Graphics

Drawing in Swing

Shape models

Transformation basics

How does Java draw the screen?

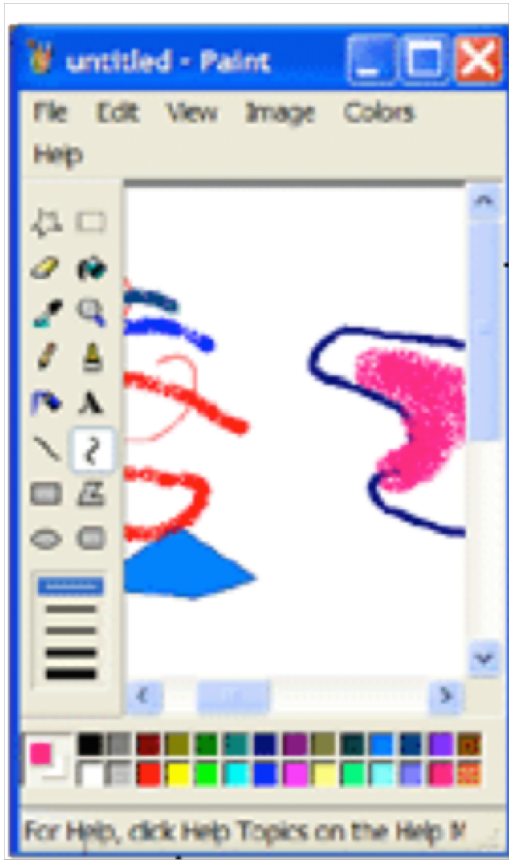
The Swing UI consists of a series of Swing widgets, in a hierarchy (interactor tree).

- The UI thread periodically walks the interactor tree, telling each component in the tree to draw itself.

Every component on-screen has it's own paintComponent method that defines how that component is drawn (derived from originally defined in JComponent).

GC (graphics context) is passed into paintComponent

- Contains state, and built-in methods for drawing primitives




```
// JButton drawing method
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    g2.setStroke(new BasicStroke(5));
    g2.setColor(Color.BLACK);
    g2.drawRect(x, y, width, height);
    g2.drawString(buttonCaption, x+10, y+25);
}
```

Drawing with Swing

Two different approaches to drawing an interface:

1. **Use Swing components**
   - Build your UI entirely out of Swing (GUI) components
   - Don't need to modify paintComponent method, already defined
   - UI is limited to the components that you are given
2. **Draw on a canvas**
   - Use a basic Swing component as a *canvas*, and draw your UI on it using primitives (rectangles, arcs, etc.)
   - Can build anything that you can draw using primitives (including "custom widgets")

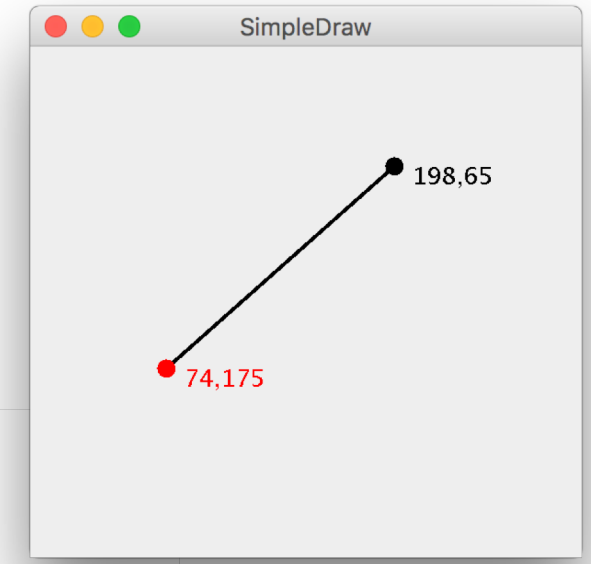
You can also combine these e.g. build a UI partially out of Swing components, and then build custom classes to draw other parts of the UI.

- This is what you're likely to do for A2!

Drawing with Swing

To draw on a canvas using Swing:

- Create a canvas, using the JComponent as the base class.
- Override the paintComponent method, and define your own behaviour.




```
// JComponent is a base class for custom components
public class SimpleDraw extends JComponent {

    Run | Debug
    public static void main(String[] args) {
        JFrame f = new JFrame("SimpleDraw"); // jframe is the app window
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 400); // window size

        SimpleDraw canvas = new SimpleDraw();
        f.setContentPane(canvas); // add canvas to jframe
        f.setVisible(true); // show the window
    }
```

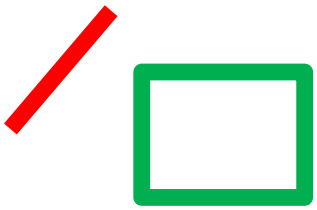
```
// custom graphics drawing
// overrode method from the JComponent class
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    g2.setStroke(new BasicStroke(32));
    g2.setColor(Color.BLUE);
    g2.drawLine(0, 0, 50, 75);
    g2.drawString("some text", 100, 150);
}
```

Drawing Primitives in Java

- The Graphics Context (GC) has methods to support all of our approaches.
  - See java.awt.Graphics in JDK for full documentation

Pixel

```
drawImage(image, x, y)
```



Stroke

```
drawLine(x1, y1, x2, y2)
drawRect(x, y, width, height)
drawOval(x1, x2, width, height)
```

Region

```
drawString(char, x, y)
```

Graphic Models and Images

We need a better way to model our shapes.

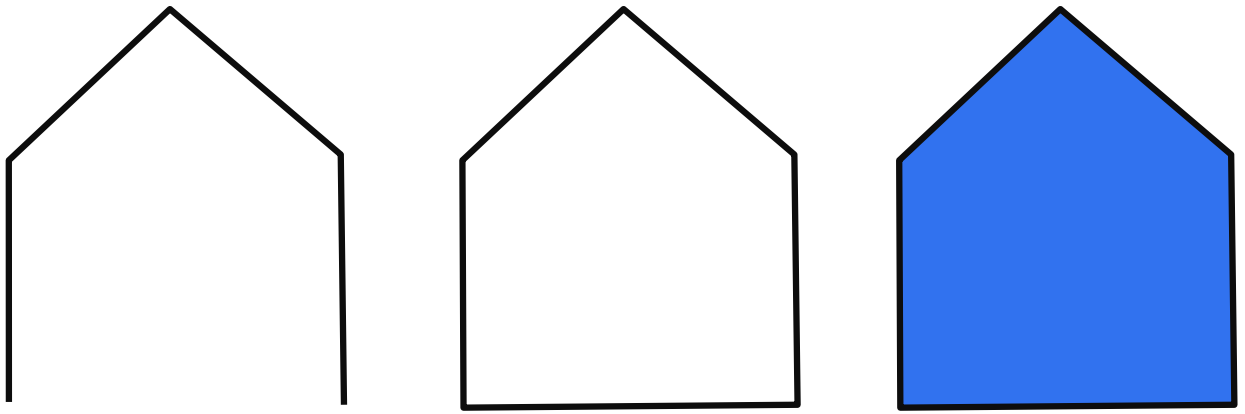
Computer Graphics is the creation, storage, and manipulation of images and their models. We can use principles from graphics when defining our UI elements.

- **Model:** a mathematical representation of an image containing the important properties of an object (location, size, orientation, color, texture, etc.) in data structures
- **Rendering:** Using the properties of the model to create an image to display on the screen
- **Image:** the rendered model

Shape Model

We'll use a **shape model** to represent primitive shapes. A shape model consists of

- an array of points: $\{P_1, P_2, \ldots, P_n\}$ that we connect to draw a shape
- properties that determine how the shape is drawn
  - isClosed flag (shape is polyline or polygon)
  - isFilled flag (polygon is filled or not)
  - stroke thickness, colours, etc.

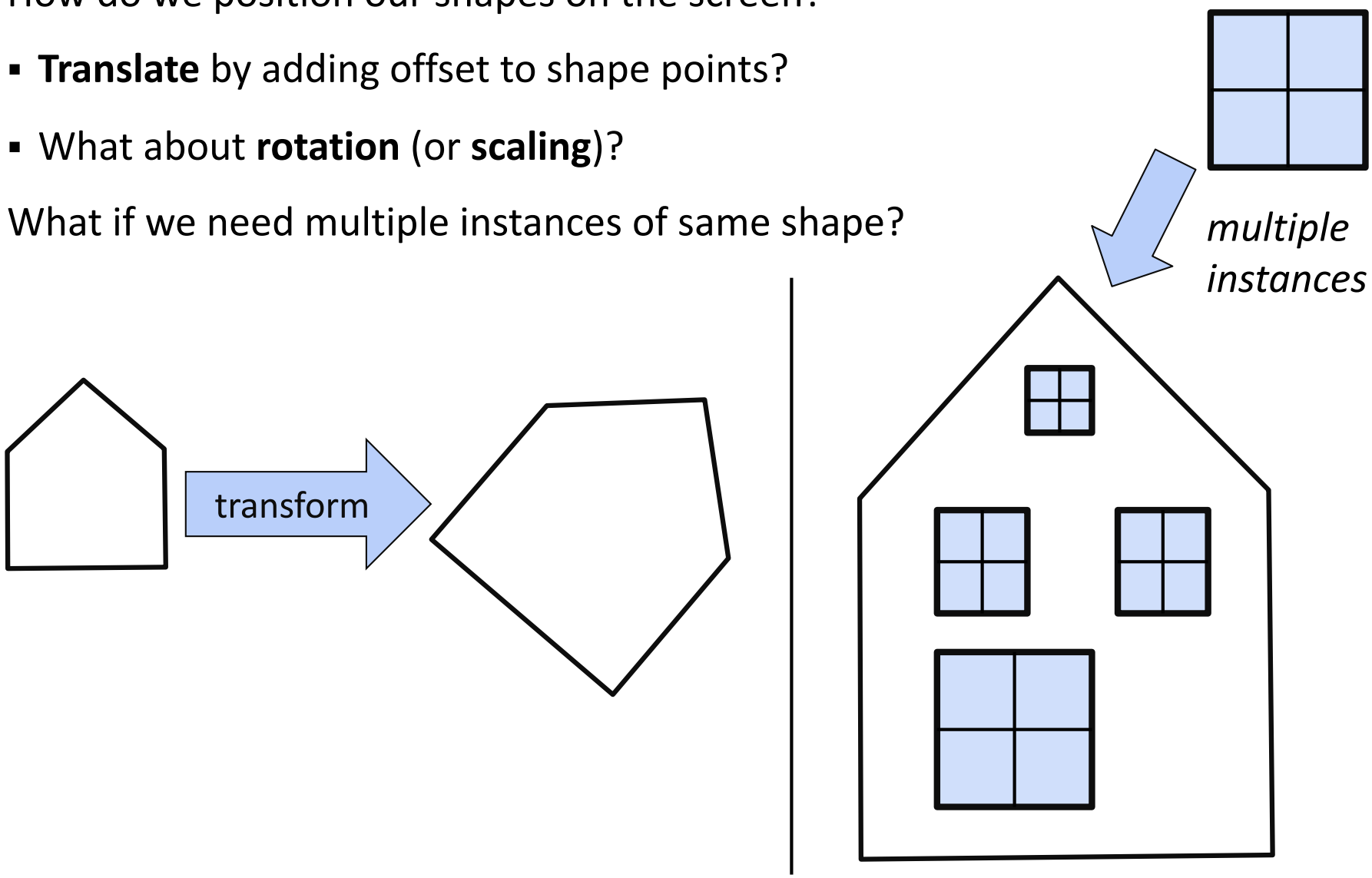


Manipulating a Shape Model

How do we position our shapes on the screen?

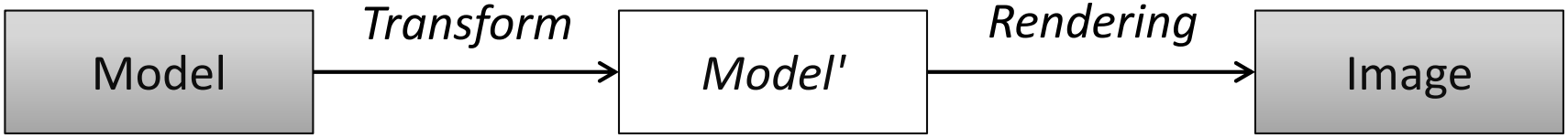
- **Translate** by adding offset to shape points?
- What about **rotation** (or **scaling**)?

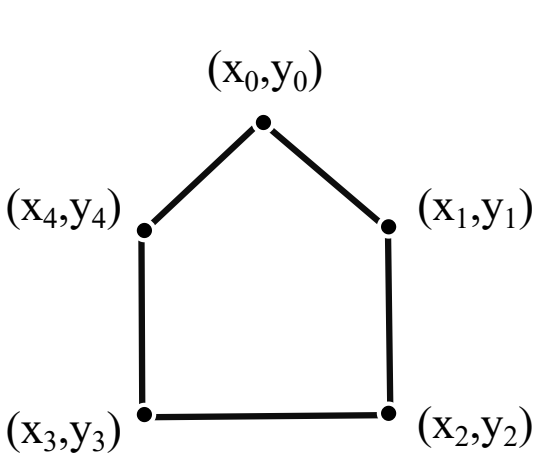
What if we need multiple instances of same shape?

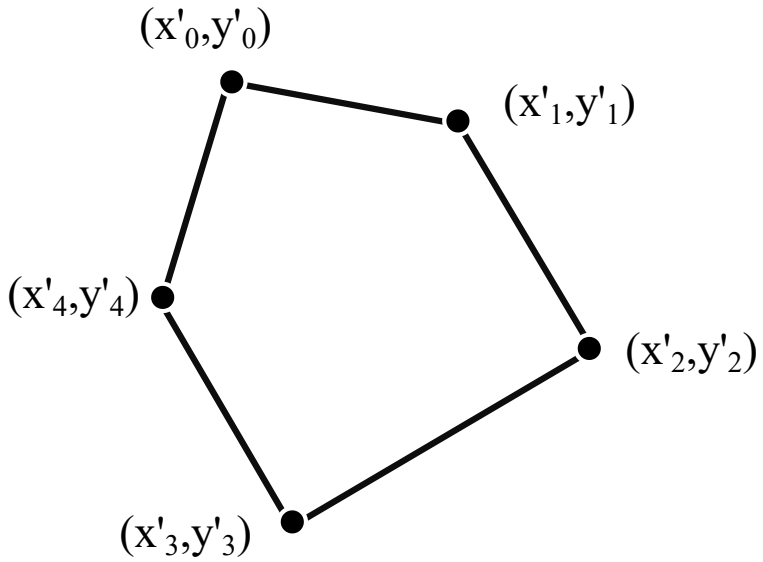


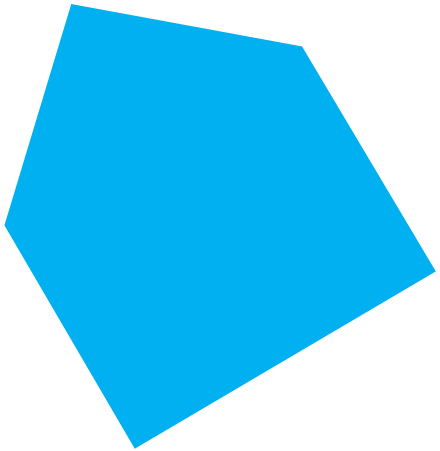
Transforming Shape Models

- Shape model is in a base coordinate frame
- The model is transformed to a location before rendering
  - **Translation**, **Rotation** and **Scaling** are typical operations that we support





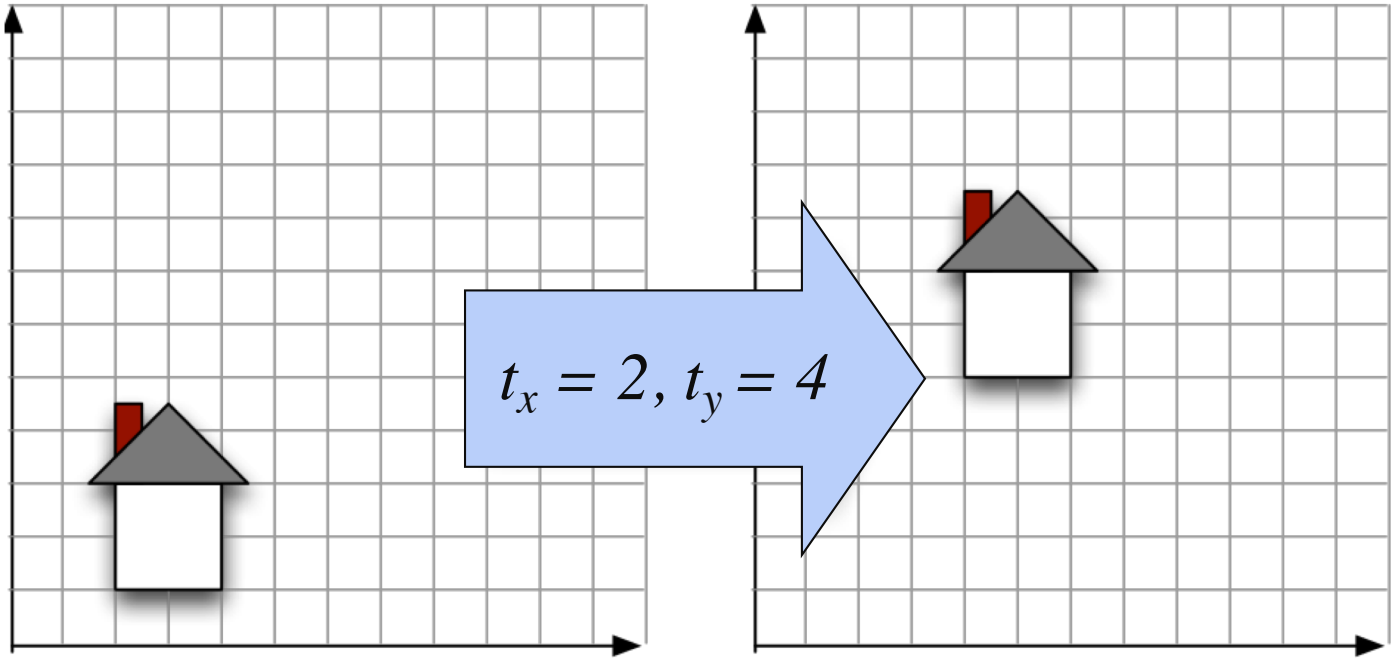




$$(x'_i,y'_i) = \boldsymbol{f}(x_i,y_i)$$

Translation

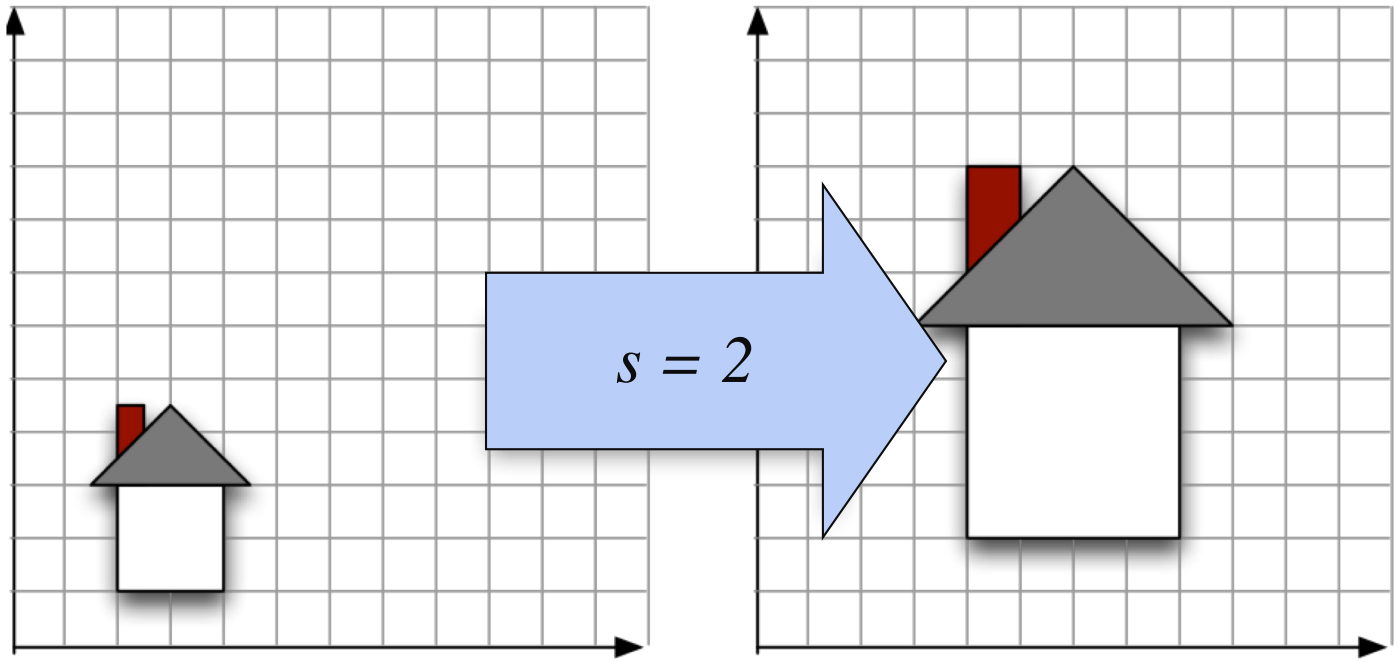
- translate: add a scalar to coordinates of each component



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

Uniform Scaling

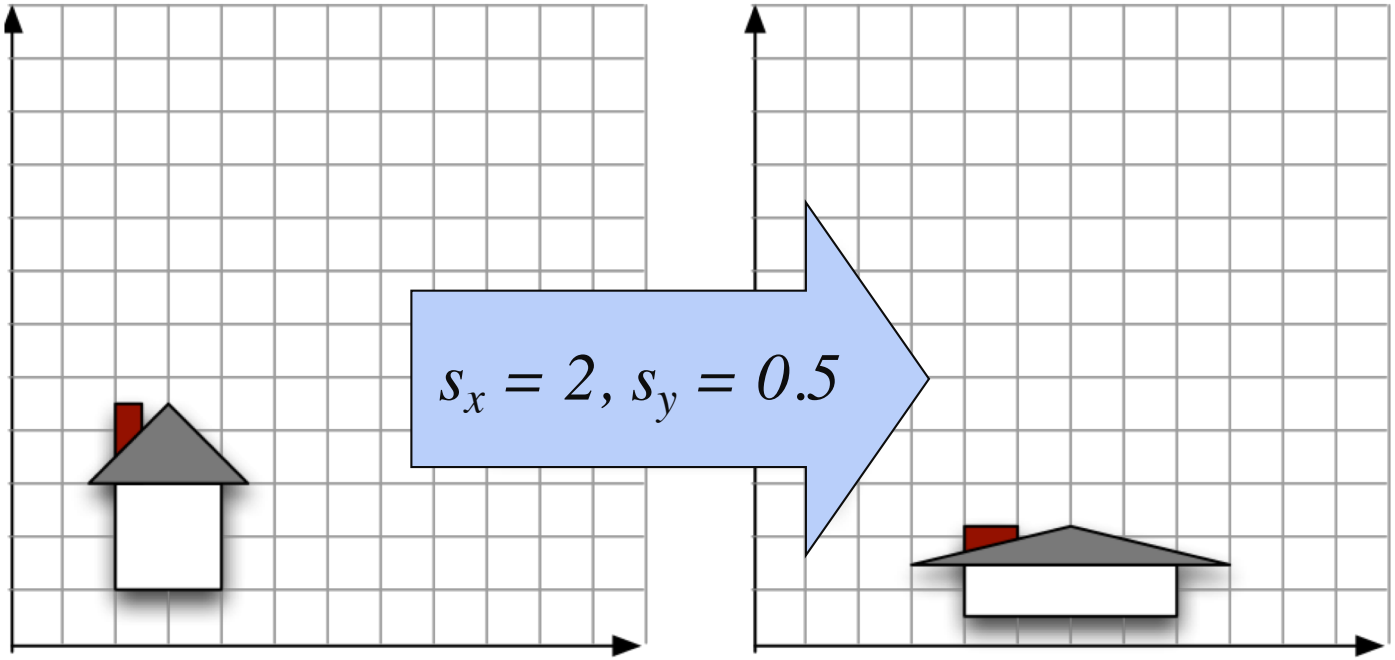
- uniform scale: multiply each component by same scalar



$$x' = x \times s$$
$$y' = y \times s$$

Non-Uniform Scaling

- scale: multiply each component by different scalar

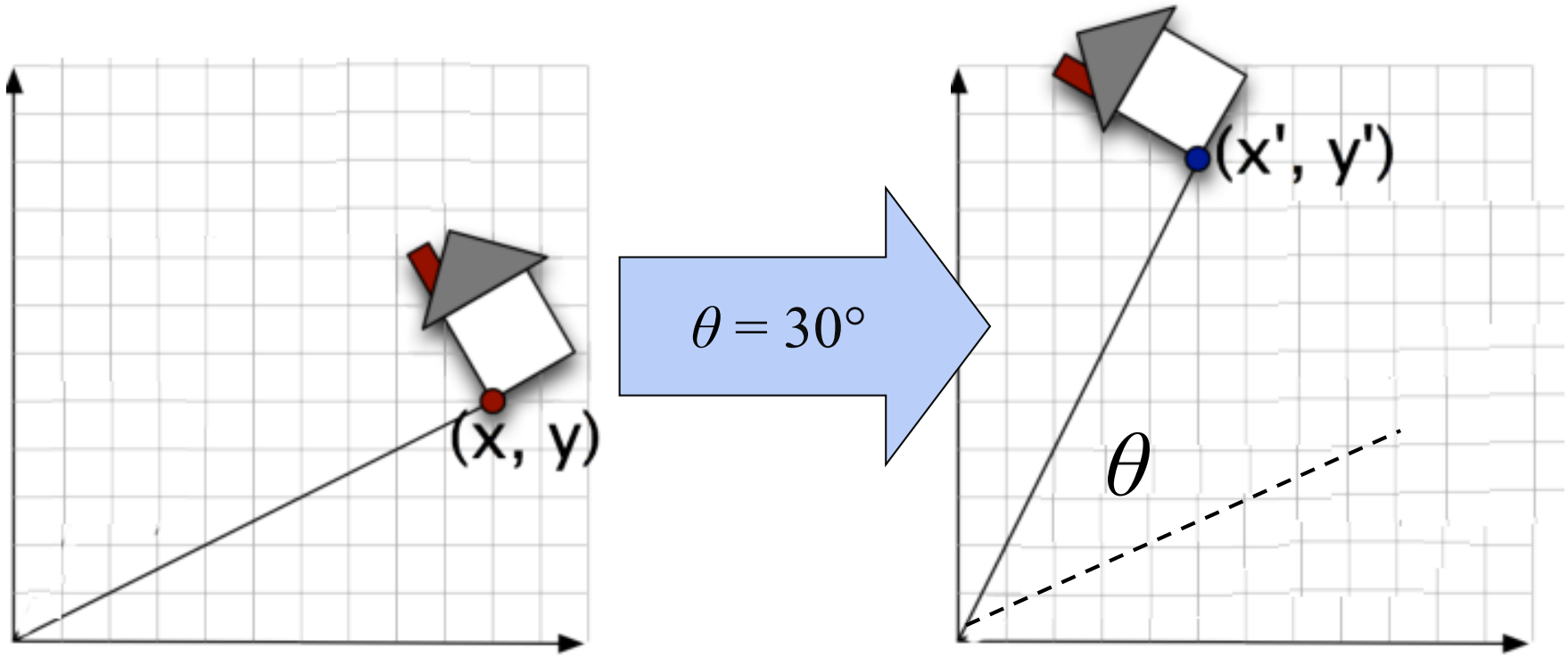


$$x' = x \times s_x$$

$$y' = y \times s_y$$

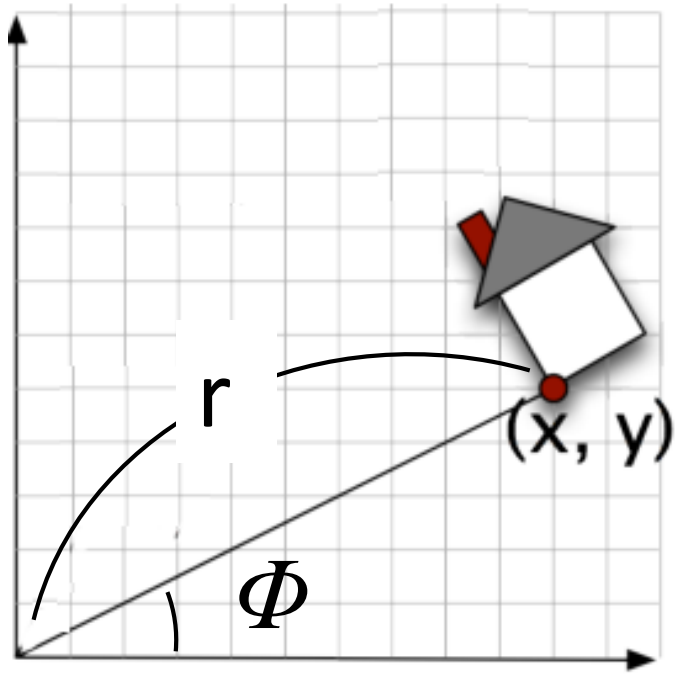
Rotation

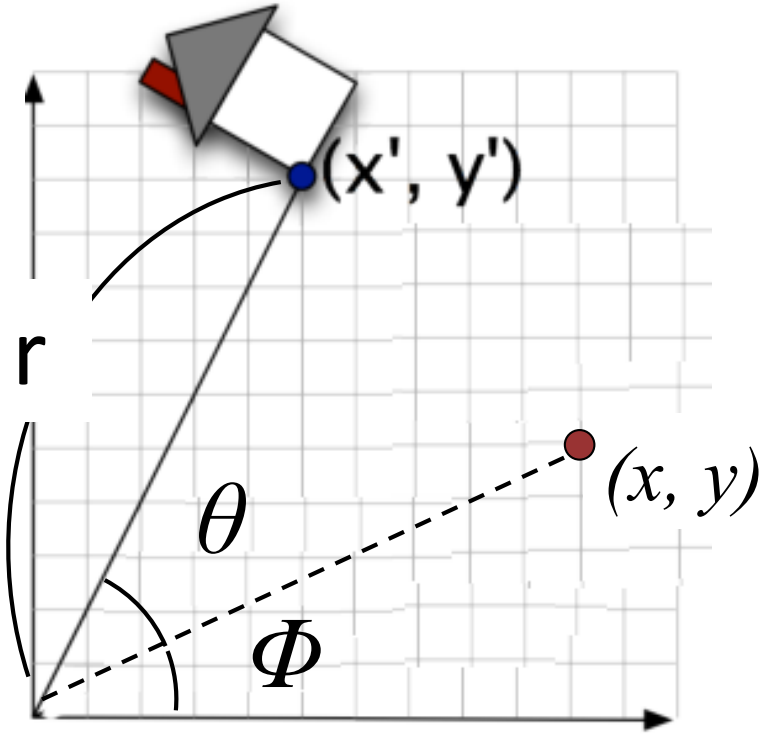
- rotate: component is some function of x, y, Θ



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

Rotation





$$(1)\ x = r\cos(\Phi)$$

$$(2)\ y = r\sin(\Phi)$$

$$(3)\ x' = r\cos(\Phi+\theta)$$

$$(4)\ y' = r\sin(\Phi+\theta)$$

Rotation

- Use these Identities

$$\cos(\Phi+\theta) = \cos(\Phi)\cos(\theta) - \sin(\Phi)\sin(\theta)$$
$$\sin(\Phi+\theta) = \cos(\Phi)\sin(\theta) + \sin(\Phi)\cos(\theta)$$

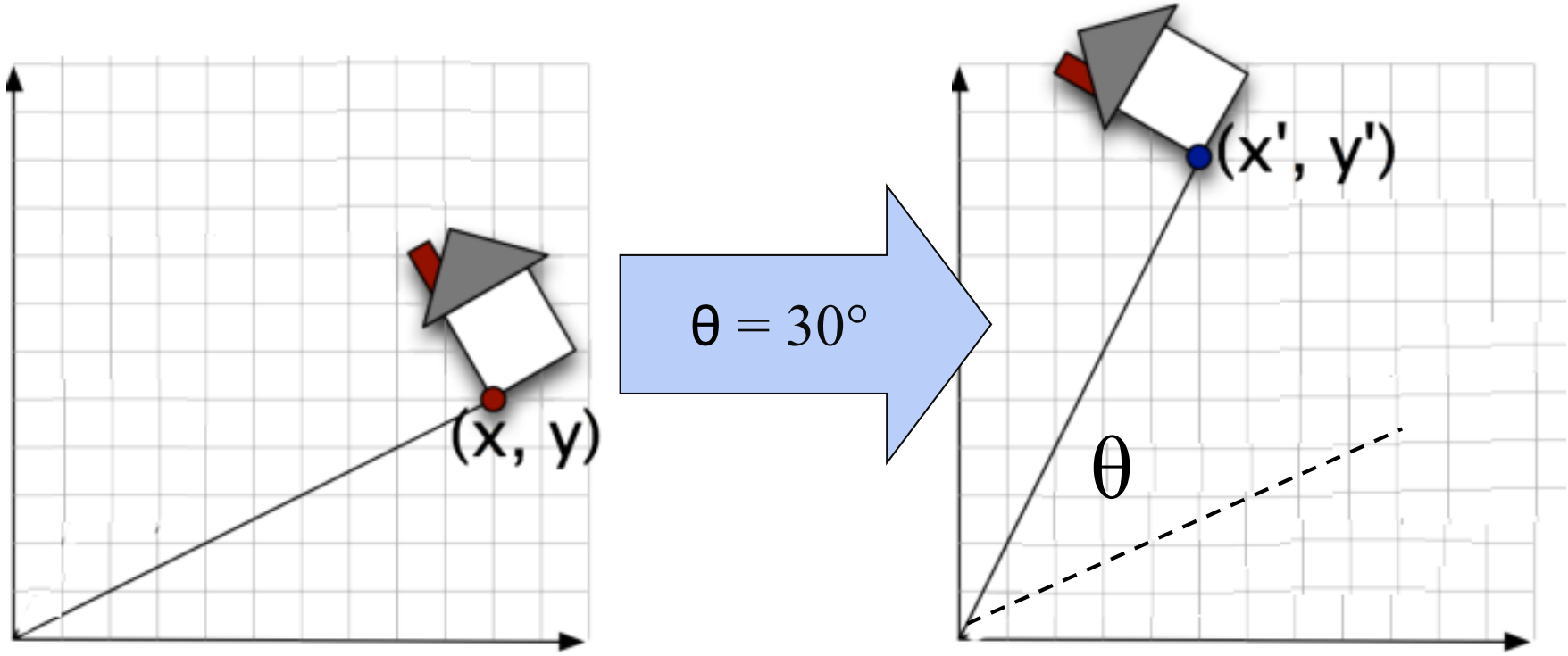
$$(3)\ x' = r\cos(\Phi+\theta)$$
$$= r\cos(\Phi)\cos(\theta) - r\sin(\Phi)\sin(\theta)$$
$$= x\cos(\theta) - y\sin(\theta)$$

$$(4)\ y' = r\sin(\Phi+\theta)$$
$$= r\cos(\Phi)\sin(\theta) + r\sin(\Phi)\cos(\theta)$$
$$= x\sin(\theta) + y\cos(\theta)$$

$$(1)\ x = r\cos(\Phi)$$
$$(2)\ y = r\sin(\Phi)$$

Rotation

- rotate: component is a function of x, y, Θ



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

Combining Transformations

- Rotate:

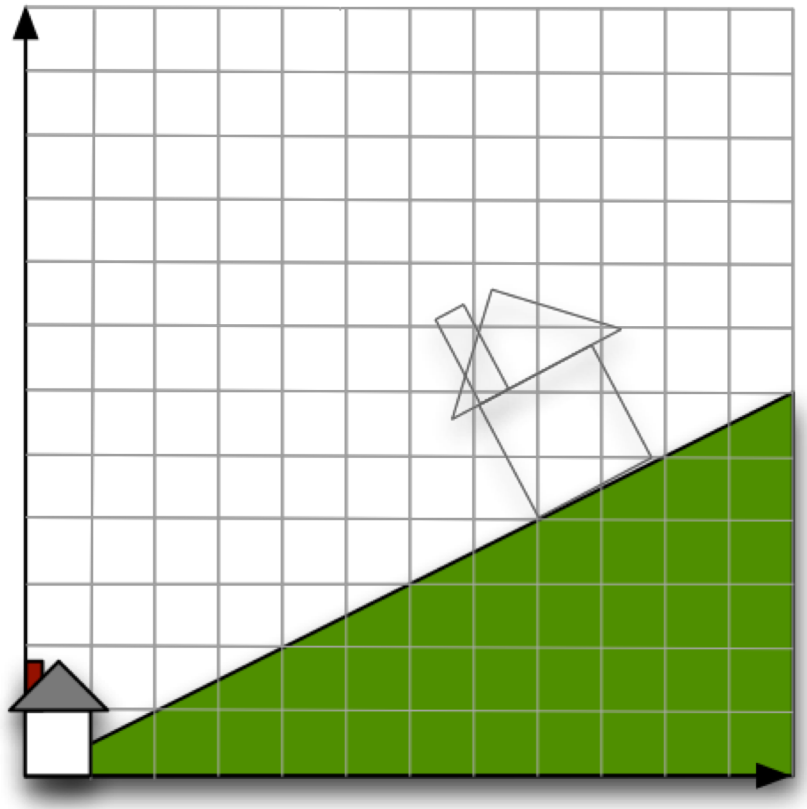
$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

- Translate:

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

- Scale:

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



Combining Transformations: Step 1 - Scale

- Rotate:

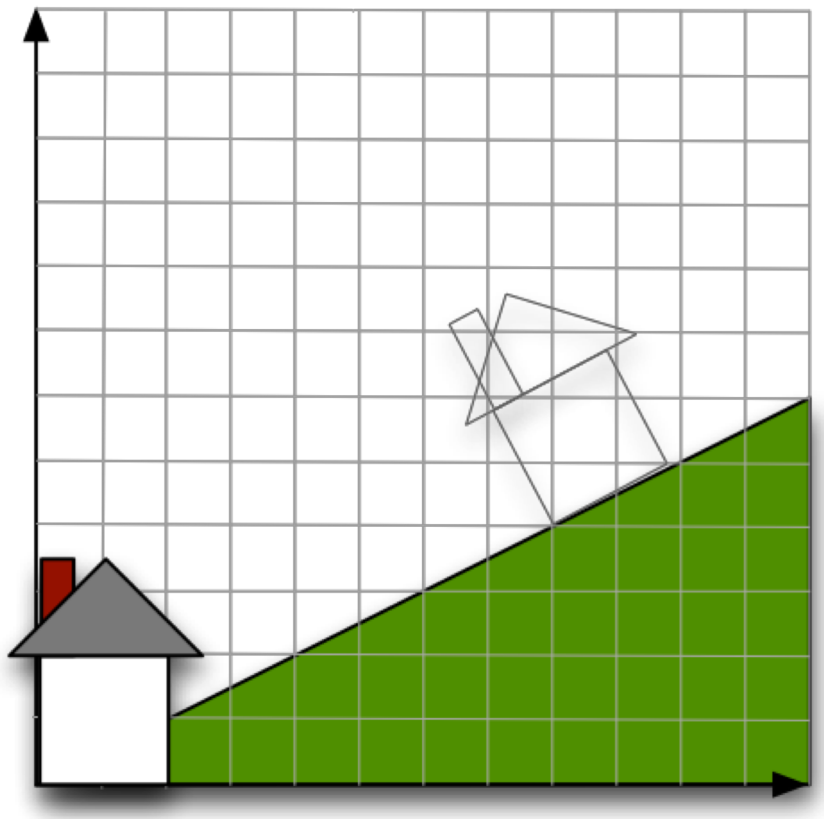
$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

- Translate:

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

- Scale:

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



$$x' = 2x$$
$$y' = 2y$$

Combining Transformations: Step 2 - Rotate

- Rotate:

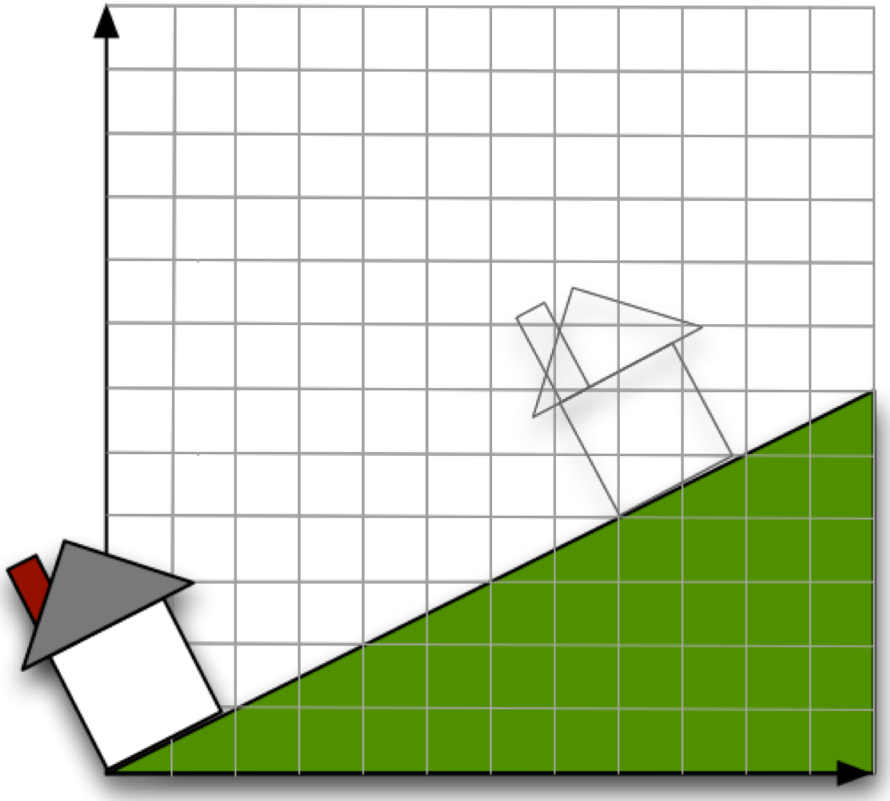
$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

- Translate:

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

- Scale:

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



$$x'' = 2x\cos(30) - 2y\sin(30)$$
$$y'' = 2x\sin(30) + 2y\cos(30)$$

Combining Transformations: Step 3 - Translate

- Rotate:

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

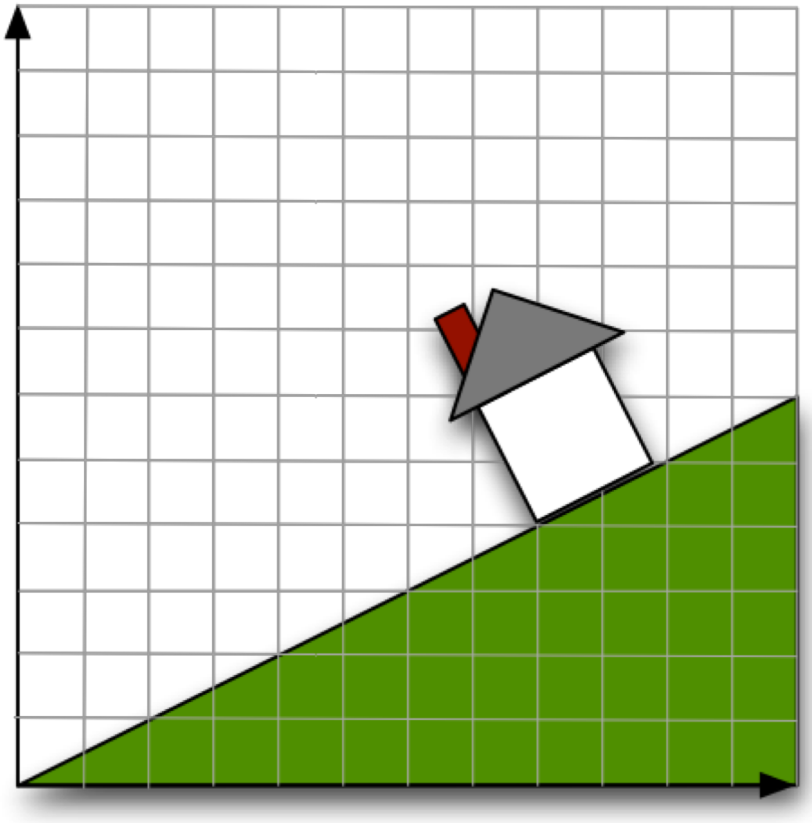
- Translate:

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

- Scale:

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

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



$$x''' = 2x\cos(30) - 2y\sin(30) + 8$$
$$y''' = 2x\sin(30) + 2y\cos(30) + 4$$

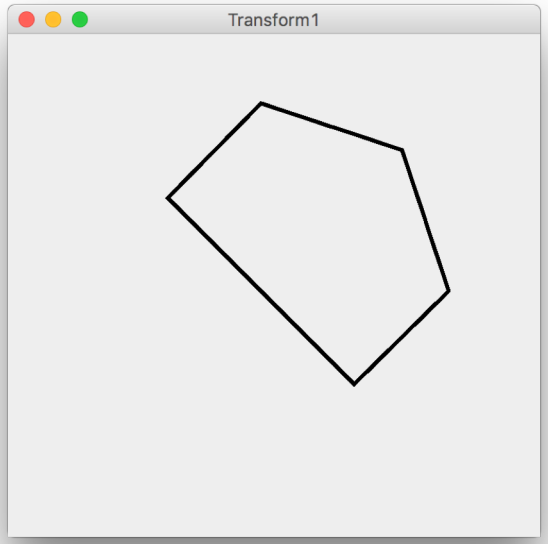
Transform1.java

```
// the house shape model (centred at top left corner)
private Polygon s= new Polygon(new int[] {-50, 50,  50, 0, -50},
                               new int[] {75, 75, -25, -75, -25}, 5);
...

// get copy of shape
Polygon ts = new Polygon(s.xpoints, s.ypoints, s.npoints);

// transform by hand
scale(ts, 2, 1);
rotate(ts, 45);
translate(ts, M.x, M.y);

g2.setStroke(new BasicStroke(3));
g2.drawPolygon(ts.xpoints, ts.ypoints, ...);
```



NOTE: Doing transformations in this way is not optimal.

vecmath.jar

- you need vecmath.jar to run these demos
- vecmath.jar needs to be included when compiling and running

```
javac -cp vecmath.jar Transform1.java
java -cp "vecmath.jar:." Transform1
```

- Makefile should have everything you need