Lecture 9: Thursday, 2 October 1997

Administrative:

- Assignment 3B (Scene Modeling):
  - Observers of R.H. can take extension to Monday
    - (Must send email to staff stating this)
  - Limit render time to 10 seconds on an Indy
  - OK to edit others' objects, add your own, etc.

Today:

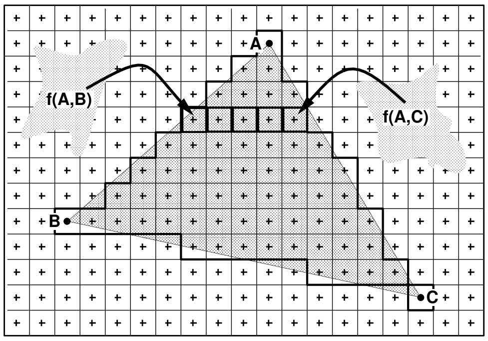
- 3D Polygon Scan Conversion (H&B S3.11, 13.5)
  - Interpolation of Color, Depth
- Continuation of Lecture 2 (2D Polygon Filling)
- Assignment 4 (`ivscan`)

L10, etc.

- Some changes, due to guest lecture
- Watch (reload !) the on-line calendar

Scan conversion

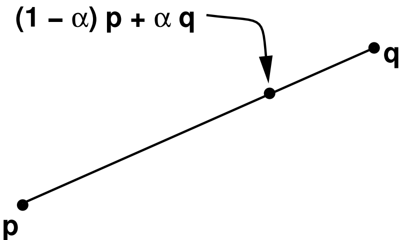
Conversion of *continuous* primitive
  to its discrete *pixel coverage*
Color (depth, etc.) associated with vertices
  interpolate across polygon interior



Step along left, right triangle edges
  interpolate color (depth, etc.) at pixels

Linear Interpolation

Consider interpolation along line segment

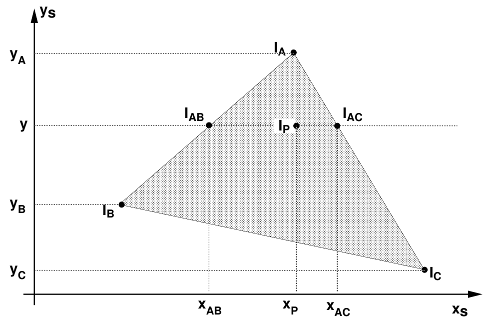


Coefficients non-negative, sum to 1 $\Rightarrow$ interpolation
  yields a *convex combination* of input samples
Can interpolate just about anything this way:

- Points
- Vectors (i.e., Normals)
- Colors
- Transparency
- Texture coordinates
- ...

Gouraud Interpolation

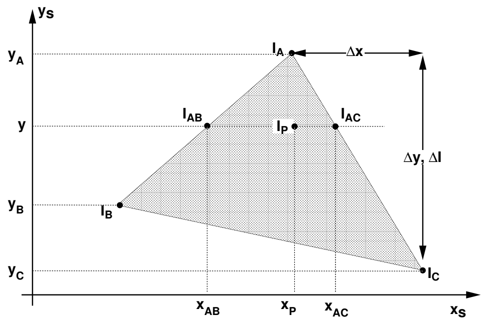
Depth, color associated with polygon vertices
  Depth: z coordinate, from perspective transform
  Color: radiometry (shading), from lighting model
Interpolate polygon edges to find *span* endpoints
  interpolate span interiors to find pixel values



Terminology: Shading, Interpolation

Scan Converting One Triangle

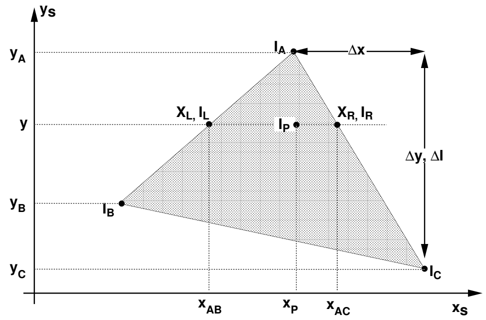
Scanline loop (x, y, intensity I per vertex):
FillPoly $(x_A, y_A, I_A, x_B, y_B, I_B, x_C, y_C, I_C)$



$\Delta X_{AB} = \frac{x_B - x_A}{y_B - y_A}$; $\Delta X_{AC} = \frac{x_C - x_A}{y_C - y_A}$
$\Delta I_{AB} = \frac{I_B - I_A}{y_B - y_A}$; $\Delta I_{AC} = \frac{I_C - I_A}{y_C - y_A}$
$x_{AB} = x_{AC} = x_A$; $I_{AB} = I_{AC} = I_A$
For $y \in y_A..y_B$
    $\mathit{FillSpan}(y, x_{AB}, I_{AB}, x_{AC}, I_{AC})$
    x_{AB} += ΔX_{AB}
    x_{AC} += ΔX_{AC}
    I_{AB} += ΔI_{AB}
    I_{AC} += ΔI_{AC}
For $y \in y_B..y_C$
    ...

Pseudo code

Span filling:



$\mathit{FillSpan}(y, x_L, I_L, x_R, I_R)$
$\Delta I = \frac{I_R - I_L}{x_R - x_L}$
$I = I_L$
For $x \in x_L..x_R$
    $\mathit{setPixel}(x, y, I)$
    I += ΔI

Analogous interpolation of depth (not yet needed)

Recap: High-Level Strategy

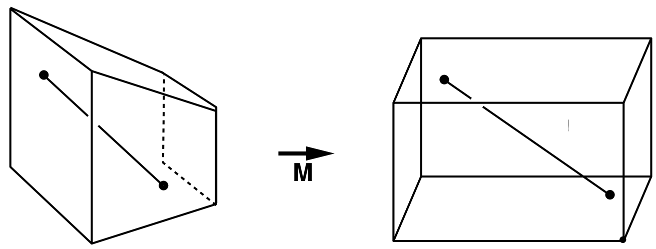
1) Shade vertices
2) Transform, project vertices into screen space
3) Interpolate screen space depth, color

Gouraud Flaw – Convex Interpolation

What if polygon interior has sharp highlight?
What if polygon vertex has sharp highlight?

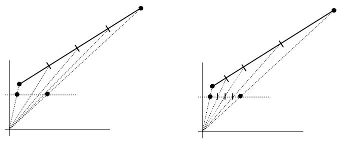
Gouraud Flaw – Not Perspective Correct

Recall: Perspective transformation preserves lines, planes
Planar polygons remain planar under perspective
  ⇒ polygon edges remain linear

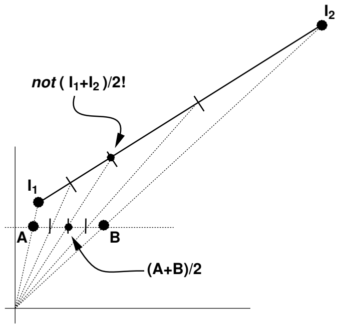


Screen Space Color Interpolation

Perspective transformation does *not* preserve distances!

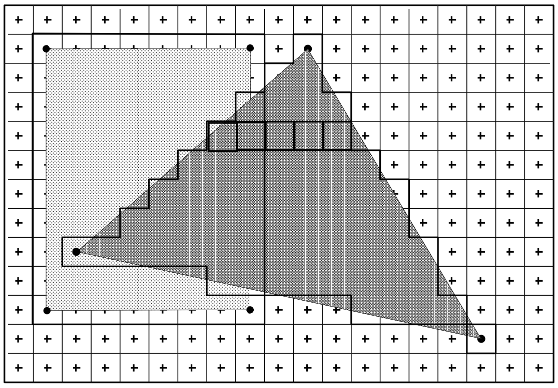


Interpolation before, after perspective
yields different results!
Shading parameters should *not* be
interpolated in screen space



Hidden-Surface Elimination

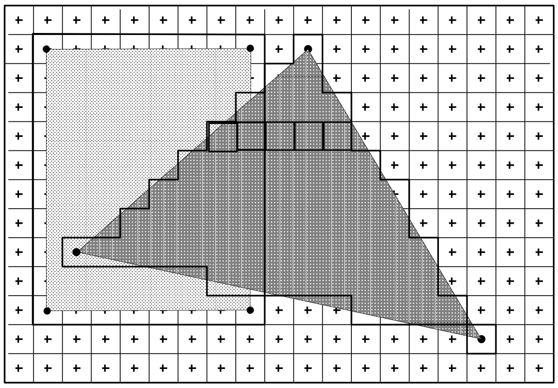
So far, have *scan-converted* only one polygon
If polygons overlap, must eliminate *hidden* portions



FillSpan() bottomed out to *setPixel*(x, y, I)
Now, must arbitrate among competing spans
Two methods:
Brute force: z buffer
Elegant: maintain ASL (Active Span List)

Visibility resolution with a Z-buffer

Keep an array of depth values, one per pixel



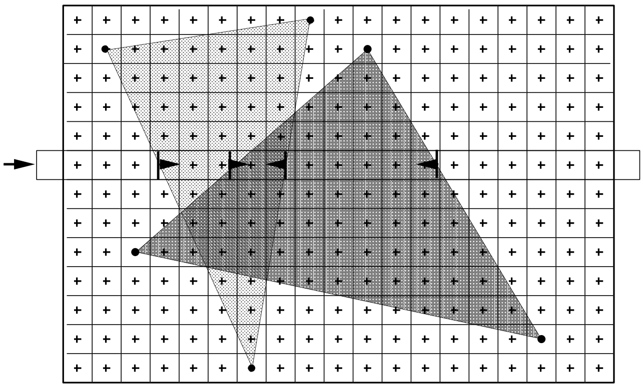
At start of frame, all depths initialized to "far away"
Color value written *conditionally* at each pixel:
If $z < z_{\text{stored}}$ then
update stored color
$z_{\text{stored}} = z$
Both full-frame, single-raster z buffers possible

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 cost about $80,000 (late '60s)!
Even today, z-buffer, logic runs ~$250+
Other reasons not to use a z-buffer?

Scan-line Hidden Surface Algorithms

"Sweep" horizontal scanline downward over scene



Assume polygons are **convex**

Exactly 0 or 2 edges at each scanline

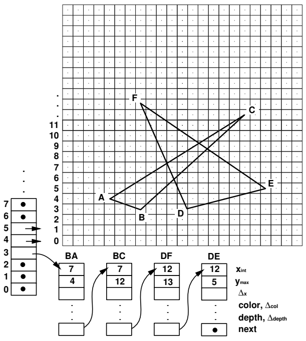
Algorithm Invariants:

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

Output all pixels on scanline before proceeding

Initialization 1

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

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

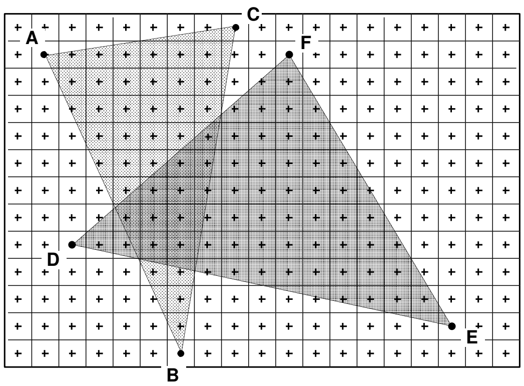
z_{curr}, Δ_z: current depth, delta

Pointer to polygon from which edge came (for matching)

next: pointer to next record, or NULL

At BOT, insert all non-horizontal edges

Initialization 2



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

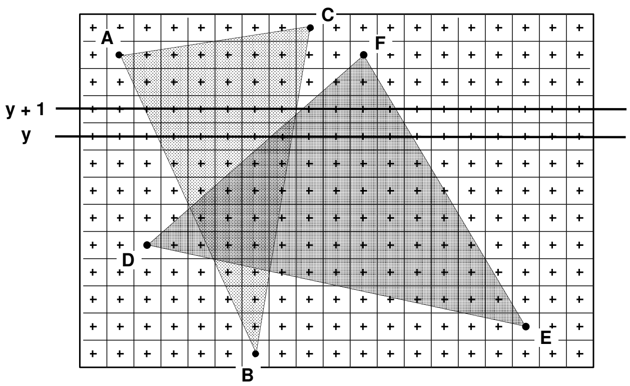
Event: start/finish of an edge (i.e., vertex)

Within y, 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

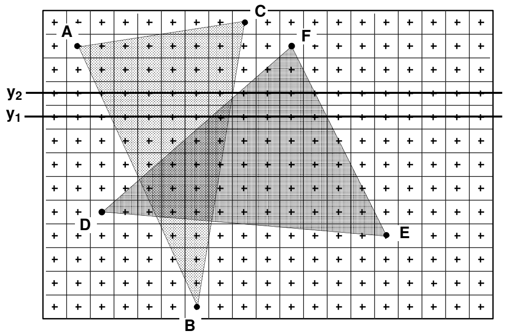
All such events pre-stored in event list!

When two edges change order along a scanline

I.e., when edges cross each other!

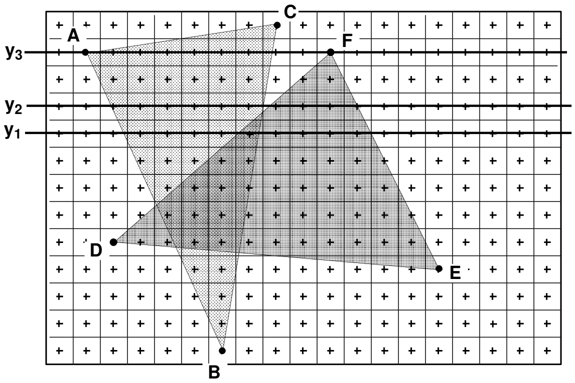
How to detect this efficiently?

Processing a Scanline 1



Add any edges which start at y (merge sort)
Traverse AEL, left to right:
- When edge encountered, toggle polygon IN/OUT flag
  - If entering, scan for matching edge (how?)
- Determine visible polygon *at each pixel*
  - 1) Interpolate depth along each span; find min
  - 2) Elegant method (in a minute)
- Extract interpolated color from visible span
- Write winning color using `setPixel()`

Processing a Scanline 2

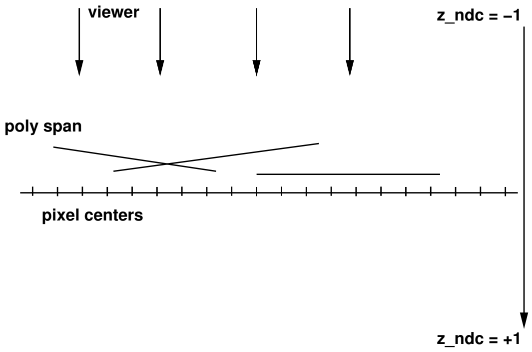


For each edge in AEL:
- If edge ends at y, delete it
- Otherwise, update x, color, depth for next y

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)



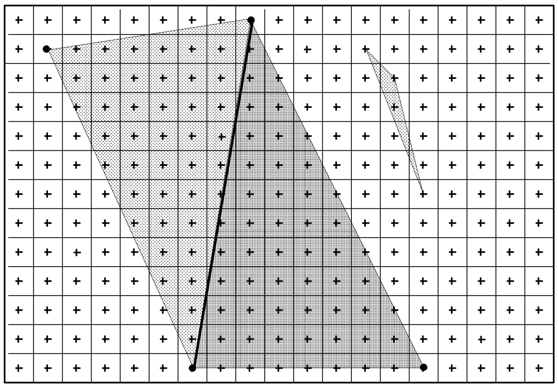
Completely analogous to AEL !

Algorithm:

Initialize **ET, AEL**
For each scanline y
- Update AEL (insert edges from ET[y])
- Assign raster of pixels from AEL
- Update AEL (delete, increment, resort)

Scan Conversion Pitfalls

Careful: singularities!



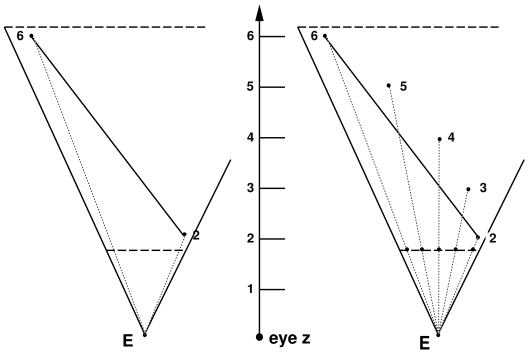
- Omitted pixels (gaps)
- Twice-filled pixels (problem when blending)
- Sliver polygons (aliasing)
- Horizontal edges (consistent coverage rule)

Coherence

Exploit spatial structure of data
- Across scanlines: edge, polygon coherence
  - Edge intersects scanline → likely that edge intersects subsequent scanline
- Within scanline: span, depth coherence
- Other kinds of coherence?

Screen Space Depth Interpolation

Pitfall of naive depth interpolation:



This is what happens when you interpolate
- **Eye** z in **screen space**

How to correctly interpolate depth? Hints:
- Work through homogeneous computation of z
- Read extensive comments in `EdgeRec.h`

Extensions

Non-convex polygons
- Multiple active intervals per polygon

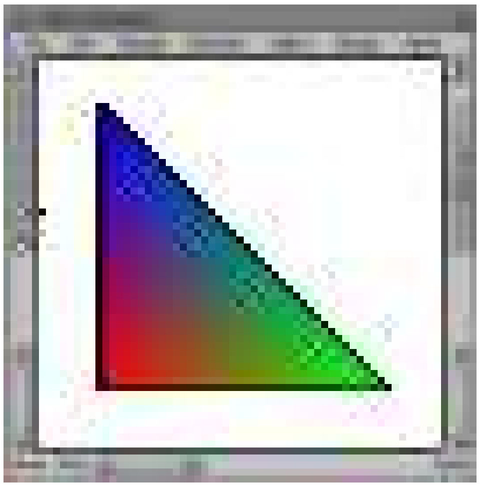
Non-simple (i.e., self-intersecting) polygons
- More general “inside-outside” rule

More sophisticated visibility maintenance
- Use of coherence to defer resorting

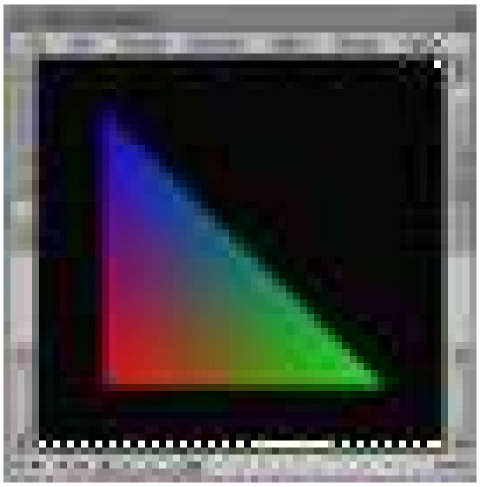
Writing fast, robust scan-converter is hard!

Assignment 4: `ivscan`

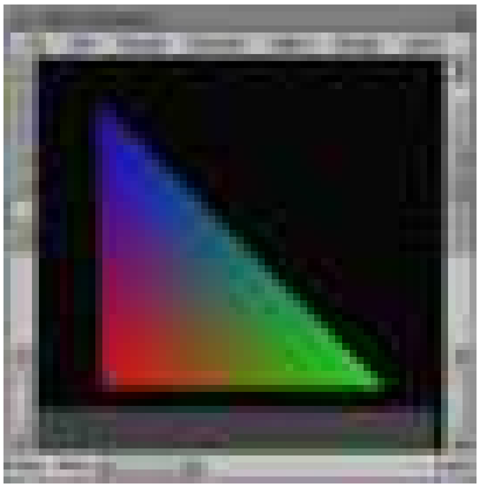
Inventor input:



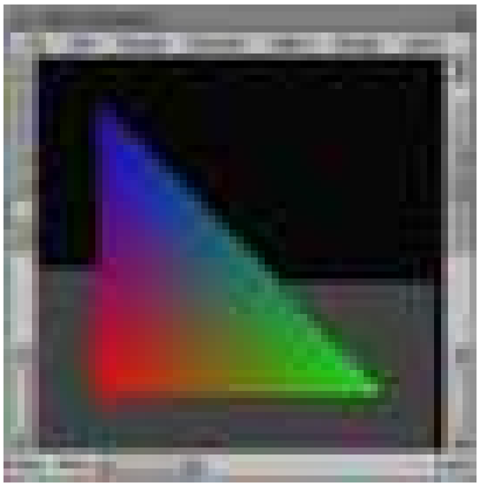
Pixels outlined:



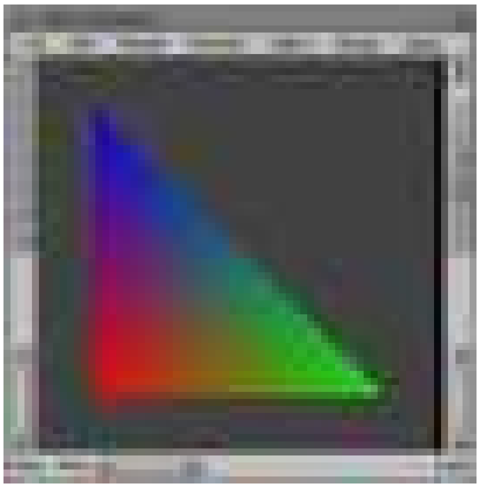
Edges outlined:



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



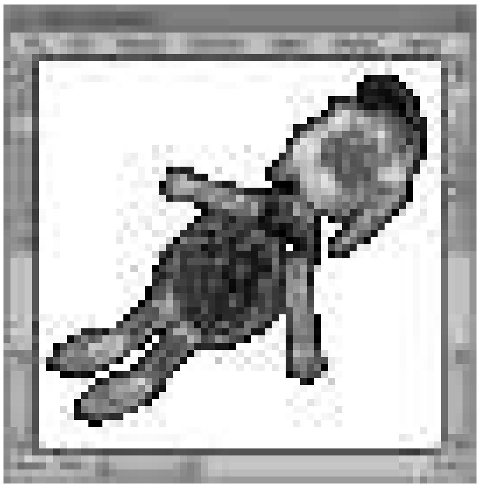
Resulting image:



Another Example

Inventor input (`rickie_snoopy.1.iv`):

Pixels, Edges outlined:



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