



University of British Columbia  
 CPSC 314 Computer Graphics  
 Jan-Apr 2005

Tamara Munzner

## Visibility II

Week 7, Fri Feb 25

<http://www.ugrad.cs.ubc.ca/~cs314/Vjan2005>

## News

- midterm scores will be scaled
  - stay tuned for details
- demo signups continue
  - you can check your time slot from scans posted to course page
    - (student numbers blocked out)
  - Mon 1-5, Tue 10-1, 3-5, Wed 1-5
- final date/time posted
  - April 19, 8:30-12:30

2

## News

- Written assignment 2 out

3

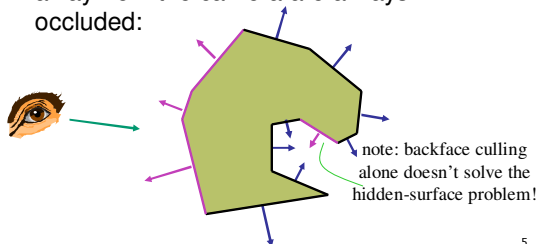
## Review: Invisible Primitives

- *why might a polygon be invisible?*
  - polygon outside the *field of view / frustum*
    - solved by *clipping*
  - polygon is *backfacing*
    - solved by *backface culling*
  - polygon is *occluded* by object(s) nearer the viewpoint
    - solved by *hidden surface removal*

4

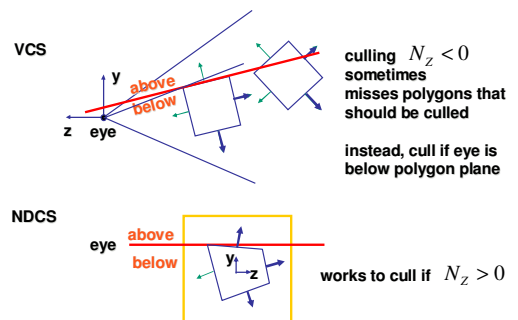
## Review: Back-Face Culling

- on the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:



5

## Review: Back-face Culling



6

### Review: Painter's Algorithm

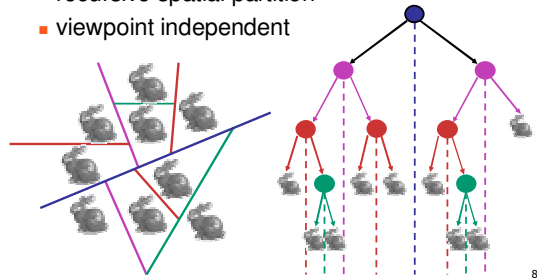
- draw objects from back to front
- problems: no valid visibility order for
  - intersecting polygons
  - cycles of non-intersecting polygons possible



7

### Review: BSP Trees

- preprocess: create binary tree
  - recursive spatial partition
  - viewpoint independent



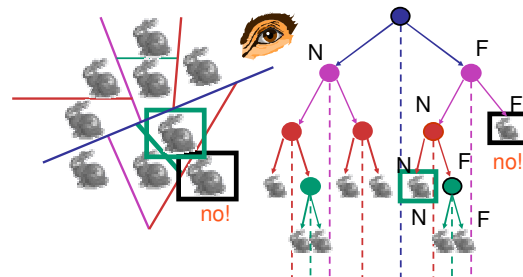
8

### Review: BSP Trees

- runtime: correctly traversing this tree enumerates objects from back to front
  - viewpoint dependent
    - check which side of plane viewpoint is on
  - draw far, draw object in question, draw near
- pros
  - simple, elegant scheme
  - works at object or polygon level
- cons
  - computationally intense preprocessing stage restricts algorithm to static scenes

9

### Correction BSP Trees : Viewpoint B



10

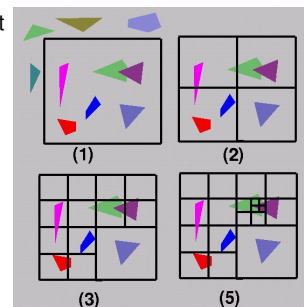
### Warnock's Algorithm (1969)

- based on a powerful general approach common in graphics
  - if the situation is too complex, **subdivide**
- BSP trees was object space approach
- Warnock is image space approach

11

### Warnock's Algorithm

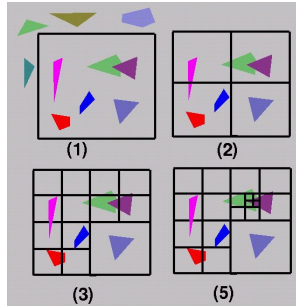
- start with root viewport and list of all objects
- recursion:
  - clip objects to viewport
  - if only 0 or 1 objects
    - done
  - else
    - subdivide to new smaller viewports
    - distribute objects to new viewpoints
    - recurse



12

### Warnock's Algorithm

- termination
  - viewport is single pixel
  - explicitly check for object occlusion



13

### Warnock's Algorithm

- pros:
  - very elegant scheme
  - extends to any primitive type
- cons:
  - hard to embed hierarchical schemes in hardware
  - complex scenes usually have small polygons and high **depth complexity** (number of polygons that overlap a single pixel)
    - thus most screen regions come down to the single-pixel case

14

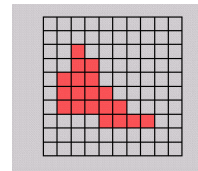
### The Z-Buffer Algorithm (mid-70's)

- both BSP trees and Warnock's algorithm were proposed when memory was expensive
  - first 512x512 framebuffer was >\$50,000!
- Ed Catmull proposed a radical new approach called **z-buffering**.
- the big idea:
  - resolve visibility **independently at each pixel**

15

### The Z-Buffer Algorithm

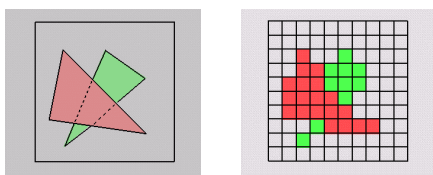
- we know how to rasterize polygons into an image discretized into pixels:



16

### The Z-Buffer Algorithm

- what happens if multiple primitives occupy the same pixel on the screen?
  - which is allowed to paint the pixel?



17

### The Z-Buffer Algorithm

- idea: retain depth after projection transform
  - each vertex maintains z coordinate
    - relative to eye point
  - can do this with canonical viewing volumes

18

## The Z-Buffer Algorithm

- augment color framebuffer with **Z-buffer** or **depth buffer** which stores Z value at each pixel
  - at frame beginning, initialize all pixel depths to  $\infty$
  - when rasterizing, interpolate depth (Z) across polygon
  - check Z-buffer before storing pixel color in framebuffer and storing depth in Z-buffer
  - don't write pixel if its Z value is more distant than the Z value already stored there

19

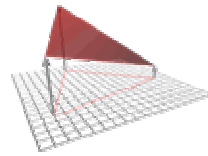
## Interpolating Z

- edge equations: Z just another planar parameter:

- $z = (-D - Ax - By) / C$
- if walking across scanline by  $(D_x)$   
 $z_{\text{new}} = z_{\text{old}} - (A/C)(D_x)$

- total cost:

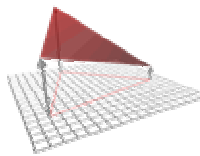
- 1 more parameter to increment in inner loop
- 3x3 matrix multiply for setup



20

## Interpolating Z

- edge walking
  - just interpolate Z along edges and across spans
- barycentric coordinates
  - interpolate Z like other parameters



21

## Z-Buffer

- store (r,g,b,z) for each pixel
- typically 8+8+8+24 bits, can be more

```

for all i,j {
  Depth[i,j] = MAX_DEPTH
  Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) {
      Image[i,j] = C_pixel
      Depth[i,j] = Z_pixel
    }
  }
}
    
```

22

## Depth Test Precision

- reminder: projective transformation maps eye-space z to generic z-range (NDC)
- simple example:

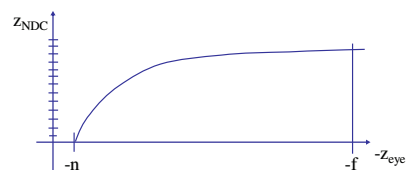
$$T \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- thus:  $z_{\text{NDC}} = \frac{a \cdot z_{\text{eye}} + b}{z_{\text{eye}}} = a + \frac{b}{z_{\text{eye}}}$

23

## Depth Test Precision

- therefore, depth-buffer essentially stores  $1/z$ , rather than z!
- issue with integer depth buffers
  - high precision for near objects
  - low precision for far objects



24

### Depth Test Precision

- low precision can lead to **depth fighting** for far objects
  - two different depths in eye space get mapped to same depth in framebuffer
  - which object “wins” depends on drawing order and scan-conversion
- gets worse for larger ratios  $f:n$ 
  - *rule of thumb*:  $f:n < 1000$  for 24 bit depth buffer
- with 16 bits cannot discern millimeter differences in objects at 1 km distance

25

### Z-Buffer Algorithm Questions

- how much memory does the Z-buffer use?
- does the image rendered depend on the drawing order?
- does the time to render the image depend on the drawing order?
- how does Z-buffer load scale with visible polygons? with framebuffer resolution?

26

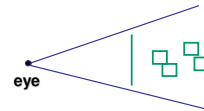
### Z-Buffer Pros

- simple!!!
- easy to implement in hardware
  - hardware support in all graphics cards today
- polygons can be processed in arbitrary order
- easily handles polygon interpenetration
- enables **deferred shading**
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

27

### Z-Buffer Cons

- poor for scenes with high depth complexity
  - need to render all polygons, even if most are invisible



- shared edges are handled inconsistently
  - *ordering dependent*

28

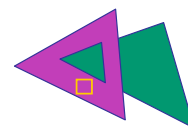
### Z-Buffer Cons

- requires lots of memory
  - (e.g. 1280x1024x32 bits)
- requires fast memory
  - Read-Modify-Write in inner loop
- hard to simulate translucent polygons
  - we throw away color of polygons behind closest one
  - works if polygons ordered back-to-front
    - extra work throws away much of the speed advantage

29

### Hidden Surface Removal

- two kinds of visibility algorithms
  - object space methods
  - image space methods



30

### Object Space Algorithms

- determine visibility on object or polygon level
  - using camera coordinates
- resolution independent
  - explicitly compute visible portions of polygons
- early in pipeline
  - after clipping
- requires depth-sorting
  - painter's algorithm
  - BSP trees

31

### Image Space Algorithms

- perform visibility test for in screen coordinates
  - limited to resolution of display
  - Z-buffer: check every pixel independently
  - Warnock: check up to single pixels if needed
- performed late in rendering pipeline

32