Lecture 8: 5 October 1999

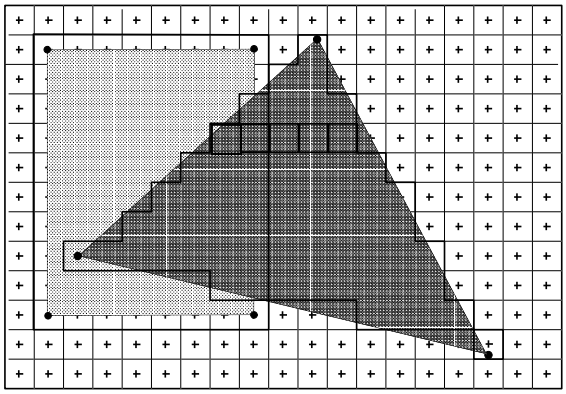
Administrative:
  Asst 4 (scene modeling) due Friday 5pm
Today:
  Inventor note – modifying DEF'ined objects
  Polygon Scan Conversion (H&B §14.5)
  Hidden Surface Elimination
  Assignment 3 (`uid_object` exhibition)
  Assignment 5 (`ivscan`) demo, handout
Thursday:
  Generalized Polygon Clipping
  Generalized Back-Face Removal

Q: "How can I instance objects in my own color?"
A: "You can't." (Fundamental Inventor limitation.)
Workaround: edit object (e.g. as ascii):

```
sed < foo.iv > bar.iv
  -e "s/diffuseColor  0.8 0.7 0.2
      /diffuseColor  0.3 0.6 0.8/g"
```


Hidden-Surface Elimination

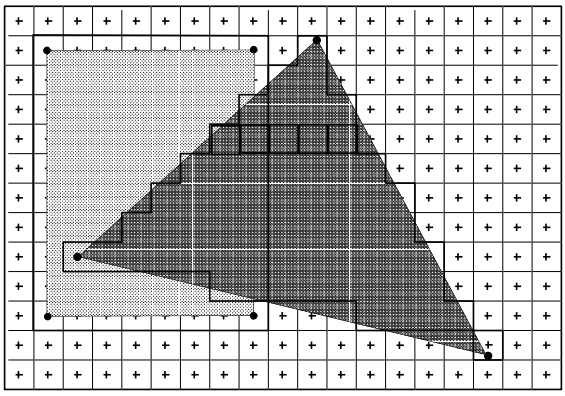
So far, have *scan-converted* only one polygon
If polygons overlap, must eliminate *hidden* portions



FillSpan() bottomed out to *setPixel*(x, y, I)
  Now, must arbitrate among competing spans
Several methods:
  Full-frame z buffer
  No z buffer – maintain ASL (Active Span List)
  Single-span z buffer – intermediate choice

Visibility resolution with full Z-buffer

Keep an array of depth values, one per pixel



At start of frame, all depths initialized to "far away"
Now, scan convert each polygon *independently*
Write color value *conditionally* at each pixel:
  If $z < z_{\text{stored}}$ then
    update stored color
    $z_{\text{stored}} = z$
Advantages?
Simple; Running time function only of screen area

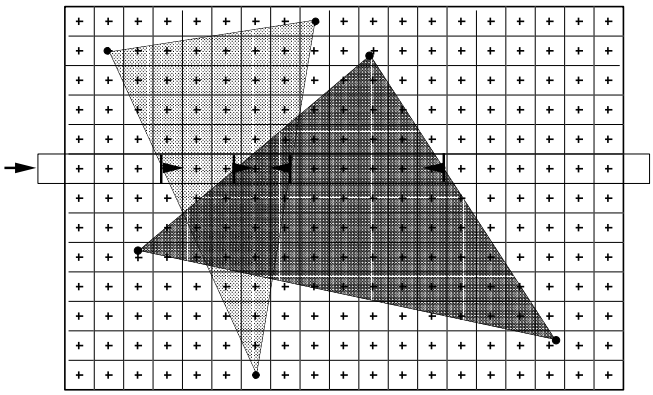
Z-buffer Memory Requirements

What are framebuffer memory requirements?
  $1024 \times 1280 = 1\frac{1}{4}$ Mpixels
  R, G, B, α channels 8 bits / pixel = 4 bytes / pixel
  Depth (z) values 32 bits / pixel = 4 bytes / pixel
  Double-buffering doubles storage requirement
  Total: [5Mb (RGBα) + 5Mb (z)] $\times 2$ = 20 Mb!
First framebuffer (512) cost $80,000 (late '60s)!
Even today, z-buffer, logic runs ~$50+
Is there any reason not to use a z-buffer?

Today, we'll assume no z-buffer
  Classical techniques, lots of literature
  Introduction to visible-surface algorithms
  Later in term: object-space visibility

Scan-line Hidden Surface Algorithms

"Sweep" horizontal scanline downward over scene



Assume polygons are **convex**

Exactly ☐ edges at each scanline

Each polygon has IN/OUT flag (used later)

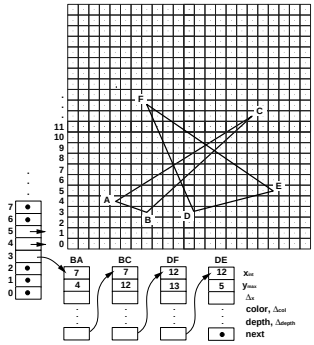
Algorithm Invariants:

Maintain portion of scene intersected by current scanline, *ordered* by x intercept of edges

Output all pixels on scanline before proceeding

Initialization

Precompute: Edge Table (ET), one entry per scan line



Each entry is a linked list of `EdgeRec`s, sorted by x_{int}:

y_{end}: y of top edge endpoint

x_{int}, Δ_x: current x intersection, delta wrt y

col_{curr}, Δ_{col}: current color, delta wrt y

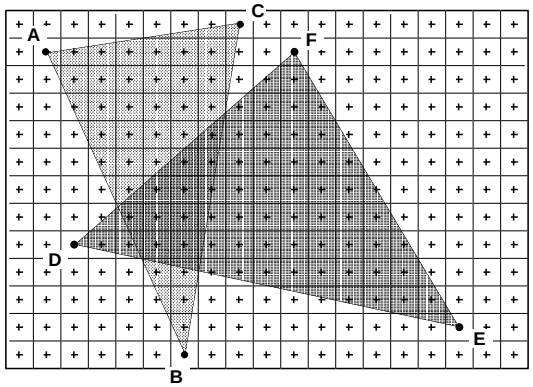
z_{curr}, Δ_z: current depth, delta wrt y

Pointer to polygon from which edge came (for matching edges across shared vertices)

next: pointer to next record, or NULL

At BOT, insert all non-horizontal edges into ET

Initialization (cont.)



Event list of **buckets**, one per scan line

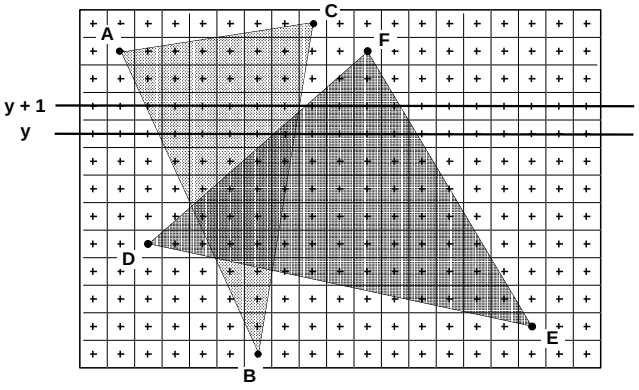
Event: start/finish of an edge (i.e., vertex) occurs within interval ☐

In each bucket, events sorted by x coordinate

Active edge list (AEL): initially empty

(Will be incrementally maintained to store all edges intersecting scanline, ordered by x)

When Does AEL Change State?



When a vertex is encountered

I.e., when an edge begins or ends

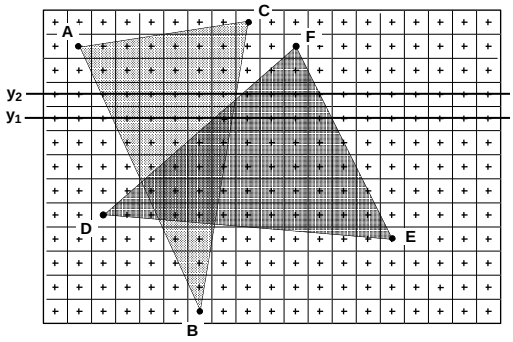
All such events pre-stored in event list!

When two edges change order along a scanline

I.e., when edges cross each other!

How to detect this efficiently?

Processing a Scanline 1



Add any edges which start in $[y, y+1)$

How can x, y pre-sorting be exploited ?

Traverse AEL, left to right:

When edge encountered, toggle poly's IN/OUT flag

If entering, scan for matching edge (how?)

Determine visible polygon *at each pixel center*

1) Interpolate z along each span, & find min; or...

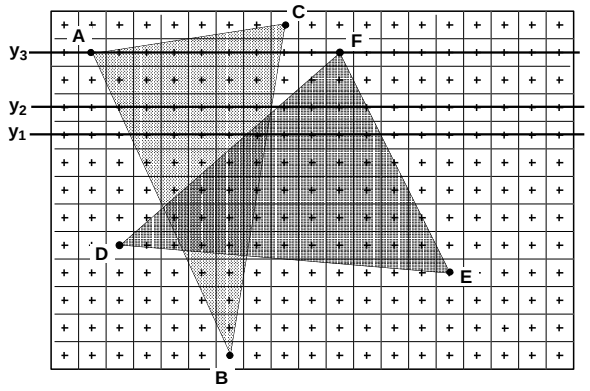
2) Render each span into one-raster z buffer; or...

3) Use more elegant ASL method (in a minute)

Extract interpolated color from visible span

Write winning color using `setPixel(`x, y, C`)`

Processing a Scanline 2



For each edge in AEL:

If edge ends in $(y-1, y]$, [redacted]

Otherwise, for next y: update x, color, z

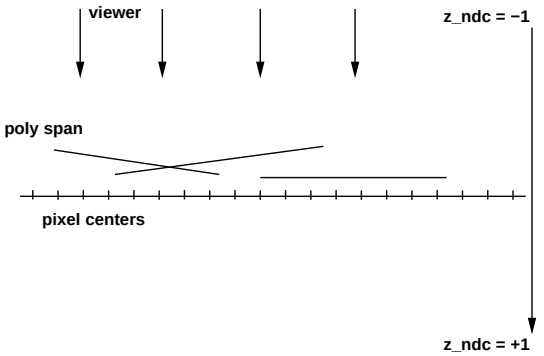
Once after each scanline is processed:

Sort AEL by x (mostly sorted, so...)

Increment scanline variable y

Raster Filling Without a z Buffer:

Use Active Span List (ASL)



Completely analogous to AEL ...

Except one dimension lower !

Algorithm Summary:

Initialize Polygons, **ET, AEL**

For each scanline y

Update AEL (insert edges from ET[y])

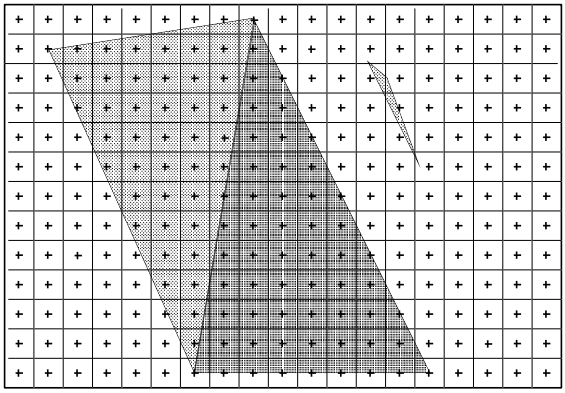
Assign raster of pixels from AEL

Update AEL (delete, increment, resort)

Clean up data structures

Scan Conversion Pitfalls

Careful: singularities!

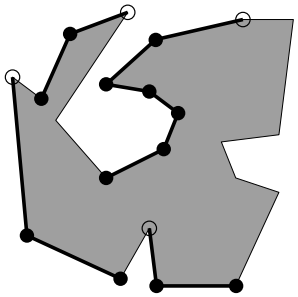


Omitted pixels (gaps)
Twice-filled pixels (problem when blending)
Sliver polygons (aliasing)
Horizontal edges (consistent coverage rule)

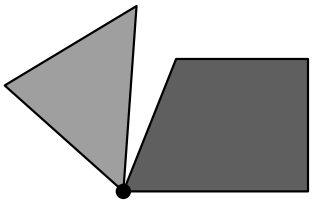
Each polygon should *own* certain pixels:
don't want pixels owned by multiple polygons
don't want to "drop" pixels
polygon owns all pixels centered in poly interior
polygon owns no pixels centered outside poly interior

Shadow Rule

"Shadow" Rule: polygon owns boundary pixel unless:
Edge is horizontal, and on "top" of poly
Edge is right-facing (normal has $n_x > 0$)
Pixel at upper extremum of polygon



Still must handle special case of shared vertices!

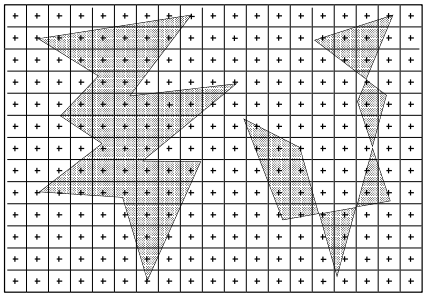


Spatial Coherence

Exploit spatial problem structure for efficiency
Across scanlines: edge, polygon coherence
Edge intersects scanline → likely that
edge will intersect subsequent scanline
Within scanline: span, depth coherence
Other kinds of coherence?

Extensions

Non-convex polygons
Multiple active intervals per polygon
Non-simple (i.e., self-intersecting) polygons
More general "inside-outside" rule

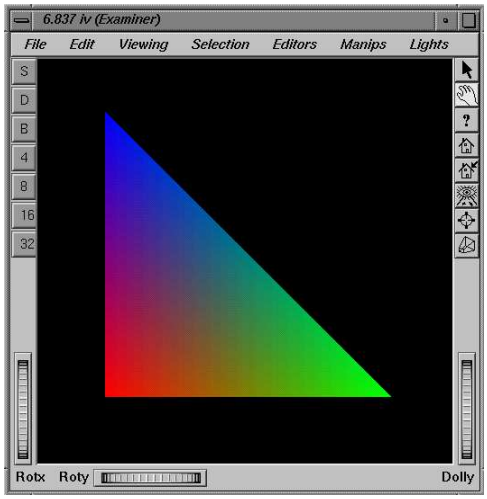


More sophisticated visibility maintenance
Use of coherence to defer resorting

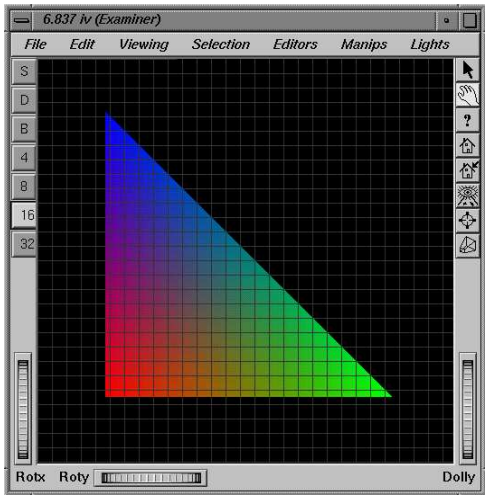
Writing fast, robust scan-converter is hard!

Assignment 4: `ivscan`

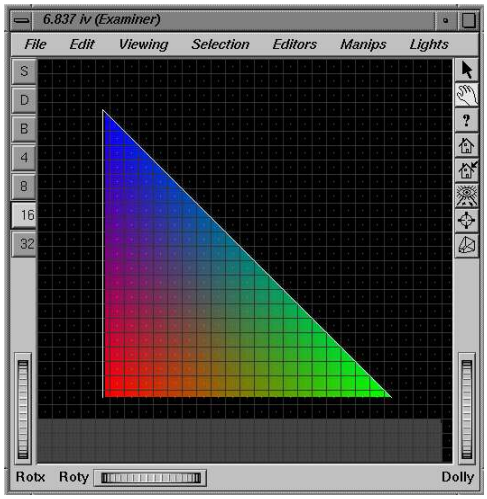
Inventor input:



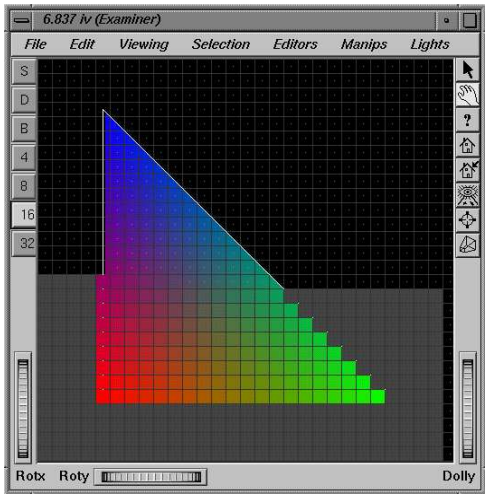
Pixels outlined:



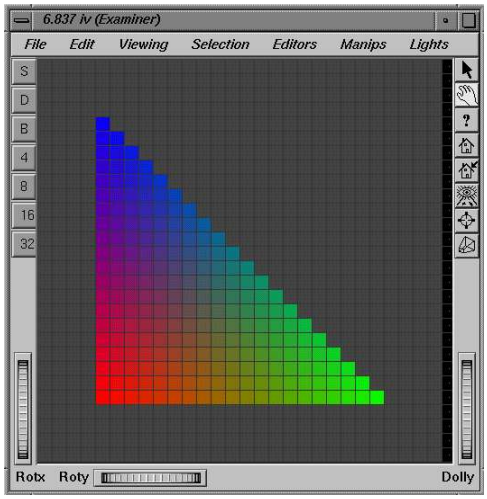
Edges outlined:



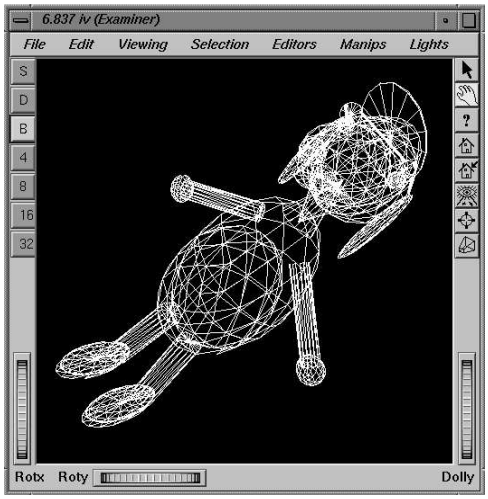
Scan conversion [calls to `setPixel()`]:



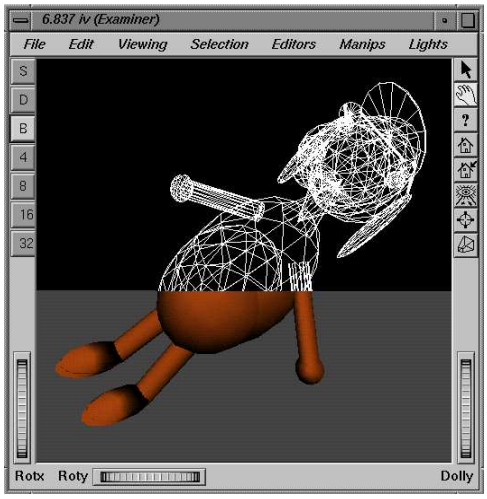
Resulting image:



Another Example

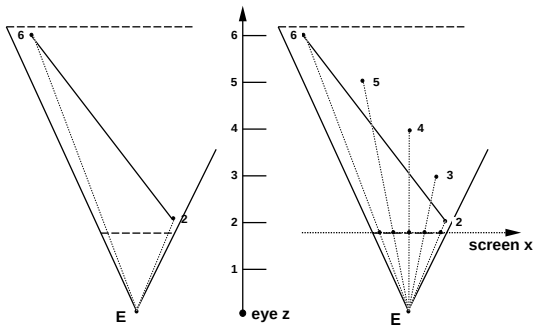


Scan conversion [calls to `setPixel()`]:



Screen Space Depth Interpolation

Pitfall of naive (eye-z) depth interpolation:



This is what happens when you interpolate
Eye z in **screen space**
(Geometric equivalent of Gouraud flaw from L7)
How to correctly interpolate depth? Hints:
Work through homogeneous computation of z
Read through comments in `ivscan` source:
`EdgeRec.h`
`ScanWrap.C`