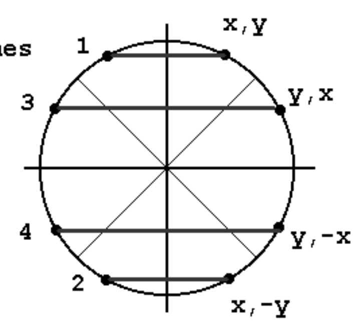
Adapting Scanline Polygon Fill to other primitives

- 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

-- i.e., until pixel color == boundary color

Push left-most unfilled, nonborder pixel above

-->new "above" seed

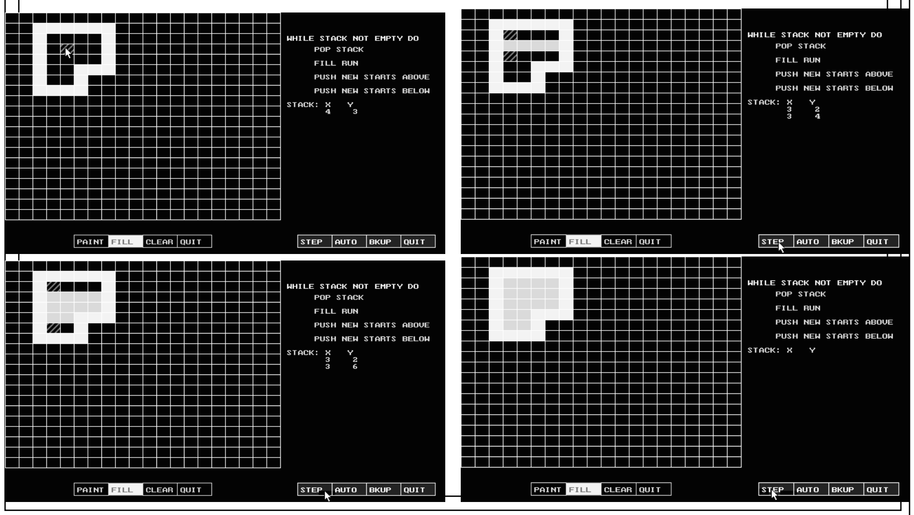
Push left-most unfilled, nonborder pixel below

-->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)**

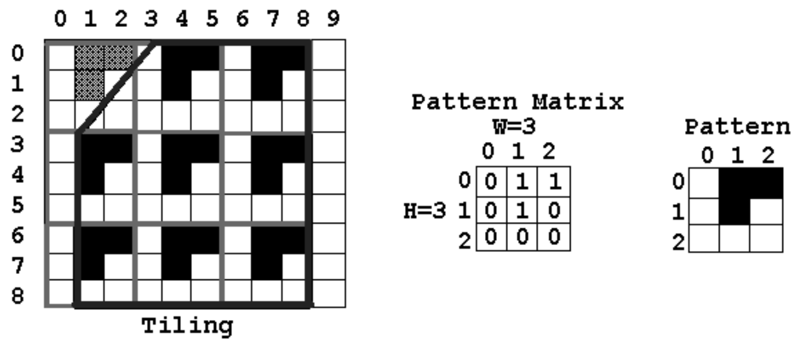
Dino Demo of Scanline Boundary Fill Algorithm



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



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

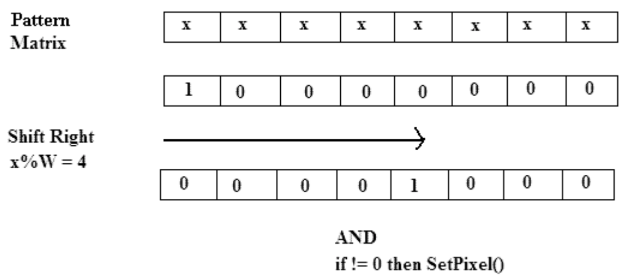
Shift a mask by x%W

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])

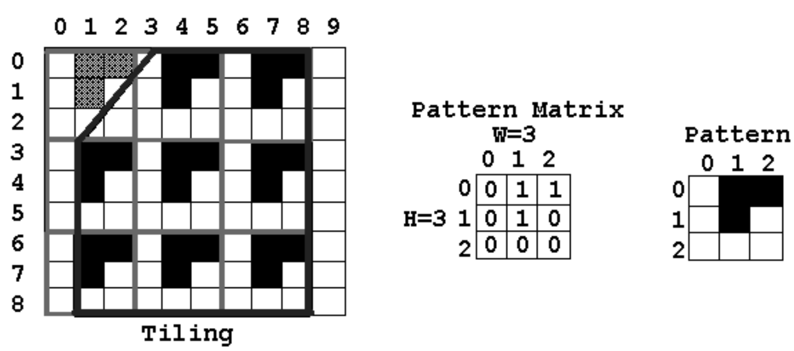
Moving the Filled Polygon

- 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



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

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

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

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--

$$V = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ . \\ . \\ . \\ a_{m1} \end{bmatrix}$$ or: $$V = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ . \\ a_m \end{bmatrix}$$

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

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

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

$S = a_{11}$

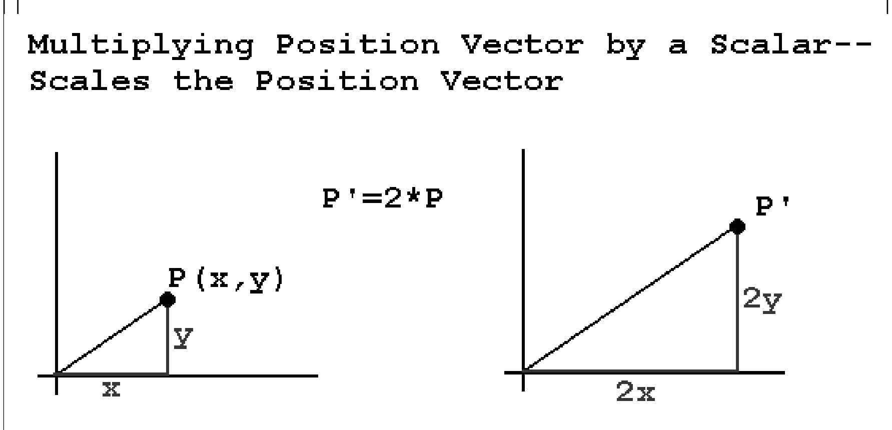
Matrix Operations-- Multiplication by a Scalar

$C = k*A$

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

- 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

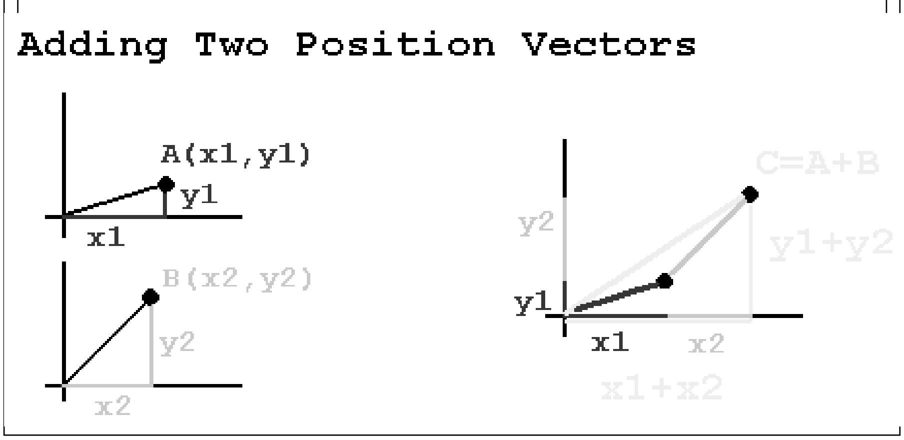


Adding two Matrices

- 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

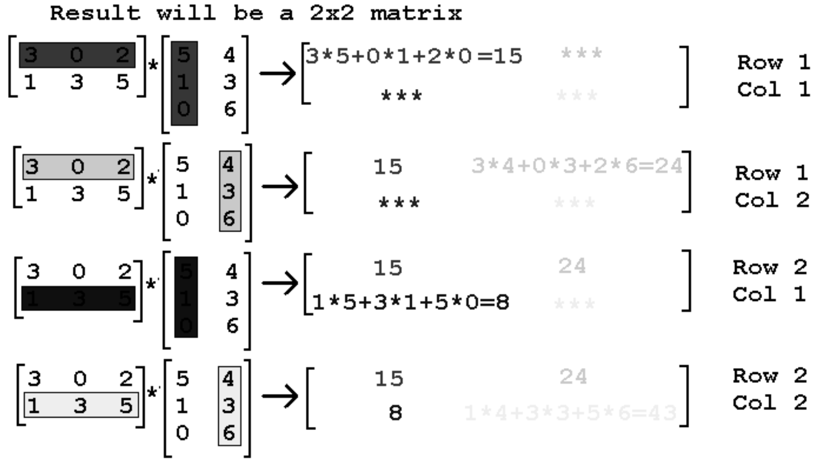


Multiplying Two Matrices

- m x n = (m x p) * (p x n)
- C = A * B
- $c_{ij} = \Sigma\ a_{ik} * b_{kj}$, 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



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
- $V'_i = \sum a_{ik} * v_k$, 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

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

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

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

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

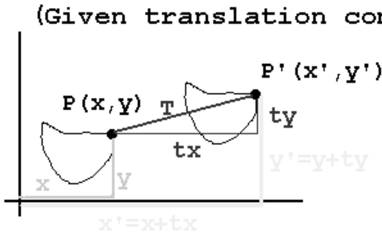
Geometrical Transformations

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

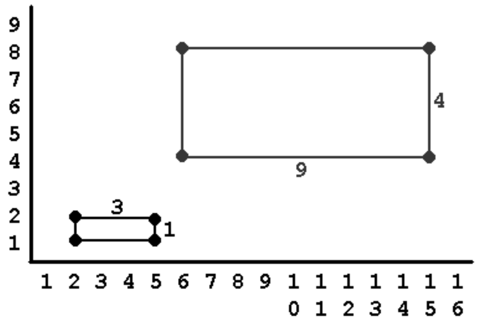
General rule:

$P' = P + T$

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

Scaling: Sizing Objects

AN EXAMPLE OF SCALING



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

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s11 & s12 \\ s21 & s22 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

or

x' = s11*x + s12*y
y' = s21*x + s22*y

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

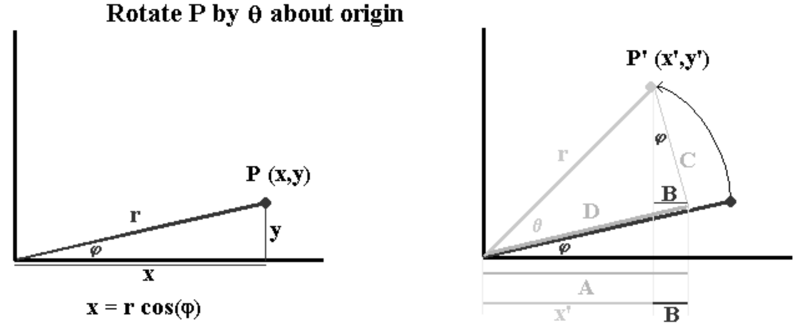
Therefore: $S = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix}$ (The scaling matrix)

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



$x = r\cos(\varphi)$

$y = r\sin(\varphi)$

$x' = A - B$

$A = D\cos(\varphi)$

$B = C\sin(\varphi)$

$D = r\cos(\theta)$

$C = r\sin(\theta)$

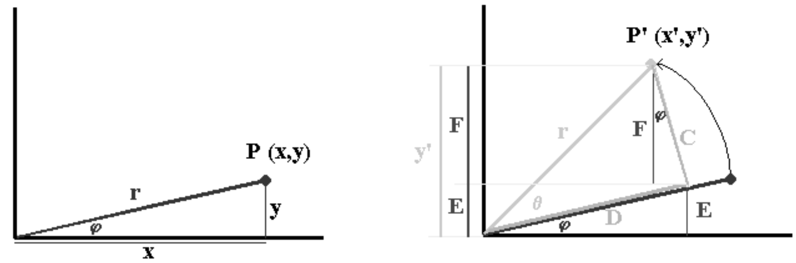
So:

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

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

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:

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

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

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.)

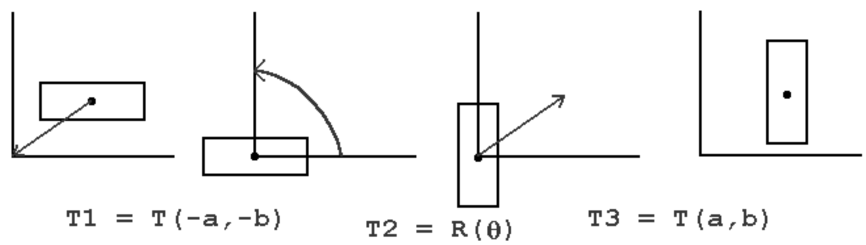
P’’ = M_{comp}*P (n matrix multiplies)

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

x'=t11*x+t12*y+t13
y'=t21*x+t22*y+t23
1 =t31*x+t32*y+t33

component eqns

x'=x+tx;
y'=y+ty;

Results

t11=1, t12=0, t13=tx
t21=0, t22=1, t23=ty
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:

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

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

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:

$r11 = \cos(\theta)$	$r12 = -\sin(\theta)$	r13=0
$r21 = \sin(\theta)$	$r22 = \cos(\theta)$	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);

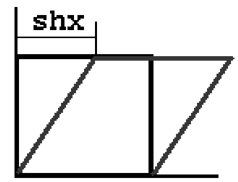
More 2-D Geometric Transformations

A. Shearing

B. Reflections

Other 2D Affine Transformations

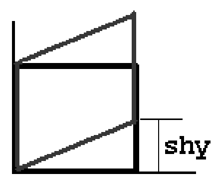
- Shearing (in x direction)
  - Move all points in object in x direction an amount proportional to y
  - Proportionality factor:
    - shx (x shearing factor)
  - Equations:




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

Shearing in y Direction

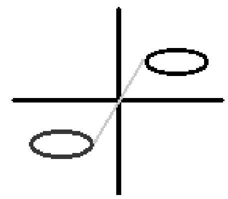
- Move all points in object in y direction an amount proportional to x
- Proportionality factor:
  - shy (y shearing factor)
- Equations:




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

Reflections

- Reflect through origin



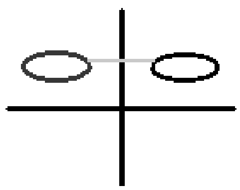
x --> -x

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 |
```

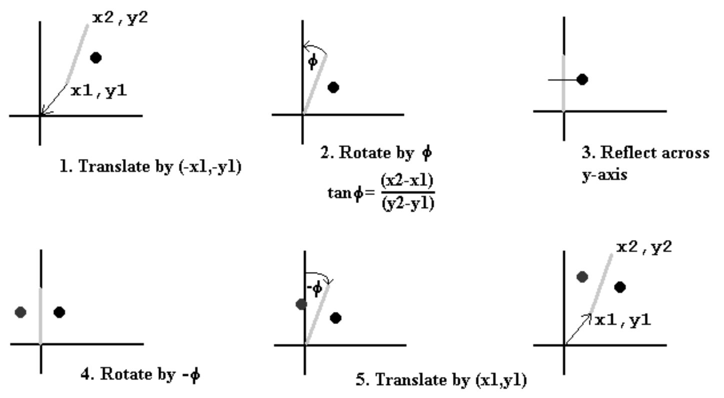
Reflect Across Arbitrary Line

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 $-\phi$
5. Translate by (x1,y1)
6. Composite transformation:

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

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

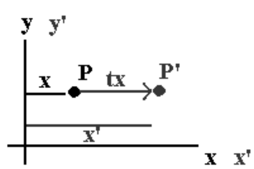


T = T5*T4*T3*T2*T1

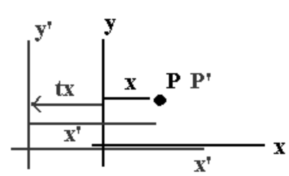
Coordinate System Transformations

- 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

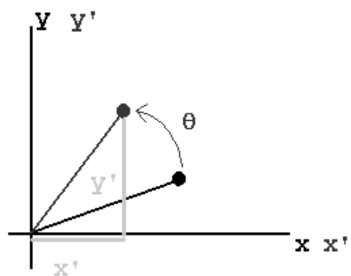
So a Coordinate System Translation by vector -P is equivalent to a Geometric Translation by vector +P

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

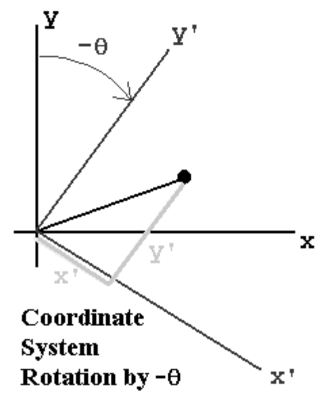
i.e., if P = (px,py): **then:**

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

Coordinate System Rotation



Geometric Rotation by θ



Coordinate System Rotation by -θ

Effect is the same

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