University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2007

Tamara Munzner

# Hidden Surfaces II

## Week 9, Mon Mar 12

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2007

## Reading for This Time

- FCG Chap 12 Graphics Pipeline
  - only 12.1-12.4



## News

- Project 3 update
  - Linux executable reposted
  - template update
    - download package again **OR**
    - just change line 31 of src/main.cpp from

```
int resolution[2];
```

to

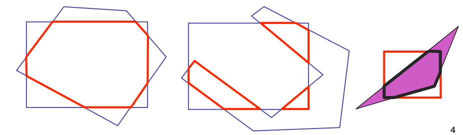
```
int resolution[] = {100,100};
```

**OR**
    - implement resolution parsing



## Review: Polygon Clipping

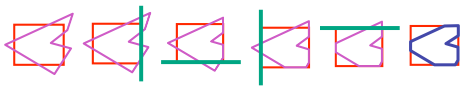
- not just clipping all boundary lines
  - may have to introduce new line segments





## Review: Sutherland-Hodgeman Clipping

- for each viewport edge
  - clip the polygon against the edge equation for new vertex list
  - after doing all edges, the polygon is fully clipped

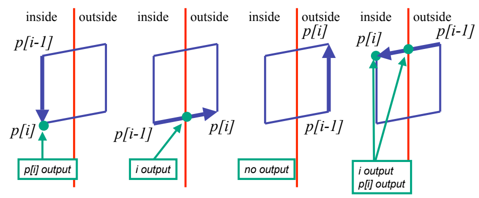


- for each polygon vertex
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?



## Review: Sutherland-Hodgeman Clipping

- edge from *p[i-1]* to *p[i]* has four cases
  - decide what to add to output vertex list





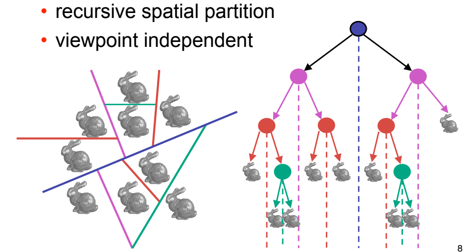

## Review: Painter's Algorithm

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



## Review: BSP Trees

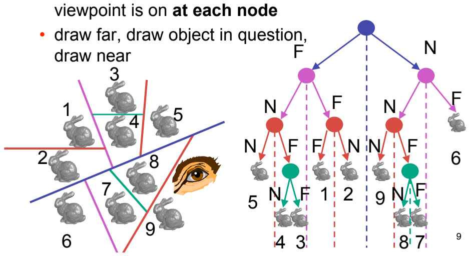
- preprocess: create binary tree
  - recursive spatial partition
  - viewpoint independent





## Review: BSP Trees

- runtime: correctly traversing this tree enumerates objects from back to front
  - viewpoint dependent: check which side of plane viewpoint is on **at each node**
  - draw far, draw object in question, draw near



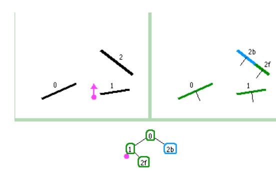


## Hidden Surface Removal II



## BSP Demo

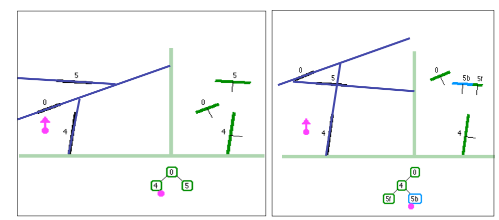
- useful demo:
  *http://symbolcraft.com/graphics/bsp*





## Clarification: BSP Demo

- order of insertion can affect half-plane extent





## Summary: BSP Trees

- pros:
  - simple, elegant scheme
  - correct version of painter's algorithm back-to-front rendering approach
  - was very popular for video games (but getting less so)
- cons:
  - slow to construct tree: O(n log n) to split, sort
  - splitting increases polygon count: O($n^2$) worst-case
  - computationally intense preprocessing stage restricts algorithm to static scenes



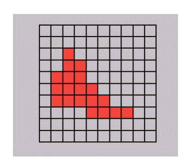
## The Z-Buffer Algorithm (mid-70's)

- BSP trees 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



## The Z-Buffer Algorithm

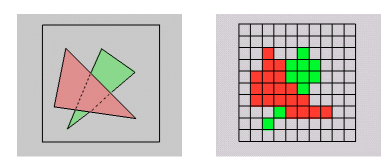
- we know how to rasterize polygons into an image discretized into pixels:





## The Z-Buffer Algorithm

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



## 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



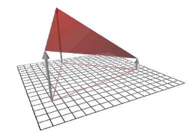
## The Z-Buffer Algorithm

- augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  - at frame beginning, initialize all pixel depths to ∞
  - 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



## Interpolating Z

- barycentric coordinates
  - interpolate Z like other planar parameters





## 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
    }
  }
}
```



## Depth Test Precision

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

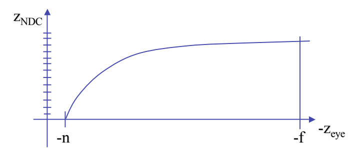
$$T\left(\begin{bmatrix}x\\y\\z\\1\end{bmatrix}\right)=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&a&b\\0&0&-1&0\end{bmatrix}\cdot\begin{bmatrix}x\\y\\z\\1\end{bmatrix}$$

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



## 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






## 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
- demo: sjbaker.org/steve/omniv/love_your_z_buffer.html



## 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?



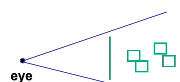
## Z-Buffer Pros

- 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



## 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*



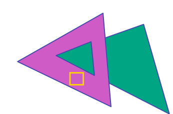
## Z-Buffer Cons

- 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



## Hidden Surface Removal

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





## 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

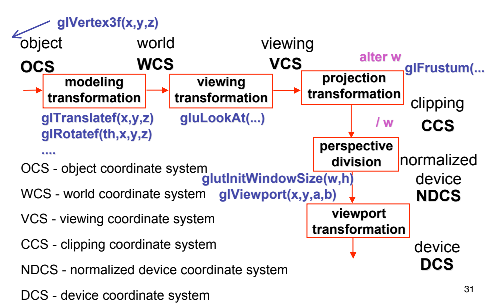


## Image Space Algorithms

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

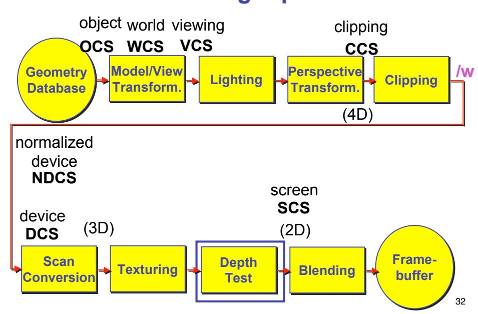


## Projective Rendering Pipeline






## Rendering Pipeline

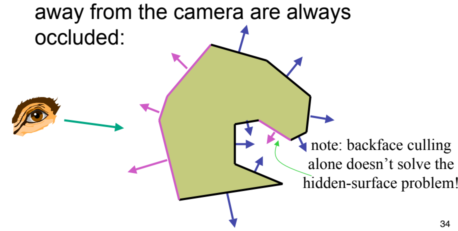


## Backface Culling



## Back-Face Culling

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





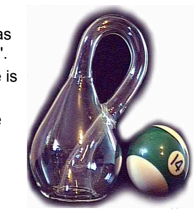

## Back-Face Culling

- not rendering backfacing polygons improves performance
  - by how much?
    - reduces by about half the number of polygons to be considered for each pixel
  - optimization when appropriate



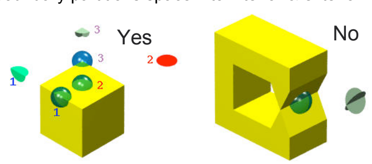
## Back-Face Culling

- most objects in scene are typically “solid”
- rigorously: orientable closed manifolds
  - orientable: must have two distinct sides
    - cannot self-intersect
    - a sphere is orientable since has two sides, 'inside' and 'outside'.
    - a Mobius strip or a Klein bottle is not orientable
  - closed: cannot “walk” from one side to the other
    - sphere is closed manifold
    - plane is not



## Back-Face Culling

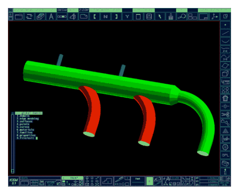
- most objects in scene are typically “solid”
- rigorously: orientable closed manifolds
  - manifold: local neighborhood of all points isomorphic to disc
  - boundary partitions space into interior & exterior






## Manifold

- examples of *manifold* objects:
  - sphere
  - torus
  - well-formed CAD part



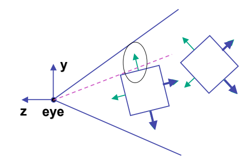


## Back-Face Culling

- examples of non-manifold objects:
  - a single polygon
  - a terrain or height field
  - polyhedron w/ missing face
  - anything with cracks or holes in boundary
  - one-polygon thick lampshade



## Back-face Culling: VCS

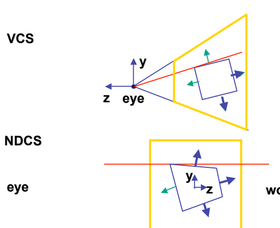



first idea: cull if $N_Z < 0$

sometimes misses polygons that should be culled



## Back-face Culling: NDCS




works to cull if $N_Z > 0$



## 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