CS 362: Computer Graphics

Scan Conversion

Dr. Samit Bhattacharya
Dept. of Comp. Sc. & Engg.
IIT Guwahati, Assam, India

What

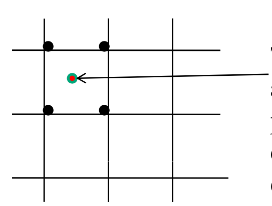
- The pipeline stages covered so far assumed continuous space
  - Methods considered points without any constraint on the coordinates – can be any real number
- To draw something on the screen, we need to consider pixel grid
  - Discreet space

What

- We need to map the representations from continuous to discreet space
- The mapping process is called *scan conversion*
  - Also called *rasterization*
- We will have a look at scan conversion of
  - Point
  - Line
  - Circle

Point Scan Conversion

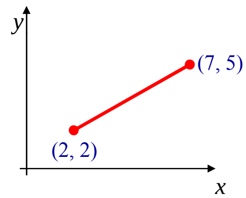
- Trivial: simply round off to the nearest pixel position



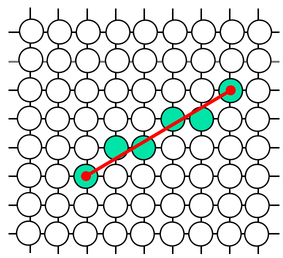
The point can be mapped to any of the four pixel positions, depending on the coordinate value
e.g. (1.1, 2.6) maps to (1, 3)

Line Scan Conversion

- A line segment is defined by the coordinate positions of the line end-points



- What happens when we try to draw this on a pixel based display?
  - How to choose the correct pixels



Line Equation

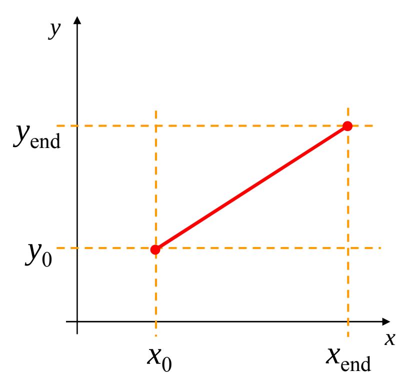
- Slope-intercept line equation

$$y = m \cdot x + b$$

where

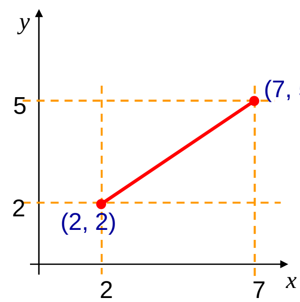
$$m = \frac{y_{end} - y_0}{x_{end} - x_0}$$

$$b = y_0 - m \cdot x_0$$

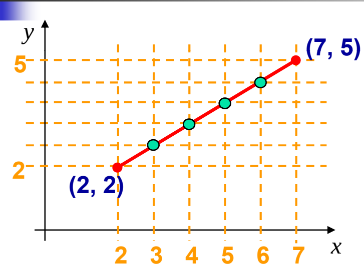


A Very Simple Solution

- Simply work out the corresponding y coordinate for each unit x coordinate
- Let's consider the following example



A Very Simple Solution (cont…)



- First work out m and b:

$$m = \frac{5-2}{7-2} = \frac{3}{5}$$

$$b = 2 - \frac{3}{5} * 2 = \frac{4}{5}$$

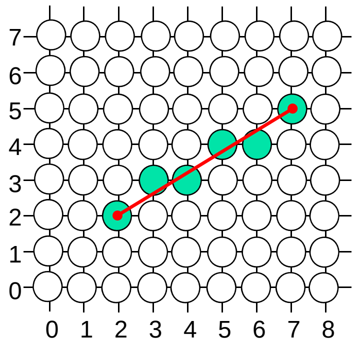
- Now for each x value work out the y value

$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2\frac{3}{5} \qquad y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3\frac{1}{5}$$

$$y(5) = \frac{3}{5} \cdot 5 + \frac{4}{5} = 3\frac{4}{5} \qquad y(6) = \frac{3}{5} \cdot 6 + \frac{4}{5} = 4\frac{2}{5}$$

A Very Simple Solution (cont…)

- Now just round off the results and turn on these pixels to draw our line



$$y(3) = 2\frac{3}{5} \approx 3$$

$$y(4) = 3\frac{1}{5} \approx 3$$

$$y(5) = 3\frac{4}{5} \approx 4$$

$$y(6) = 4\frac{2}{5} \approx 4$$

A Very Simple Solution (cont…)

- However, this approach is just way too slow
- In particular,
  - The equation $y = mx + b$ requires the multiplication of m by x
  - Rounding off the resulting y coordinates
- We need a faster solution

A Quick Note About Slopes

- In the previous example, we chose to solve the line equation to get *y* coordinate for each *x* coordinate
- What if we had done it the other way around?
- So this gives us: $x = \frac{y - b}{m}$

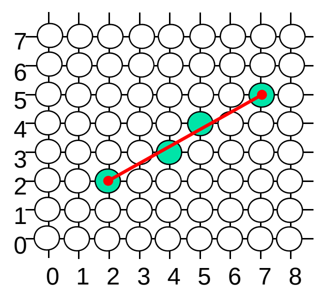
where: $m = \frac{y_{end} - y_0}{x_{end} - x_0}$ and $b = y_0 - m \cdot x_0$

A Quick Note About Slopes (cont…)

- Leaving out the details this gives us:

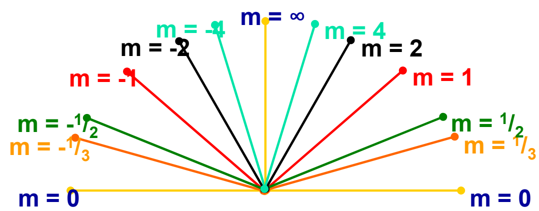
$x(3) = 3\frac{2}{3} \approx 4 \quad x(4) = 5\frac{1}{3} \approx 5$

- We can see easily that this line doesn't look very good!
- We choose which way to work out the line pixels based on the slope of the line



A Quick Note About Slopes (cont…)

- If the slope of a line is between -1 and 1, work out *y* coordinates based on *x* coordinates
- Otherwise do the opposite – *x* coordinates are computed based on *y* coordinates



The DDA Algorithm

- The *digital differential analyzer* (DDA) algorithm takes an incremental approach in order to speed up scan conversion
- Consider the list of points that we determined for the line in our previous example:

  $(2, 2), (3, 2^3/_5), (4, 3^1/_5), (5, 3^4/_5), (6, 4^2/_5), (7, 5)$
- Notice that as the *x* coordinates go up by one, the *y* coordinates simply go up by the slope of the line
  - This is the key insight in the DDA algorithm

The DDA Algorithm (cont…)

- When m is between -1 and 1, begin at the first point in the line and, by incrementing *x* by 1, calculate the corresponding *y* as follows

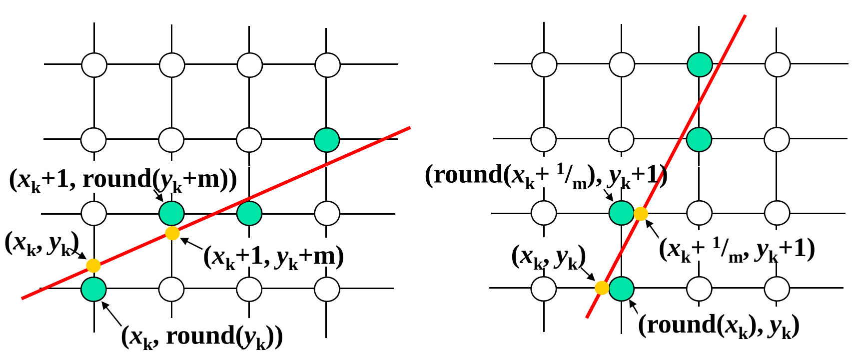
$$y_{k+1} = y_k + m$$

- When m is outside these limits, increment *y* by 1 and calculate the corresponding *x* as follows

$$x_{k+1} = x_k + \frac{1}{m}$$

The DDA Algorithm (cont…)

- Again the values calculated by the equations used by the DDA algorithm must be rounded

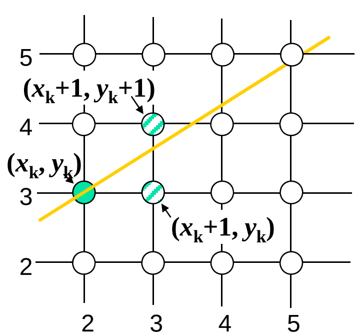


The DDA Algorithm Summary

- The DDA algorithm is much faster than our previous attempt
  - In particular, there are no longer any multiplications involved
- However, there are still two big issues
  - Accumulation of round-off errors can make the pixelated line drift away from what was intended
  - The rounding operations and floating point arithmetic involved are time consuming

The Bresenham Line Algorithm

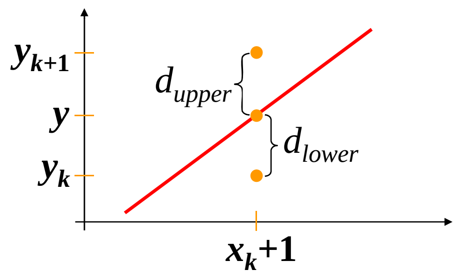
- Move across the *x* axis in unit intervals and at each step choose between two different *y* coordinates



- For example, from position (2, 3) we have to choose between (3, 3) and (3, 4)
- We would like the point that is closer to the original line

Derivation

- At sample position x_k+1 the vertical separations from the mathematical line are labelled d_{upper} and d_{lower}



• The y coordinate on the mathematical line at x_k+1 is:

$$y = m(x_k + 1) + b$$

Derivation

- So, d_{upper} and d_{lower} are given as follows:

$$\begin{aligned} d_{lower} &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned}$$

and

$$\begin{aligned} d_{upper} &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned}$$

- We can use these to make a simple decision about which pixel is closer to the mathematical line

Derivation

- This simple decision is based on the difference between the two pixel positions:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

- Let's substitute m with $\Delta y/\Delta x$ where Δx and Δy are the differences between the end-points:

$$\begin{aligned}\Delta x(d_{lower} - d_{upper}) &= \Delta x(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c\end{aligned}$$

Derivation

- So, a decision parameter p_k for the kth step along a line is given by:

$$\begin{aligned}p_k &= \Delta x(d_{lower} - d_{upper}) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c\end{aligned}$$

- The sign of the decision parameter p_k is the same as that of $d_{lower} - d_{upper}$
- If p_k is negative, then choose the lower pixel, otherwise choose the upper pixel

Derivation

- Remember coordinate changes occur along the *x* axis in unit steps, so we can do everything with integer calculations
- At step k+1 the decision parameter is given as:

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

- Subtracting p_k from this we get:

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

Derivation

- But, x_{k+1} is the same as x_k+1 so:

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

- where y_{k+1} - y_k is either 0 or 1 depending on the sign of p_k
- The first decision parameter p0, evaluated at (x0, y0), is given as:

$$p_0 = 2\Delta y - \Delta x$$

The Bresenham Line Algorithm

BRESENHAM'S LINE DRAWING ALGORITHM

1. Input the two line end-points, storing the left end-point in (x_0, y_0)
2. Plot the point (x_0, y_0)
3. Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and get the first value for the decision parameter as:

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point to plot is (x_k+1, y_k) and: $p_{k+1} = p_k + 2\Delta y$

The Bresenham Line Algorithm

Otherwise, the next point to plot is (x_k+1, y_k+1) and:

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 $(\Delta x - 1)$ times

- The algorithm and derivation above assumes slopes are less than 1 ($|m| < 1.0$), for other slopes we need to adjust the algorithm slightly

Circle Scan Conversion

- The equation for a circle is:

$$x^2 + y^2 = r^2$$

where r is the radius of the circle

- So, we can write a simple circle drawing algorithm by solving the equation for y at unit x intervals using:

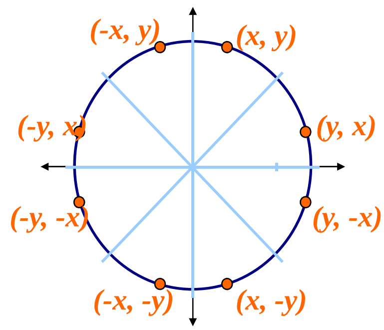
$$y = \pm\sqrt{r^2 - x^2}$$

Circle Scan Conversion

- Unsurprisingly this is not a brilliant solution
- Firstly, the resulting circle has large gaps where the slope approaches the vertical
- Secondly, the calculations are not very efficient
  - The square (multiply) operations
  - The square root operation – try really hard to avoid these!
- We need a more efficient, more accurate solution

Eight-Way Symmetry

- Circles centred at (*0*, *0*) have *eight-way symmetry* – this fact can be exploited to design an efficient algorithm

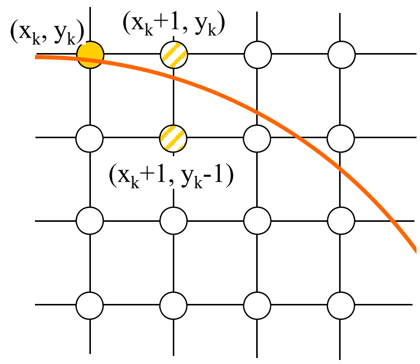


Mid-Point Circle Algorithm

- An incremental algorithm for drawing circles
- Algorithm calculates pixels only for the top right eighth of the circle
- Other points are derived using the eight-way symmetry

Mid-Point Circle Algorithm (cont…)

- Assume that we have just plotted point (x_k, y_k)
- The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)
- We would like to choose the point that is nearest to the actual circle
  - So how do we make this choice?



Mid-Point Circle Algorithm (cont…)

- Let's re-jig the equation of the circle slightly to give us:

$$f_{circ}(x, y) = x^2 + y^2 - r^2$$

- The equation evaluates as follows:

$$f_{circ}(x, y) \begin{cases} < 0, \text{ if } (x, y) \text{ is inside the circle boundary} \\ = 0, \text{ if } (x, y) \text{ is on the circle boundary} \\ > 0, \text{ if } (x, y) \text{ is outside the circle boundary} \end{cases}$$

- By evaluating this function at the midpoint between the candidate pixels we can make our decision

Mid-Point Circle Algorithm (cont…)

- Assuming we have just plotted the pixel at (x_k, y_k) so we need to choose between (x_k+1, y_k) and (x_k+1, y_k-1)
- Our decision variable can be defined as:

$$p_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$
$$= (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2$$

- If $p_k < 0$ the midpoint is inside the circle and the pixel at y_k is closer to the circle
- Otherwise the midpoint is outside and y_k-1 is closer

Mid-Point Circle Algorithm (cont…)

- To ensure things are as efficient as possible we can do all of our calculations incrementally
- First consider: $p_{k+1} = f_{circ}\left(x_{k+1} + 1, y_{k+1} - \tfrac{1}{2}\right)$

$$= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \tfrac{1}{2}\right)^2 - r^2$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

where y_{k+1} is either y_k or y_k-1 depending on the sign of p_k

Mid-Point Circle Algorithm (cont…)

- The first decision variable is given as

$$p_0 = f_{circ}(1, r - \tfrac{1}{2})$$
$$= 1 + (r - \tfrac{1}{2})^2 - r^2$$
$$= \tfrac{5}{4} - r$$

- Then if $p_k < 0$ then the next decision variable is given as: $p_{k+1} = p_k + 2x_{k+1} + 1$
- If $p_k > 0$ then the decision variable is

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_k + 1$$

Mid-Point Circle Algorithm

MID-POINT CIRCLE ALGORITHM

- Input radius r and circle centre (x_c, y_c), then set the coordinates for the first point on the circumference of a circle centred on the origin as:

$$(x_0, y_0) = (0, r)$$

- Calculate the initial value of the decision parameter as: $p_0 = \tfrac{5}{4} - r$
- Starting with $k = 0$ at each position x_k, perform the following test. If $p_k < 0$, the next point along the circle centred on $(0, 0)$ is (x_k+1, y_k) and:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

Mid-Point Circle Algorithm (cont…)

- Otherwise the next point along the circle is (x_k+1, y_k-1) and $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$
- Determine symmetry points in the other seven octants
- Move each calculated pixel position (x, y) onto the circular path centred at (x_c, y_c) to plot the coordinate values:

$$x = x + x_c \qquad y = y + y_c$$

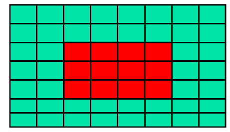
- Repeat steps 3 to 5 until $x >= y$

Fill-Area Scan Conversion

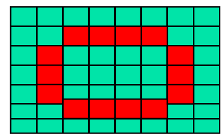
- *Region filling*: "coloring in" a definite image area or region
- Definition at pixel or geometric level
- Pixel level definitions
  - Boundary defined: region defined in terms of boundary pixels
  - Interior defined: region defined in terms of all the pixels within the interior
- Geometric region (usually for polygons): region defined in terms of edges and vertices

Region Definitions

- Pixel level - mostly used in
  - Applications having complex boundaries
  - Interactive painting systems
- The second mostly used in general graphics packages



Interior-defined region



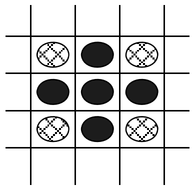
Boundary-defined region

Seed Fill (Boundary Fill) Algorithm

- Assume at least one pixel interior to a polygon or region is known – called *seed*
- Regions boundary defined
  - For interior defined regions – flood-fill algorithm
- Two conventions
  - 4-connected: each pixel connected to four adjacent pixels (Top, Bottom, Left, Right)
  - 8-connected: each pixel connected to eight adjacent pixels (Top, Top Left, Top Right, Bottom, Bottom Left, Bottom Right, Left, Right)

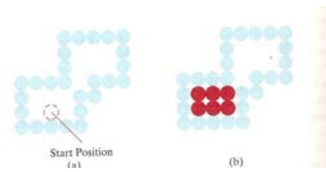
Seed Fill (Boundary Fill) Algorithm

- Pixel Adjacency



4-connected

8-connected



- Boundary-Fill Algorithm
  - Starting at a point inside the figure and painting the interior in a specified color or intensity

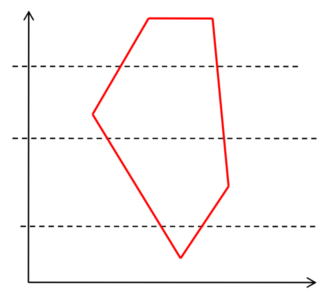
A Simple Seed Fill Algorithm

- *Push the seed pixel onto the stack*
- While the stack is not empty
  - Pop a pixel from the stack
  - Set the pixel to the required value
  - For each of the 4 connected pixels adjacent to the current pixel
    - If it is a boundary pixel or if it has already been set to the required value, ignore it
    - Else push it onto the stack
- Easy to modify for 8-connected pixels
  - It also works with holes in the polygons

Scan-Line Polygon Fill

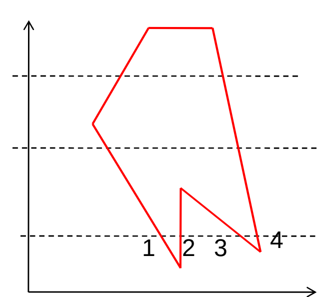
Do the following for every scan line:

1. Compute the intersection of the current scan line with every polygon edge
2. Sort the intersections in increasing order of the *x* coordinate
3. Draw every pixel that lies between each pair of intersections



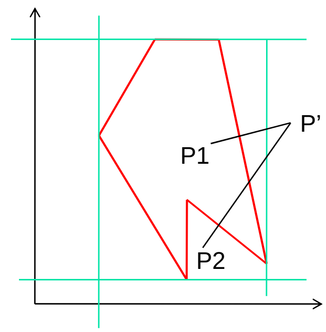
Problem

- What will happen in case of concave polygons
  - We have pairs of intersection points (1,2), (2,3) and (3,4)
  - We should not set pixels between (2,3) – it's outside
  - How can we decide?



A Simple Inside-Outside Test

- Suppose we want to know if a point P is inside
  - Determine the bounding box (max and min x and y extents)
  - Choose a point P’ outside the bounding box
  - Join P and P’
  - If the line intersects the polygon edges even number of times, P is outside. Else P is inside



P1 inside, P2 outside

Determining Edge-Scanline Intersection

- If a scan line intersects an edge at (x1, y1), the next scan line will intersect the same edge at (x1+1/m, y1+1)
  - m is the slope of the edge

Determining Edge-Scanline Intersection

- We actually don't need to calculate 1/m-a floating point operation
  - Keep a counter C, initially set it to 0
  - Increment counter each time 1/m added to x by $2\Delta x$, till $C \geq \Delta y$
  - Increment x by 1 (y=current scan line), reset C to $C- 2\Delta y$
    - Till this point, keep x same, increment y
  - Continue till ymax

Character Rendering

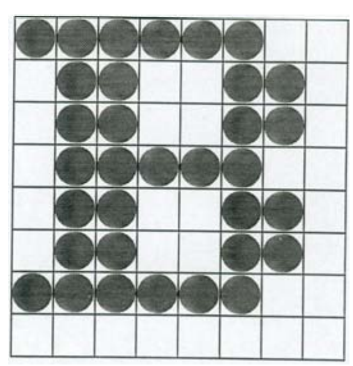
- Letters, digits, non-alphanumeric
- Terms borrowed from typography
  - Typeface: a particular style of characters (Times New Roman, Courier, Arial)
  - Font: cast metal character form to print typeface
  - In CG, the terms are used synonymously
- Fonts
  - Two broad types: Serif, Sans-Serif
  - Can vary in appearance: Normal, bold, italic
- Rendering techniques
  - Bitmap, Outlined

About "Point"

- Font size usually denoted in point (e.g. 10-point, 12-point)
  - Denotes height of the characters in inches
- A term from typography
  - Smallest unit of measure
- We are concerned with desk-top publishing (DTP) point, also called the PostScript point
  - Not the original typographical point
- 1 DTP point = 1/72 of an inch or approx 0.0139 inch

Bitmapped Fonts

- Represents each character as the *on* pixels in a bi-level pixel grid pattern known as *bitmap*
- Advantages
  - Simple
  - Fast, since the characters are defined in already scan converted form, no further processing is required



Bitmapped Fonts

- Disadvantages
  - More storage: for each character, we need to store the bitmap
  - Although different style/sizes can be generated from one font, the result is not satisfactory
  - Bitmap font size dependent on resolution (e.g. a 12 pixel high bitmap will produce a 12-point character in a 72 pixels/inch resolution, while the same bitmap will produce 9-point character in a 96 pixels/inch resolution)

Outlined Fonts

- Character outline is defined using graphical primitives (e.g. line, arcs)
  - PostScript by Adobe
  - Less storage (no need to store bitmaps any more)
  - Good for styles/sizes
    - Scaling transformation to resize
    - Shearing transformation to italicize etc
  - Slower (since scan conversion is involved)