Prof. Reuven Aviv
Department of Computer Science
Tel Hai Academic College

Computer Graphics

Raster Operations

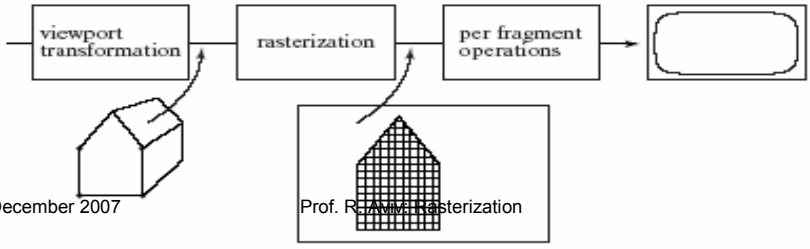
Slides adapted from F. Hill, S. Kelley Computer Graphics

Contents

Pixmaps manipulations

Creating Images

- Image: arrays of pixels displayed on a raster device.
- *how can we create images?*
- Compute a value for each pixel procedurally.
- Scan and digitize an existing image.
- Image stored in a pixmap (color values) structure



images from pipeline and other images

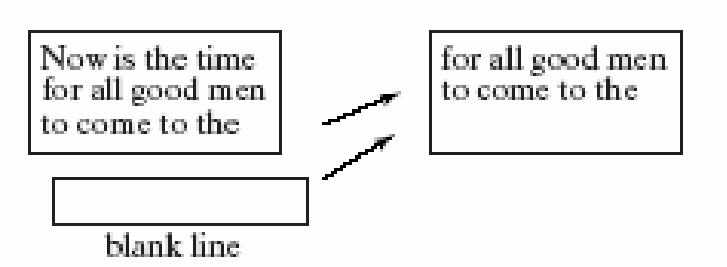
- Viewport transformation produces *fragments*
  - color, depth, & texture coordinates (s,t)
  - per vertex
- Several processing operations are performed on fragments before they are written to the screen.
- these operations can also be done directly on images on the screen, not part of the pipeline *e.g. menus. More images?*
- E.g. scrolling, moving cursor, adding text, menus
  - copy/paste images, change colors, combine images,..
- Need data structure to hold images: pixmap

Manipulating Pixmaps

- *Where pixmaps are stored?*
- main memory or frame buffer *what's the diff?*
- Pixmap in the frame buffer are visible on screen
- Other pixmaps will be put some time later in the frame buffer
- *What operations can be done on pixmaps*
- read, write between frame buffer and memory
- Copy, scale, rotate, compare, combine,
- Define regions in pixmaps: circle, lines, text
- Fill regions, draw lines

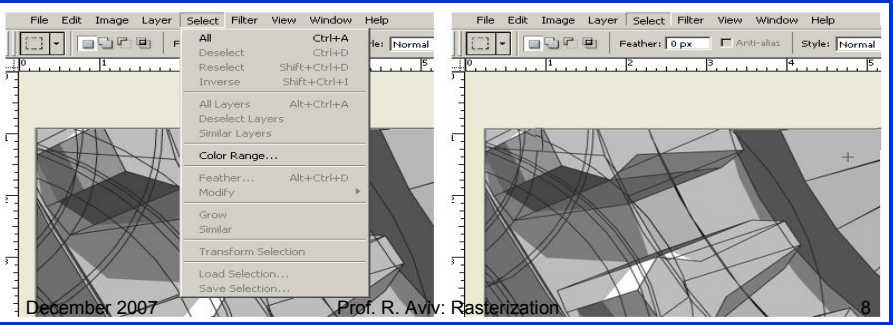
Example: Scrolling

- draw blank line at bottom
- move (copy/past) all lines up one
  - Frame buffer to frame buffer operation

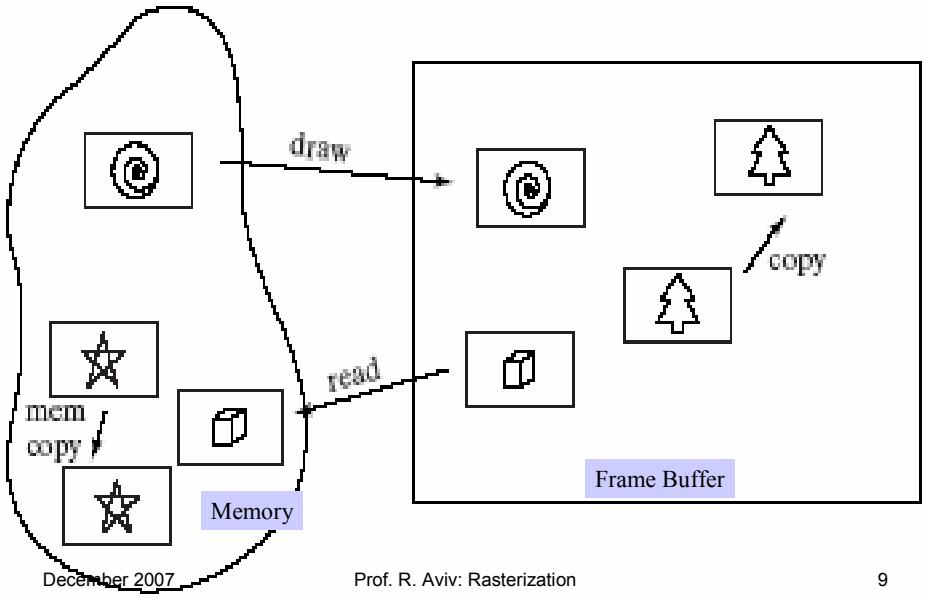


Examples (2)

- Redrawing screen after menu use *how?*
- Read region of image to off-screen pixmap
- Copy menu pixmap to frame buffer
- Later write the off-screen pixmap to frame buffer



Pixmap Operations: Copying



Pixmap Operations in OpenGL: Copying

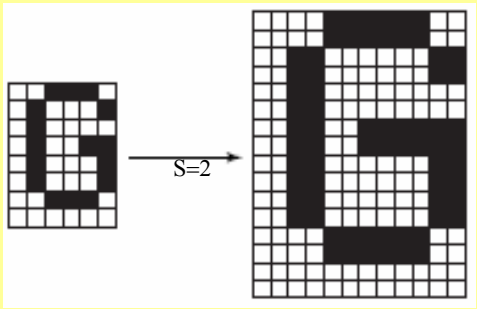
- glReadPixels () from frame buffer into memory.
- glCopyPixels() copies a region in one part of the frame buffer into another region of the frame buffer.
- glDrawPixels() draws a pixmap into frame buffer.

Pixmap Data Types

- A pixmap has rows & columns of pixels with data
- Types of pixmaps
  - Bitmap: pixel = bit, 0 or 1 (black or white)
  - Gray-scale bitmap: 1 byte, representing gray level
  - RGB pixmap: 3 bytes *for what?*
  - RGBA: 4 bytes, red, green, blue, alpha (transparancy)
  - Pixel contains an index into a lookup table (LUT);
    - usually index is a byte
    - it points to a color value

Scaling Pixmaps

- Scale by integer s:
  - New pixmap is larger
  - Pixel replication
  - 6 x 8 to 12 x 16

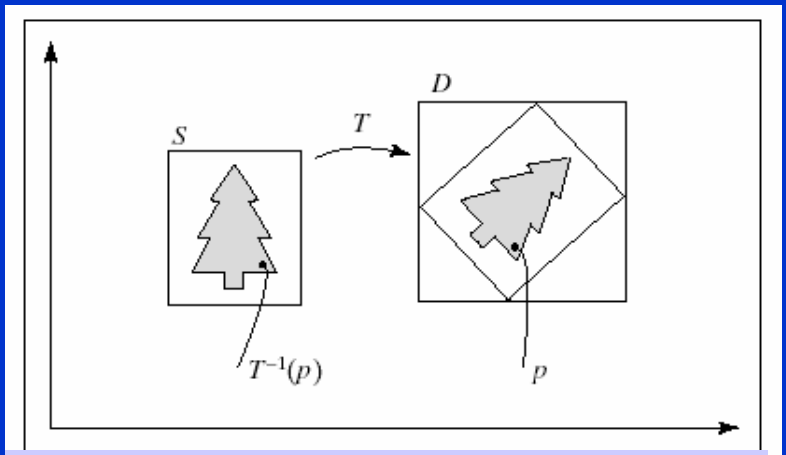


- *What if s = 1/3?*
- take every third row & column of pixmap.
  - not satisfactory
- average the values of 9 pixels in each 3 x 3 array
  - use that value for the pixel in the new pixmap

Rotating Images

- Rotating through 90°, 180°, 270° is simple:
  - copy pixels colors to new locations in new pixmap
- Other rotations difficult.
  - *why not use same algorithm?*
- several old pixels transform in part to new pixel
  - bad results *how to improve?*
- find all pixels which transform (in part) to new pixel
  - calculate average color

Rotating Images (2)



Need to find original pixels that transformed to P

Combining Pixmaps

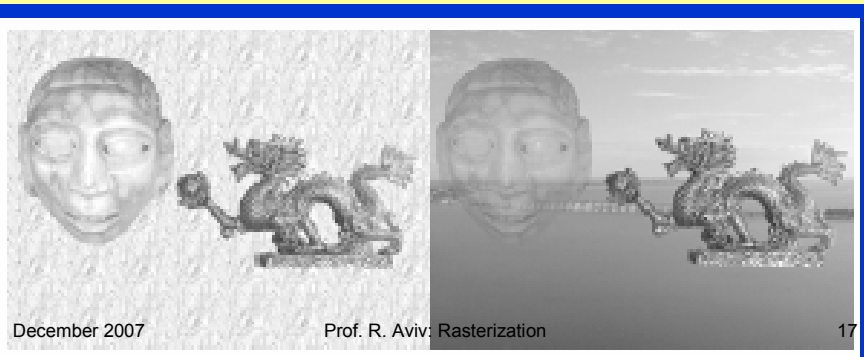
- Pixmaps are usually combined pixel by pixel
  - using corresponding pixels
- Weighted average: (1 – f)*__A__ + f*__B__ for some f < 1.0.
- Allows "dissolving" one image into another *how?*
- Prepare a series of pimaps with f varies 0 → 1
- Load pixmaps to frame buffer one after another

Alpha and Image Blending

- Blending allows you to draw a partially transparent image over another image.
- use the alpha value, "4th component of color"
  - $\alpha = 0$ transparent, $\alpha = 255$ opaque
- Source: pixel [i][j] green color: S[i][j].g, $\alpha = 200$
- Destination: pixel [i][j] green color: D[i][j].g, $\alpha = 10$
- New Destination green color blends 2 values:
- D[i][j].g = u*S[i][j].g + (1-u)*D[i][j].g 0 < u < 1
- u = S[i][j]*α/255
- Note: u changes from pixel to pixel.

Example: Forward backward images

- Background sea transparent, $\alpha = 0$.
- mask semi-transparent. $\alpha = 128$;
- Dragon opaque, $\alpha = 255$
- Dragon is forward; Mask semi transparent.



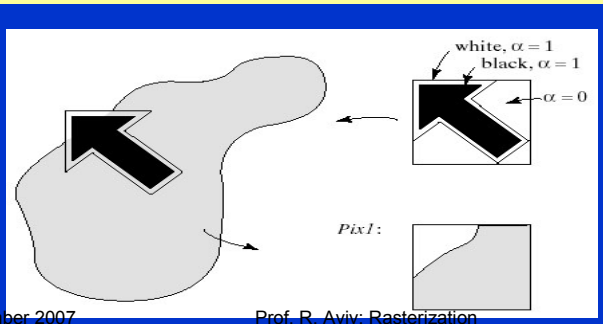
Example: Cursor Viewing

•Cursor consist of two regions:

•Black, opaque + thin transparent border. *what's that for?*

• cursor move, we see through the border the underlying image

•Makes the cursor "float" over the image



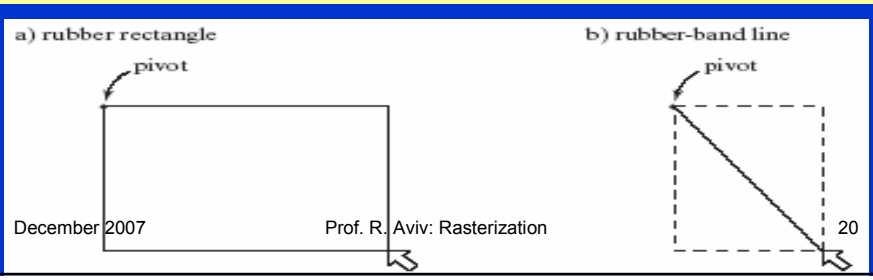
Logical Combinations of Pixmaps

- combine pixmaps in various ways.

parameter value	value written to destination
GL_CLEAR	0
GL_COPY	S
GL_NOOP	D
GL_SET	1
GL_COPY_INVERTED	NOT S
GL_INVERT	NOT D
GL_AND_REVERSE	S OR NOT D
GL_AND	S AND D
GL_OR	S OR D
GL_NAND	NOT(S AND D)
GL_NOR	NOT(S OR D)
GL_XOR	S XOR D
GL_EQUIV	NOT(S XOR D)
GL_AND_INVERTED	NOT S AND D
GL_OR_INVERTED	NOT S OR D

Application: Rubber-banding Lines and Rectangles

- user draws a line or rectangle with the mouse, adjusts them interactively
  - Line/rectangle erased and redrawn continuously
  - *how?*
- Write: XOR line or rectangle pixmap with frame buffer
- Erase: another XOR



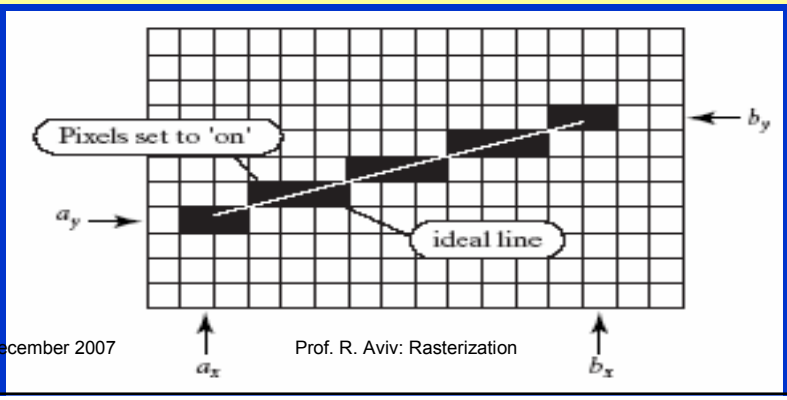
The general BitBlt Operation

- BitBlt (bit block-transfer): hardware operation
  - combines the draw, read, and copy operations
- basic operation: source rect copied to dest rect
  - Rect same size, either in memory or on the screen
- "copy" includes operation between the pixels
  - Dest is clipped to the BitBlt's clipping rectangle.
  - many other operations on blocks of bits

Line Drawing

Line Drawing

- *How line-drawing routine (e.g in OpenGL) works?*
- need to calculate which pixels need to be colored
- line to draw: from **A** = (a_x, a_y) to **B** = (b_x, b_y) (integers)
- *ideal vs. actual line?*



Brute Force Line Drawing

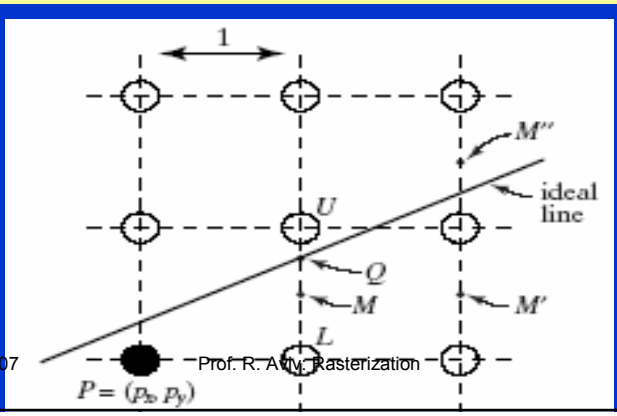
- Equation for the line: $y = m (x - a_x) + a_y$
- *What's the line slope m ?*
- $m = (b_y - a_y)/(b_x - a_x)$ calculated once
- increment x, from a_x to b_x in steps of one pixel
- for each pixel
  - calculate $y = \text{round}\ (m (x - a_x) + a_y)$
  - color the resulting pixel (x,y)
- Requires a multiplication, a subtraction, an addition, and a round for each pixel – Expensive! Slow!

Differential approach

- $y=m(x - a_x) + a_y$ *what is Δy when x --> x + Δx*
- $\Delta y = m*\Delta x$, where $\Delta x = 1$ pixel
- Algorithm
- Start: $x = a_x$, $y = a_y$
- Repeat:
  - Increment x by 1
  - New y = round(previous y + m). Color (x, y)
- requires a floating addition and a round.
- The question was: what is y for a given x
- idea: replace the question by another one!

Bresenham Approach

- *What is the question?*
- Which pixel (L or U) do we set next?
- uses no floating point arithmetic and no rounding.
- Incremental approach



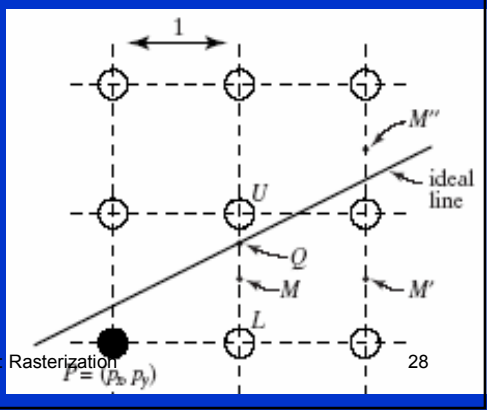
Bresenham Approach (2)

- Assume a line with $0 < \text{slope} < 1$
- Define *extents* of the line segment in x and y:
  - $W = b_x - a_x$, and $H = b_y - a_y$.
- $0 < H < W$. *Why?*
- Line equation: $H/W = (y-a_y)/(x-a_x)$
- $-W*(y-a_y) + H*(x - a_x) = 0$
- Define $F(x, y) = -2\ W*(y - a_y) + 2\ H*(x - a_x)$
  - equation of line is $F(x, y) = 0$

Bresenham Approach (3)

- Grid: Points on the grid are centers of pixels
- Let P a pixel which is currently set (chosen)
- Next step pixel: Candidates L (low) and U (up)
- Midpoint between them $M = (p_x+1, p_y+1/2)$

- if ideal line below M
  - Set next pixel L
  - (p_x+1, p_y)
- Else set next pixel U
  - (p_x+1, p_y+1)
- *how do decide?*



Bresenham Approach (4)

- if F (M_x, M_y) < 0, line below M, set L; else set U
- Need to calculate F(M_x, M_y). Or rather, changes in F

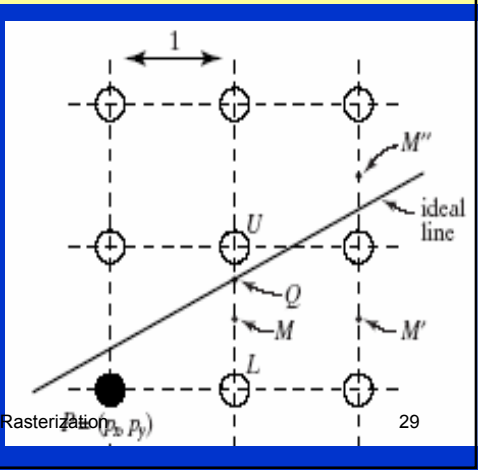
•Next M: M' or M"?

• if ideal line below M, next M will be M'

•M' = (p_x+2, p_y+1/2);

• else M"

• M" = (p_x+2, p_y+3/2).



Bresenham Approach (5)

- Case 1: line below M F(M_x, M_y) < 0
- Next M is M' (p_x+2, p_y+1/2)
- next F: F(p_x+2, p_y+1/2) = -2W(p_y+1/2 – a_y) + 2H(p_x+2-a_x)
- Calculate the change in F, from this step to next step:

□ ΔF = F(p_x+2, p_y+1/2) - F(p_x+1, p_y+1/2) =

□ ΔF = 2H(p_x+2-a_x) - 2H(p_x+1-a_x) = 2H

– F(M'$_x$, M'$_y$) = F(M_x,M_y) + 2H (constant integer)

Bresenham Approach (6)

- Case 2: line above M F(Mx, My) > 0
  - Next M is M" = (p_x+2, p_y+3/2)
- $\Delta F = F(p_x+2, p_y+3/2) - F(p_x+1, p_y+1/2) = -2(W-H)$
- $F(M"_x, M"_y) = F(M_x, M_y) - 2(W-H)$
- In either case, F changes by a constant integer:

– 2H if y was not incremented (negative F)

– -2(W-H) if y was incremented (positive F)

Bresenham Algorithm

- M = (a_x+1, a_y+1/2)
  - $F(a_x, a_y) = -2W(a_y+1/2-a_y) + 2H(a_x+1-a_x) = 2H-W$
  - Initial value: F = 2H-W
- For (i = a_x; x <= b_x; x++)
- {
  - Set pixel (x, y)
  - If F< 0 F += 2H; // no change in y
  - Else { y++;
- F += 2(H-W)
- }
- } //endFor

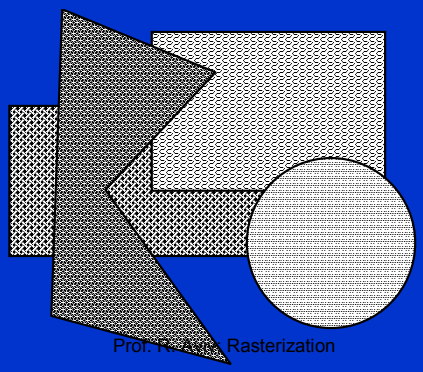
Bresenham Algorithm (2)

- Until now: $a_x < b_x$ and slope < 1
- Other cases: (slope >1; negative slopes) Exercises
  - tricks: replace roles of X and Y, and W with –W
- Drawing Patterned lines:
- Patterned lines (dotted or dashed lines)
  - store the desired line pattern in a bit mask
  - When a pixel is selected, consult the bit mask to check whether it should be drawn.

Regions in a pixmap
Polygon filling

Regions in a pixmap

- Region: a collection of pixels
  - lying next to one another in some fashion,
  - or being associated by some common propert

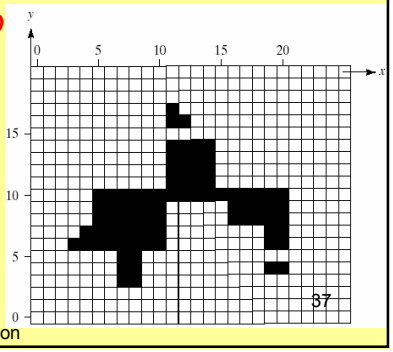


Defining Regions within pixmap

- A **symbolic** description:
- Via a property that all pixels in region *R* have:
- All pixels closer to a given point **A** than to any of the given points **B**, **C**, or **D**
- All pixels lying within a circle of radius r centered at **O**
- All pixels inside the polygon with specified vertices
- Methods of definition: List of rectangles or boundary path

Rectangle-Defined Regions (shapes)

- Region (e.g. all black pixels): a list of rectangles in pixmap
- Algorithm: scan line
  - Check every pixel if it has the property (e.g. black)
  - Look for **runs** (series of pixels with the property)
  - *What happens if set of runs in scan line differs from those in previous line?*
  - begin new set of rectangles
- Few rectangles if
  - span coherence (similar lines)
  - scan line coherence



Operations on regions

- *Translation*
  - traverse the list of rectangles
  - increment all (*x,y)* of vertices
- *Scaling (e.g. by 2)*
  - First translate it so that the first run is at the origin (0,0)
  - then double all values in the data structure.
  - translate it so first run comes to a desired position.

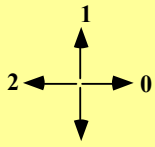
Path-Defined Regions

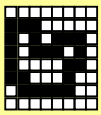
- Region defined by its boundary (a path).
- Path definitions:
  - Mathematical formula: e.g., $(x - 122)^2 + (y - 36)^2 = 25$
  - Polyline: sequence of pixel locations (x1, y1), (x2, y2),..., (xn, yn);
    - if closed ((x1, y1) = (xn, yn)), is a polygon.
  - Sequence of adjacent pixels (chain code):
  - a starting pixel, say (34, 67),
  - and a sequence of moves from pixel to pixel, such as "go up, go right, go down,...".

Path-Defined Regions (2)

- Example of chain codes:

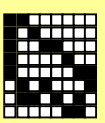
a).





11011223221122333303000 0

b).

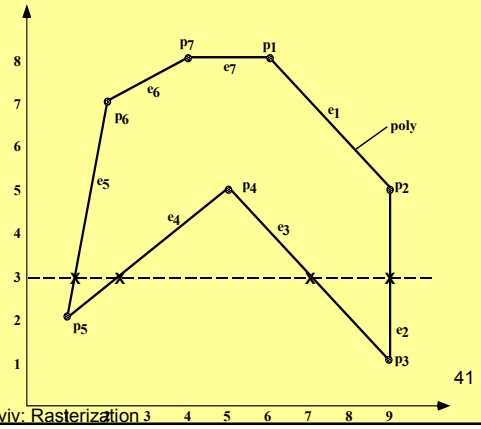


21243343346666760177 0

Filling Polygon-defined Regions

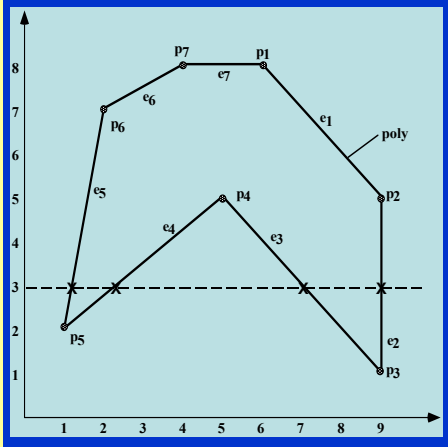
- polygon *P:* a set of vertices pixel addresses, $p_i = (x_i, y_i)$
- Filling algorithm: Scanline
- We have to find the intersections of the scan line, say, $y = 3$, with all the edges of *P*.

```
for(each scan line L)
{
   Find intersections of L
   sort intersections by x
   Fill runs between pairs
    of intersections
}
```



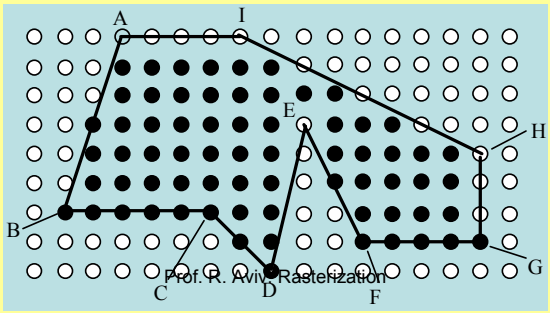
Example of Polygon Filling

- Scan line $y = 3$
- intersects edges e_2, e_3, e_4, e_5.
- intersection *x*-values are rounded up or down to integers, and sorted:
  - {1, 2, 7, 9}.
- Two runs are filled:
  - column 1 to 2 & column 7 to 9.



Intersections with Vertex and Edge Points

- If a scan line intersects a vertex (e.g. E)
- Rule: ignore intersections with *top* ends of edges
- If scanline intersect horizontal edge (e.g. BC)
- ignore edge
- No intersections occur at ***A*** & ***I***, one at ***B*** and at ***H***.

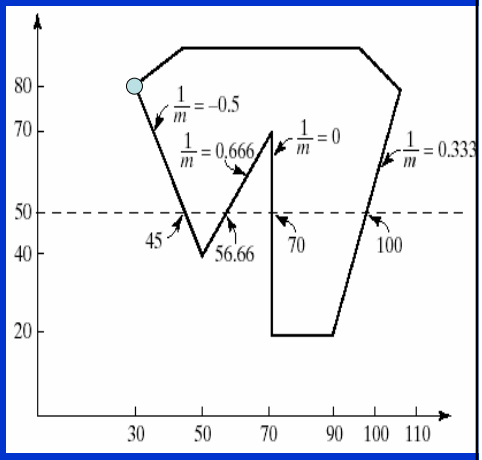


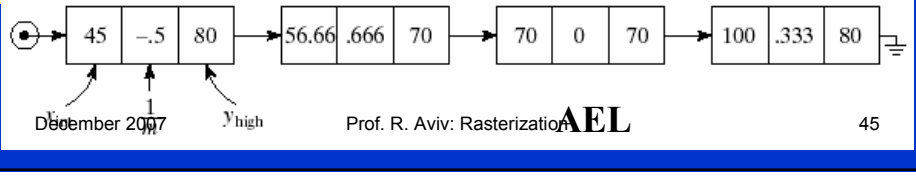
Improving Algorithm Performance

- use *edge coherence*
  - edges intersect scan line *y* likely to intersect next scanline
  - *x*-value of intersection migrates *in uniform increments* from scan line to scan line.
- maintain *Active Edge List* (AEL)
- AEL contains:
  - *x*-values of all the edge intersections for current scanline
    - in sorted order, so pairs of *x*-values define runs
  - Data for updating AEL for the next scanline

Improving Algorithm Performance (2)

- Scan-line $y = 50$
- intersects four edges
- at x = 45, 56.66, 70, 100
- AEL linked list of items
- item (one per edge):
  - x_{int}
  - inverse slope 1/m of edge
  - y_{top} of edge *why?*



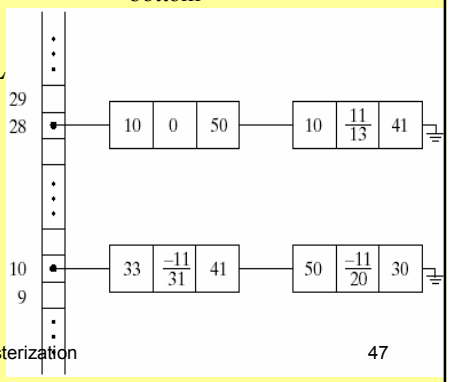


Basic step of the improved algorithm:

- Assume scanline y intersect an edge **e** at x_{int}
- Next consider scanline y + 1
  - If $y + 1 >= y_{top}(\underline{\mathbf{e}})$
    - Then scanline y + 1 does not intersect edge **e**
  - else scanline y+1 intersect edge **e** at $x_{int}+1/m$ (easy)
  - If $y+1= y_{bottom}(\underline{\mathbf{f}})$ scanline y+1 meets new edge **f**
  - Then edge **f** is added to Active Edge List *(update AEL)*
    - Entry added: $(x_{bottom}(\underline{\mathbf{f}}), 1/m(\underline{\mathbf{f}}), y_{top}(\underline{\mathbf{f}}))$
  - Might need to re-sort the entries of AEL *why?*

Updating AEL via Edge Table (ET)

- Pre-prepared table ET
- For each y, a linked list
- Entries in list y: info about edges with $y_{bottom} = y$
  - which will be copied to AEL
  - when scanline y starts
  - $(x_{bottom}, 1/m, y_{top})$



Skeleton Code for Polygon-Fill

```
Prepare ET when traversing polygon to eliminate horizontals
AEL = NULL;                          // AEL is initially empty
for (y = 0; y <= maxRow; y++)  {
   <add all edges in ET[y] to AEL>
   if( AEL != NULL) {                // any edges to process?
        <sort AEL by x_int value>
        <fill runs, identified by AEL info, with pixel values>
        <delete from AEL records where y_top == y>
        <update each x_int value by its inverse slope>
   }// end if
} // end for()
```