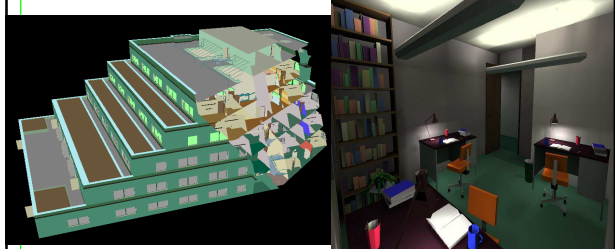
Disadvantage

- Waste time drawing hidden objects
- Z-precision errors
- May have to use point sampling

Z-buffer performance

- Brute-force image-space algorithm – easy to implement and is very general.
- Memory overhead: $O(1)$
- Time to resolve visibility to screen precision: $O(n)$
 - n : number of polygons
 - Need to be combined with other culling methods to reduce complexity

Ex. Architectural scenes



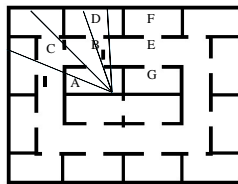
Portal Culling

Model scene as a graph:

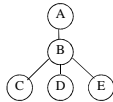
- Nodes: Cells (or rooms)
- Edges: Portals (or doors)

Graph gives us:

- Potentially visible set



1. Render the room
2. If portal to the next room is visible, render the connected room in the portal region
3. Repeat the process along the scene graph



Summary

- Z-buffer is easy to implement on hardware and is an important tool
- We need to combine it with an object-based method especially when there are too many polygons
 - portal culling, frustum culling

References for hidden surface removal

- Foley et al. Chapter 15, all of it.
- Introductory text, Chapter 13, all of it
- Or equivalents in other texts, look out for:
 - (as well as the topics covered today)
 - Depth sort – Newell, Newell & Sancha
 - Scan-line algorithms