CSE 413
(Computer Graphics)
Hidden Surface Removal

Iyolita Islam

Department of Computer Science and Engineering
Military Institute of Science and Technology

Last Updated: October 5, 2020

Unintentional Mistakes

Best efforts have been exercised in order to keep the slides error-free, the preparer does not assume any responsibility for any unintentional mistakes. The text books must be consulted by the user to check veracity of the information presented.

Outline

1. Hidden Surface Removal
2. Back-Face Culling
3. Painter's Algorithm
4. z-buffer Alogorithm
5. Scan-Line Algorithm

Hidden Surface Removal

- For a set of 3D objects and a viewing specification, determining which lines or surfaces are visible is called visible surface/ line determination, or hidden surface/ line removal.
- So, the surface(s) that are blocked or hidden from the view point must be "removed" in order to construct a realistic view of the 3D objects.
- There are two general approaches: (a) Image Precision (b) Object Precision

Why might a polygon be invisible?

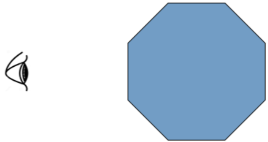
1. Polygon outside the field of view
2. Polygon is backfacing
3. Polygon is occluded by object(s) nearer the viewpoint

Hidden Surface Removal Algorithms

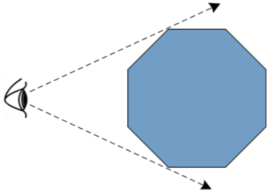
- **Conservative visibility testing**: only trivial reject – does not give final answer!
  i.e.: back-face culling, canonical view volume clipping, spatial subdivision
- **Image Precision**: For each pixel, examine all n objects, find which is the closest.
  $O(np)$ where, p = number of pixels.
  i.e.: ray tracing, or Z-buffer and scan-line algo
- **Object Precision**: For each object, examine all n objects.
  $O(n^2)$
  i.e.: 3-D depth sort, BSP trees

Back-Face Culling

- See an object from an viewpoint.

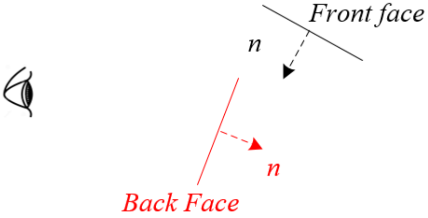


- Viewpoint can view an object from only one side.



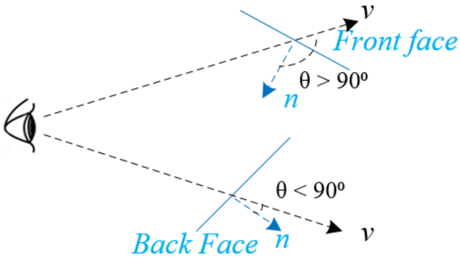
Back-Face Culling

- Front face: If the normal to a face is pointing towards the viewpoint.
- Back face: If the normal to a face is pointing away from the viewpoint.



Back-Face Culling (for an object)

- Determine for each face, if it's front or back.



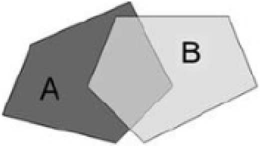
- $v.n = |v||n|cos\theta$
- if front face, then $\theta > 90\,\text{deg}, cos\theta < 0$ and $v.n < 0$
  if back face, then $\theta < 90\,\text{deg}, cos\theta > 0$ and $v.n > 0$

Back-Face Culling Algorithm

Given a viewpoint P and a set of polygons

```
for all polygons in the real world do
    calculate the normal vector n for current polygon
    calculate the centre C of the current polygon
    calculate the viewing vector v = C - P
    if v . n < 0 then
        render current polygon
    endif
end for
```

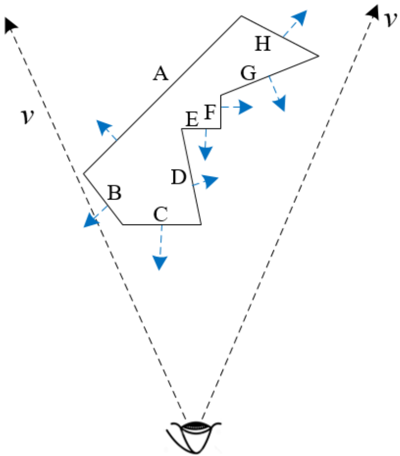
- On the surface of a closed manifold, polygons whose normals are pointing away from the camera are always occluded.
- One polygon can obscure another.



Back-Face Culling Algorithm - Example

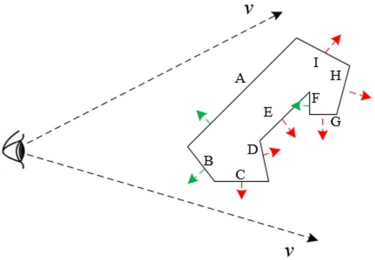
Determine the front and back faces for the given object from the viewpoint.

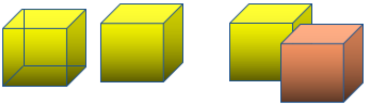
- calculate normal n for all the faces of the polygon.
- calculate view direction vector v.
- if $v.n < 0$, then show the current face.
  otherwise, ignore the current face.
- Front face: B, C, F, G
  Back face: A, D, E, H



Back-Face Culling Algorithm

- Advantages:
  - On average, approximately one-half of a polyhedron's polygons are back-facing.
  - Back-Face culling halves the number of polygons to be considered for each pixel in an image precision algorithm.
- Disadvantages:
  - On the surface of a closed manifold, polygons whose normals point away from the camera are always occluded.
  - One polyhedron may obscure another



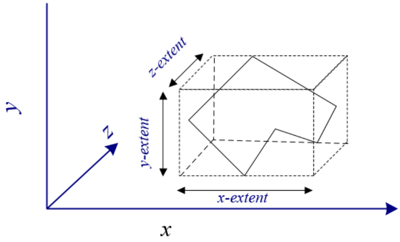


Painter's Algorithm

- A depth sorting algorithm.
- Surfaces are sorted according to the decreasing depth.
- The polygons are processed as if they are painted on the view plane in the order of their distance from the viewer.
- So, the polygons at the most distance are painted first.
- To implement this concept, a priority order is to be determined.
- For most scenes, some polygons will overlap.
- To render the correct image, we need to determine which polygons occlude which.

Some basics

- Priority assigning: By determining if a polygon P obscures any other. If the answer is NO, then the polygon should be painted at first. So, we need to determine if polygon P is obscure polygon Q.
- z-extent of a polygon: The region between $z = z_{min}$ to $z = z_{max}$. Here, z_{min} is the smallest of the z-coordinates of all the vertices of the polygon. and, z_{max} is the largest.
  similarly, x-extent and y-extent.
- Extent/ bounding box of the polygon: The intersection of the x, y, and z extents.



Painter's Algorithm

For each two polygons:

- Examine Z-extents.
- Check if polygon P obscure polygon Q (For any YES of the following tests (applied in sequential order), P can be drawn before Q):
  1. Are the X-extents of P and Q are disjoint?
  2. Are the Y-extents of P and Q are disjoint?
  3. Is P entirely on the opposite side of Q's plane from the eye?
  4. Is Q entirely on the same side of P's plane as the eye?
  5. Are the projections of P and Q on the screen are disjoint?
- If all answers are NO, we assume that P actually obscure Q.
- Therefore test whether Q can be scan converted before P.
  - Is Q entirely on the opposite side of P's plane from the eye?
  - Is P entirely on the same side of Q's plane as the eye?
- If the answers are NO, then split.

Painter's Algorithm

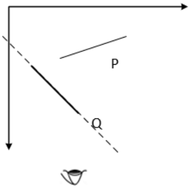


Figure: Question 03

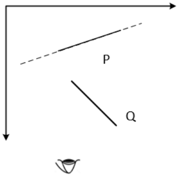
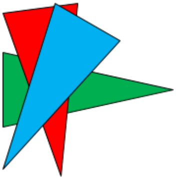
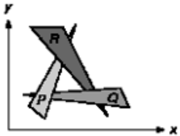


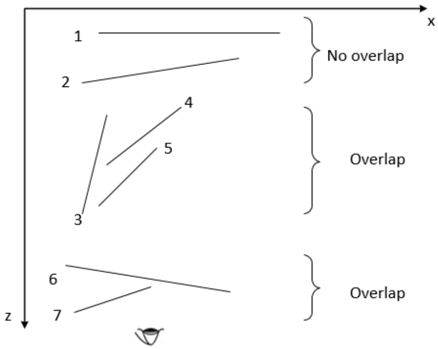
Figure: Question 04

- Order- Green, Red, Blue.
- But intersecting polygons can make a cycle with no valid visibly order.





Painters Algorithm - Example



(For practice, we will perform only 1, 3 and 4 no tests.)

Painters Algorithm

- Advantages:
  - Fast enough for simple scenes
  - Fairly intuitive
- Disadvantages:
  - Slow for even moderately complex scenes
  - Hard to implement and debug
  - Lots of special cases

z-buffer Alogorithm

- Requires two buffers:
- Intensity Buffer
    - our familiar RGB pixel buffer
    - initialized to background color
- Depth ("Z") Buffer
    - depth of scene at each pixel
    - initialized to far depth
- Polygons are scan-converted in arbitrary order. When pixels overlap, use Z-buffer to decide which polygon "gets" that pixel

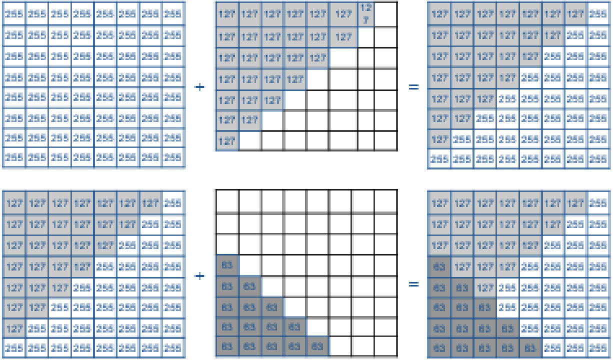
z-buffer Alogorithm

```
Initialize:
    Each z-buffer cell = Max z value
    Each frame buffer cell = background color
for each polygon
    Compute z(x, y) as polygon depth at the pixel (x, y)
    if z(x, y) < z-buffer value at pixel (x, y)
        z-buffer(x, y) = z(x, y)
        pixel(x, y) = color of polygon at (x, y)
    end if
end for
```

z-buffer Alogorithm

- Initially, z-buffer $\leftarrow \infty(255)$
  color-buffer $\leftarrow$ BLACK
- Update z-buffer for all (x, y) points with z-values of all points of an object.
  Update the color-buffer for all (x, y) points with the color of that object.
- For the next object, for those (x, y) points z-buffer and color-buffer will be updated where the z-value $<$ current value of z-buffer.

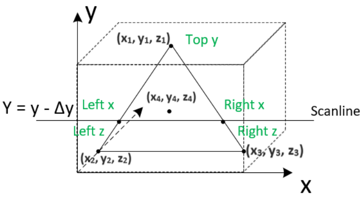
z-buffer Algorithm



Above: example using integer Z-buffer with near = 0, far = 255

How to compute z-values of an object efficiently?

- Start from Top y and decrement Δy.
  Y = y - Δy
- Calculate X:
  LeftX $\Rightarrow \frac{x-x_1}{x_1-x_2} = \frac{Y-y_1}{y_1-y_2}$
  RightX $\Rightarrow \frac{x-x_1}{x_1-x_3} = \frac{Y-y_1}{y_1-y_3}$
- Calculate Z:
  LeftZ $\Rightarrow \frac{LeftZ-z_1}{z_1-z_2} = \frac{Y-y_1}{y_1-y_2}$ RightZ $\Rightarrow \frac{RightZ-z_1}{z_1-z_3} = \frac{Y-y_1}{y_1-y_3}$



- Incrementing value of X, calculate Z for all points on scanline:
  $Z_p = \frac{Z_p-LeftZ}{LeftZ-RightZ} = \frac{X_p-LeftX}{LeftX-RightX}$

z-buffer Algorithm

- Advantages:
  - Simplicity lends itself well to hardware implementations: used by all graphics cards
  - Polygons do not have to be compared in any particular order: no presorting in z necessary, big gain!
  - Only consider one polygon at a time
  - Z-buffer can be stored with an image; allows to correctly composite multiple images without having to merge models
  - Can be used for non-polygonal surfaces
- Disadvantages:
  - A pixel may be drawn many times
  - High amount of memory required
  - Lower precision for higher depth

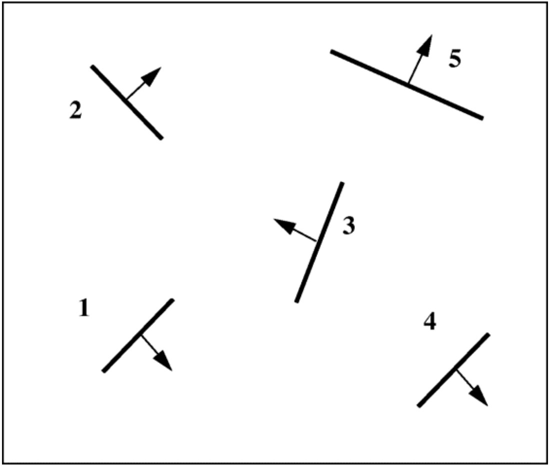
Binary Space Partioning (BSP) Tree

- Provides spatial subdivision and draw order
- Split space with any line (2D) or plane (3D)
- Divide and conquer: to display any polygon correctly:
    1. display all polygons on "far" (relative to viewpoint) side of polygon,
    2. then that polygon,
    3. then all polygons on polygon's "near" side.
- Always need to check the view point with respect to current node.
- If viewpoint is infront of the root, then at first back of the root, then root and then front of the root.
- If viewpoint is backward of the root, then at first front of the root, then root and then back of the root.

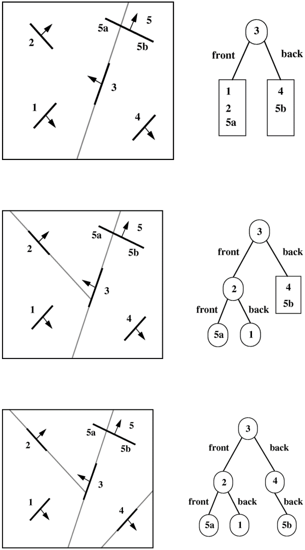
Binary Space Partioning (BSP) Tree

- Trades off view-independent preprocessing step (extra time and space) for low run-time overhead each time view changes
- Perform view-independent step once each time scene changes:
  - recursively subdivide environment into a hierarchy of half-spaces by dividing polygons in a half-space by the plane of a selected polygon
  - build a BSP tree representing this hierarchy
  - each selected polygon is the root of a sub-tree

Binary Space Partioning (BSP) Tree - Example

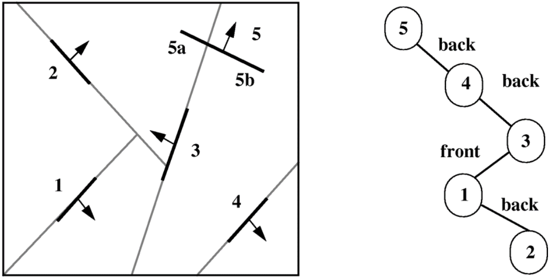


- Step-1: Choose any polygon (e.g., polygon 3) and subdivide others by its plane, splitting polygons when necessary.
- Step-2: Process front sub-tree recursively
- Step-3: Process back sub-tree recursively



Binary Space Partioning (BSP) Tree - Example

(Alternative Solution)



Binary Space Partioning (BSP) Tree

- Each face has form $Ax + By + Cz + D$
- Plug in coordinates and determine:
  - Positive: front side
  - Zero: on plane
  - Negative: back side
- Back-to-front: postorder traversal, farther child first
- Front-to-back: inorder traversal, near child first
- Do backface culling with same sign test
- Clip against visible portion of space (portals)

Scan-Line Algorithm

- Image precision algorithm
- Renders scene scan line by scan line
- Maintain various lists
    - Edge Table
    - Active Edge Table
    - Polygon Table
    - Active Polygon Table

Scan-Line Algorithm Basics

- Edge Table (ET) (for non-horizontal edge): Sorted into buckets based on each edge's smaller y-coordinate
  Within buckets are ordered by increasing x-coordinate of their lower end point.
- Polygon Table (PT): List of the polygons
- Active Edge Table (AET): Stores the list of edges intersecting current scanline in increasing order of current x-coordinate
- Active Polygon Table (APT): At each x-scan value this table contains the list of polygons whose in-out flag is set to true
- Naming of the edges:: For two points of an edge, point with minimum y value will come first.
  for same y values, point with minimum x will come first.

Scan-Line Algorithm Basics

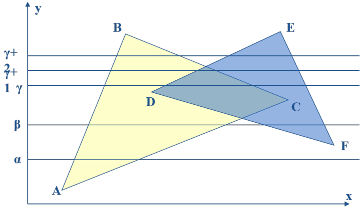


Table: AET Contents

Scan Line	Entries			
α	AB	AC		
β	AB	AC	FD	FE
$\gamma, \gamma + 1$	AB	DE	CB	FE
$\gamma + 2$	AB	CB	DE	FE

End of Slides