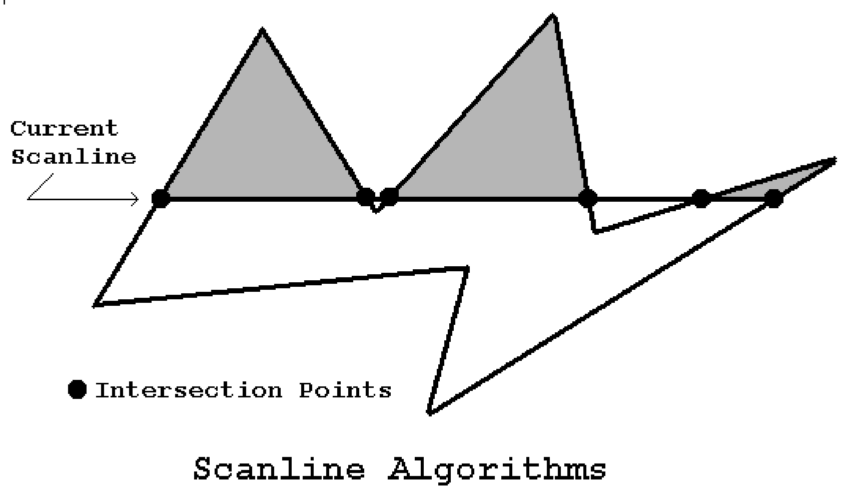
Scanline Polygon Fill Algorithm

- Look at individual scan lines
- Compute intersection points with polygon edges
- Fill between alternate pairs of intersection points

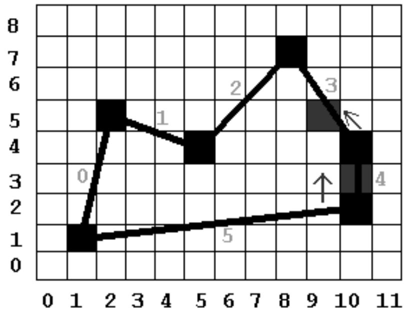


Scanline Polygon Fill Algorithm

1. Set up edge table from vertex list; determine range of scanlines spanning polygon (miny, maxy)
2. Preprocess edges with nonlocal max/min endpoints
3. Set up activation table (bin sort)
4. For each scanline spanned by polygon:
   - Add new active edges to AEL using activation table
   - Sort active edge list on x
   - Fill between alternate pairs of points (x,y) in order of sorted active edges
   - For each edge e in active edge list:
     - If (y != ymax[e]) Compute & store new x (x+=1/m)
     - Else Delete edge e from the active edge list

Scanline Polygon Fill Algorithm Example

`poly={1,1, 2,5, 5,4, 8,7, 10,4, 10,2, 1,1}`



edge	x1	y1	x2	y2	sgn(Dy)
0	1	1	2	5	+
1	2	5	5	4	−
2	5	4	8	7	+
③	8	7	10	4	− ←
④	10	4	10	2	− ←
5	10	2	1	1	−
0	1	1	2	5	+

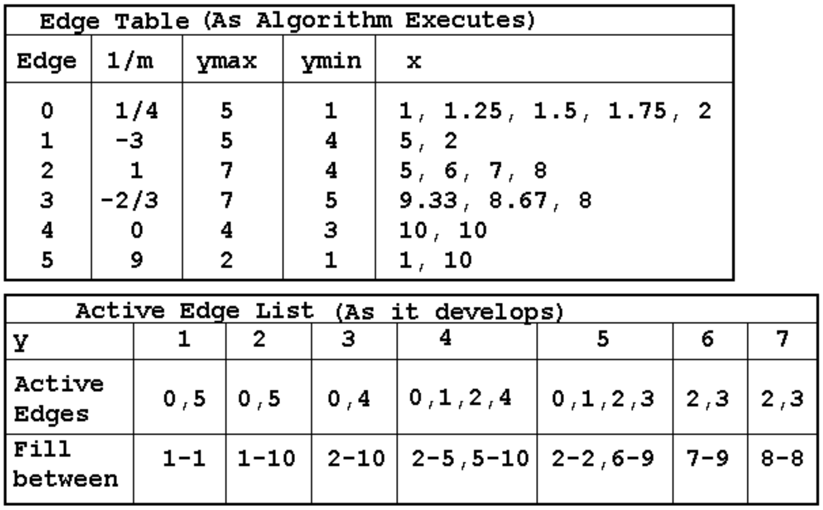
Activation Table

y	1	2	3	4	5	6	7
activated edge #s	0 5		4	1 2	3		

Edge Table

edge	1/m	ymin	x	ymax
0	1/4	1	1	5
1	-3	4	5	5
2	1	4	5	7
3	-2/3	4→5	10→9 1/3	7
4	0	2→3	10→10	4
5	9	1	1	2

Scanline Poly Fill Alg. (with example Data)



Edge Table (As Algorithm Executes)

Edge	1/m	ymax	ymin	x
0	1/4	5	1	1, 1.25, 1.5, 1.75, 2
1	-3	5	4	5, 2
2	1	7	4	5, 6, 7, 8
3	-2/3	7	5	9.33, 8.67, 8
4	0	4	3	10, 10
5	9	2	1	1, 10

Active Edge List (As it develops)

y	1	2	3	4	5	6	7
Active Edges	0,5	0,5	0,4	0,1,2,4	0,1,2,3	2,3	2,3
Fill between	1-1	1-10	2-10	2-5,5-10	2-2,6-9	7-9	8-8

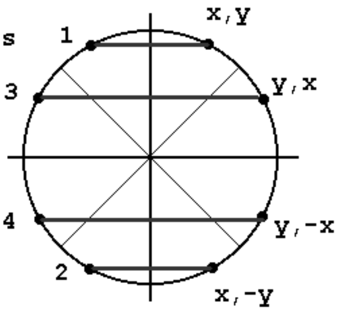
Adapting Scanline Polygon Fill to other primitives

- Example: a circle or an ellipse
  - Use midpoint algorithm to obtain intersection points with the next scanline
  - Draw horizontal lines between intersection points
  - Only need to traverse part of the circle or ellipse

Scanline Circle Fill Algorithm

```
Modify midpoint circle algorithm
For each step draw 4 horizontal lines

Line4(x,y,h,k)
{
 Line(-x+h,y+k,x+h,y+k);    // 1
 Line(-x+h,-y+k,x+h,-y+k);  // 2
 Line(-y+h,x+k,y+h,x+k);    // 3
 Line(-y+h,-x+k,y+h,-x+k);  // 4
}
```



The Scanline Boundary Fill Algorithm for Convex Polygons

Select a Seed Point (x,y)

Push (x,y) onto Stack

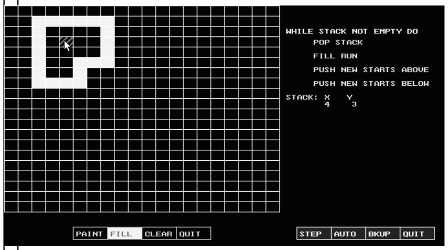
While Stack is not empty:

- Pop Stack (retrieve x,y)
- Fill current run y (iterate on x until borders are hit)
- Push left-most unfilled, nonborder pixel above
  -->new "above" seed
- Push left-most unfilled, nonborder pixel below
  -->new "below" seed

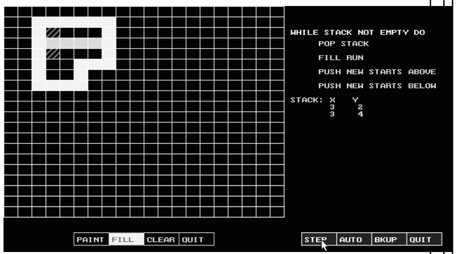
Demo of Scanline Polygon Fill Algorithm vs. Boundary Fill Algorithm

- Polyfill Program
    - Does:
        - Boundary Fill
        - Scanline Polygon Fill
        - Scanline Circle with a Pattern
        - Scanline Boundary Fill (Dino Demo)

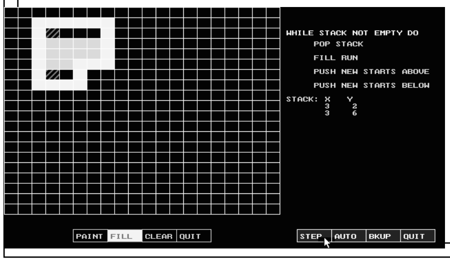
Dino Demo of Scanline Boundary Fill Algorithm



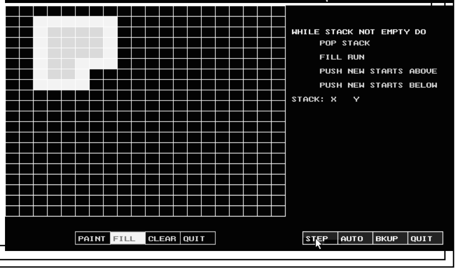
```
WHILE STACK NOT EMPTY DO
     POP STACK
     FILL RUN
     PUSH NEW STARTS ABOVE
     PUSH NEW STARTS BELOW
STACK: X   Y
       4   3
```



```
WHILE STACK NOT EMPTY DO
     POP STACK
     FILL RUN
     PUSH NEW STARTS ABOVE
     PUSH NEW STARTS BELOW
STACK: X   Y
       3   2
       3   4
```



```
WHILE STACK NOT EMPTY DO
     POP STACK
     FILL RUN
     PUSH NEW STARTS ABOVE
     PUSH NEW STARTS BELOW
STACK: X   Y
       3   2
       3   6
```

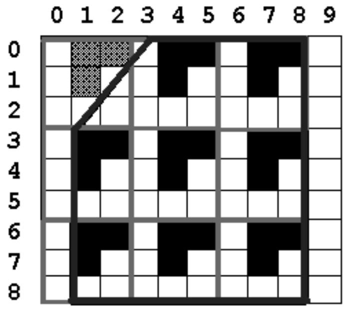


```
WHILE STACK NOT EMPTY DO
     POP STACK
     FILL RUN
     PUSH NEW STARTS ABOVE
     PUSH NEW STARTS BELOW
STACK: X   Y
```

Pattern Filling

- Represent fill pattern with a <u>Pattern Matrix</u>
- Replicate it across the area until covered by non-overlapping copies of the matrix
  - Called Tiling

Pattern Filling--Pattern Matrix

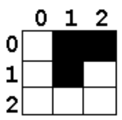


Tiling

Pattern Matrix
W=3

		0	1	2
	0	0	1	1
H=3	1	0	1	0
	2	0	0	0

Pattern



x	0	1	2	3	4	5	6	7	8
x posn	0	1	2	0	1	2	0	1	2
y	0	1	2	3	4	5	6	7	8
y posn	0	1	2	0	1	2	0	1	2

In general, posn in matrix:
xpos = x%W, ypos = y%H

Using the Pattern Matrix

- Modify fill algorithm
- As (x,y) pixel in area is examined:

```
if(pat_mat[x%W][y%H] == 1)
    SetPixel(x,y);
```

A More Efficient Way

Store pat_matrix as a 1-D array of bytes or words, e.g., WxH

y%H --> byte or word in pat_matrix

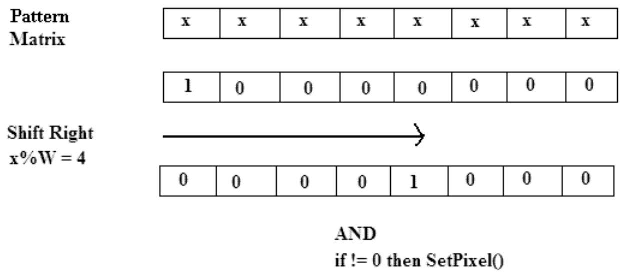
Shift a mask by x%W

e.g. 10000000 for 8x8 pat_matrix

--> position of bit in byte/word of pat_matrix

"AND" byte/word with shifted mask

if result != 0, Set the pixel



Color Patterns

- Pattern Matrix contains color values
- So read color value of pixel directly from the Pattern Matrix:

```
SetPixel(x, y, pat_mat[x%W][y%H])
```

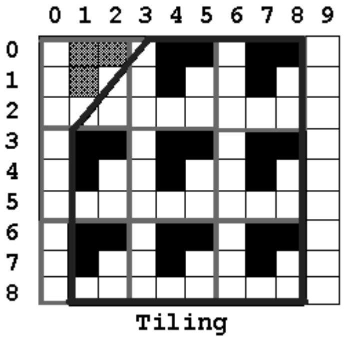
Moving the Filled Polygon

- As done above, pattern doesn’t move with polygon
- Need to “anchor” pattern to polygon
- Fix a polygon vertex as “pattern reference point”, e.g., (x0,y0)

```
If (pat_matrix[(x-x0)%W][(y-y0)%H]==1)
    SetPixel(x,y)
```

- Now pattern moves with polygon

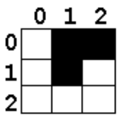
Pattern Filling--Pattern Matrix



Pattern Matrix
W=3

		0	1	2
	0	0	1	1
H=3	1	0	1	0
	2	0	0	0

Pattern



x	0	1	2	3	4	5	6	7	8
x posn	0	1	2	0	1	2	0	1	2
y	0	1	2	3	4	5	6	7	8
y posn	0	1	2	0	1	2	0	1	2

In general, posn in matrix:
xpos = x%W, ypos = y%H

OpenGL Line/Polygon Attributes

- Line Width
  - glLineWidth(width);
    - width: floating pt value rounded to nearest integer
- Line Style
  - glLineStipple(repeat factor, pattern);
    - pattern: 16-bit integer describes style (1 on, 0 off)
    - repeat factor: integer expressing how many times each bit in pattern repeats before next bit is applied – default 1
  - Must activate Line Style feature
    - glEnable(GL_LINE_STIPPLE)

Other OpenGL Line Effects

- Color Gradations
  - Vary color smoothly between line endpoints
  - Assign different color to each endpoint
    - System interpolates between colors

```
glShadeModel(GL_SMOOTH);
glBegin(GL_LINES);
   glColor3f(0.0, 0.0, 1.0);   // one end blue
   glVertex2i(50, 50);
   glColor3f(1.0, 0.0, 0.0);   // other end red
   glVertex2i(100, 100);
glEnd();
```

Area Fill in OpenGL

- Only available for convex polygons
- Steps:
  - Define fill pattern
  - Invoke polygon-fill routine
  - Activate polygon-fill feature in OpenGL
  - Describe the polygon(s) to be filled

OpenGL Pattern Fill

- Default: convex polygon displayed in solid color using current color setting
- Pattern fill:
  - – Use a 32 X 32 bit mask
    - 1: pixel set to current color
    - 0: background color
    - Glubyte fp[ ] = {0xff,0xff,0xff,0xff,0,0,0,0…}
      - – Bottom row first
  - – glPolygonStipple(fp);
  - – glEnable(GL_POLYGON_STIPPLE);

OpenGL Interpolation Patterns

- Can assign different colors to vertices
- OpenGL will interpolate interior colors

```
glShadeModel(GL_SMOOTH);
glBegin(GL_POLYGON)
    glColor3f(0.0, 0.0, 1.0);   // one vertex blue
    glVertex2i(25, 25);
    glColor3f(1.0, 0.0, 0.0);   // another red
    glVertex2i(75, 75);
    glColor3f(0.0,1.0, 0.0);    // last one green
    glVertex2i(75, 25);
glEnd();
```

Geometric Transformations

- Moving objects relative to a stationary coordinate system
- Common transformations:
  - Translation
  - Rotation
  - Scaling
- Implemented using vectors and matrices

Quick Review of Matrix Algebra

- Matrix--a rectangular array of numbers
- a_{ij}: element at row i and column j
- Dimension: m x n

  m = number of rows

  n = number of columns

A Matrix

An m x n matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & & & \ddots & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

Degenerate case: m = 1 (a row vector)

$$V = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \end{bmatrix}$$ or:

$$V = \begin{bmatrix} a_{1} & a_{2} & a_{3} & \cdots & a_{n} \end{bmatrix}$$

Vectors and Scalars

Degenerate Case (n=1) a column vector--

$$V = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \\ a_{m1} \end{bmatrix} \quad \text{or:} \quad V = \begin{bmatrix} a_{1} \\ a_{2} \\ a_{3} \\ \vdots \\ a_{m} \end{bmatrix}$$

Point in space
(x,y) or (x,y,z)--
Use vectors:

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

2D 3D

Transpose of a Matrix A^T

$a^T_{ij} = a_{ji}$ The transpose of a row vector is a column vector.

Degenerate Case: m=n=1, a scalar

$S = a_{11}$

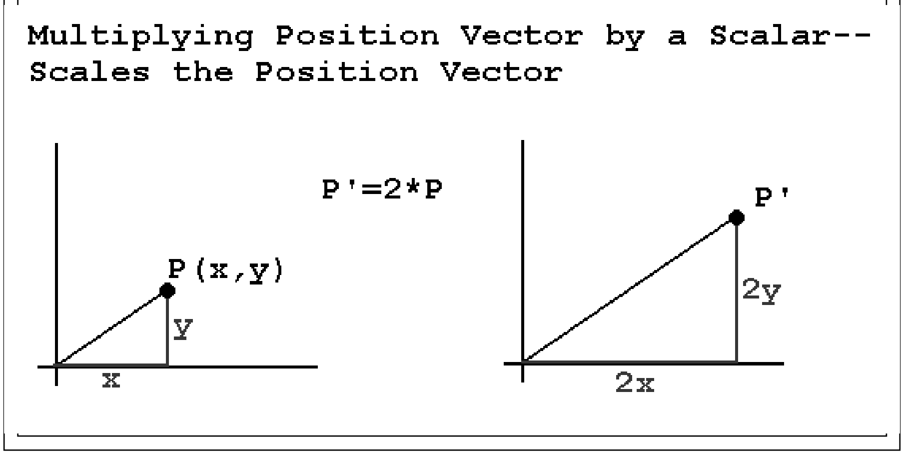
Matrix Operations-- Multiplication by a Scalar

C = k*A

$c_{ij} = k * a_{ij}$, $1 <= i <= m$, $1 <= j <= n$

- Example: multiplying position vector by a constant:
  - Multiplies each component by the constant
  - Gives a scaled position vector (k times as long)

Example of Multiplying a Position Vector by a Scalar



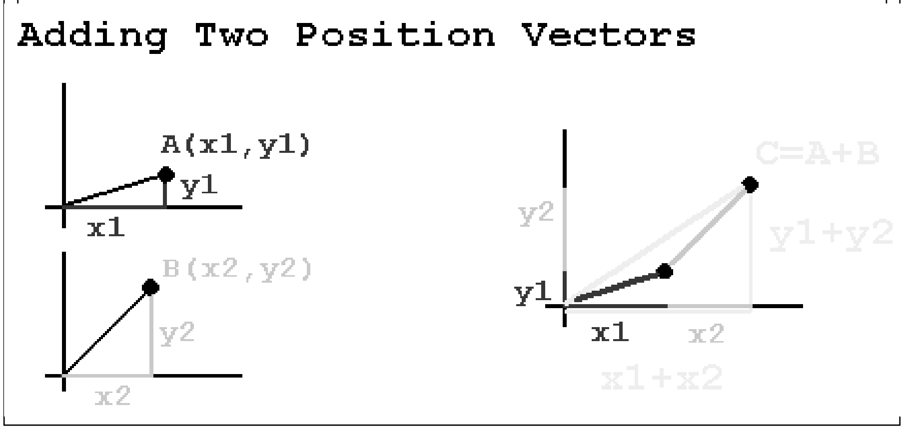
Multiplying Position Vector by a Scalar-- Scales the Position Vector

Adding two Matrices

- Must have the same dimension
- C = A + B

  $c_{ij} = a_{ij} + b_{ij}$, $1<=i<=m$, $1<=j<=n$
- Example: adding two position vectors
  - Add the components
  - Gives a vector equal to the net displacement

Adding two Position Vectors: Result is the Net Displacement

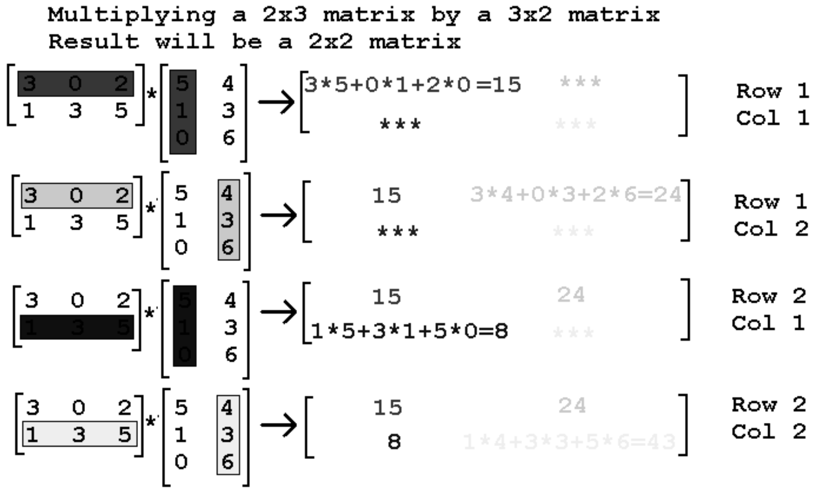


Multiplying Two Matrices

- m x n = (m x p) * (p x n)
- C = A * B
- $c_{ij} = \sum a_{ik} * b_{kj}$, 1<=k<=p
- In other words:
  - To get element in row i, column j
    - Multiply each element in row i by each corresponding element in column j
    - Add the partial products

Matrix Multiplication

An Example



Multiplying a 2x3 matrix by a 3x2 matrix
Result will be a 2x2 matrix

$$\begin{bmatrix}3&0&2\\1&3&5\end{bmatrix} * \begin{bmatrix}5&4\\1&3\\0&6\end{bmatrix} \rightarrow \begin{bmatrix}3*5+0*1+2*0=15 & *** \\ *** & ***\end{bmatrix}$$

Row 1
Col 1

$$\begin{bmatrix}3&0&2\\1&3&5\end{bmatrix} * \begin{bmatrix}5&4\\1&3\\0&6\end{bmatrix} \rightarrow \begin{bmatrix}15 & 3*4+0*3+2*6=24 \\ *** & ***\end{bmatrix}$$

Row 1
Col 2

$$\begin{bmatrix}3&0&2\\1&3&5\end{bmatrix} * \begin{bmatrix}5&4\\1&3\\0&6\end{bmatrix} \rightarrow \begin{bmatrix}15 & 24 \\ 1*5+3*1+5*0=8 & ***\end{bmatrix}$$

Row 2
Col 1

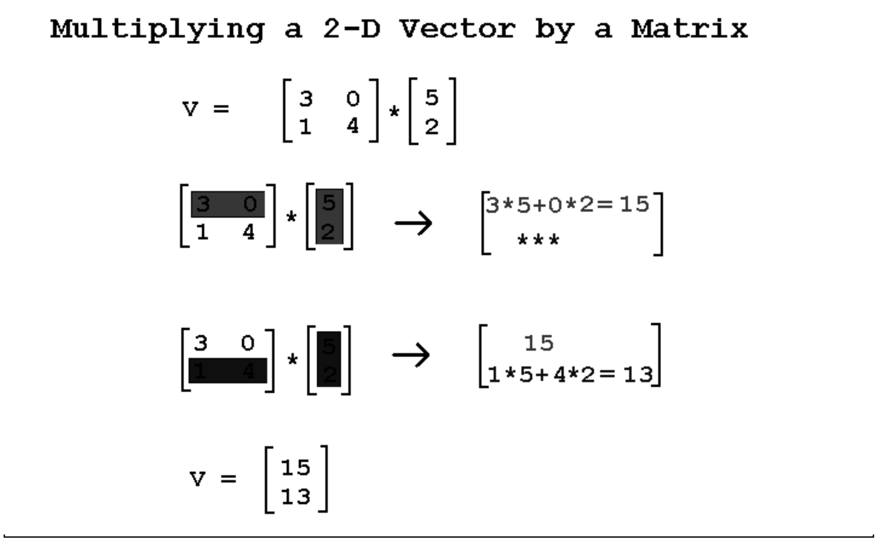
$$\begin{bmatrix}3&0&2\\1&3&5\end{bmatrix} * \begin{bmatrix}5&4\\1&3\\0&6\end{bmatrix} \rightarrow \begin{bmatrix}15 & 24 \\ 8 & 1*4+3*3+5*6=43\end{bmatrix}$$

Row 2
Col 2

Multipy a Vector by a Matrix

- $V' = A*V$
- If V is a m-dimensional column vector, A must be an m x m matrix
- $V'_i = \Sigma\, a_{ik} * v_k$, $1<=k<=m$
  - So to get element i of product vector:
    - Multiply each row i matrix element by each corresponding element of the vector
    - Add the partial products

An Example



Multiplying a 2-D Vector by a Matrix

$$V = \begin{bmatrix} 3 & 0 \\ 1 & 4 \end{bmatrix} * \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 0 \\ 1 & 4 \end{bmatrix} * \begin{bmatrix} 5 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 3*5+0*2=15 \\ *** \end{bmatrix}$$

$$\begin{bmatrix} 3 & 0 \\ 1 & 4 \end{bmatrix} * \begin{bmatrix} 5 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 15 \\ 1*5+4*2=13 \end{bmatrix}$$

$$V = \begin{bmatrix} 15 \\ 13 \end{bmatrix}$$

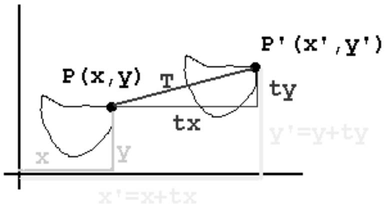
Geometrical Transformations

- Alter or move objects on screen
- Affine Transformations:
  - Each transformed coordinate is a linear combination of the original coordinates
  - Preserve straight lines
- Transform points in the object
  - Translation:
    - A Vector Sum
  - Rotation and Scaling:
    - Matrix Multiplies

Translation: Moving Objects

TRANSLATIONS IN 2-D

(Given translation components, tx, ty)



$$P = \begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Component rule:

$x' = x + tx$

$y' = y + ty$

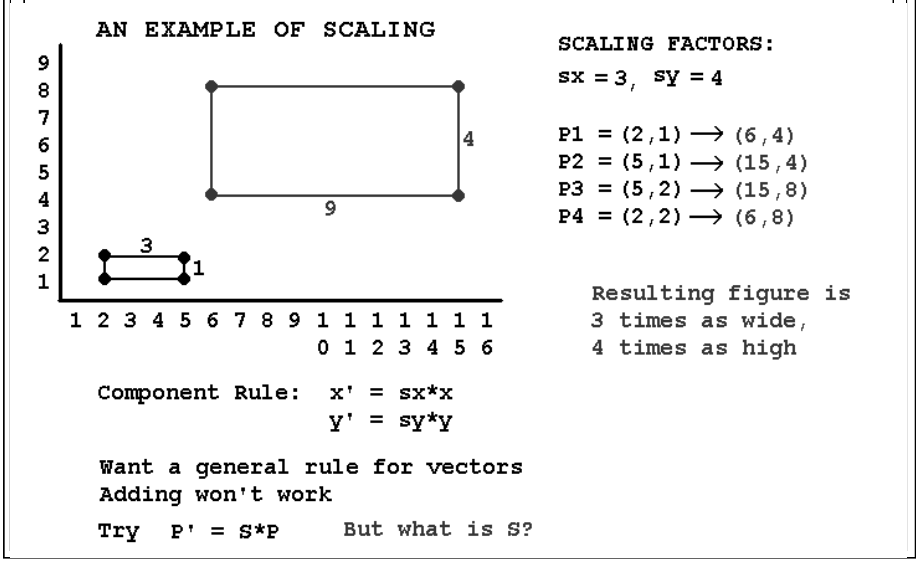
So: $P' = \begin{bmatrix} x+tx \\ y+ty \end{bmatrix}$

General rule:

$P' = P + T$

where $T = \begin{bmatrix} tx \\ ty \end{bmatrix}$

Scaling: Sizing Objects



SCALING FACTORS:

sx = 3, sy = 4

P1 = (2,1) ⟶ (6,4)
P2 = (5,1) ⟶ (15,4)
P3 = (5,2) ⟶ (15,8)
P4 = (2,2) ⟶ (6,8)

Resulting figure is
3 times as wide,
4 times as high

Component Rule: x' = sx*x
y' = sy*y

Want a general rule for vectors
Adding won't work

Try P' = S*P But what is S?

Scaling, continued

P’ = S*P

P, P’ are 2D vectors, so S must be 2x2 matrix

Component equations:

x’ = sx*x, y’ = sy*y

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s11 & s12 \\ s21 & s22 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad \begin{aligned} x' &= s11*x + s12*y \\ y' &= s21*x + s22*y \end{aligned}$$

So: s11=sx, s12=0, s21=0, s22=sy

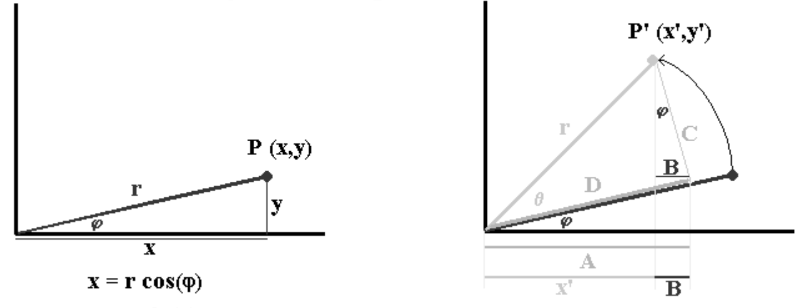
Therefore: $S = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix}$ (The scaling matrix)

Rotation about Origin

- Rotate point P by θ about origin
- Rotated point is P’
- Want to get P’ from P and θ
- P’ = R*P
- R is the rotation matrix
- Look at components:

Rotation: X Component

Rotate P by θ about origin



$x = r\cos(\varphi)$

$y = r\sin(\varphi)$

$x' = A - B$

$A = D\cos(\varphi)$

$B = C\sin(\varphi)$

$D = r\cos(\theta)$

$C = r\sin(\theta)$

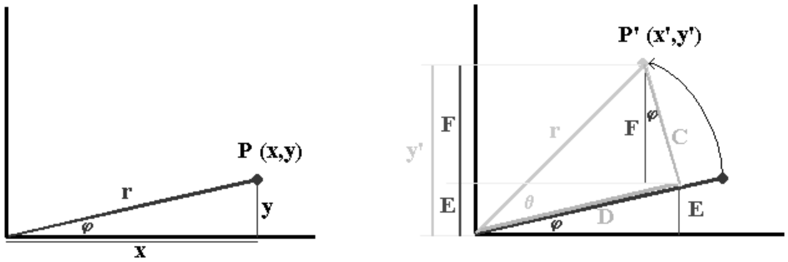
So:

$$x' = r\cos(\theta)\cos(\varphi) - r\sin(\theta)\sin(\varphi)$$

$$x' = x\cos(\theta) - y\sin(\theta)$$

Rotation: Y Component

Rotate P by θ about origin



$x = r\cos(\varphi)$

$y = r\sin(\varphi)$

$y' = E + F$

$E = D\sin(\varphi)$

$F = C\cos(\varphi)$

$D = r\cos(\theta)$

$C = r\sin(\theta)$

So:

$y' = r\cos(\theta)\sin(\varphi) + r\sin(\theta)\cos(\varphi)$

$y' = y\cos(\theta) + x\sin(\theta)$

Rotation: Result

P’ = R*P

R must be a 2x2 matrix

Component equations:

- x’ = x cos(θ) - y sin(θ)
- y’ = x sin(θ) + y cos(θ)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} r11 & r12 \\ r21 & r22 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad \begin{array}{l} x' = r11*x + r12*y \\ y' = r21*x + r22*y \end{array}$$

So: r11=cos(θ), r12=-sin(θ), r21=sin(θ), r22=cos(θ)

$$\text{Therefore: } R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

The Rotation Matrix

Transforming Objects

- For example, lines
  1. Transform every point & plot (too slow)
  2. Transform endpoints, draw the line
     - Since these transformations are affine, result is the transformed line

---

Composite Transformations

- Successive transformations
- e.g., scale then rotate an n-point object:

1. Scale points: P’ = S*P (n matrix multiplies)
2. Rotate pts: P’’ = R*P’ (n matrix multiplies)

But:

P’’ = R*(SP), & matrix multiplication is associative

P’’ = (R*S)*P = M_{comp}*P

So Compute M_{comp} = R*S (1 matrix mult.)

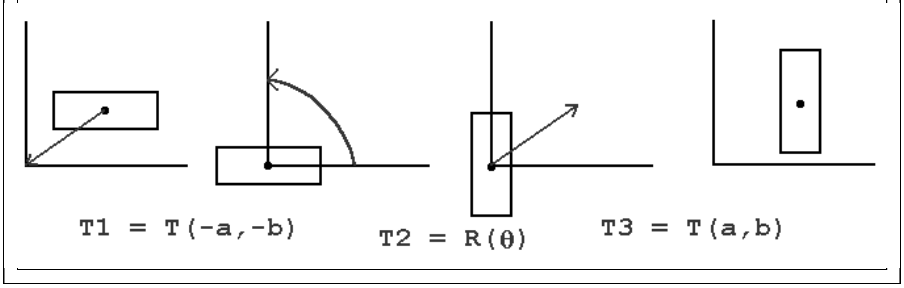
P’’ = M_{comp}*P (n matrix multiplies)

n+1 multiplies vs. 2*n multiplies

Composite Transformations

Another example: Rotate in place

- center at (a,b)
- 1. Translate to origin: T(-a.-b)
- 2. Rotate: R(θ)
- 3. Translate back: T(a,b)



Rotation in place:

- 1. P’ = P + T1
- 2. P’’ = R*P’ = R*(P+T1)
- 3. P’’’ = P’’+T3 = R*(P+T1) + T3
- Can’t be put into single matrix mult. form:
  - i.e., P’’’ != T_{comp} * P
- But we want to be able to do that!!
  - Problem is: translation--vector add
  - rotation/scaling--matrix multiply

Homogeneous Coordinates

- Redefine transformations so each is a matrix multiply
- Express each 2-D Cartesian point as a triple:
  - A 3-D vector in a “homogeneous” coordinate system

$$\begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{bmatrix} xh \\ yh \\ w \end{bmatrix}$$

where we define:

xh = w*x,

yh = w*y

- Each (x,y) maps to an infinite number of homogeneous 3-D points, depending on w
- Take w=1
- Look at our affine geometric transformations

Homogeneous Translations

P' = P + T (Cartesian 2-D coordinates)

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad T = \begin{bmatrix} tx \\ ty \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

(Homogeneous coords)

P' = T*P

What matrix is T?

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t11 & t12 & t13 \\ t21 & t22 & t23 \\ t31 & t32 & t33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

matrix multiplication	component eqns	Results
x'=t11*x+t12*y+t13	x'=x+tx;	t11=1, t12=0, t13=tx
y'=t21*x+t22*y+t23	y'=y+ty;	t21=0, t22=1, t23=ty
1 =t31*x+t32*y+t33		t31=0, t32=0, t33=1

So:

$$T = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Scaling (wrt origin)

P' = S*P

Component Equations:

x' = sx*x, y' = sy*y

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s11 & s12 & s13 \\ s21 & s22 & s23 \\ s31 & s32 & s33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Doing the matrix multiplication:

x' = s11*x + s12*y + s13

y' = s12*x + s22*y + s23

1 = s31*x + s32*y + s33

Comparing with component eqns:

s11=sx, s12=0, s13=0

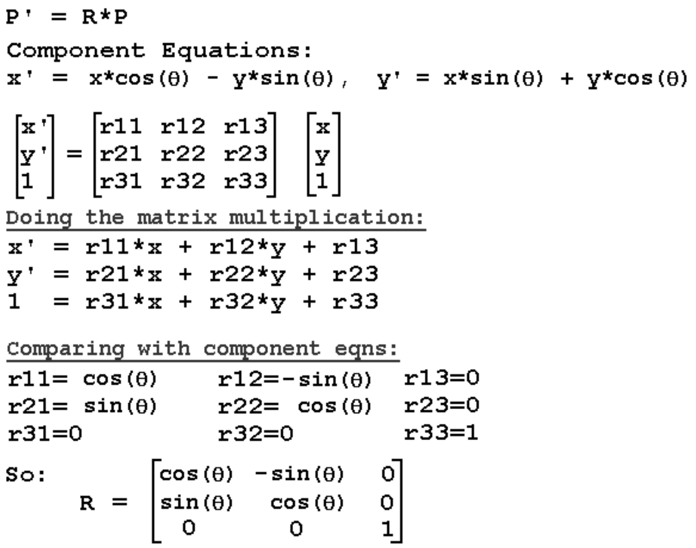
s21=0, s22=sy, s23=0

s31=0, s32=0, s33=1

So:

$$S = \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Rotation (about origin)



P' = R*P

Component Equations:

x' = x*cos(θ) - y*sin(θ), y' = x*sin(θ) + y*cos(θ)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ r31 & r32 & r33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Doing the matrix multiplication:

x' = r11*x + r12*y + r13
y' = r21*x + r22*y + r23
1 = r31*x + r32*y + r33

Comparing with component eqns:

r11= cos(θ) r12=-sin(θ) r13=0
r21= sin(θ) r22= cos(θ) r23=0
r31=0 r32=0 r33=1

So:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Composite Transformations with Homogeneous Coordinates

- All transformations implemented as homogeneous matrix multiplies
- Assume transformations T1, then T2, then T3:

  Homogeneous matrices are T1, T2, T3

  P’ = T1*P

  P’’ = T2*P’ = T2*(T1*P) = (T2*T1)*P

  P’’=T3*P’’=T3*((T2*T1)*P)=(T3*T2*T1)*P

  Composite transformation: T = T3*T2*T1

  Compute T just once!

Example

Rotate line from (5,5) to (10,5) by 90° about (5,5)
T1=T(-5,-5), T2=R(90), T3=T(5,5)
T=T3*T2*T1

```
     _     _   _              _   _      _
    | 1 0 5 | | cos90 -sin90 0 | | 1 0 –5 |
T = | 0 1 5 |*| sin90  cos90 0 |*| 0 1 –5 |
    |_0 0 1_| |_  0     0    1_| |_0 0  1_|
```

```
    _          _
   | 0  -1   10 |
T= | 1   0    0 |
   |_0   0    1_|
```

Example, continued

```
P1′ = T*P1

       _        _   _ _     _ _
      | 0 -1  10 | | 5 |   | 5 |
P1′ = | 1  0   0 |*| 5 | = | 5 |
      |_0  0   1_| |_1_|   |_1_|

       _        _   _  _     _ _
      | 0 -1  10 | | 10 |   | 5 |
P2′ = | 1  0   0 |*|  5 | = |10 |
      |_0  0   1_| |_ 1_|   |_1_|
i.e., P1′ = (5,5),  P2′ = (5,10)
```

Setting Up a General 2D Geometric Transformation Package

Multiplying a matrix & a vector: General 3D Formulation

```
 _  _      _        _     _ _
| x’ |    | a0 a1 a2 |   | x |
| y’ |  = | a3 a4 a5 | * | y |
|_z’_|    |_a6 a7 a8_|   |_z_|
So: x’ = a0*x + a1*y + a2*z
    y’ = a3*x + a4*y + a5*z
    z’ = a6*x + a7*y + a8*z
(9 multiplies and 6 adds)
```

Multiplying a matrix & a vector: Homogeneous Form

```
 _  _     _        _     _ _
| x’ |   | a0 a1 a2 |   | x |
| y’ | = | a3 a4 a5 | * | y |
|_1 _|   |_0  0  1 _|   |_1_|
So: x’ = a0*x + a1*y + a2
    y’ = a3*x + a4*y + a5
(4 multiplies and 4 adds)
MUCH MORE EFFICIENT!
```

Multiplying 2 3D Matrices: General 3D Formulation

```
 _        _   _        _   _        _
| c0 c1 c2 | | a0 a1 a2 | | b0 b1 b2 |
| c3 c4 c5 |=| a3 a4 a5 |*| b3 b4 b5 |
|_c6 c7 c8_| |_a6 a7 a8_| |_b6 b7 b8_|
So: c0 = a0*b0 + a1*b3 + a2*b6
    Eight more similar equations
(27 multiplies and 18 adds)
```

Multiplying 2 3D Matrices: Homogeneous Form

```
 _        _   _        _   _        _
| c0 c1 c2 | | a0 a1 a2 | | b0 b1 b2 |
| c3 c4 c5 |=| a3 a4 a5 |*| b3 b4 b5 |
|_0  0  1 _| |_0  0  1 _| |_0  0  1 _|
So: c0 = a0*b0 + a1*b3 + 0
  (Similar equations for c1, c3, c4)
And: c2 = a0*b2 + a1*b5 +a2
  (Similar equation for c5)
(12 multiplies and 8 adds)
MUCH MORE EFFICIENT!
```

Much Better to Implement our Own Transformation Package

- In general, obtain transformed point P' from original point P:
- P' = M * P
- Set up a a set of functions that will transform points
- Then devise other functions to do transformations on polygons
  - since a polygon is an array of points

- Store the 6 nontrivial homogeneous transformation elements in a 1-D array A
  - The elements are a[i]
    - `a[0], a[1], a[2], a[3], a[4], a[5]`
- Then represent any geometric transformation with the following matrix:

```
      _               _
     | a[0] a[1] a[2] |
M =  | a[3] a[4] a[5] |
     |_ 0    0    1  _|
```

- Define the following functions:
  - Enables us to set up and transform points and polygons:

```
settranslate(double a[6], double dx, double dy); // set xlate matrix
setscale(double a[6], double sx, double sy);   //  set scaling matrix
setrotate(double a[6], double theta);              // set rotation matrix
combine(double c[6], double a[6], double b[6]);   // C = A * B
xformcoord(double c[6], DPOINT vi, DPOINT* vo);  // Vo=C*Vi
xformpoly(int n, DPOINT inpts[], DPOINT outpts[], double t[6]);
```

- The “set” functions take parameters that define the translation, scaling, rotation and compute the transformation matrix elements a[i]

- The combine() function computes the composite transformation matrix elements of the matrix C which is equivalent to the multiplication of transformation matrices A and B

  (C = A * B)

- The xformcoord(c[ ],Vi,Vo) function
  - Takes an input DPOINT (Vi, with x,y coordinates)
  - Generates an output DPOINT (Vo, with x',y' coordinates)
  - Result of the transformation represented by matrix C whose elements are c[i]

- The xformpoly(n,ipts[ ],opts[ ],t[ ]) function
  - – takes an array of input DPOINTs (an input polygon)
  - – and a transformation represented by matrix elements t[i]
  - – generates an array of ouput DPOINTs (an output polygon)
    - result of applying the transformation t[ ] to the points ipts[ ]
  - – will make n calls to xformcoord()
    - n = number of points in input polygon

An Example--Rotating a Polygon about one of its Vertices by Angle θ

- Rotation about (dx,dy) can be achieved by the composite transformation:

  1. Translate so vertex is at origin (-dx,-dy); Matrix T1
  2. Rotate about origin by θ; Matrix R
  3. Translate back (+dx,+dy); Matrix T2

- The composite transformation matrix would be: T = T2*R*T1

Some Sample Code: Rotating a Polygon about a Vertex

Example Code: rotating a polygon about a vertex

```
DPOINT p[4];                   // input polygon
DPOINT px[4];                  // transformed polygon
int n=4;                        // number of vertices
int pts[ ]={0,0,50,0,50,70,0,70};  // poly vertex coordinates
float theta=30;                // the angle of rotation
double dx=50,dy=70;            // rotate about this vertex
double xlate[6];               // the transformation 'matrices'
double rotate[6];
double temp[6];
double final[6];
```

```
for (int i=0; i<n; i++)   // set up the input polygon
   { p[i].x=pts[2*i];
     p[i].y=pts[2*i+1]; }
Polygon(p,n);                        // draw original polygon
settranslate(xlate,-dx,-dy);     // set up T1 trans matrix
setrotate(rotate,theta);            // set up R rotaton matrix
combine (temp,rotate,xlate);  // compute R*T1  &...
                                      // save in temp
settranslate(xlate,dx,dy);         // set up T2 trans matrix
combine(final,xlate,temp);       // compute T2*(R*T1) &...
                                      // save in final
xformpoly(n,p,px,final);     // get transformed polygon px
Polygon(px,n);                  // draw transformed polygon
```

Setting Up More General Polygon Transformation Routines

- trans_poly() could translate a polygon by tx,ty
- rotate_poly() could rotate a polygon by θ about point (tx,ty)
- scale_poly() could scale a polygon by sx, sy wrt (tx,ty)
- These would make calls to previously defined functions

General Polygon Transformation Function Prototypes

- void trans_poly(int n, DPOINT p[], DPOINT px[], double tx, double ty);
- void rotate_poly(int n, DPOINT p[], DPOINT px[], double theta, double x, double y);
- void scale_poly(int n, DPOINT p[], DPOINT px[], double sx, double sy, double x, double y);

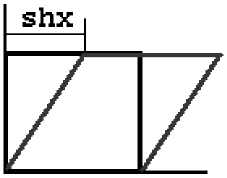
More 2-D Geometric Transformations

A. Shearing

B. Reflections

Other 2D Affine Transformations

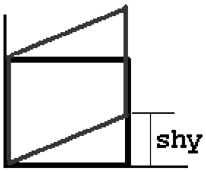
- Shearing (in x direction)
  - – Move all points in object in x direction an amount proportional to y
  - – Proportionality factor:
    - shx (x shearing factor)
  - – Equations:



```
y’ = y
x’ = x + shx*y       |1  shx  0 |
P’ = SHX*P    SHX = |0   1   0 |
                    |0   0   1 |
```

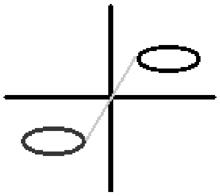
Shearing in y Direction

- – Move all points in object in y direction an amount proportional to x
- – Proportionality factor:
  - shy (y shearing factor)
- – Equations:



```
x’ = x
y’ = shy*x + y       |1   0   0 |
P’ = SHX*P    SHX = |shy 1   0 |
                    |0   0   1 |
```

Reflections



- Reflect through origin

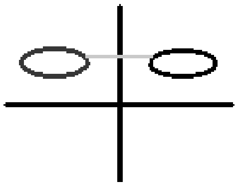
x --> -x

y--> -y

Equations:

```
  x’ = -x
  y’ = -y              |-1  0  0 |
  P’ = Ro*P   Ro = | 0 -1  0 |
                       | 0  0  1 |
```

Reflect Across y-axis



y --> y

x --> -x

Equations:

```
  x’ = -x              |-1  0  0 |
  y’ = y      Ry = | 0  1  0 |
  P’ = Ry*P            | 0  0  1 |
```

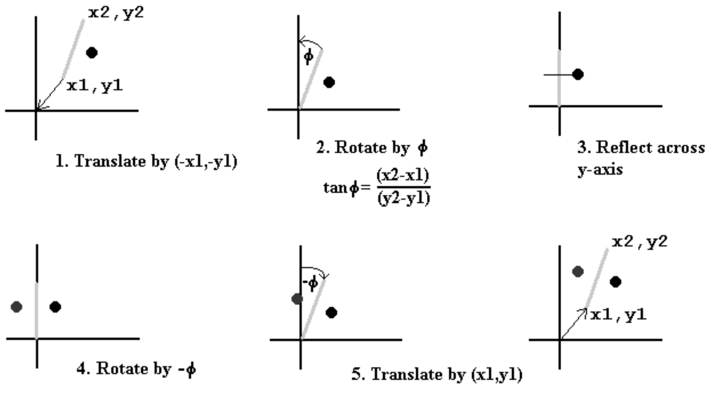
Reflect Across Arbitrary Line

Given line endpoints: (x1,y1), (x2,y2)

1. Translate by (-x1,-y1) [endpoint at origin]
2. Rotate by ϕ [line coincides with y-axis]
3. Reflect across y-axis
4. Rotate by -ϕ
5. Translate by (x1,y1)
6. Composite transformation:

   T = T(x1,y1)*R(-ϕ)*Ry*R(ϕ)*T(-x1,-y1)

Reflect Across a Line
Endpoints (x1,y1), (x2,y2)



1. Translate by (-x1,-y1)

2. Rotate by ϕ

$\tan\phi = \frac{(x2-x1)}{(y2-y1)}$

3. Reflect across y-axis

4. Rotate by -ϕ

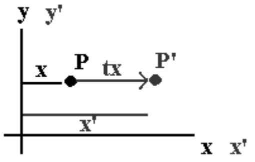
5. Translate by (x1,y1)

T = T5*T4*T3*T2*T1

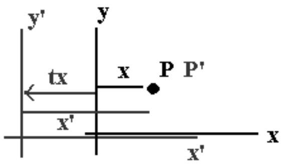
Coordinate System Transformations

- Geometric Transformations:
  - Move object relative to stationary coordinate system (observer)
- Coordinate System Transformation:
  - Move coordinate system (observer) & hold objects stationary
  - Two common types
    - Coordinate System translation
    - Coordinate System rotation
  - Related to Geometric Transformations

Coordinate System Translation



Geometric Translation
by + tx
x' = x + tx



Coordinate system
Translation by -tx
x' = x + tx

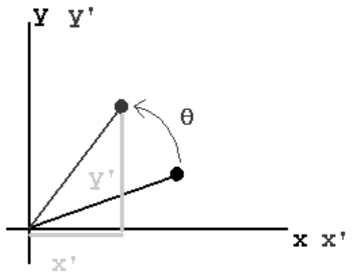
So a Coordinate System Translation by vector -P
is equivalent to a Geometric Translation by vector +P

$$T_G(P) \Longleftrightarrow T_C(-P)$$

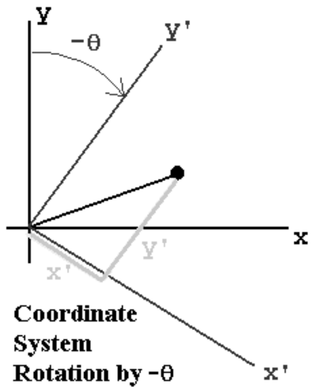
i.e., if P = (px,py): **then:**

$$T_G = \begin{vmatrix} 1 & 0 & px \\ 0 & 1 & py \\ 0 & 0 & 1 \end{vmatrix} \qquad T_C = \begin{vmatrix} 1 & 0 & -px \\ 0 & 1 & -py \\ 0 & 0 & 1 \end{vmatrix}$$

Coordinate System Rotation



Geometric Rotation by θ



Coordinate System Rotation by −θ

Effect is the same

$R_C(\theta)$ <==> $R_G(-\theta)$