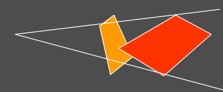
Computer Graphics

Hidden Surface Removal

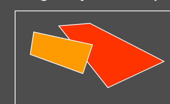
Hidden Surface Removal

Hidden Surface Removal for Polygonal Scenes

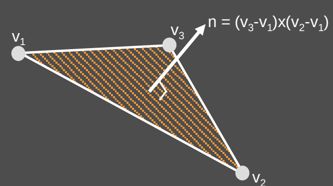
- Input: Set of polygons in three-dimensional space + a viewpoint



- **Output:** A two-dimensional image of projected polygons, containing only visible portions



The Normal Vector



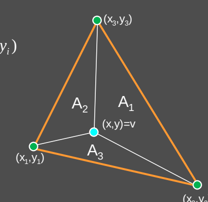
n(1,2,3) = n(2,3,1) = -n(2,1,3)

Barycentric Coordinates

$$(x,y)=\sum_{i=1}^{3}\alpha_i\cdot(x_i,y_i)$$

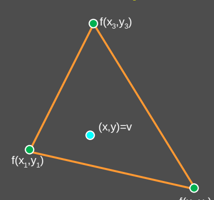
$$\alpha_i = A_i / \sum_{i=1}^{3} A_i$$

$$\sum_{i=1}^{3}\alpha_i = 1$$



Barycentric coordinates of v = $(\alpha_1, \alpha_2, \alpha_3)$
B.C. are unique.
B.C. of all interior points are ≥ 0.
Triangle centroid = (1/3,1/3,1/3).

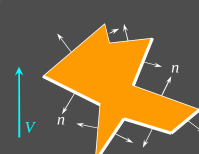
Linear Interpolation



$$(x,y)=\sum_{i=1}^{3}\alpha_i\cdot(x_i,y_i)$$

$$f(x,y)=\sum_{i=1}^{3}\alpha_i\cdot f(x_i,y_i)$$

Back Face Culling (object space)



- In closed polyhedron you don't see object "back" faces
- Assumption
  - Normals of faces point *out* from the object
- Object space algorithm

Computer Graphics

Hidden Surface Removal

Back Face Culling

- Determine back & front faces using sign of inner product $< n, V >$

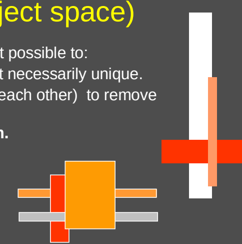
$$\langle n, v\rangle = n_x v_x + n_y v_y + n_z v_z = \|n\| \cdot \|v\| \cos\theta$$

- In a convex object :
  - Invisible back faces
  - All front faces entirely visible ⇒ solves hidden surfaces problem
- In non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible

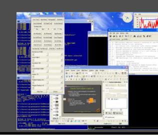
7

Depth Sort (object space)

- **Question:** Given a set of polygons, is it possible to:
  - sort them by depth. The order is not necessarily unique.
  - then paint them back to front (over each other) to remove the hidden surfaces ?
  - This is called the **painter algorithm.**



- **Answer:** Usually not
- Works for special cases
  - E.g. polygons with constant z (where do we have polygons with constant z!?)



8

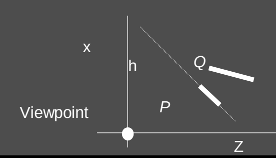
Depth Sort (object space)

- Will fail for:
  - Intersecting polygons
  - Mutually occluding polygons

9

Plane containing *P*

- Since every polygon is planar, we can speak about the plane *h* of a polygon *P.*
- *Observation:* If polygon *Q* does not intersect h, then
  - If Q is in the side of *h* containing the viewpoint, then during the painter algorithm, we can draw *P* before drawing *Q*
  - Otherwise P can be drawn before Q



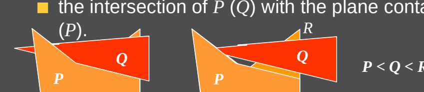
Depth Sort by Splitting

- Given two polygons, P and Q, we can order them in z if:
  1. P and Q do not overlap in their x extents
  2. Or P and Q do not overlap in their y extents
  3. Or P is totally on one side of Q's plane
  4. Or Q is totally on one side of P's plane
  5. Or P and Q do not intersect in projection plane
- Can we always resolve the relation between P and Q using steps 1-5?

11

Depth Sort by Splitting

- What steps 1-5 all fail ?
- Split P (Q) along:
  - the intersection with Q (resp P) into two smaller polygons – (how could one compute this intersection!?)
  - the intersection of P (Q) with the plane containing Q (P).



- Object space algorithm

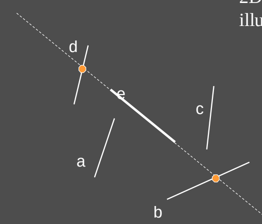
12

BSP - trees

- Construct a tree that gives a rendering order
- Tree recursively splits 3D world into cells, each of which contain at most one piece of polygon.
- Constructing tree:
  - choose polygon (arbitrary)
  - split its cell using plane on which polygon lies
  - continue until each cell contains only one polygon

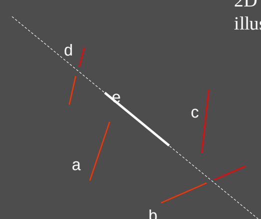
BSP - trees

2D version for illustration



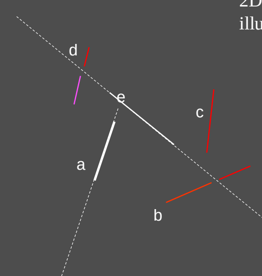
BSP - trees

2D version for illustration



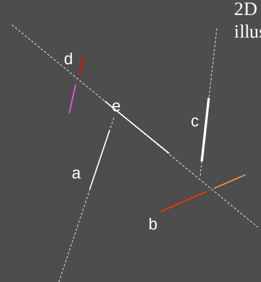
BSP - trees

2D version for illustration



BSP - trees

2D version for illustration

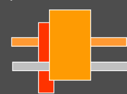


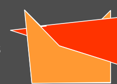
BSP - trees

- Rendering tree:
  - recursive descent
  - render back, node polygon, front
    - back/front is determined by what side of the plane the camera is on
- Disadvantages:
  - many small pieces of polygon (more splits than depth sort!)
  - over rendering (does not work well for complex scenes with lots of depth overlap)
- Advantages:
  - one tree works for all focal points (good for cases when scene is static)
  - filter anti-aliasing works fine, as does transparency
  - data structure is worth knowing about
- Comment
  - expensive to get approximately optimal tree, but for many applications this can be "off-line" in a pre-processing step.

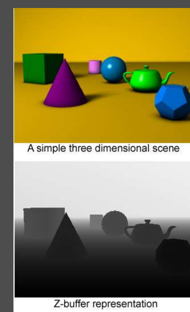
Z-Buffer Algorithm (image space)

- **Basic Idea: resolve the visibility at the pixel level, using depth sort.**
- For each image pixel - store both the color and the current *z* depth
- Instead of always painting the pixels while scan-converting a polygon, do so only if polygon's depth is less than current *z* depth at that pixel

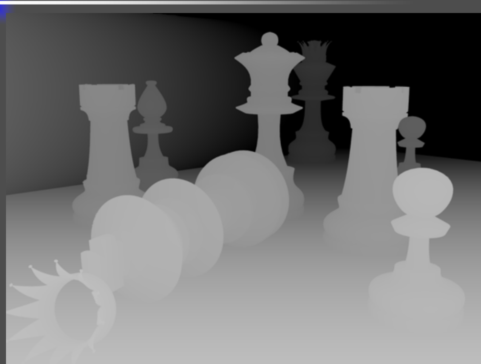




19



20

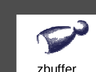


21

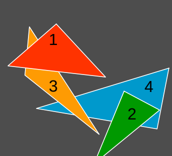
Z-Buffer

```
ZBuffer(Scene)
For every pixel (x,y) do PutZ(x,y,MaxZ);
For each polygon P in Scene do
   Q := Project(P);
   For each pixel (x,y) in Q do
      z := Depth(Q,x,y);
      if (z < GetZ(x,y)) then
         PutZ(x,y,z);
         PutColor(x,y,Col(P));
      end;
   end;
end;
```

Questions: How can one compute **Project**(P) and **Depth**(Q,x,y)?

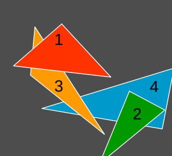


22

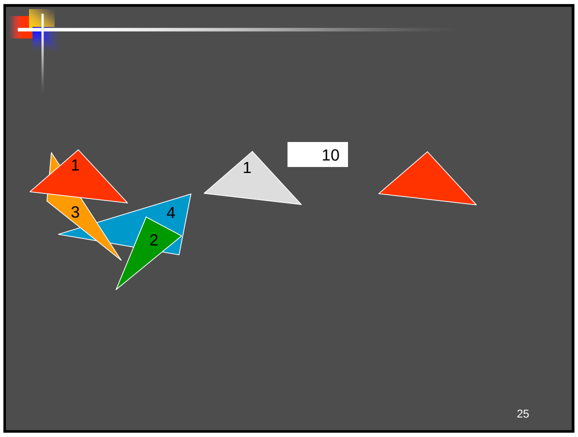


23

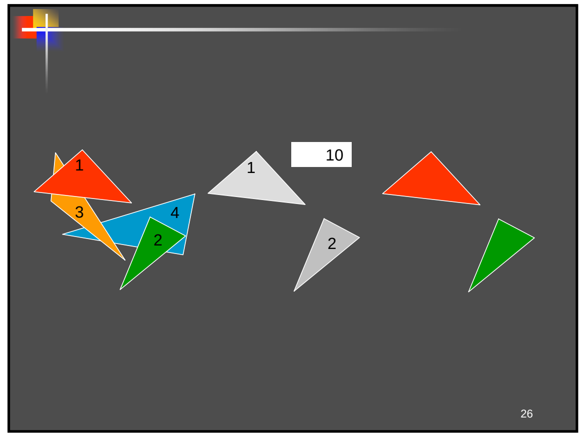
10



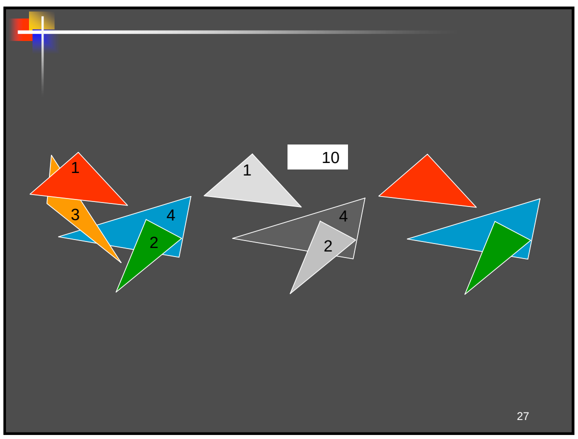
24



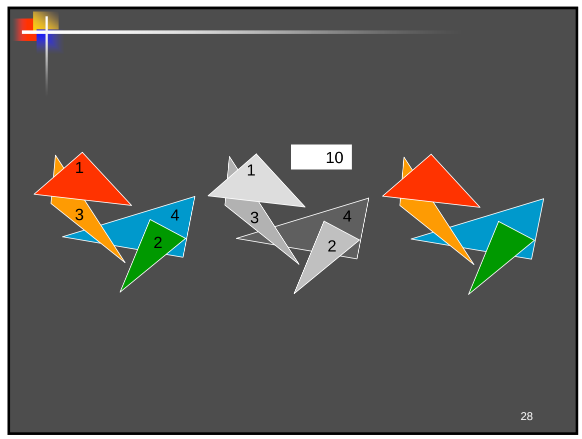
1
3
4
2
1
10
25



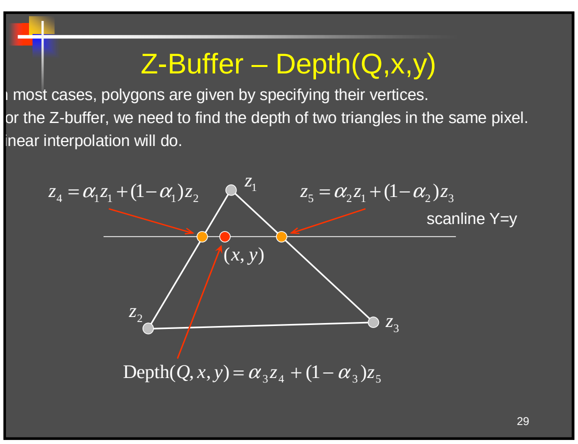
1
3
4
2
1
10
2
26



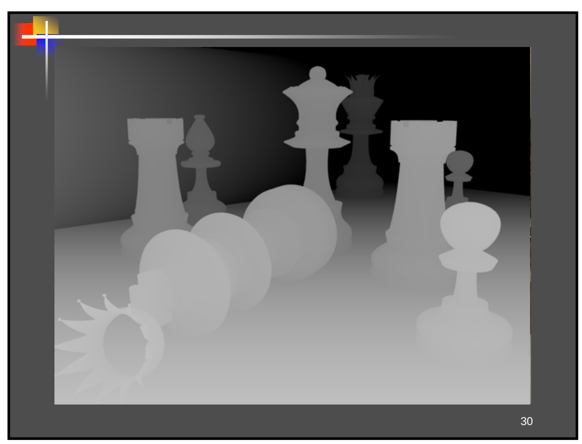
1
3
4
2
1
10
4
2
27



1
3
4
2
1
10
3
4
2
28



Z-Buffer – Depth(Q,x,y)
most cases, polygons are given by specifying their vertices.
or the Z-buffer, we need to find the depth of two triangles in the same pixel.
inear interpolation will do.
$z_4 = \alpha_1 z_1 + (1-\alpha_1) z_2$
z_1
$z_5 = \alpha_2 z_1 + (1-\alpha_2) z_3$
scanline Y=y
(x, y)
z_2
z_3
$\text{Depth}(Q, x, y) = \alpha_3 z_4 + (1-\alpha_3) z_5$
29



30

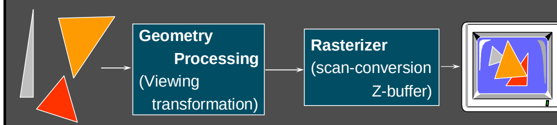
Z-Buffer Algorithm

- Image space algorithm
- Data structure: Array of depth values
- Common in hardware due to simplicity
- Depth resolution of 32 bits is common
- ***Scene may be updated on the fly, adding new polygons***

31

The Graphics Pipeline

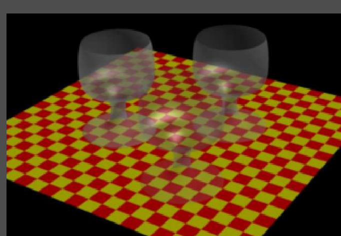
- Hardware implementation of screen Z-buffer:
  - Polygons sent through pipeline one at a time
  - Display updated to reflect each new polygon



32

Transparency Z-Buffer

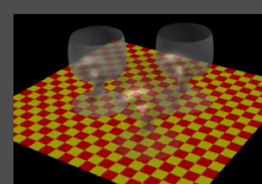
How can we emulate transparent objects?



33

Transparency Buffer

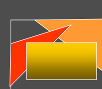
- Extension to the basic Z-buffer algorithm
- Save *all* pixel values
- At the end – have list of polygons & depths (order) for each pixel
- Simulate transparency by weighting the different list elements, in order



34

The A - buffer

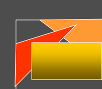
- For transparent surfaces and filter based anti-aliasing:
- Algorithm (1): filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don't remove farther polygons



pixel

The A - buffer

- Algorithm (2): rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used for either for calculations involving transparency or for filtering for aliasing

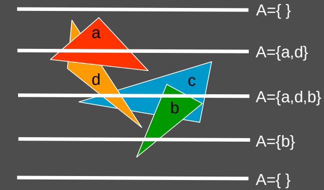


pixel

Scan-Line Z-Buffer Algorithm

- In software implementations - amount of memory required for screen Z-buffer may be prohibitive
- Scan-line Z-buffer algorithm:
  - Render the image one line at a time
  - Take into account only polygons affecting this line
- Combination of polygon scan-conversion & Z-buffer algorithms
- Only Z-buffer the size of scan-line is required.
- Entire scene must be available a-priori
- Image cannot be updated incrementally

37



38

Scan-Line Z-Buffer Algorithm

```
ScanLineZBuffer(Scene)
Scene2D := Project(Scene);
Sort Scene2D into buckets of polygons P in  increasing YMin(P) order;
A := EmptySet;
for y := YMin(Scene2D) to YMax(Scene2D) do
    for each pixel (x, y) in scanline Y=y do  PutZ(x, MaxZ);
    A := A + {P in Scene : YMin(P)<=y};
    A := A - {P in A : YMax(P)<y};
    for each polygon P in A
       for each pixel (x, y) in P's spans on the scanline
           z := Depth(P, x, y);
           if (z<GetZ(x)) then
              PutZ(x, z);
              PutColor(x, y, Col(P));
          end;
       end;
    end;
end;
```