Affine Transformations

Transforming shape models

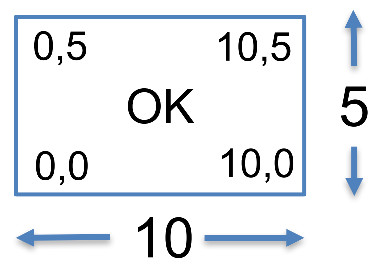
Combining affine transformations

Scene graphs, interactor trees

Hit tests on transformed shapes

Reacll: Shape Models

- We can think of widgets as shape models
  - Consist of an array of points {P1, P2, …, Pn}
  - Properties describing how to draw it
- e.g. Button
  - Points
    - (0,0), (10,0), (5,0), (10,5)
  - Properties
    - Text: “OK”
    - Border width: 1 pixel
    - Border color: blue



Recall: Interactor Tree

- The **interactor tree** describes the hierarchy of widgets
  – Widget location is always specified in terms of parent's coordinate system (i.e. *relative* or local coordinates).
    - We need to do some math before painting
    - Performing hit test requires taking parent location into account (i.e. determining which components a given location corresponds to, relative to its parent)

- In this lecture, we'll show how to use **affine transformations** with an interactor tree to:
  – Manipulate and transform widgets (shape models).
  – Tell widgets how to draw themselves on-screen, relative to their parent.
  – Determine if a mouse-click intersects one of these widgets in the interactor tree.

Linear Algebra: Review of Terms

(s) Scalar: a single value (usually real number)

(v) Vector: directed line segment (represents direction and magnitude)

(P) Point: a fixed location in space (represents a position)

Legal operations:

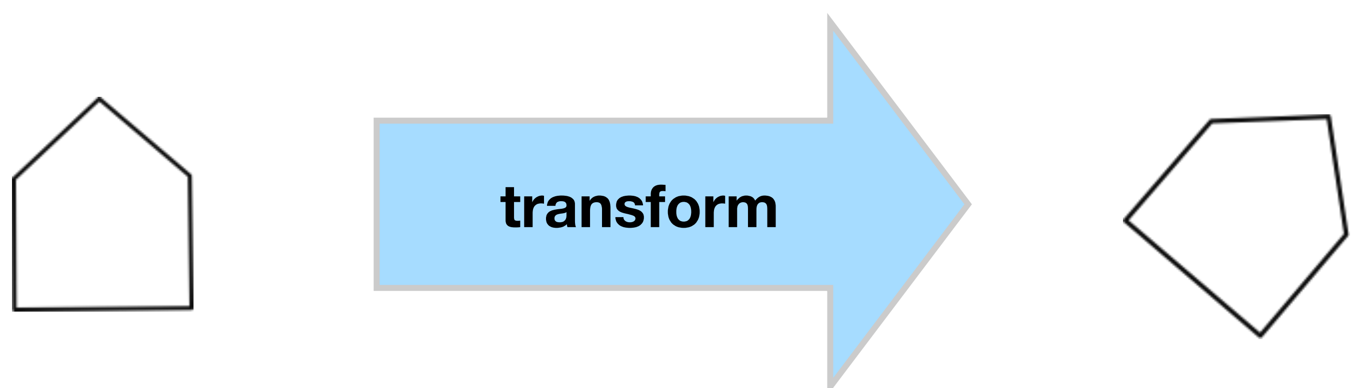
- vector + vector: **v1** + **v2** = **v3**
- vector multiplied by a scalar: **v1** x s1 = **v4**
- point minus point = P1 - P2 = v5
- point + vector: P2 + **v5** = P1
- 2 ways to “multiply” vector by vector
  - dot (inner) product: **v1** o **v2** = s2
  - cross (outer) product: **v1** x **v2** = v6

Moving Shapes, Multiple Instances

Translate by adding offset to shape points

What about scaling?

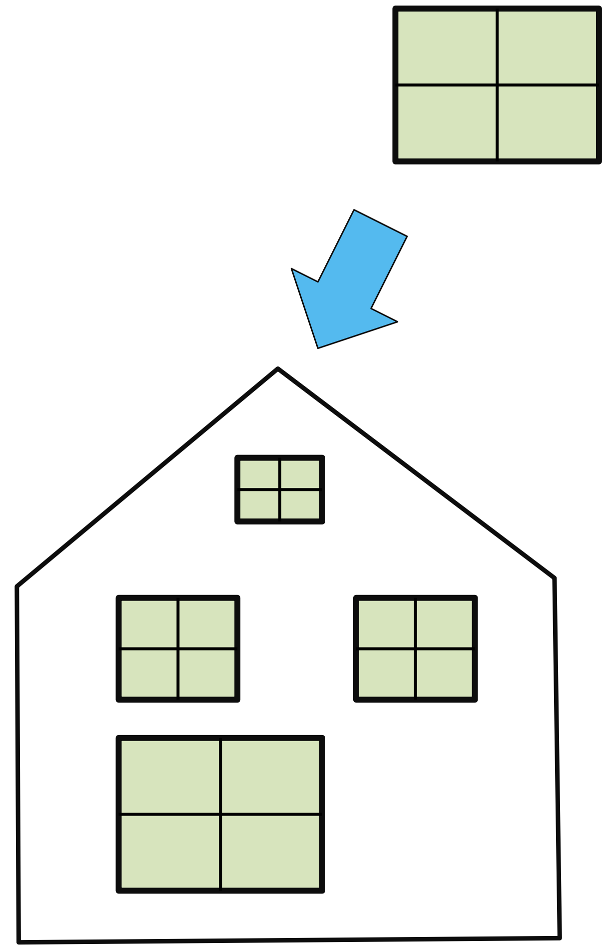
Or rotation?



Moving Shapes, Multiple Instances

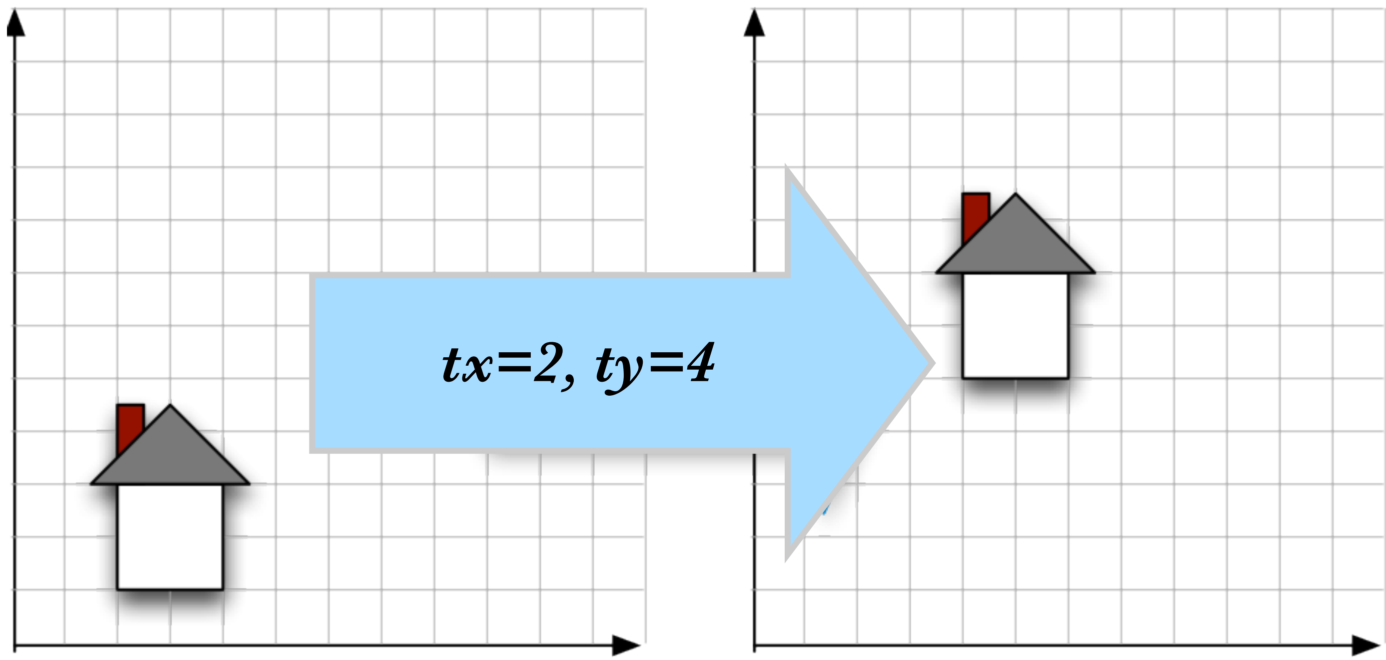
How to create multiple instances of a shape?

Rotation + Scaling + Translation



Translation

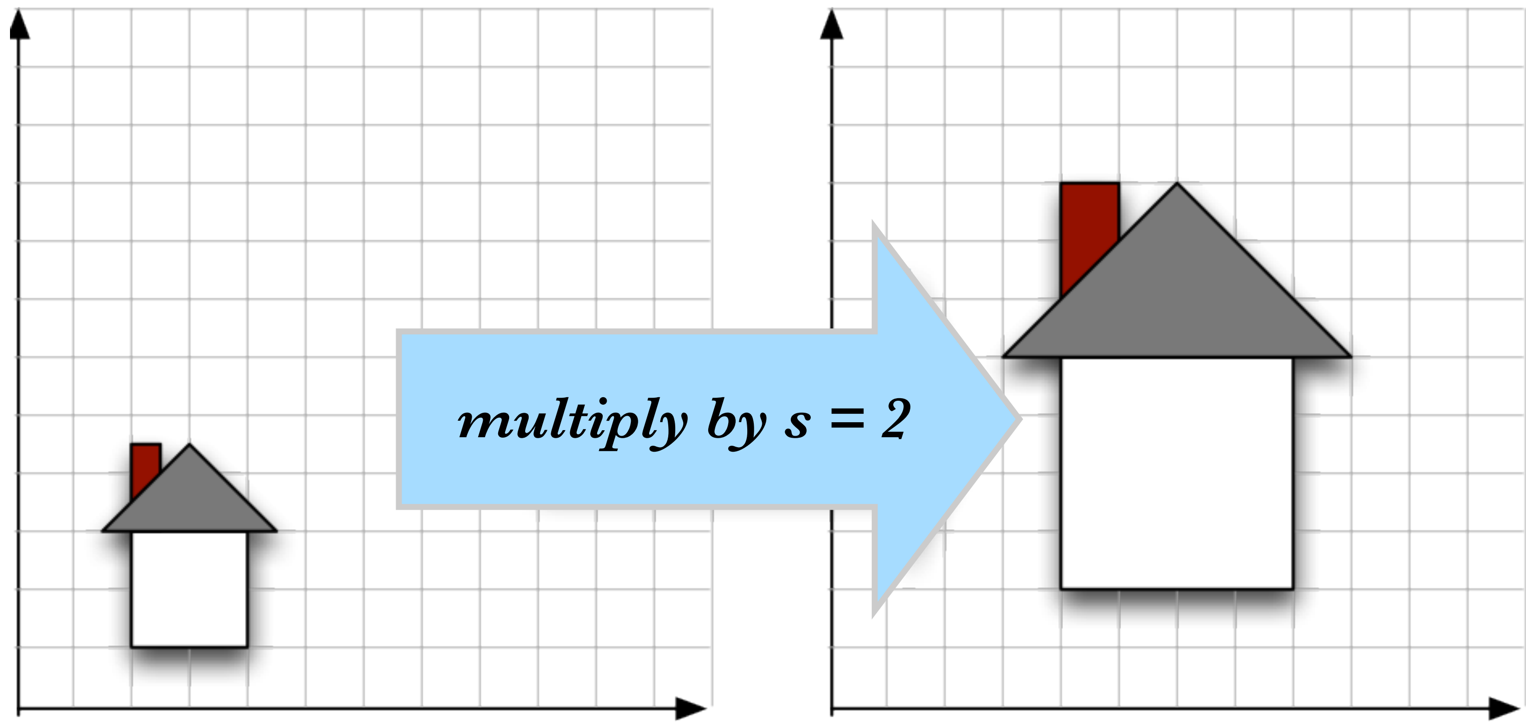
Translate: add a scalar to each of the components



$$x' = x + t_x$$
$$y' = y + t_y$$

Uniform Scaling

Uniform scale: multiply each component by the same scalar

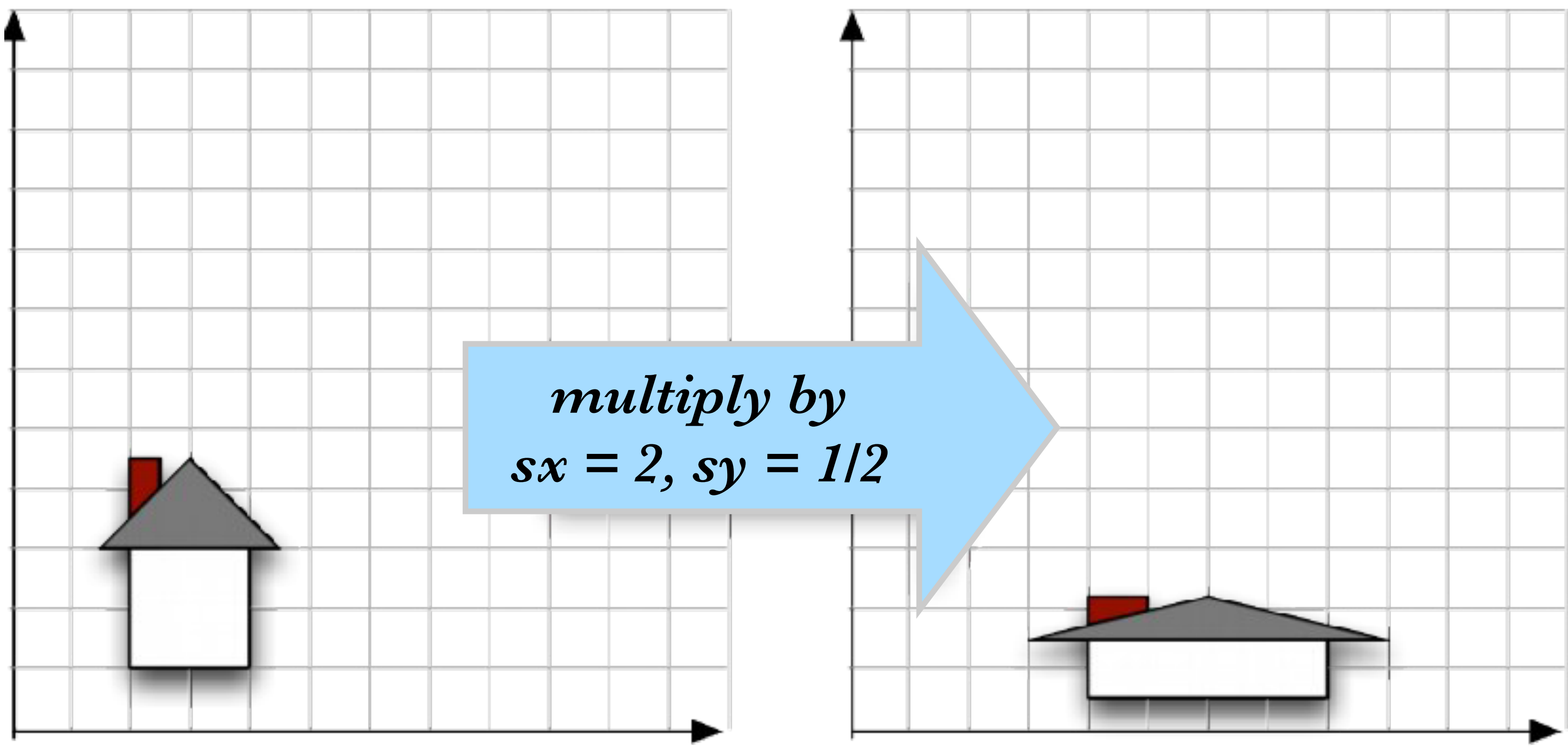


$$x' = x \times s$$

$$y' = y \times s$$

Non-Uniform Scaling

Scale: multiply each component by different scalar

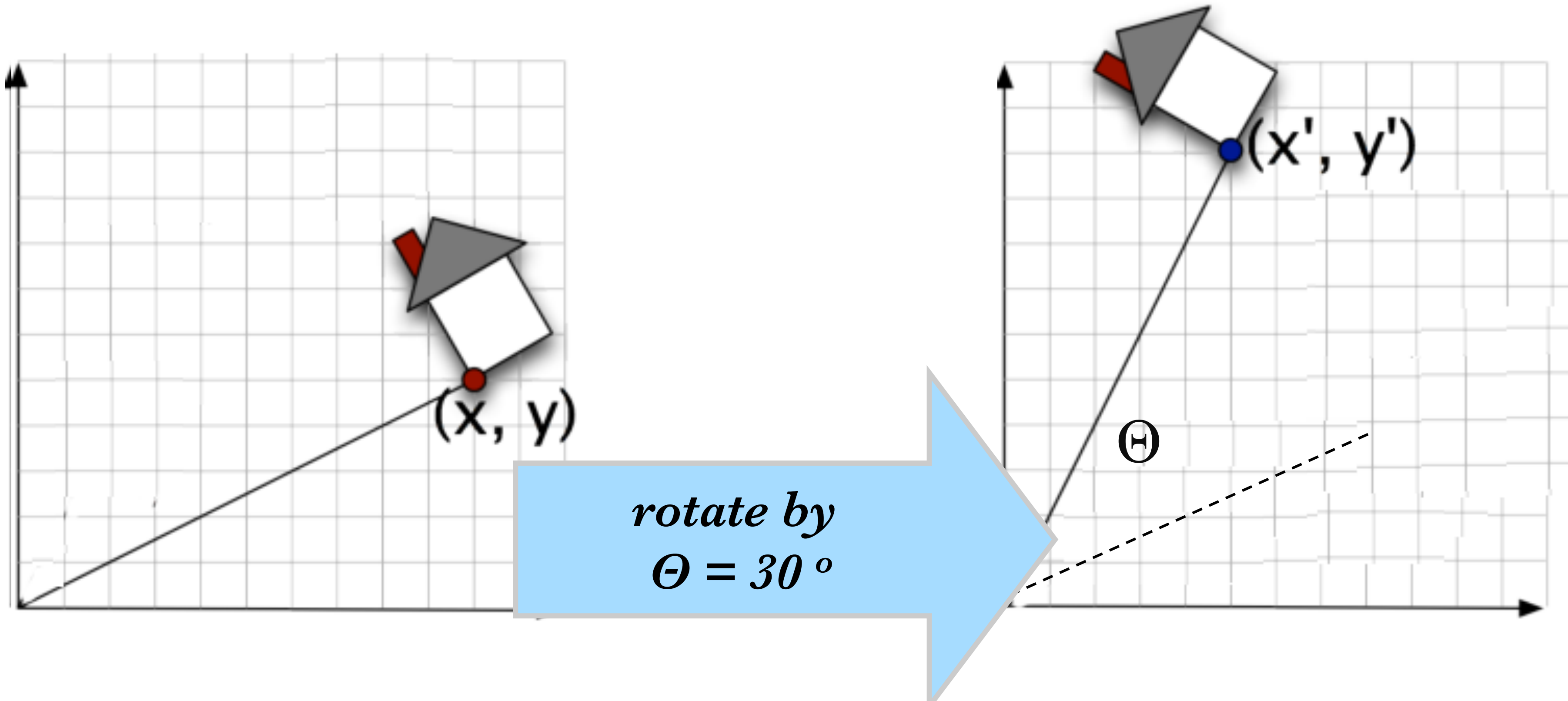


$$x' = x \times s_x$$

$$y' = y \times s_y$$

Rotation

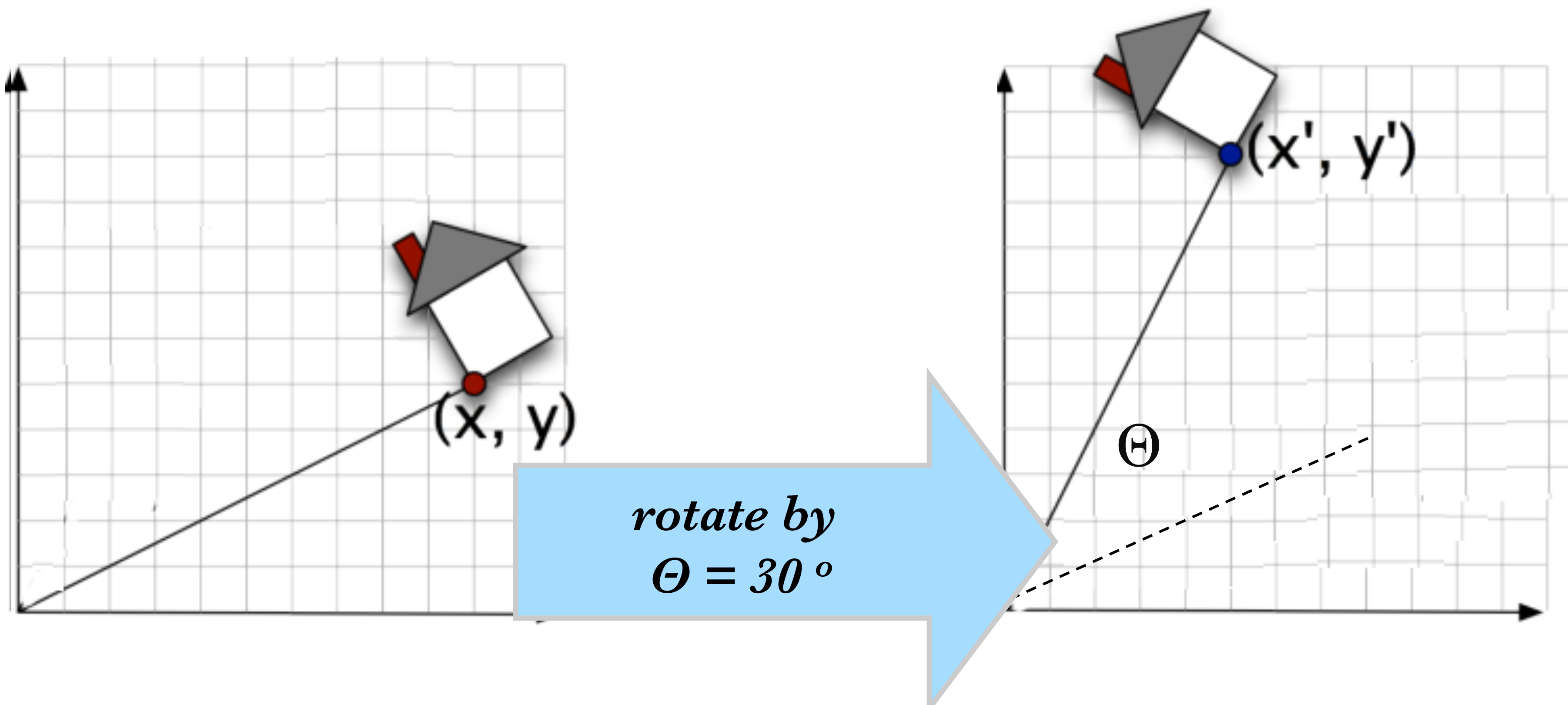
Rotate: each component is some function of x, y, Θ



$$x' = f(x, y, \Theta)$$
$$y' = f(x, y, \Theta)$$

Rotation

Rotate: each component is some function of x, y, Θ



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

Combining Transformations

Rotate:

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

Translate:

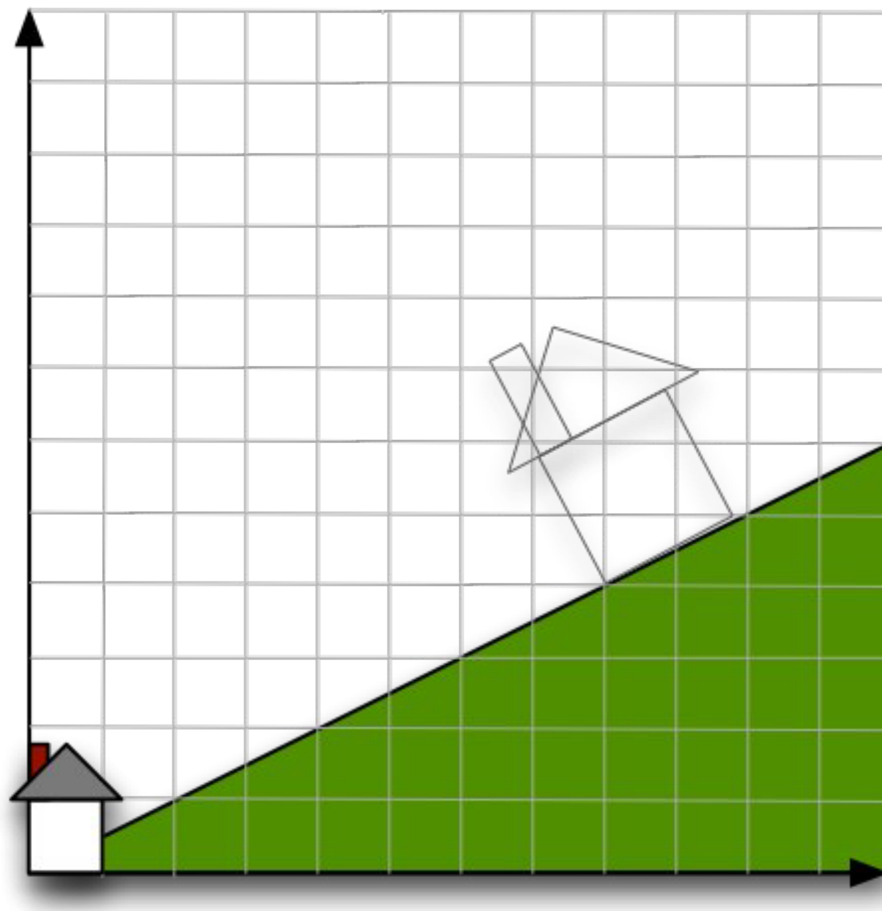
$$x' = x + t_x$$
$$y' = y + t_y$$

Scale:

$$x' = x \times s_x$$
$$y' = y \times s_y$$

<u>Goal</u>: Paint the house at 2x the size, up the hill.

What's our strategy?



Combining Transformations: (1) Scale

Rotate:

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

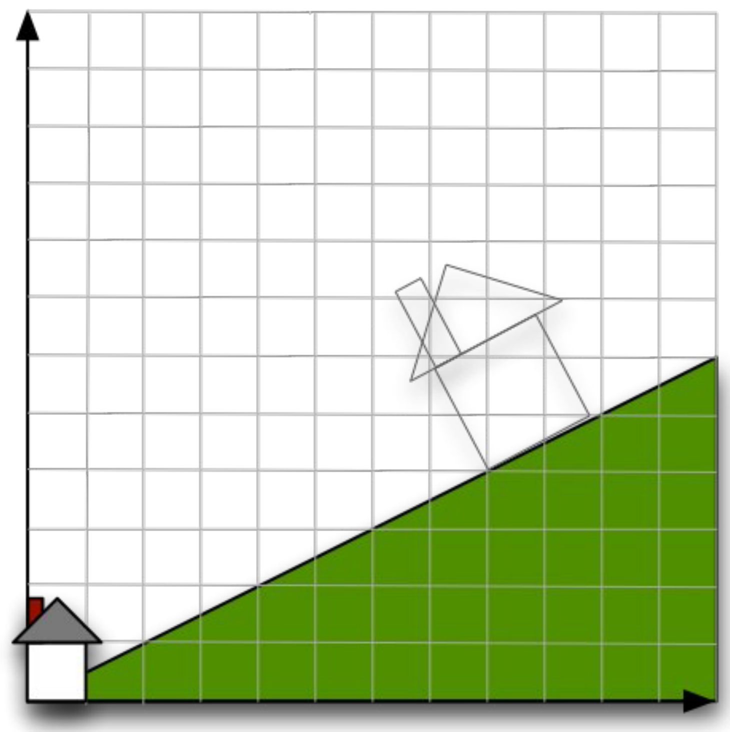
Translate:

$$x' = x + t_x$$
$$y' = y + t_y$$

Scale:

$$x' = x \times s_x$$
$$y' = y \times s_y$$

Steps
1. Scale



$$x_1 = 2x$$
$$y_1 = 2y$$

Combining Transformations: (2) Rotate

Rotate:

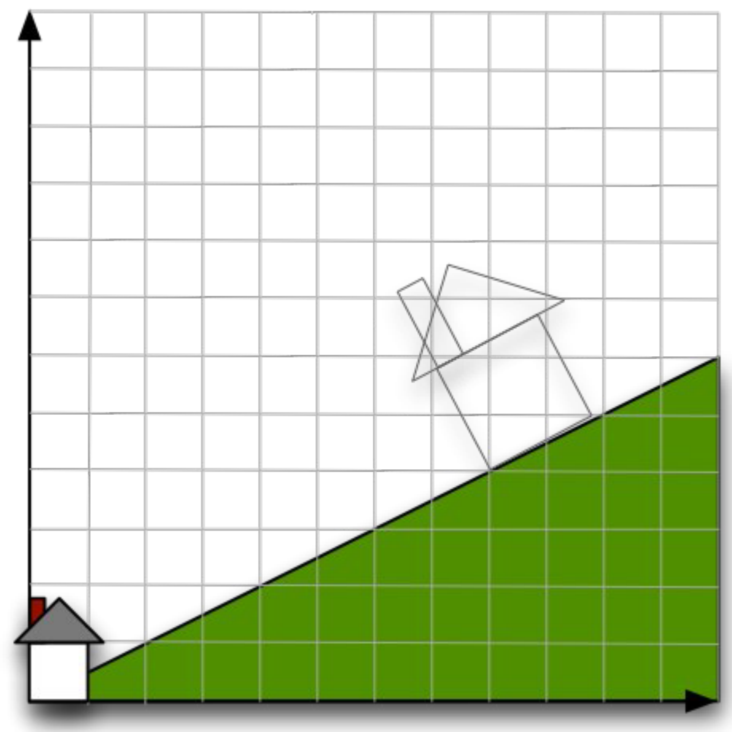
$$x' = x \cos(\Theta) - y \sin(\Theta)$$
$$y' = x \sin(\Theta) + y \cos(\Theta)$$

Translate:

$$x' = x + t_x$$
$$y' = y + t_y$$

Scale:

$$x' = x \times s_x$$
$$y' = y \times s_y$$



Steps

1. Scale
2. Rotate

$$x_2 = 2(\, x \cos(30) - y \sin(30) \,)$$
$$y_2 = 2(\, x \sin(30) + y \cos(30) \,)$$

Combining Transformations: (3) Translation

Rotate:

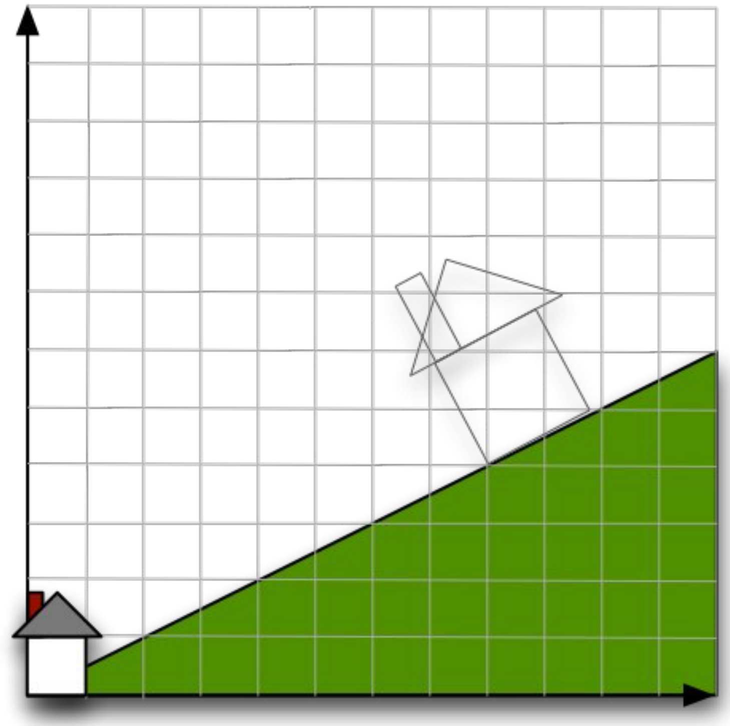
$$x' = x\cos(\Theta) - y\sin(\Theta)$$
$$y' = x\sin(\Theta) + y\cos(\Theta)$$

Translate:

$$x' = x + t_x$$
$$y' = y + t_y$$

Scale:

$$x' = x \times s_x$$
$$y' = y \times s_y$$



Steps

1. Scale
2. Rotate
3. Translate

$$x_3 = 2(\,x\cos(30) - y\sin(30)\,) + 8$$
$$y_3 = 2(\,x\sin(30) + y\cos(30)\,) + 4$$

Order of operations is important. What if you translate first?

Matrix Representation

Goal: represent each 2D transformation with a matrix

This is a succinct way of expressing a single transformation, that can be used to transform all of the points in our shape model

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Multiply matrix by column vector <=> apply transformation to point

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned} \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Matrix Representation

Matrix notation also supports combining transformations by multiplication (i.e. transformations are *associative*)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

We can multiply transformation matrices together

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} aei + bgi + afk + bhk & aej + bgj + ael + bgl \\ cei + dgi + cfk + dhk & cej + dgj + cfl + dhl \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- This single matrix can then be used to transform many points.
- Can be sent to a GPU to speed the process.

Why types of transformations can be represented by a 2x2 matrix?

2D Scale around (0,0)?

$$\begin{aligned} x' &= x \times s_{\mathrm{x}} \\ y' &= y \times s_{\mathrm{y}} \end{aligned} \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_{\mathrm{x}} & 0 \\ 0 & s_{\mathrm{y}} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Why types of transformations can be represented by a 2x2 matrix?

2D Rotate around (0,0)?

$$\begin{aligned} x' &= x\cos(\theta) - y\sin(\theta) \\ y' &= x\sin(\theta) + y\cos(\theta) \end{aligned} \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Why types of transformations can be represented by a 2x2 matrix?

2D Mirror about Y axis?

$$\begin{aligned} x' &= -x \\ y' &= y \end{aligned} \quad \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2 x 2 Matrices

Why types of transformations can be represented by a 2x2 matrix?

2D Translation

$$
\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
$$

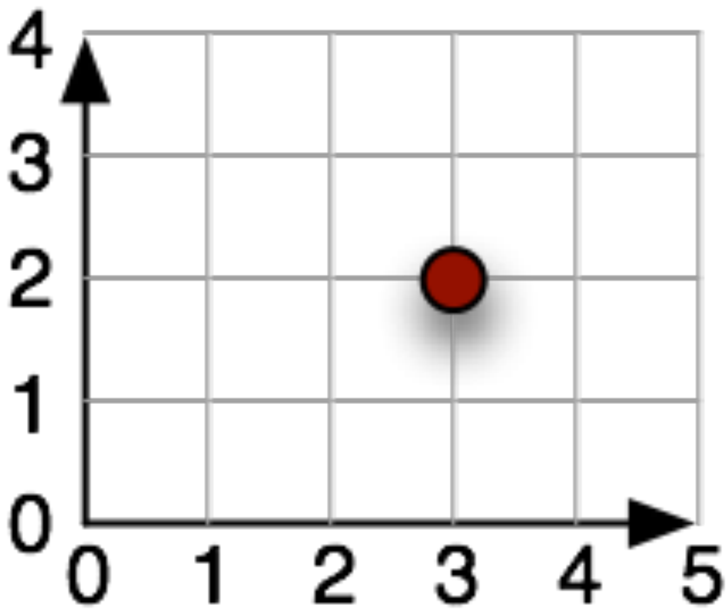
Maybe this?

$$
\begin{matrix} x' \\ y' \end{matrix} = \begin{bmatrix} 1 & t_x / y \\ t_y / x & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
$$

<u>We can’t generalize 2D translation this way</u>; this only works for a specific point! We need a general solution.

Homogeneous Coordinates

- Solution: add an extra component (*w*) to each coordinate
- $[x, y, w]^T$ represents a point at location $[x/w, y/w]^T$
- convenient coordinate system to represent many useful transformations



$[3, 2, \boxed{1}]^T$

$[6, 4, \boxed{2}]^T$

$[7.5, 5, \boxed{2.5}]^T$

…

(divide by *w* to get cartesean coords)

Homogeneous Coordinates

- The *w* component is needed to represent translation (we're not working in 3D space, it just looks like it!)

$$\begin{bmatrix} x \\ y \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \\ y \\ \boxed{1} \end{bmatrix}$$

extra row w
(just set to 1)

- This gives us an **Affine Transformation Matrix** that we can use to represent all combined transformations: translation, scaling and rotation

$$M = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- a, d: *scaling*
- a, b, c, d: *rotation*
- tx, ty: *translation*

last row is always (0,0,1)

- This gives us a general representation of a set of transformations, that we can apply to a point or a vector.

vector
$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

point
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

What we typically use for manipulating shape models..

Vector and Point Homogeneous Coordinates

$$\vec{v} + \vec{w} = \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} + \begin{bmatrix} w_x \\ w_y \\ 0 \end{bmatrix} = \begin{bmatrix} v_x + w_x \\ v_y + w_x \\ 0 \end{bmatrix}$$

add vectors

$$\vec{v} \times s = \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} \times s = \begin{bmatrix} v_x \times s \\ v_y \times s \\ 0 \end{bmatrix}$$

scalar multiply

$$p - q = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} - \begin{bmatrix} q_x \\ q_y \\ 1 \end{bmatrix} = \begin{bmatrix} p_x - q_x \\ p_y - q_x \\ 0 \end{bmatrix}$$

subtract points

$$p + \vec{v} = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_x \\ 1 \end{bmatrix}$$

point + vector

Translating Vectors

A vector has no position so translating it shouldn't change anything.

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{\mathrm{x}} \\ 0 & 1 & t_{\mathrm{y}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

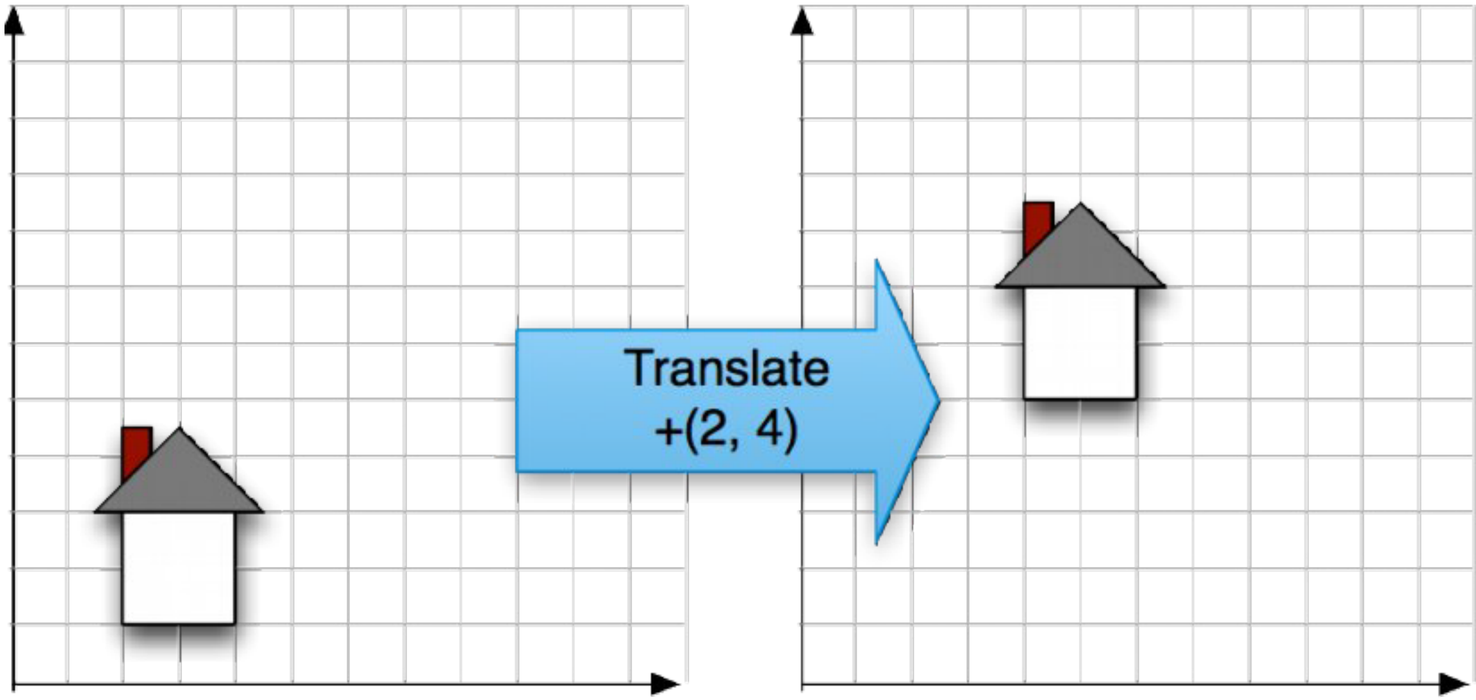
Translation Matrix

We can represent 2D point translation with a 3x3 matrix

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_{\mathrm{x}} \\ y + t_{\mathrm{y}} \\ 1 \end{bmatrix} \Leftrightarrow \begin{matrix} Ax + By + C = x + t_{\mathrm{x}} \\ Dx + Ey + F = y + t_{\mathrm{y}} \\ Gx + Hy + I = 1 \end{matrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{\mathrm{x}} \\ 0 & 1 & t_{\mathrm{y}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_{\mathrm{x}} \\ y + t_{\mathrm{y}} \\ 1 \end{bmatrix}$$

Translation Matrix: Example



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+2 \\ y+4 \\ 1 \end{bmatrix}$$

Rotation Matrix

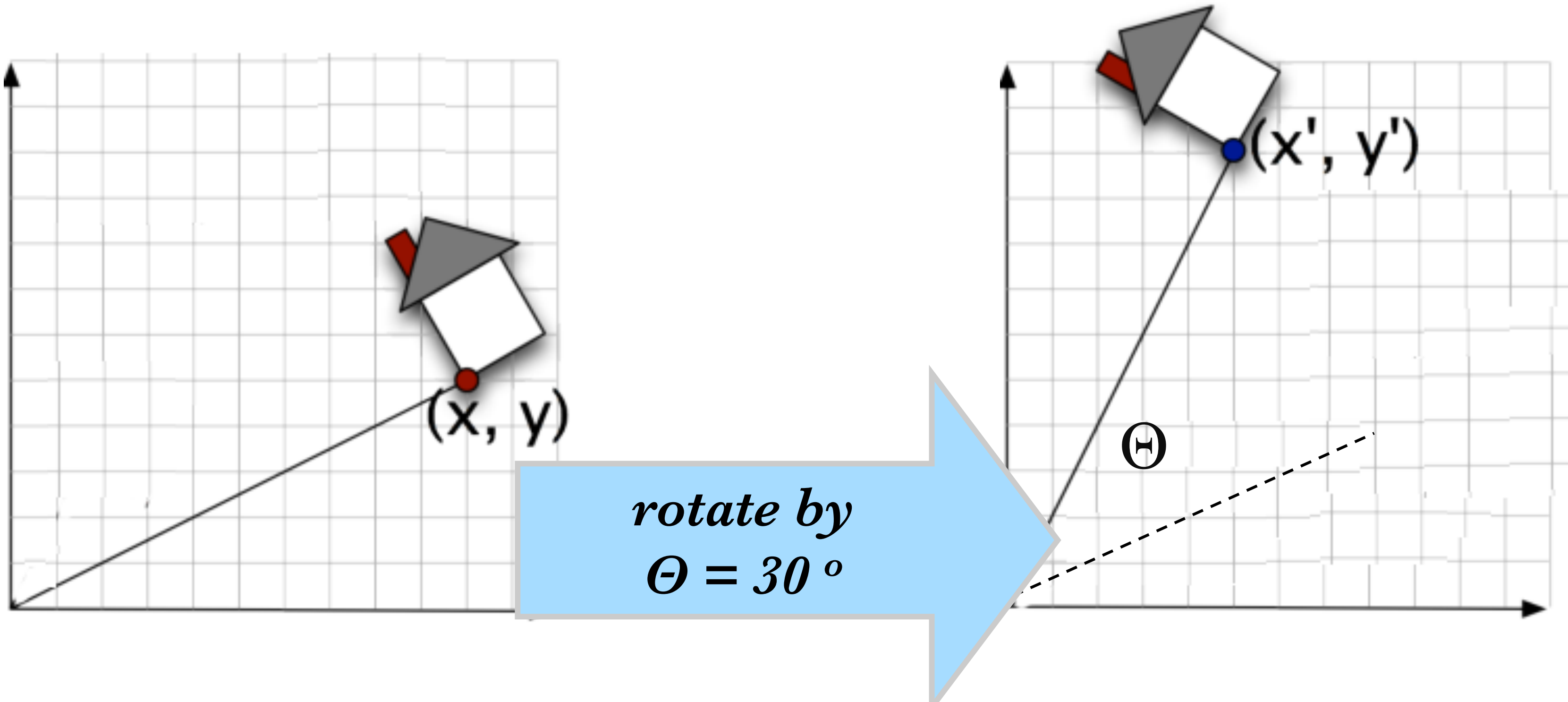
Vectors:

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \\ 0 \end{bmatrix}$$

Points:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \\ 1 \end{bmatrix}$$

Rotation Matrix: Example



$$\begin{bmatrix} cos(30) & -sin(30) & 0 \\ sin(30) & cos(30) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling Matrix

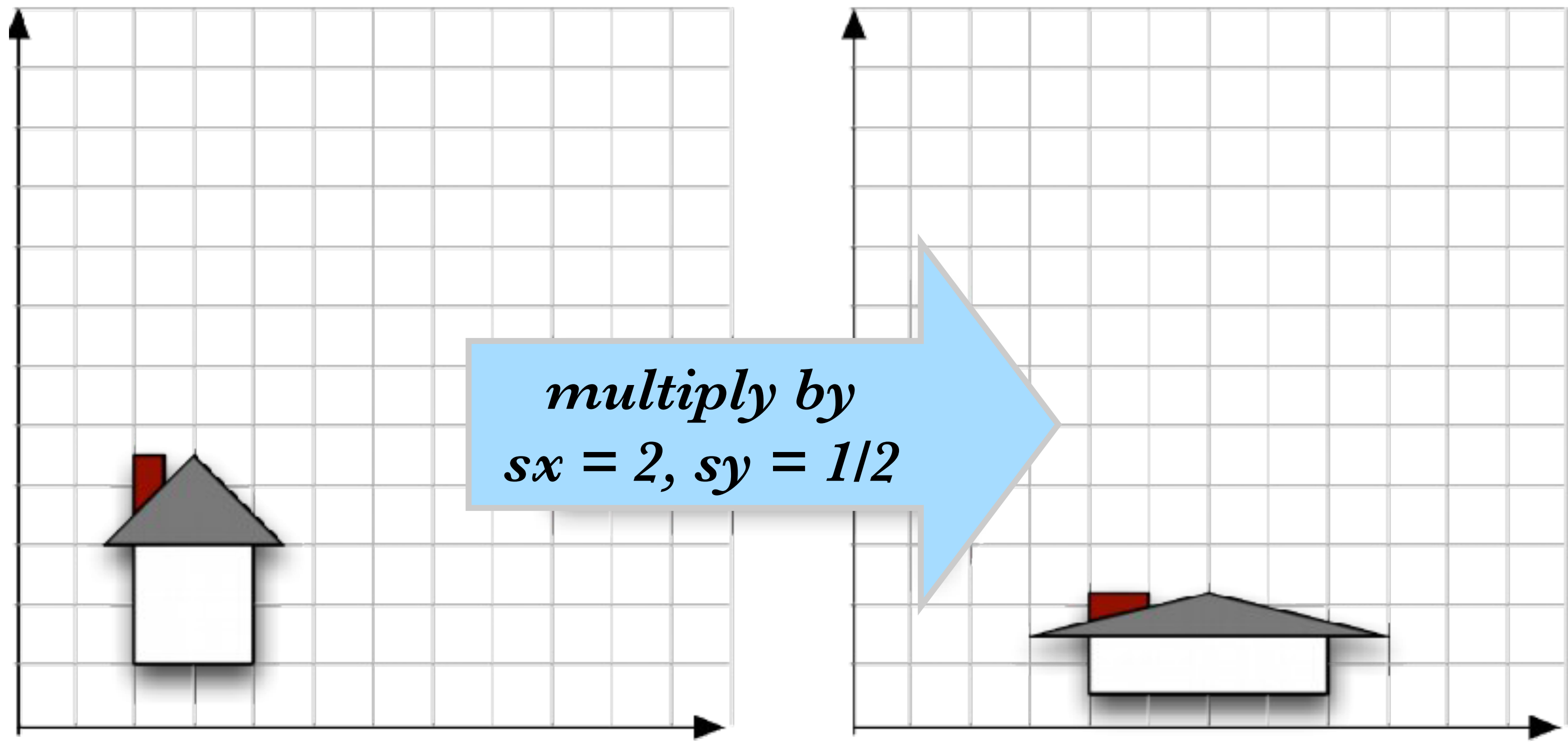
Vectors:

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \cdot s_x \\ y \cdot s_y \\ 0 \end{bmatrix}$$

Points:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot s_x \\ y \cdot s_y \\ 1 \end{bmatrix}$$

Scaling Matrix: Example



$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix Composition

Transformations can be combined by matrix multiplication

$$p' = T(t_x, t_y) \cdot R(\theta) \cdot S(s_x, s_y) \cdot p$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}}^{\text{Translation}} \overbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{\text{Rotation}} \overbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{\text{Scaling}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

apply transformations to point, from right-to-left

Matrix Composition Order

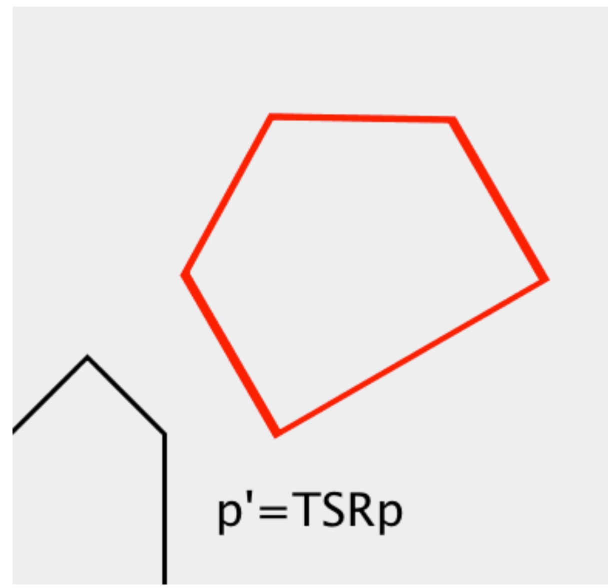
- Associative: A(BC) = (AB)C
- <u>Not</u> Commutative: AB ≠ BA
  - The order of transformations matters!

$$p' = T \cdot R \cdot S \cdot p$$

$$p' = (T \cdot (R \cdot (S \cdot p)))$$

$$p' = (T \cdot R \cdot S) \cdot p$$

T = T(100, 0)
R = R(30°)
S = S(2, 1.2)

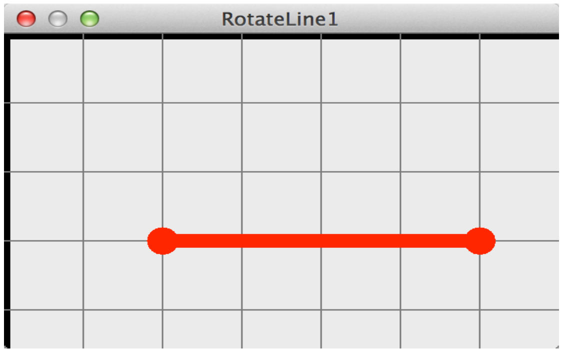


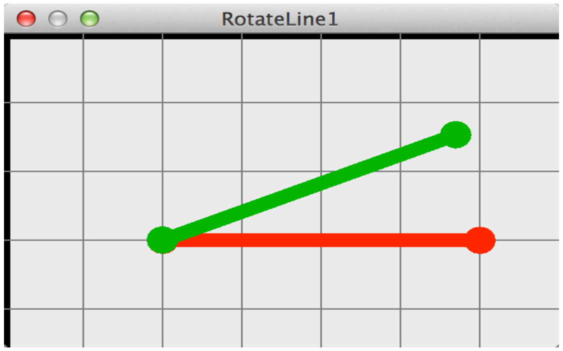
TransConDemo.java

Exercise

- A bar is drawn at (100, 150). Rotate it by 22.5 degrees (π/8 rads) about the left endpoint.

```
private static final int x1 = 100;
private static final int y1 = 150;
private static final int len = 200;

// draw the horizontal bar
private void drawBar(Graphics2D g2) {
   g2.drawLine(x1, y1, x1+len, y1);
   g2.fillOval(x1-10, y1-10, 20, 20);
   g2.fillOval(x1+len-10, y1-10, 20, 20);
}
```

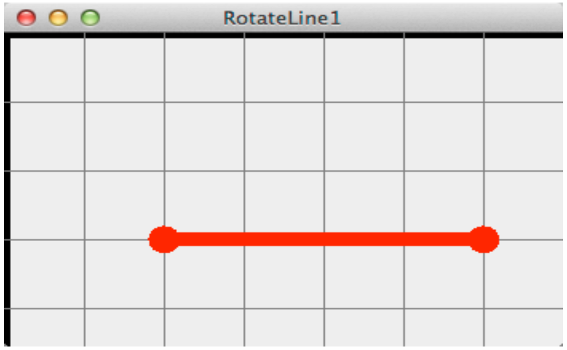




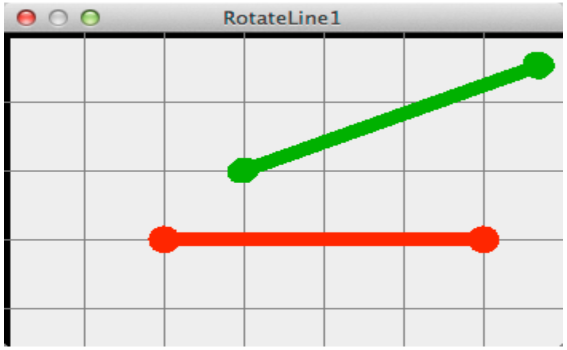
What steps do I need to take to draw the transformed bar?

Multiplication Order – Wrong Way

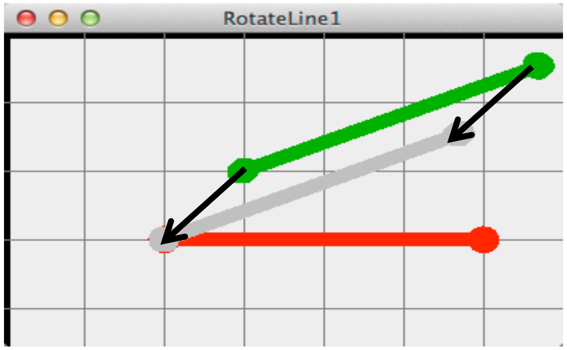
- Beginning situation



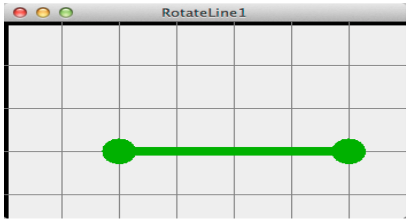
- Rotate 22.5 degrees (π/8 rads)
    - Oops, both endpoints moved



- Could try translating to return **a** to its original position
    - But by how much?

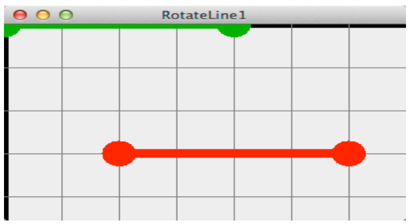


Multiplication Order - Correct Way



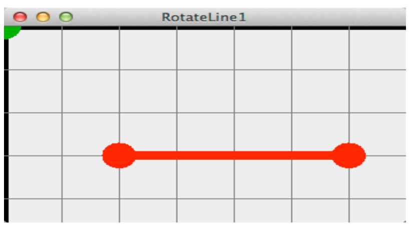
$$p' = I \cdot p$$

<u>Remember</u>: Scaling and rotation are both about the origin



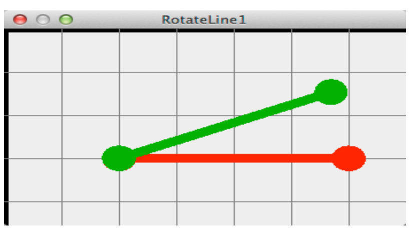
$$p' = T_{(-x1,-y1)} I \cdot p$$

1. Translate shape to the origin



$$p' = R_{(\pi/8)} T_{(-x1,-y1)} \cdot p$$

2. Rotate

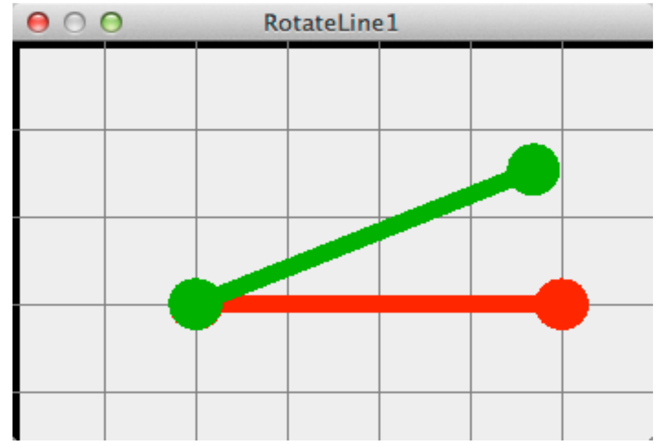


$$p' = T_{(x1,y1)} R_{(\pi/8)} T_{(-x1,-y1)} \cdot p$$

3. Translate back to where you want it

Approach 1: Do it in Java

- We can mimic this approach in Java
  - Create a shape model, containing points (e.g. line)
  - Create instances of this shape model (e.g. red, green)
  - Use the Java `AffineTransform` static methods to create transform, scale or rotation matrices reflecting the operations needed.
    - AffineTransform.getRotateInstance
    - AffineTransform.getTranslateInstance
    - AffineTransform.getScaleInstance
  - For each of these operations, use the resulting matrix to transform the points in your model.
    - matrix.transform (p1, p1);

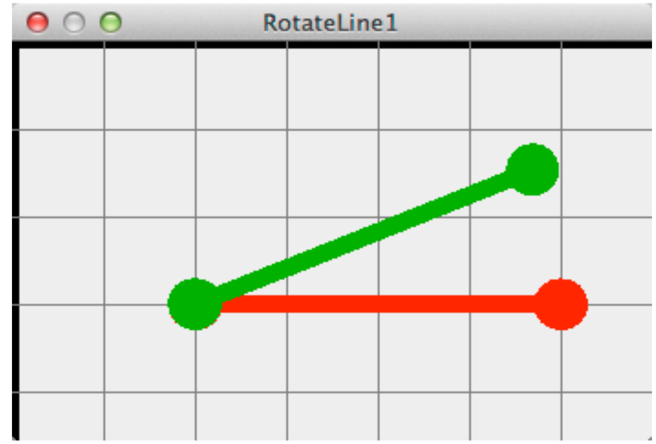


ShapeModel1.java

Approach 2: Do it in Java

- We can also use the Graphics Context to describe transformations that should be applied to shapes before they are drawn.
  - g2.rotate(r)
  - g2.translate(x,y)
  - g2.scale(s)

- Notice that in Java, the origin is in the top-left corner of the screen.
  - Previous demos showed standard Cartesean coordinate system
  - In computer graphics, the origin is typically top-left



ShapeModel2.java

- We can draw at fixed locations, or use transformations.
- `RotateLine1` mixes affine transforms and drawing at fixed positions. It's easier if you don't.
- `RotateLine2` is a cleaner implementation that positions the shape model at the origin, and relies on transformations (trsnslations) to position the shapes.
  - Recommendation: draw at the origin and then translate to the position you want!.

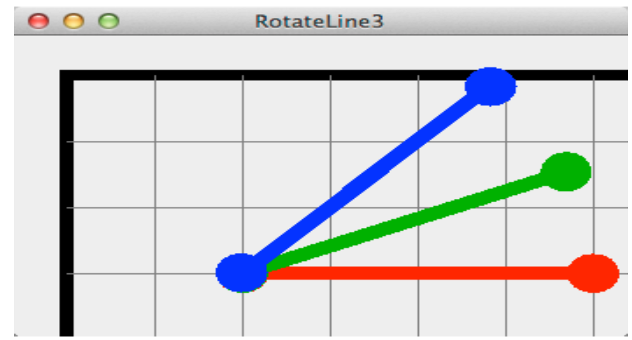
```
private static final int x1 = 100;
private static final int y1 = 150;
private static final int len = 200;

// Draw bar at the origin
private void drawBar(Graphics2D g2) {
    g2.drawLine(0, 0, len, 0);
    g2.fillOval(-15, -15, 30, 30);
    g2.fillOval(len - 15, -15, 30, 30);
}
```

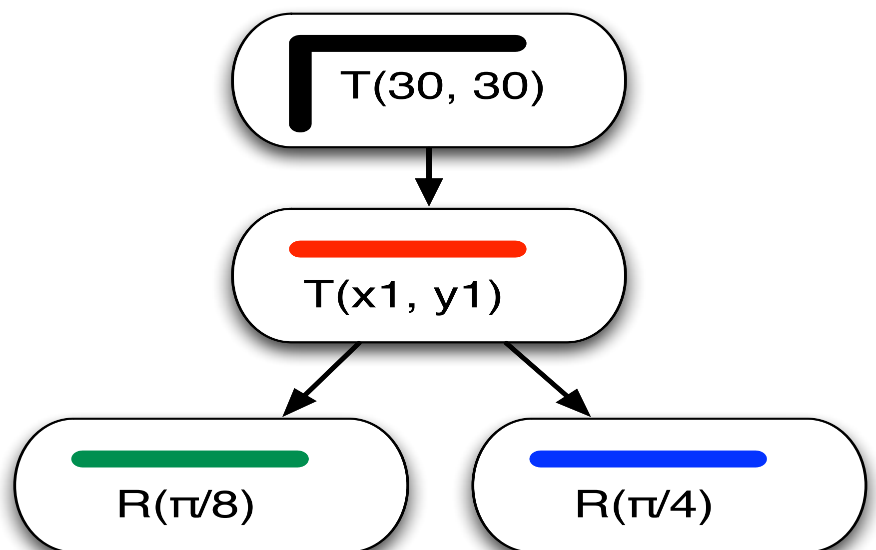
Code Review: RotateLine2 (2 of 3)

```
/* Draw everything. */
   public void paintComponent(Graphics g) {
       Graphics2D g2 = (Graphics2D) g;
       g2.setStroke(new BasicStroke(8));
       this.drawAxes(g2);

       g2.setStroke(new BasicStroke(10));
       g2.setColor(Color.RED);
       // Transform WRT previous transformations (none)
       g2.translate(x1, y1);
       this.drawBar(g2);

       // Transform WRT previous transformations
       g2.rotate(-Math.PI/8);
       g2.setColor(Color.GREEN.darker());
       this.drawBar(g2);

   }
```

Code Review: RotateLine3 (1 of 3)

```
public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(8));
    g2.translate(30, 30);
    this.drawAxes(g2);

        g2.setStroke(new BasicStroke(10));
        g2.setColor(Color.RED);
        g2.translate(x1, y1);
    this.drawBar(g2);

    AffineTransform at = g2.getTransform();

    g2.rotate(-Math.PI/8);
    g2.setColor(Color.GREEN.darker());
    this.drawBar(g2);

    g2.setTransform(at);

    g2.rotate(-Math.PI/4);
    g2.setColor(Color.BLUE);
    this.drawBar(g2);
}
```

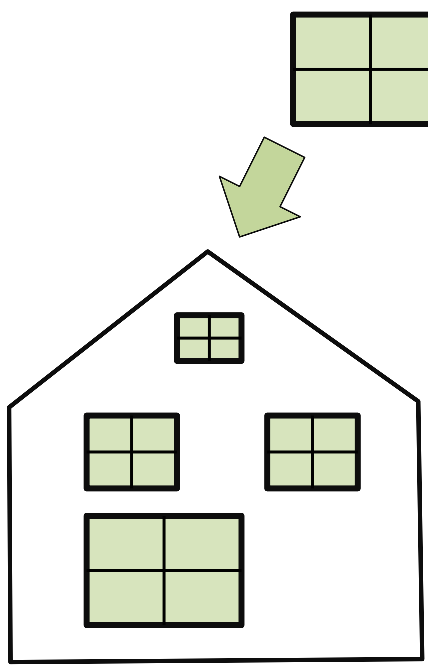


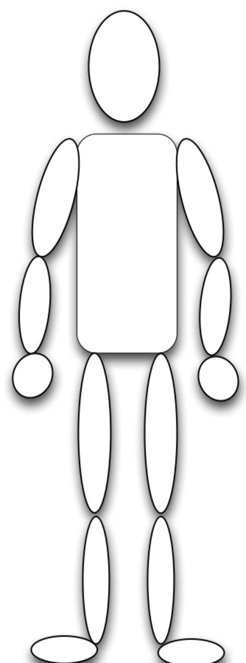


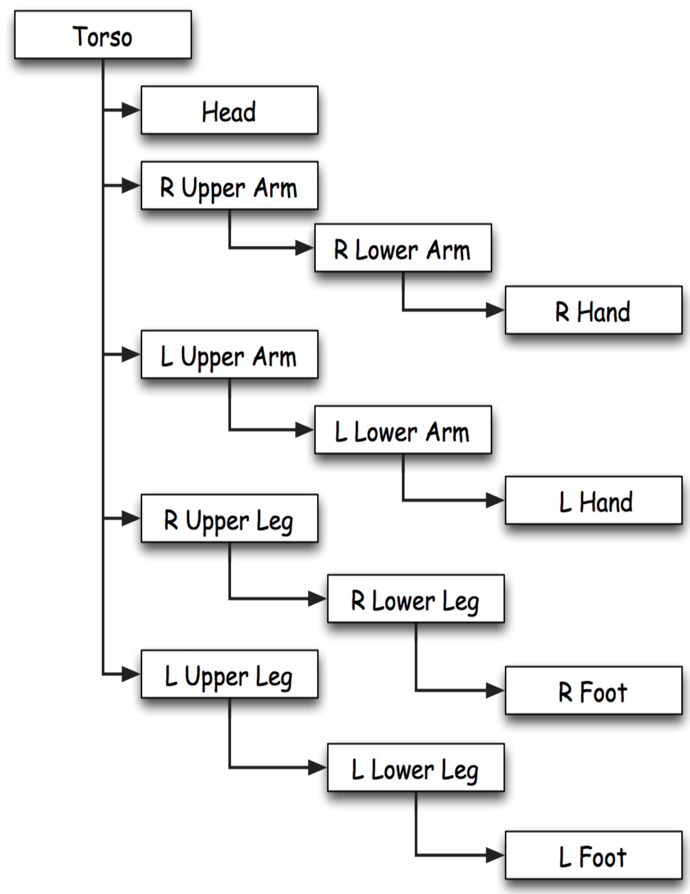
Scene Graphs

Scene Graph

- Mechanism for drawing a series of connected components
  - Each part has a transform matrix
  - Each part draws its children relative to itself

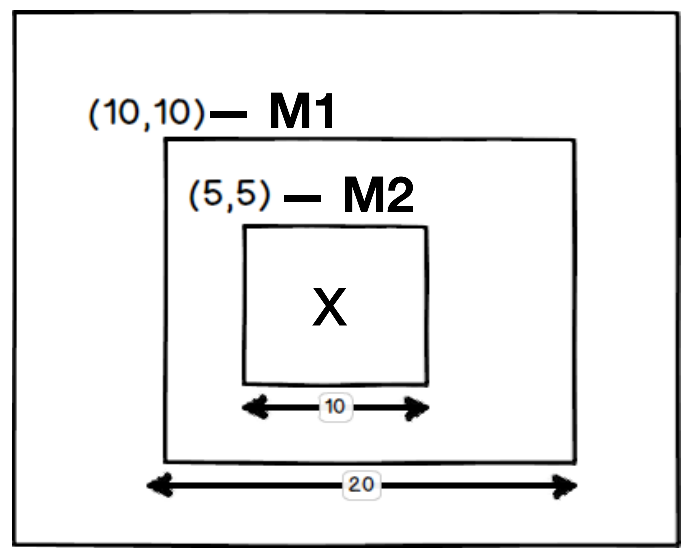






Affine Transformations + Interactor Tree

- An interactor tree is a type of scene graph.
- Each component has an affine transformation matrix, and a paint routine.
  - Matrix describes the components location relative to its parent (i.e. what translations should be performed on it)
  - The paint routine concatenates the parents affine transformation matrix with the components matrix and then paints using the resulting matrix.
  - e.g.
    - M1: translate (10,10)
    - M2: translate (5,5)



As we navigate the interactor tree, we combine successive matrices to reflect the way that each component is positioned relative to it's parent.

Each component should:

1. paint itself (using it's affine transformation matrix)
2. for each child

- save the current affine transform
- calculate a new transform matrix using the current transform matrix and the location of the child (i.e. current * child's translation transformation)
- tell child to paint themselves using the new affine transform matrix
- return the original affine transform matrix

Useful Graphics2D methods

- AffineTransform getTransform(),
  void setTransform(AffineTransform Tx)
  - Returns/sets a copy of the current Transform in the Graphics2D context.
- void rotate(double theta),
  void rotate(double theta, double x, double y)
  - Concatenates the current Graphics2D Transform with a rotation transform.
  - Second variant translates origin to (x,y), rotates, and translates origin (-x, -y).
- void scale(double sx, double sy)
  - Concatenates the current Graphics2D Transform with a scaling transformation. Subsequent rendering is resized according to the specified scaling factors relative to the previous scaling.
- void translate(double tx, double ty)
  - Concatenates the current Graphics2D Transform with a translation transform.

Java2D AffineTransform Class

- AffineTransform handles all matrix manipulations
  - A bit more control than Graphics2D

- Static Methods
  - static AffineTransform getRotateInstance(double theta)
  - static AffineTransform getRotateInstance(double theta,
    double anchorx, double anchory)
  - static AffineTransform getScaleInstance(
    double sx, double sy)
  - static AffineTransform getTranslateInstance(
    double tx, double ty)

Java2D AffineTransform Class

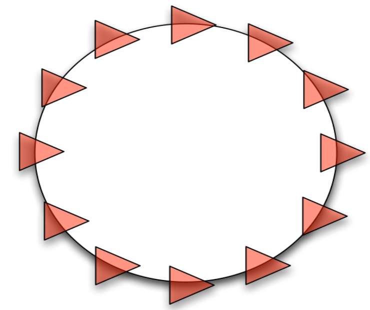
- Concatenation methods
  - void rotate(double theta),
    void rotate(double theta, double anchorx, double anchory)
  - void scale(double sx, double sy)
  - void translate(double tx, double ty)
  - void concatenate(AffineTransform Tx)

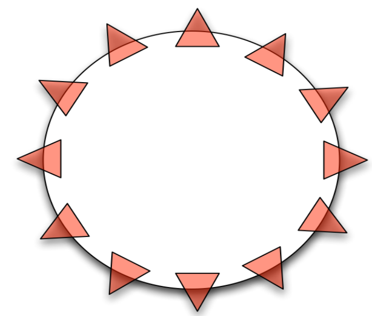
- Other Methods
  - AffineTransform createInverse()
  - void transform(Point2D[ ] ptSrc, int srcOff, *
    Point2D[ ] ptDst, int dstOff, int numPts)

Example: Rotate Triangle

We can develop the transformations to animate a triangle (drawn at the origin) in a circle in two different ways:

```
public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;

        g2.translate(x_center, y_center);
        g2.rotate(cur_angle);
        //g2.translate(radius, 0);
        //g2.rotate(-cur_angle);

        g2.setColor(Color.RED);
        g2.fillPolygon(triangle);
        g2.setColor(Color.BLACK);
        g2.drawPolygon(triangle);
}
```





RotateTriangle.java

Benefits of Geometric Manipulations

- Allow reuse of objects in scenes
  - Can create multiple instances by translating model of object and re-rendering
- Allows specification of object in its own coordinate system
  - Don’t need to define object in terms of its screen location or orientation
- Simplifies remapping of models after a change
  - e.g. animation

Inside Tests

We also need to transform any coordinates in events as the events are passed down the interactor tree.

- This allows us to translate between global and local (or relative) coordinate systems.

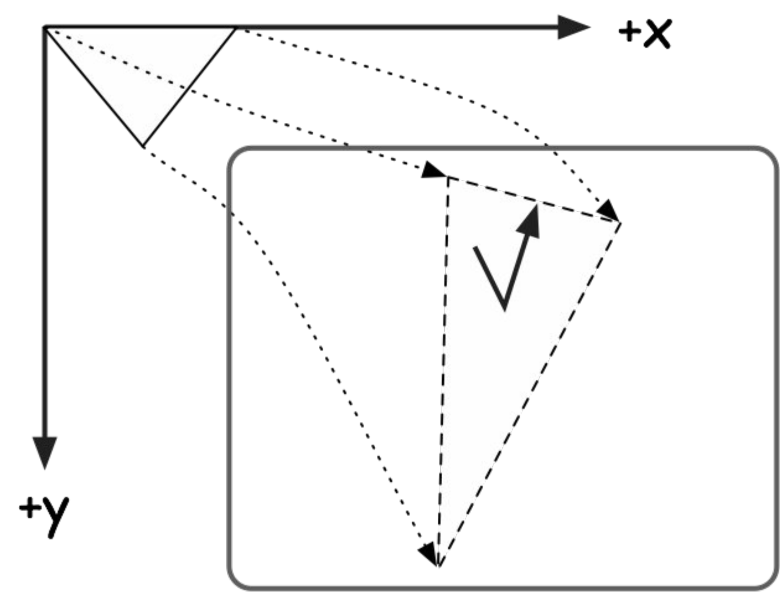
For example, a mouse event in the window’s coordinates will need its coordinates translated to a child component’s local (relative) location prior to passing it down.

- Affine transforms can be used for this, too.

Mouse and shape model must use the same coordinate system

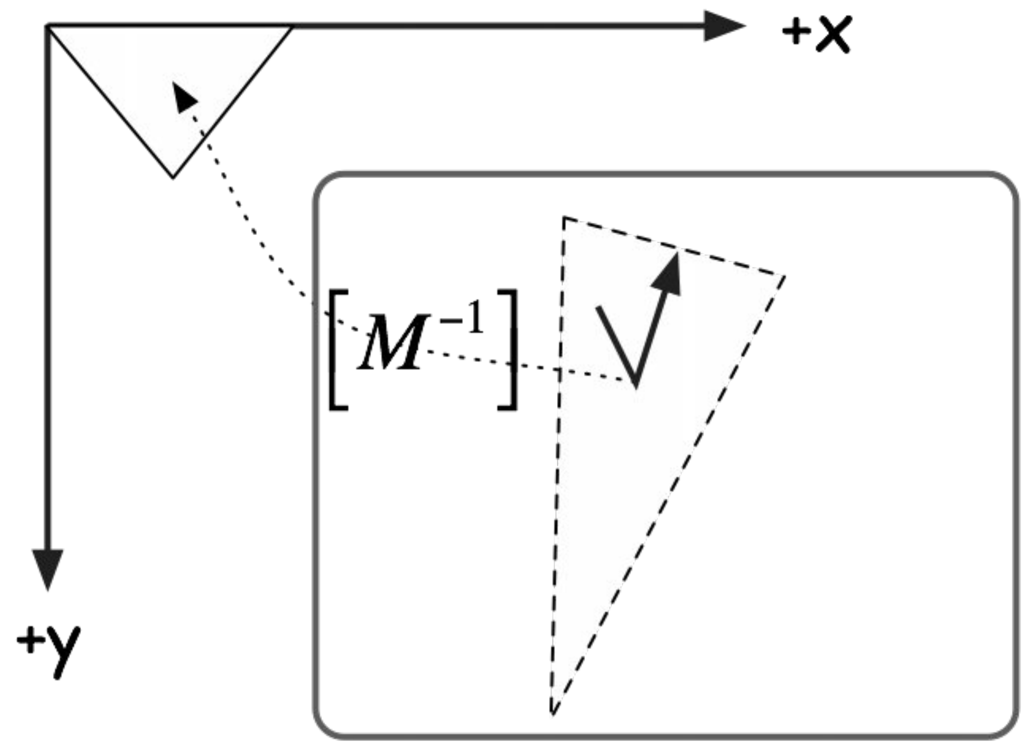
Two options:

1. Transform **mouse to model** coordinates, or
2. Transform **model to mouse** coordinates



Option 1: Transform Mouse to Model Coordinates

- Only one transformation
- Selection: Within 3 pixels of a line in screen coordinates is how far in model coordinates?
- Uniform scaling…
- Maintaining the inverse



Option 2: Transform Model to Mouse Coordinates

- Many transformations
- Manipulations (e.g. dragging) must be transformed back into model coordinates
- Not recommended