CS770/870 Fall 2017
Visible Surface Algorithms

Related material in
Cunningham (superficial): p. 1.13, 1.19
Hill and Kelley: 8.3-8.4
Angel 5e: 7.8-7.11
Angel 6e: 6.11

Preview

- Image space algorithms
- Object space algorithms
- Coherence
- Optimization options
- Backface culling
- Depth buffer algorithm

Evaluation Issues

- There are a few key issues to consider regarding all visible surface algorithms
  - Is algorithm general purpose?
  - What is the space/time performance?
  - How easy is it to incorporate improved realism?
  - Is it appropriate for the particular task

Image Precision v. Object Precision

- Image precision algorithms

  Work in *image space* (in floating point)

```
for each pixel
    find closest object that projects onto pixel
    color the pixel with color of object
```

  Computations done at the precision of the image

- Object precision algorithms

```
for each object in the world
    find the parts not obscured by other objects
    draw these parts where they belong on display
```

  Computations done at the precision of the object space

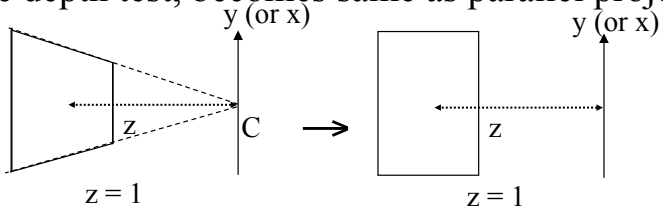
  have to deal with a pixel partially covered by the object

Coherence

- Different algorithms take advantage of 1 or more forms of *coherence* to improve efficiency
  - *object coherence*: objects tend to be compact and separated from other objects
  - *face coherence*: surface properties vary smoothly across object faces
  - *edge coherence*: edge visibility can only change if the edge crosses a visible edge or intersects a visible face
  - *implied edge coherence*: the intersection of two planar faces is defined by a simple edge with 2 end points
  - *scanline coherence*: neighboring scanlines have similar visibility

Comparing Depths

- Determining visibility relies on comparing the "depths" of 2 objects with respect to the display
  - Transform world so display screen located in xy plane
  - z-values can then be used to determine closeness to display
  - *parallel* projections: all points that project to pixel (x,y), have same x,y values
  - *perspective* projections: if apply *perspective transformation* before depth test, becomes same as parallel projection



Bounding Areas and Volumes

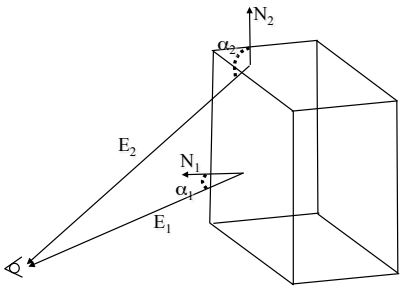
- Lots of optimizations can be achieved using *bounding boxes* in either 2D screen or 3D world. E.g.,
  - If bounding boxes of the projections of 2 objects don't overlap in the images, the objects cannot obscure each other
  - If bounding boxes of objects in 3D don't overlap, can simplify many calculations
  - *minmax* tests compare bounds of 1 dimension of the box
- Other bounding volumes
  - spheres
  - cylinders
  - parallel planes

Self-Obscuring Tests

- Objects in the scene are composed of faces that are obscured by other (opaque) faces
  - on average, half the faces are obscured by other faces
  - these are called backfaces
- Compute the normal to the polygon that points outside the volume, the *outward normal*, N
- Create a vector from the polygon to the eye point, E
- If the angle between N and E is > 90°, the viewer is looking at the "back" of the face, so it is invisible
- This is called *backface culling*

Backface Culling

- Compute the outward normal
  - Ordering of vertices determines direction of normal
- Create vector from any point on polygon to eye
- A back face normal makes an angle > 90° with eye vector
- The computation can be done either in world coordinates or viewing coordinates



$\alpha_1 = E_1 \bullet N_1 < 90°$ front face

$\alpha_2 = E_2 \bullet N_2 > 90°$ backface

Can only do this if object is composed of opaque convex parts!

Backface Culling Details

- Don't need to compute the angle, just the cosine
  - $\alpha > 90°$ iff $\cos \alpha < 0$
- Don't need to compute cosine either
  - $E \bullet N = ||E||\ ||N|| \cos \alpha$
  - So, $\cos \alpha < 0$ iff $E \bullet N < 0$
- Can apply the same logic to a parallel projection
  - the eye is at ∞ along the *direction of projection*
  - so face is a backface if **dop** • **N** < 0.
  - if dop = (0 0 1), **dop** • **N** = $0*N_x + 0*N_y + 1*N_z = N_z$
  - also true if apply test after perspective transformation

Plane Normal from Planar Vertices

- Compute cross product of any 3 non-collinear vertices:
  Pick any vertex, define vectors from that vertex to neighbors
  Let P = (V_2-V_1) and Q = (V_0-V_1), then

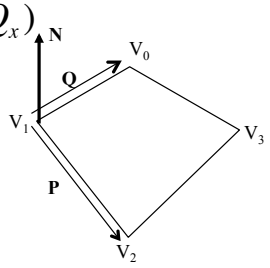
Vertex order important:
CCW from outside

$$\mathbf{N} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ P_x & P_y & P_z \\ Q_x & Q_y & Q_z \end{vmatrix} = \mathbf{i}\begin{vmatrix} P_y & P_z \\ Q_y & Q_z \end{vmatrix} - \mathbf{j}\begin{vmatrix} P_x & P_z \\ Q_x & Q_z \end{vmatrix} + \mathbf{k}\begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix}$$

$$= \mathbf{i}(P_yQ_z - P_zQ_y) - \mathbf{j}(P_xQ_z - P_zQ_x) + \mathbf{k}(P_xQ_y - P_yQ_x)$$

$$= (P_yQ_z - P_zQ_y,\ -P_xQ_z + P_zQ_x,\ P_xQ_y - P_yQ_x\ )$$

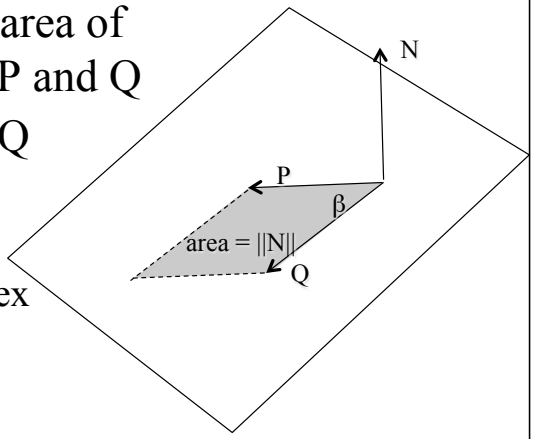
$$= (P_yQ_z - P_zQ_y,\ P_zQ_x - P_xQ_z,\ P_xQ_y - P_yQ_x\ )$$



- **N** is normal to both P and Q and PQN is rhs

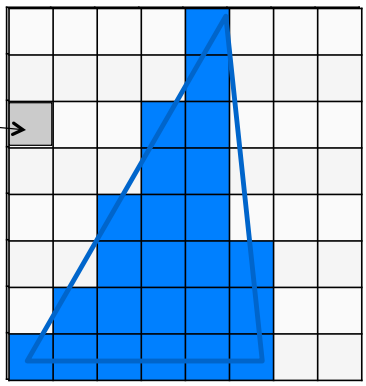
Cross Product Details

- How do we make sure N is an *outward* normal?
  - Order vertices of the polygon so that:
    as viewed from outside object,
    vertices are in counter-clockwise order
- || **N** || = (||P|| ||Q|| sin β) is the area of the parallelogram formed by P and Q
- β is the angle between P and Q
- Collinear vertices test?
  - Could just test ||N||
  - if close to 0, move to next vertex
  - stop when loop around



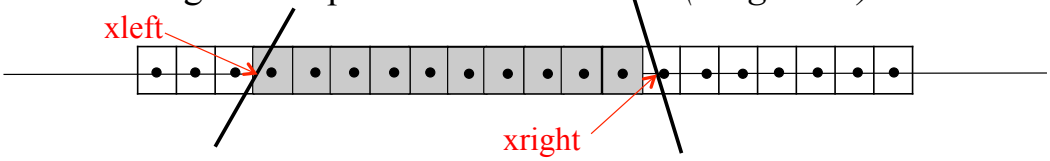
Triangle Scan Conversion

- Depth buffer visibility done during *scan conversion*:
  - 3D Polygons are converted to 3D (planar) triangles
  - Triangle vertices are mapped to 3D projection space
  - Scan conversion: which pixels are in each triangle
- What *is* a pixel in this context?
  - It's an *area* whose "address" is at the center of the area
- When should a pixel be filled?
  - If its *center* is *inside* the triangle
  - Or if it is *ON* a *left* edge of the triangle



Scan Line Details

- A *span* is the portion of a scan line defined by 2 successive x-intersection values, *xleft* and *xright*
- To obey the "fill rule"
  - the left most pixel to be filled is *ceil(xleft)*
  - the right most pixel to be filled is *ceil(xright - 1)*



  - Note: *trunc(xright)* doesn't work: it would fill if the edge intersects the center of the pixel, which is against the rule!
  - If left edge intersects the pixel center, *ceil(xleft)* will be at the center and that pixel will be filled -- as it should be.

Depth Buffer
Visible Surface Algorithm

- Brute force image precision algorithm
  - Polygon *scan conversion* determines all pixels that are "inside" the bounds of a polygon and assigns the polygon color to them
    - Uses a 2D *color* array that is *xResolution* by *yResolution*; *color[y,x]* stores the color to be assigned to the pixel at (x,y).
  - To add visibility to this algorithm
    - Define a 2D *depth* buffer that is the same size as the *color* buffer; *depth[y,x]* stores the z-value of the last polygon whose color is in *pixel[y,x]*.

```
writePixel(x,y,z,color)
{
  if ( z<depth[y,x] )
    depth[y,x] = z
    color[y,x] = color
}
```

- This is a standard feature on modern graphics cards
  - requires 3D coordinates and perspective transformation

Edge *x* intersections with scanlines

- How do we get *xleft* and *xright* for each edge for each scan line (y_s)?
  - From 3D parametric line equations for <u>each</u> edge

$$x(t) = x_1 + (x_2-x_1)*t = x_1 + d_x*t$$
$$y(t) = y_1 + (y_2-y_1)*t = y_1 + d_y*t$$
$$z(t) = z_1 + (z_2-z_1)*t = z_1 + d_z*t$$

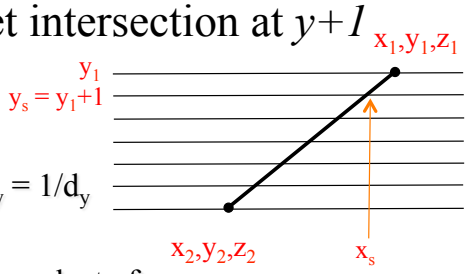
  - Given y_s (scanline), we could solve for *t* and compute *x*:

$$t_s = (y_s-y_1) / d_y$$
$$x_s = x_1 + d_x*t_s$$

    This is <u>4</u> arithmetic operations per edge per scan line
  - We can easily do better

Better edge/scanline intersections

- Once we have an edge's *x* intersection value for scanline *y*, can use that to get intersection at *y+1*



Start: $t = 0$, $y = y_1$, $x = x_1$ and $z = z_1$

Increment y: $y_s = y_1+1$

Solve for t: $t_s = (y_s-y_1) / d_y = ((y_1+1)-y_1) / d_y = 1/d_y$

Solve for x_s: $x_s = x_1 + d_x/d_y$

Let $\Delta x = d_x/d_y$ and note that this term is independent of y

Every time y_s increases by 1, x_s changes by Δx:

$x_s \mathrel{+}= \Delta x$

This reduces computation from 4 arithmetic operations to 1!

Computing *z* at each pixel

- As we fill pixels across the scan line, we also need to compute the *z* values for each pixel center. How do we do that?
  - With the implicit *plane* equation
    $Ax + By + Cz + D = 0$
  - Which we rewrite as
    $z = -(A/C)*x - (B/C)*y - D = ax + by + d$
  - y is constant across the scan line
  - as we move from pixel to pixel, x changes by exactly 1 so z += a
  - This is an example of *depth coherence*

Depth Coherence

- Faces are planar; z-values change linearly across a face.
  - Once we compute a z-value at the left end of a scan line, subsequent z-values for neighboring pixels are cheaper

$$z = -\frac{Ax + By + D}{C} = -\frac{A}{C}x - \frac{B}{C}y - \frac{D}{C} = ax + by + d$$

So, if $z\big|_{x=x_{left}} = z_{left} = ax_{left} + by + d$

then $z\big|_{x=x_{left}+1} = a(x_{left}+1) + by + d = ax_{left} + a + by + d = z_{left} + a = z_{left} - \frac{A}{C}$

In general, $z\big|_{x+\Delta x,y} = z\big|_{x,y} - \frac{A}{C}\Delta x$

and $z\big|_{x,y+\Delta y} = z\big|_{x,y} - \frac{B}{C}\Delta y$

Depth Buffer Issues

- What if z-values are the same?
  - Doesn't matter which you choose:
    - it will be wrong half the time!
  - If there were truly 2 different objects at the same place, the scene would be invalid
  - Round-off error, however, is a real issue
    - Consider polygons that share an edge: one is a back face and backface culling has **not** been applied:
    - It will seem like a random decision as to which *z* value along the edge is determined to be "in front", so you'll get scattered pixels along the edge of the front face with the color of the otherwise invisible back face.