# Graphics

## Graphics Pipeline & Transformations

CS 349

## June 7

# Graphics Pipeline & Transformations

# Graphic Models and Images

Computer Graphics is the creation, storage, and manipulation of images and their models.

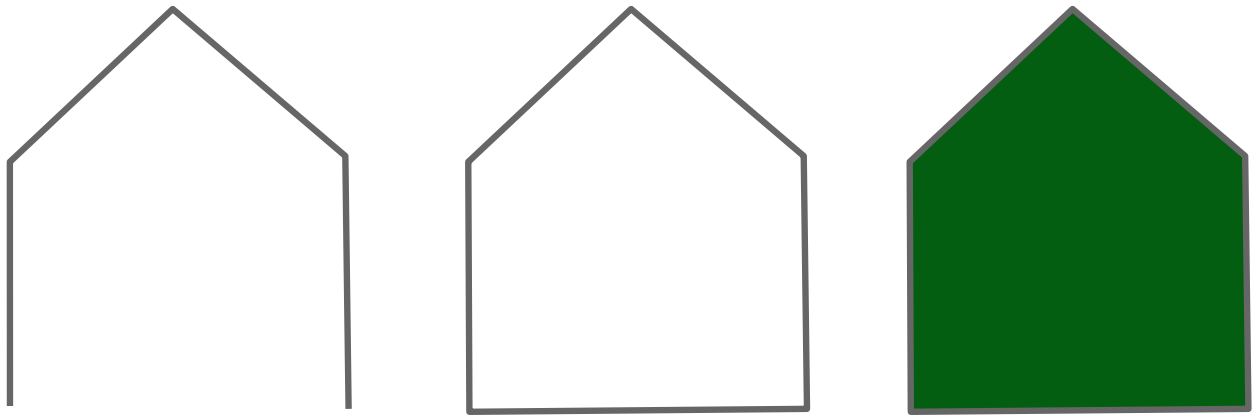
- **Model:** a mathematical representation of an image containing the important properties of an object (location, size, orientation, color, texture, etc.) in data structures
- **Rendering:** Using the properties of the model to create an image to display on the screen
- **Image:** The rendered model

# Shape Model

An array of points (or vertices) $\{P_1, P_2, \ldots, P_n\}$ defines a shape.

Properties that determine how the shape is drawn

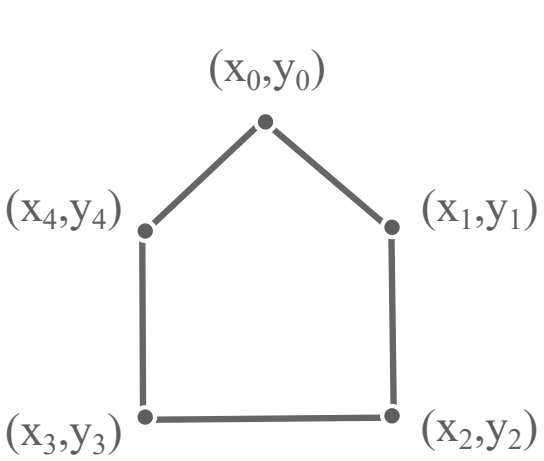
- isClosed flag (shape is polyline or polygon)
- isFilled flag (polygon is filled or not)
- stroke thickness, colours, etc.

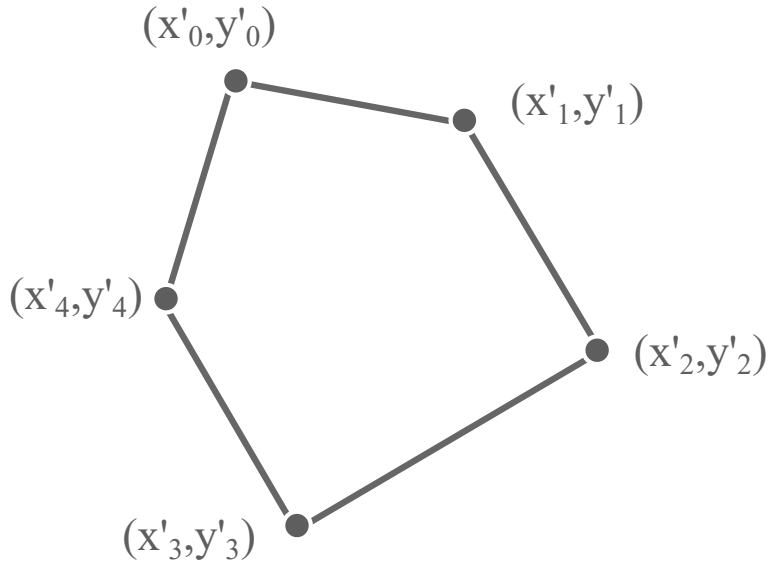


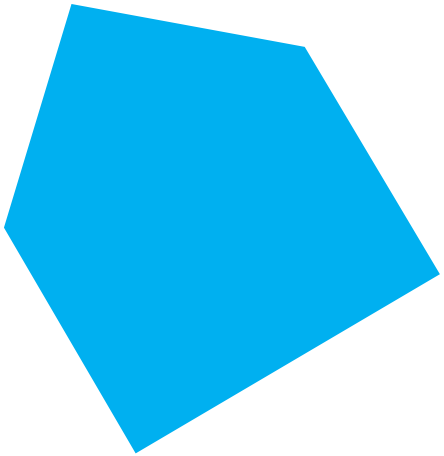
# Transforming Shape Models

Shape model points are defined relative to a base coordinate system.

The model is transformed to a location before rendering through translation, rotation, and scaling.

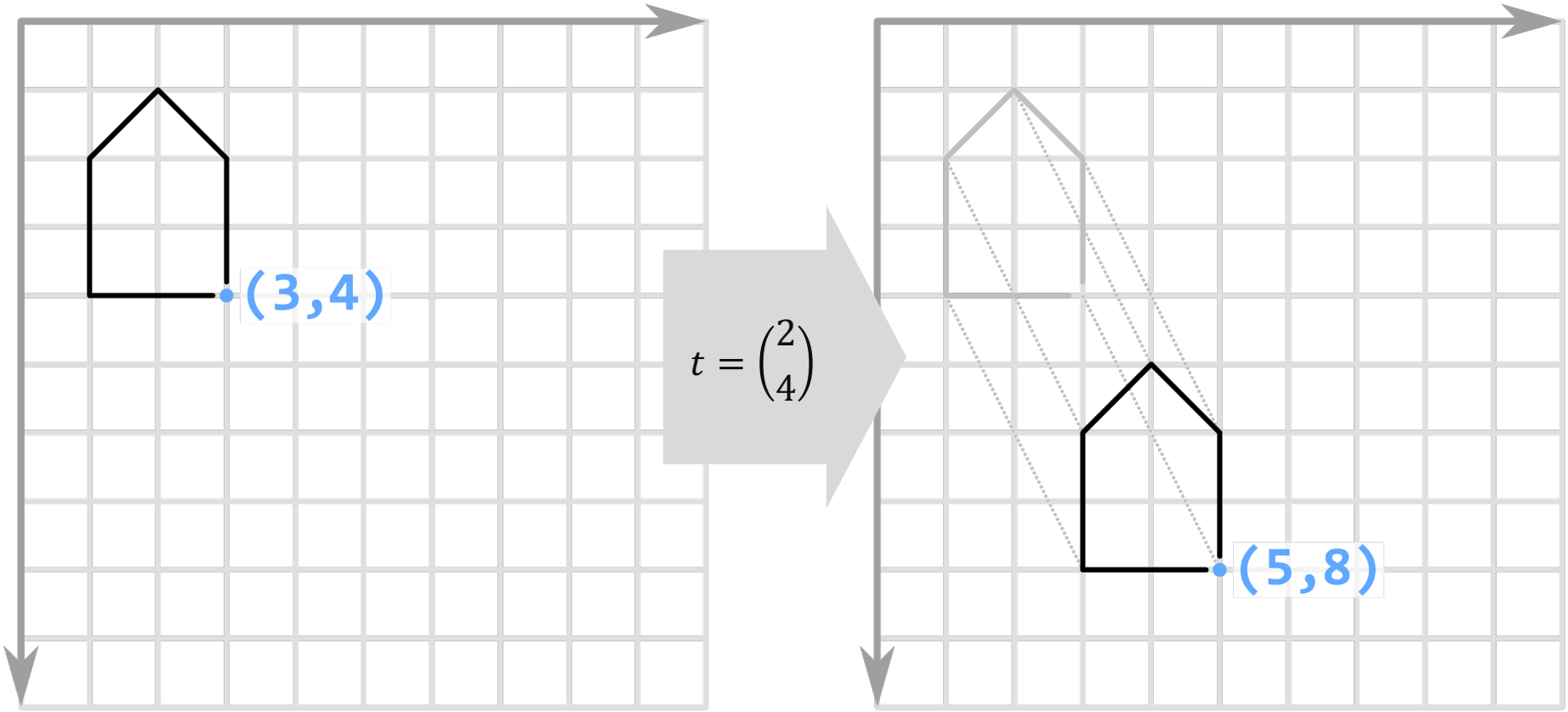



$$(x'_i,y'_i) = \boldsymbol{f}(x_i,y_i)$$

# Translation

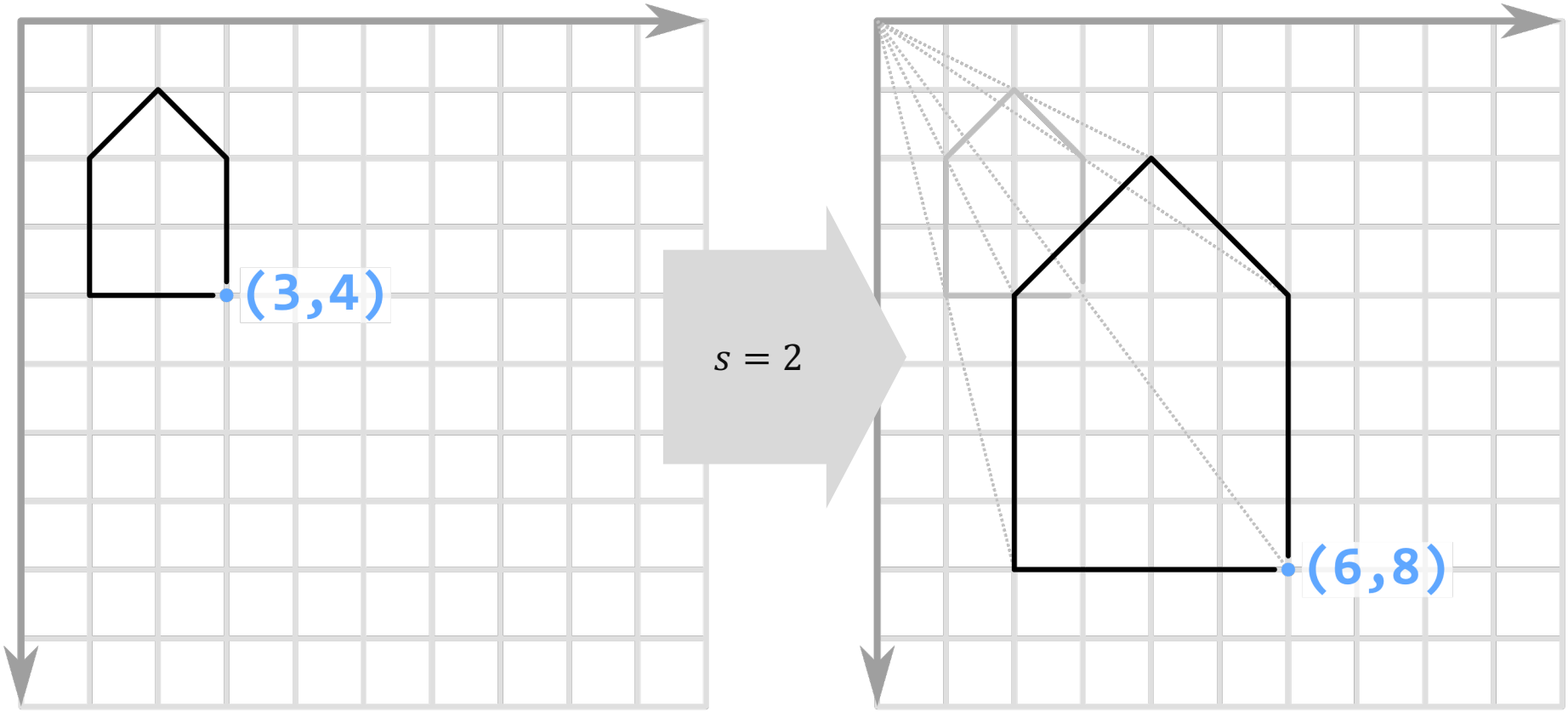
Translation adds the same vector $t$ to each vertex $v$.




$$v' : \begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

# Uniform Scaling

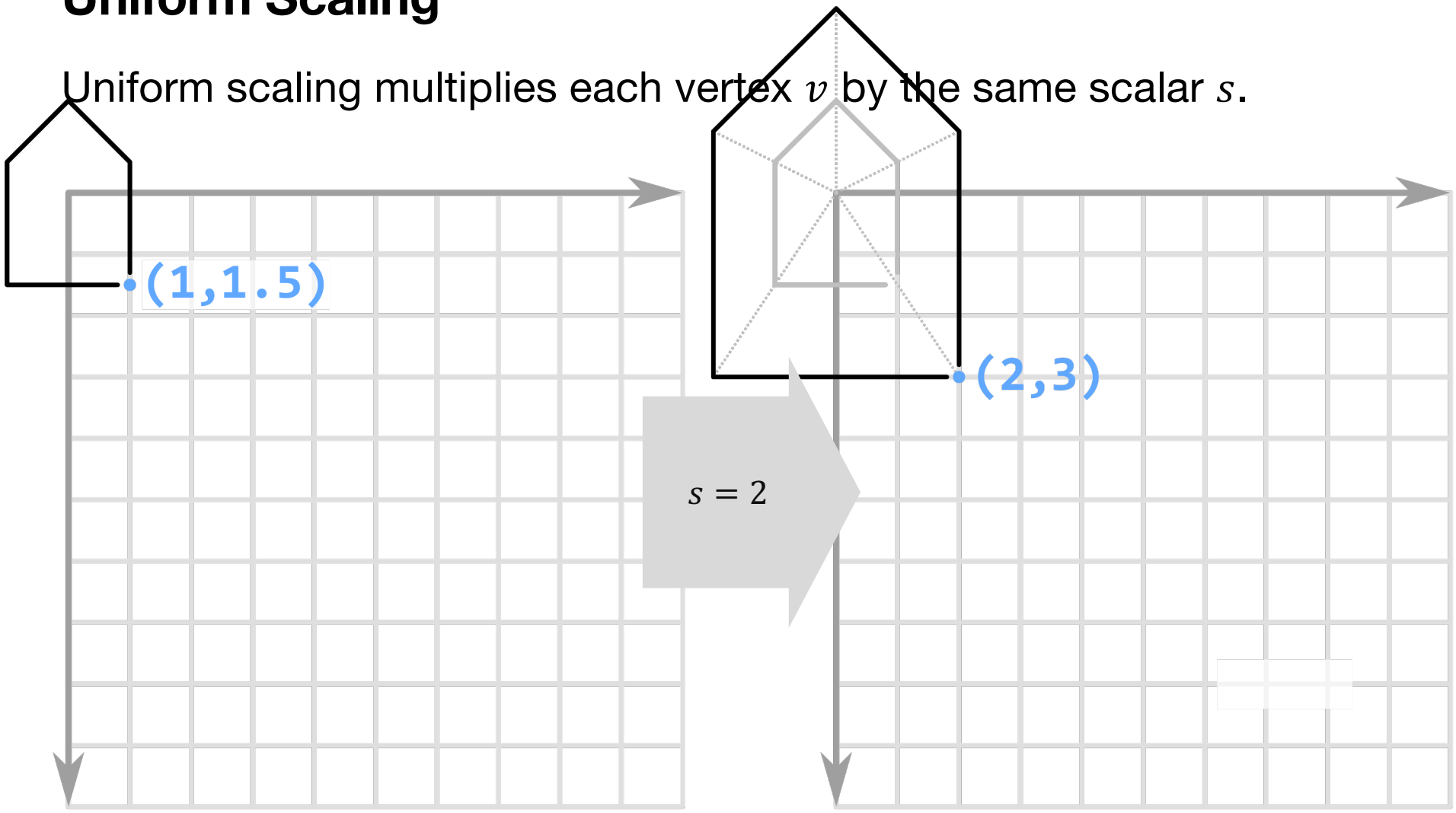
Uniform scaling multiplies each vertex $v$ by the same scalar $s$.




$$v' : \begin{cases} x' = s \times x \\ y' = s \times y \end{cases}$$

# Uniform Scaling

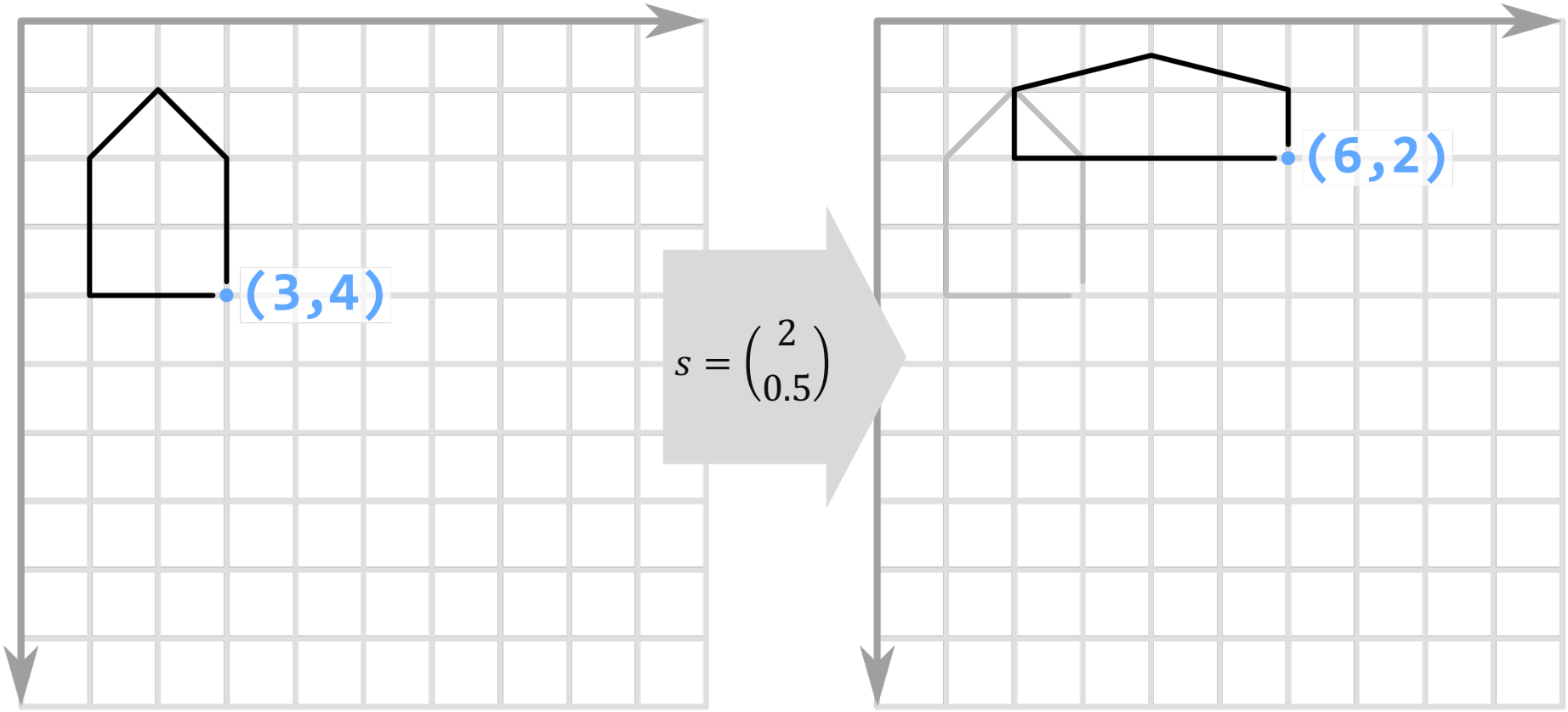
Uniform scaling multiplies each vertex $v$ by the same scalar $s$.




$$v' : \begin{cases} x' = s \times x \\ y' = s \times y \end{cases}$$

# Non-Uniform Scaling

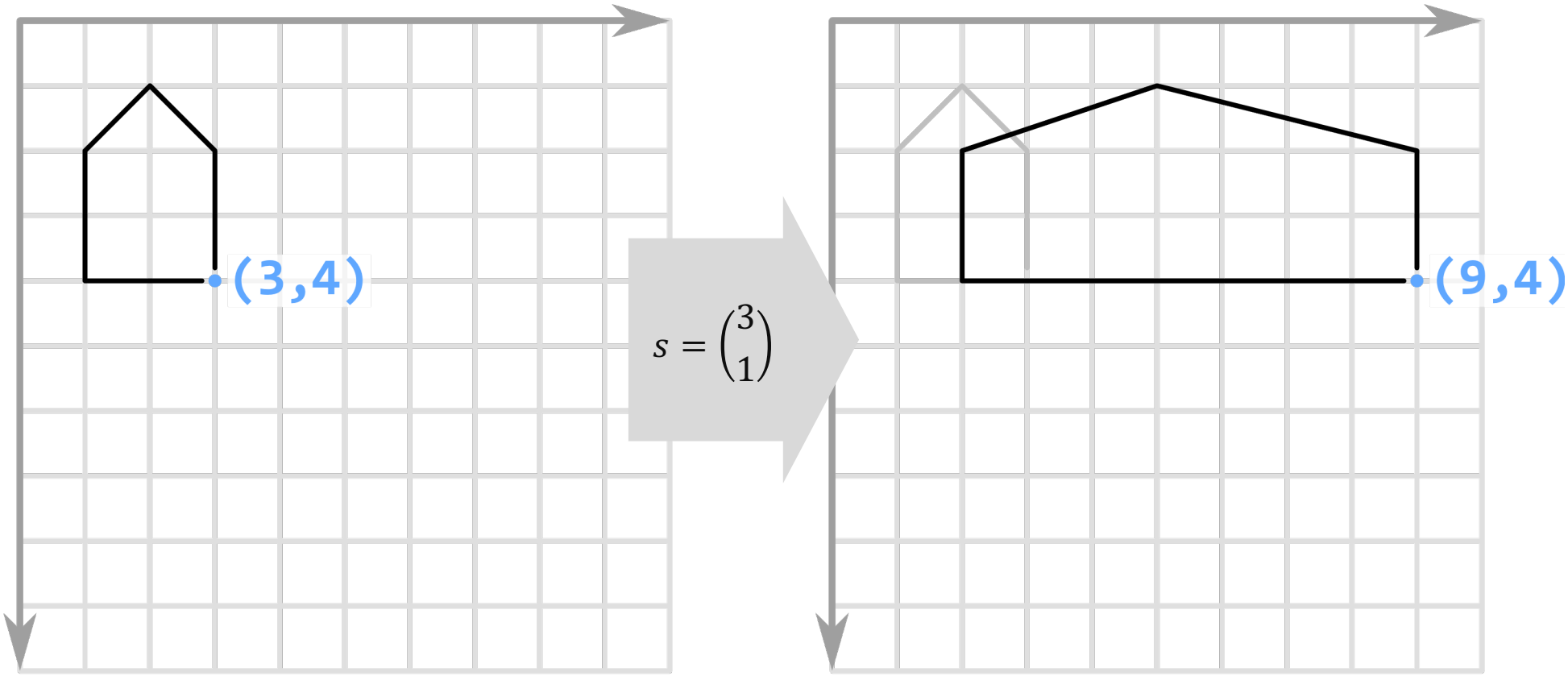
Non-uniform scaling applies the same scalar $s$ to each vertex $v$.




$$v' : \begin{cases} x' = s_x \times x \\ y' = s_y \times y \end{cases}$$

# Non-Uniform Scaling

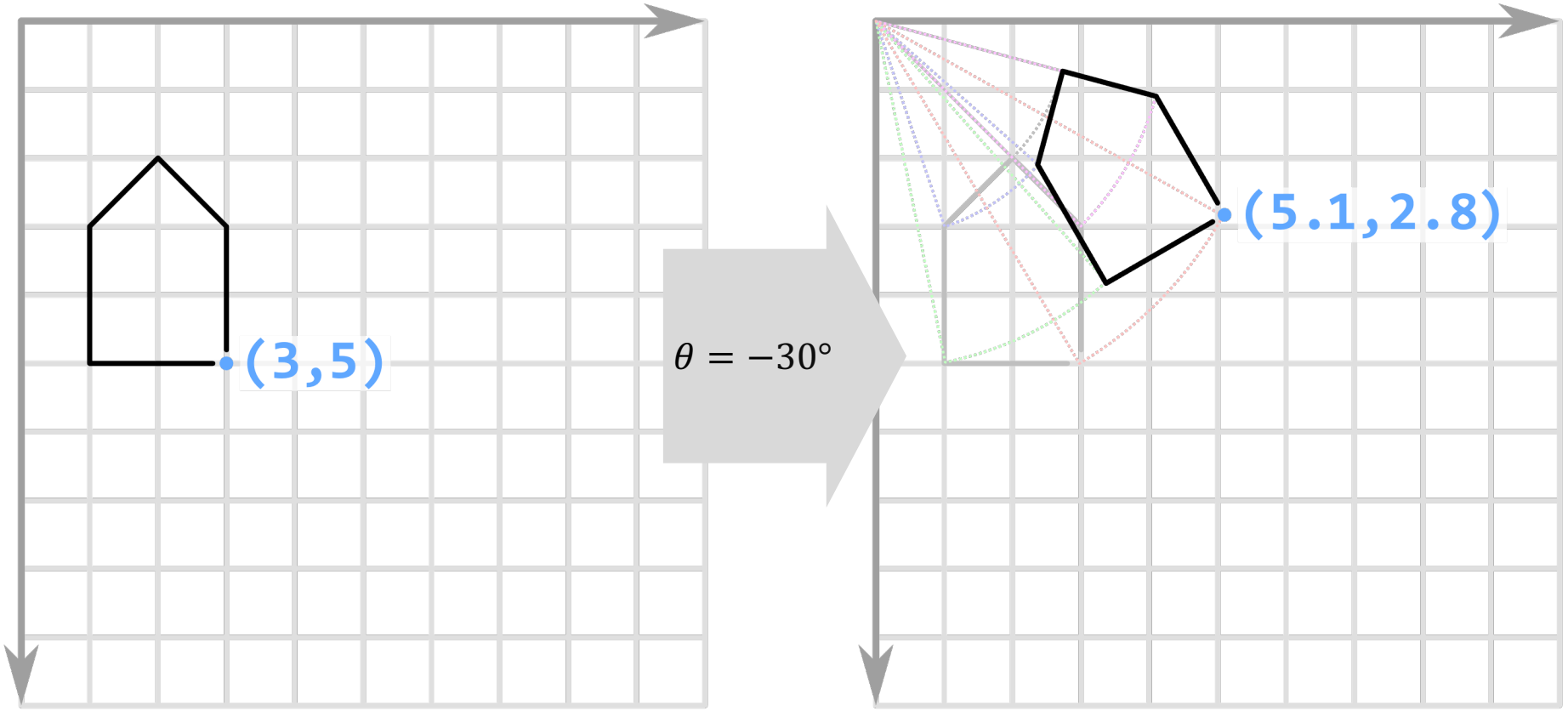
Non-uniform scaling applies the same scalar $s$ to each vertex $v$.




$$v' : \begin{cases} x' = s_x \times x \\ y' = s_y \times y \end{cases}$$

# Rotation

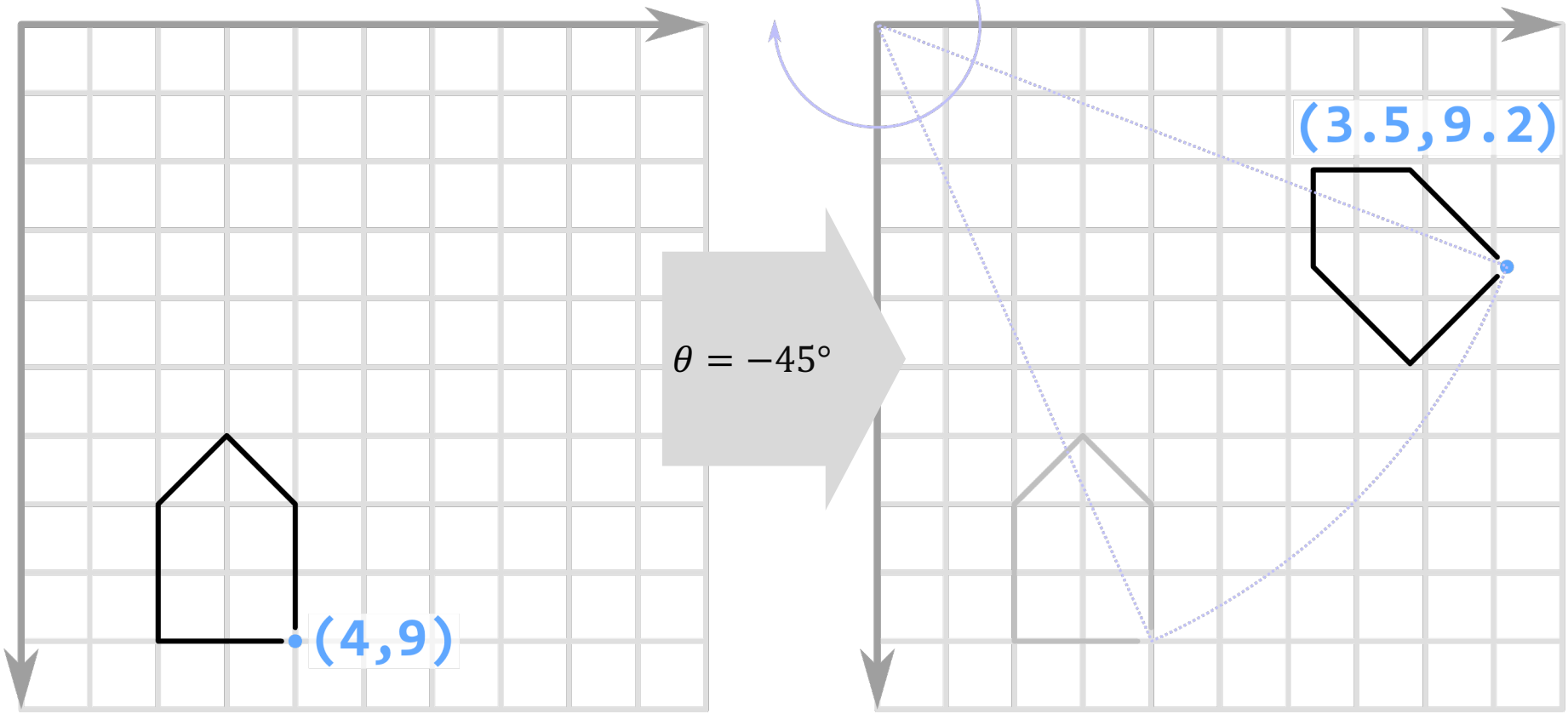
Rotation rotates each vertex $v$ around the origin of the base coordinate system by $\theta$ degrees.




$$v' : \begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

# Rotation

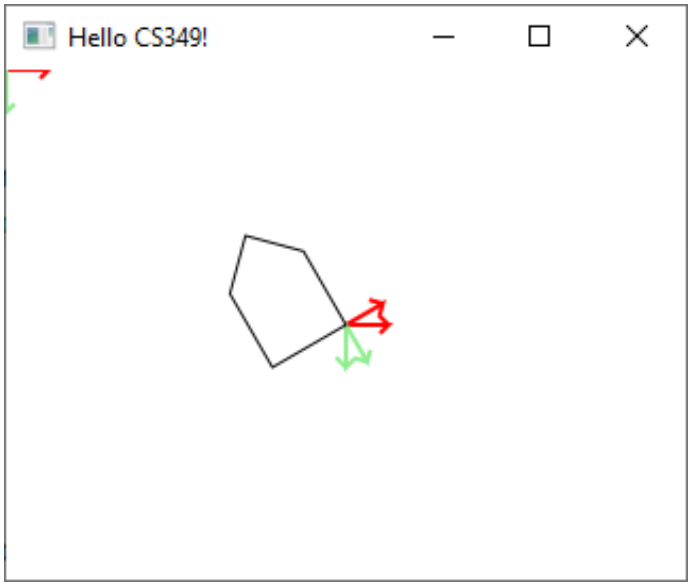
Rotation direction, i.e., clockwise (CW) or counter-clockwise (CCW), depends on the “handedness” if the coordinate system (LHS or RHS)




$$v' : \begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

# Transforms: Using the Graphics Context (GC)

We can use the Graphics Context to draw and transform. This has the effect of setting up common transformations that all shapes will use. Transformations are relative to the base coordinate system.

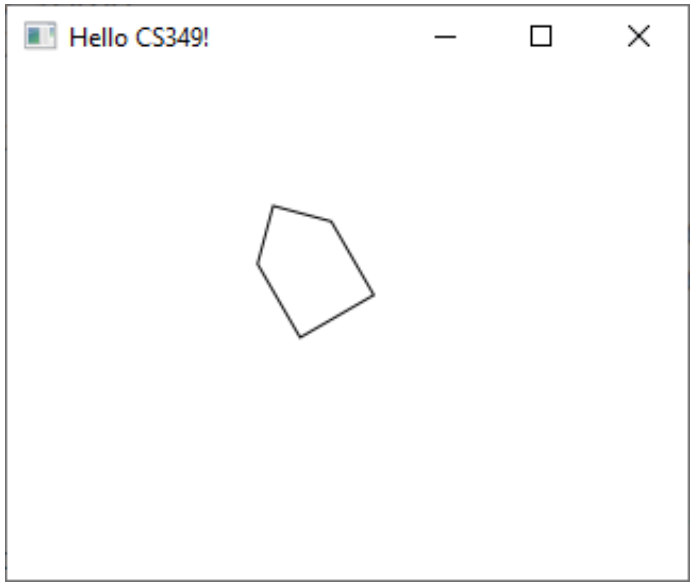



```
val canvas = Canvas(640.0, 480.0)
canvas.graphicsContext2D.apply {
    drawCoords(this)
    translate(160.0, 120.0)
    drawCoords(this)
    rotate(-30.0)
    drawCoords(this)
    strokePolygon(
        listOf(0.0, 0.0, -20.0, -40.0, -40.0).toDoubleArray(),
        listOf(0.0, -40.0, -60.0, -40.0, 0.0).toDoubleArray(), 5)
}
val scene = Scene(Group(canvas), 320.0, 240.0)
```

# Transforms: Using Shape Properties

We can also use the built-in functions in a Shape class.
Transformations are relative to the centre of the shape's bounding box.

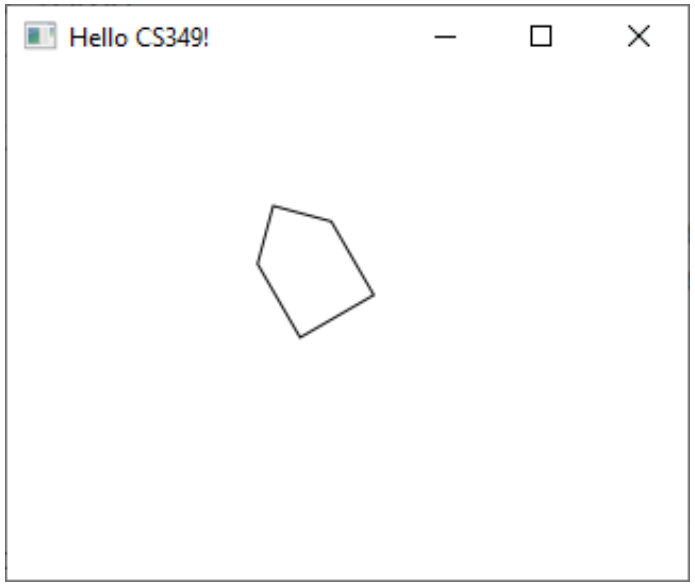
```
val house = Polygon(0.0, 0.0, 0.0, -40.0, -20.0, -60.0,
                    -40.0, -40.0, -40.0, 0.0).apply {
    translateX = 160.0
    translateY = 120.0
    rotate = -30.0
    fill = Color.TRANSPARENT
    stroke = Color.BLACK
}

val scene = Scene(Group(house),
                  320.0, 240.0)
```



# Transforms: Using Containers

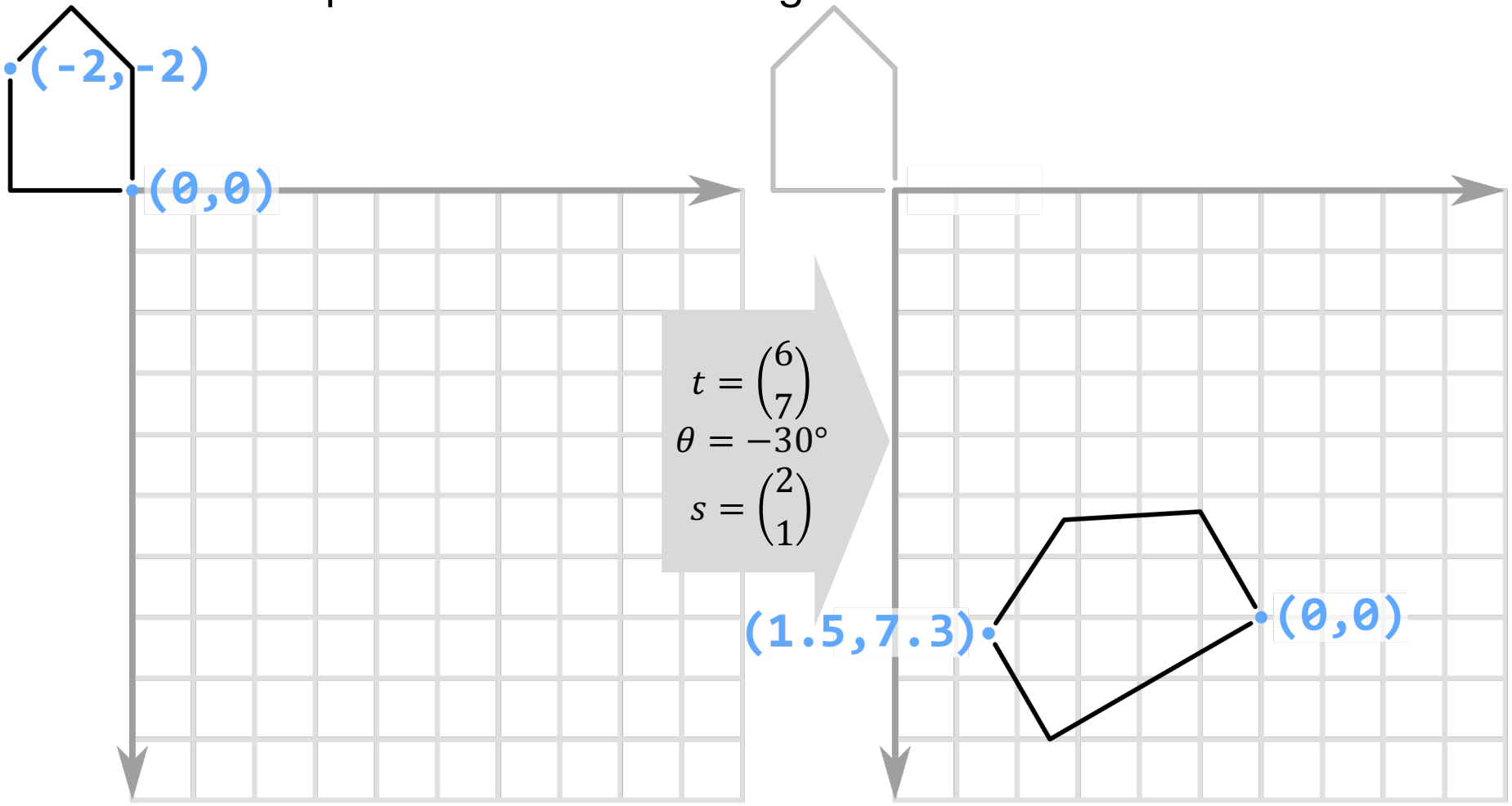
Transformations can also be applied to a container. They will be applied to all its children. Transformations are relative to the centre of the shape's bounding box.

```
val house = Polygon(0.0, 0.0, 0.0, -40.0, -20.0, -60.0,
                    -40.0, -40.0, -40.0, 0.0).apply {
    fill = Color.TRANSPARENT
    stroke = Color.BLACK
}

val houseGroup = Group(house).apply {
    rotate = -30.0
    translateX = 160.0
    translateY = 120.0
}

val scene = Scene(houseGroup,
                  320.0, 240.0)
```



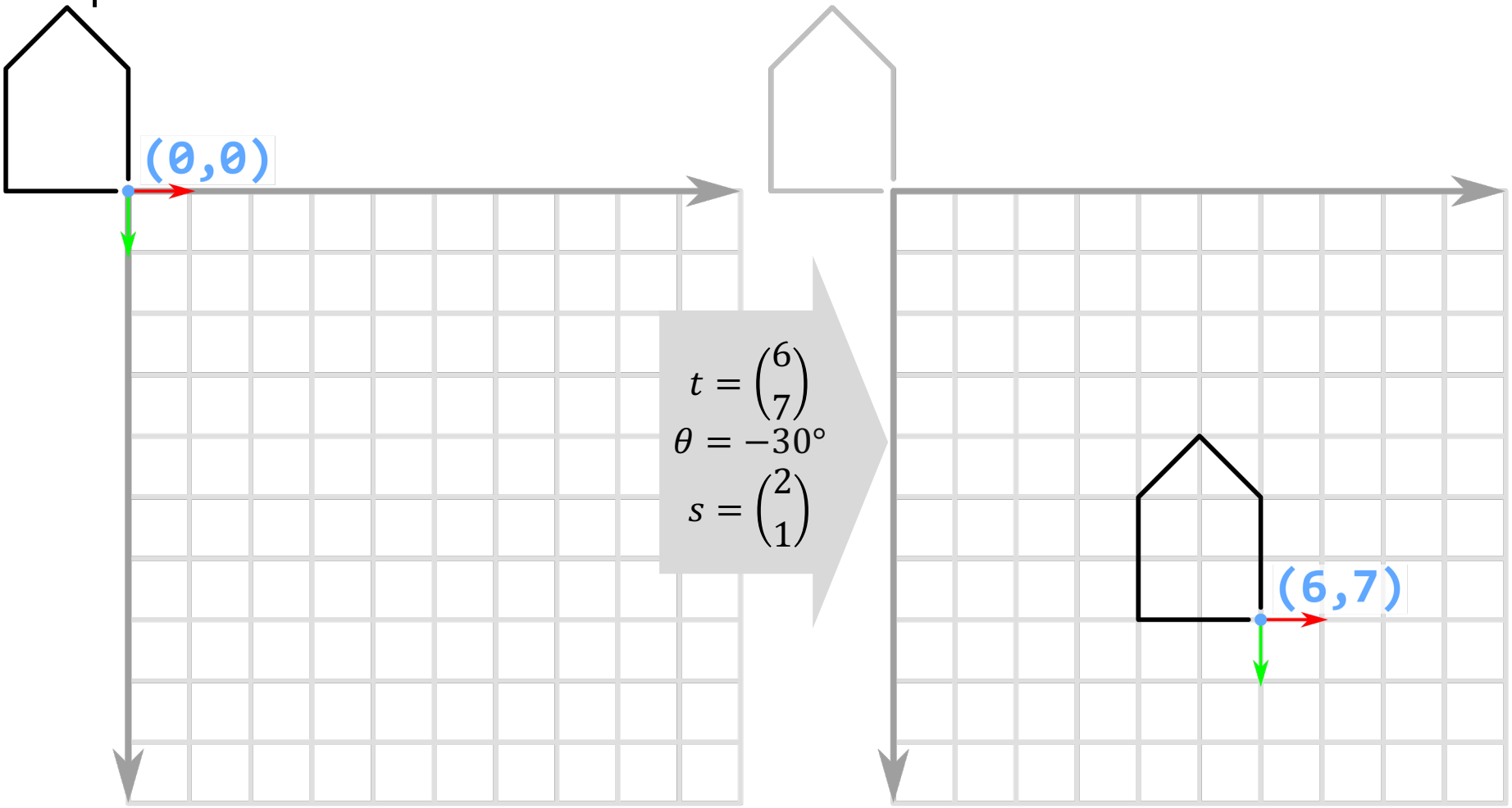
# Combining Transformations

Combine multiple transformations together:



# Combining Transformations

Step 1: Translate

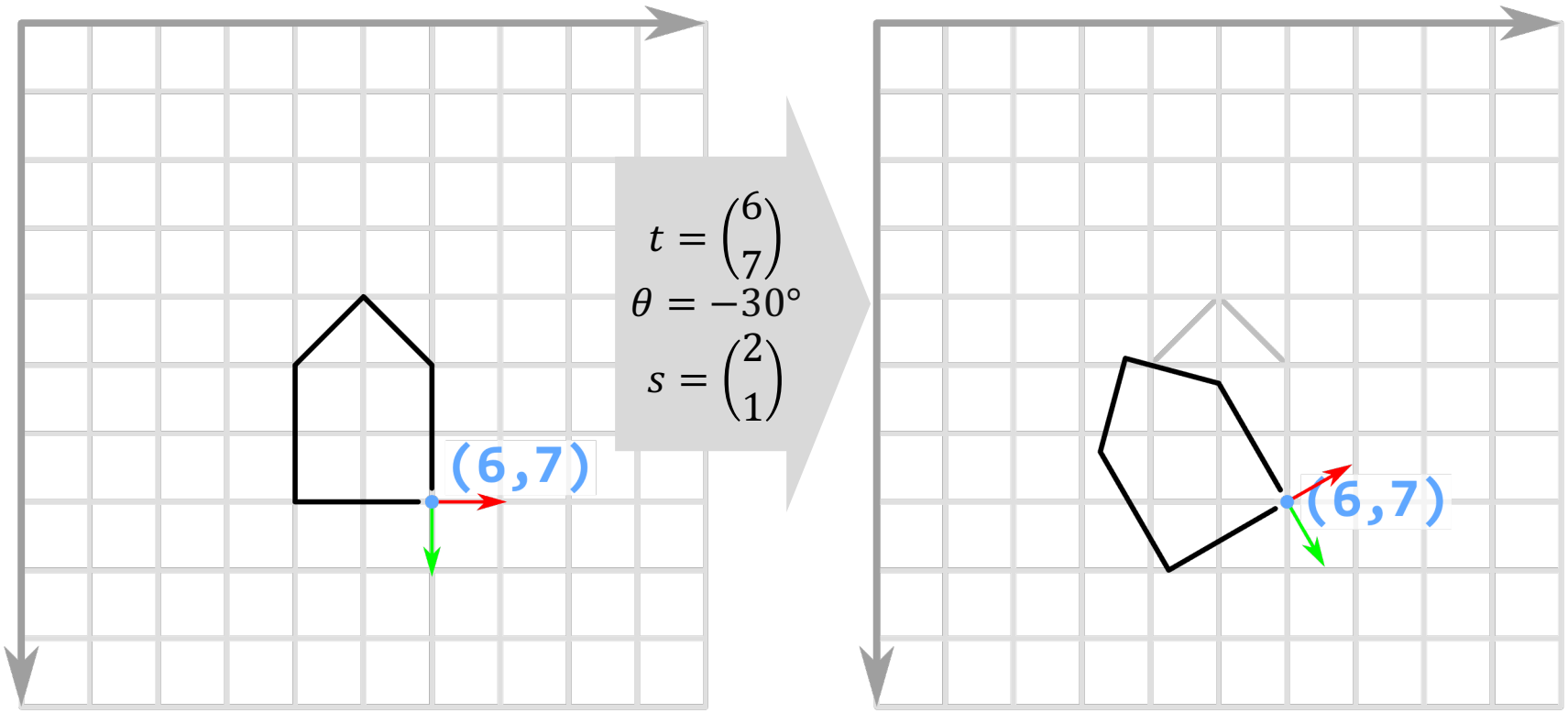



$$v' : \begin{cases} x' = x + 6 \\ y' = y + 7 \end{cases}$$

Graphics/07.HouseDemo

# Combining Transformations

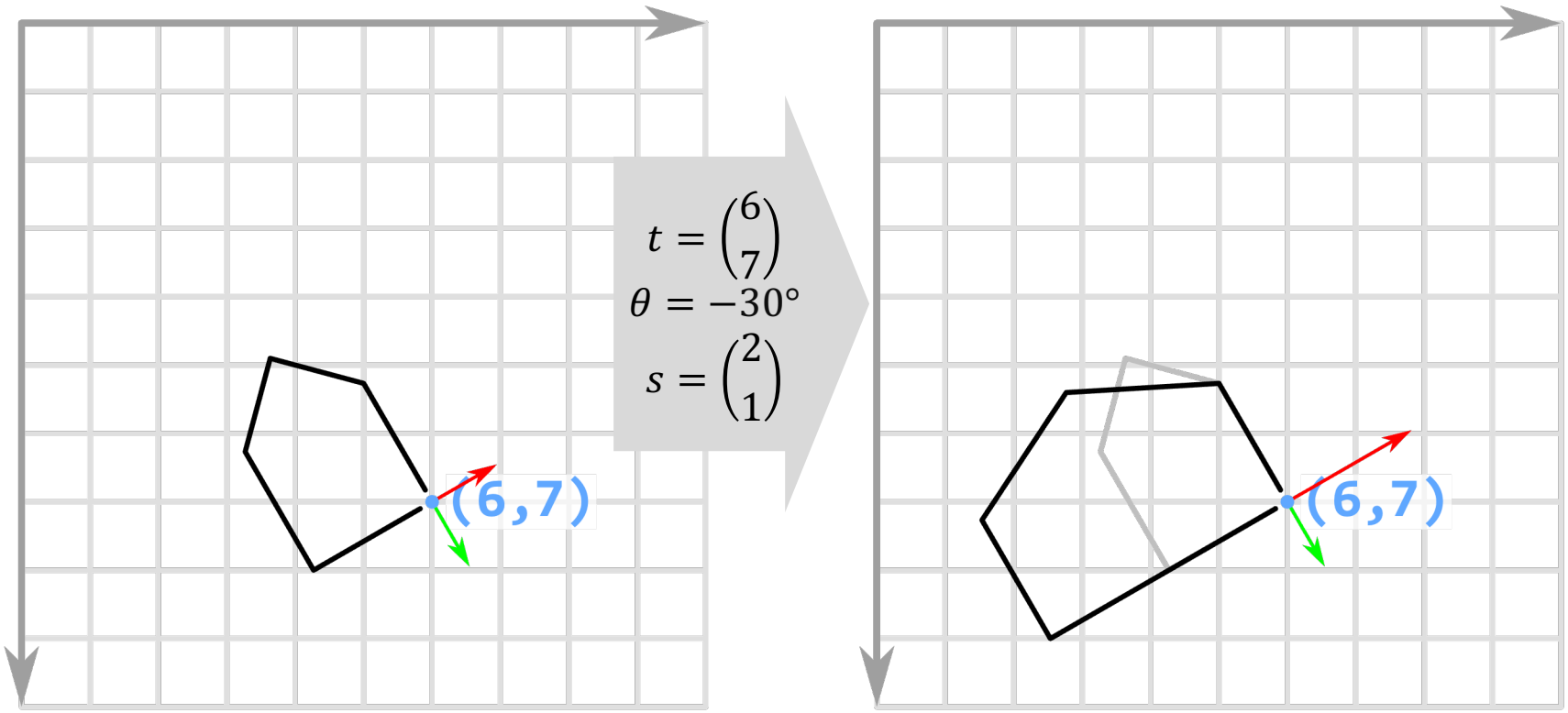
Step 2: Rotate




$$v'' : \begin{cases} x'' = x\cos(-30) - y\sin(-30) + 6 \\ y'' = x\sin(-30) + y\cos(-30) + 7 \end{cases}$$

# Combining Transformations

Step 3: Scale

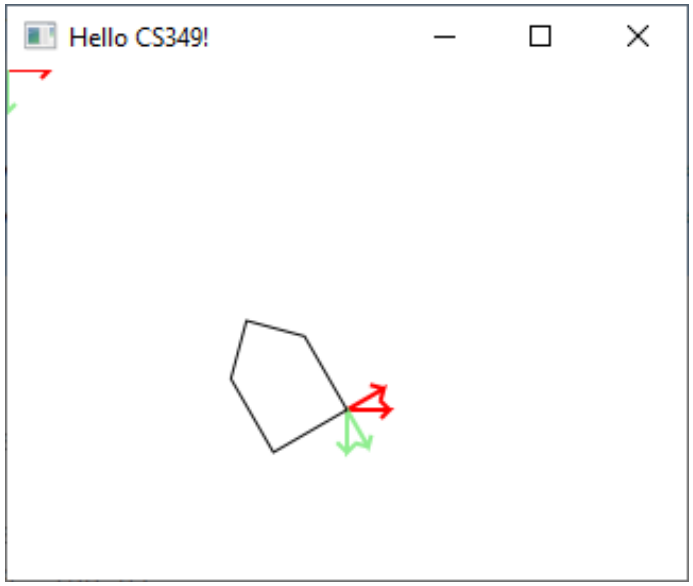



$$v''' : \begin{cases} x''' = (2x)\cos(-30) - (1y)\sin(-30) + 6 \\ y''' = (2x)\sin(-30) + (1y)\cos(-30) + 7 \end{cases}$$

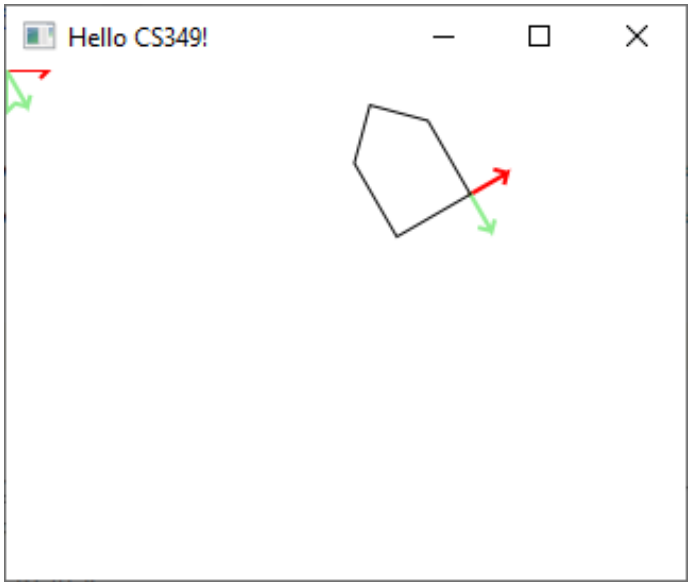
# Combining Transformations: GC Order

The order in which transformations are applied can matter!

```
translate(160.0, 160.0)
rotate(-30.0)
strokePolygon(
    listOf(...).toDoubleArray(),
    listOf(...).toDoubleArray(),
    5)
```



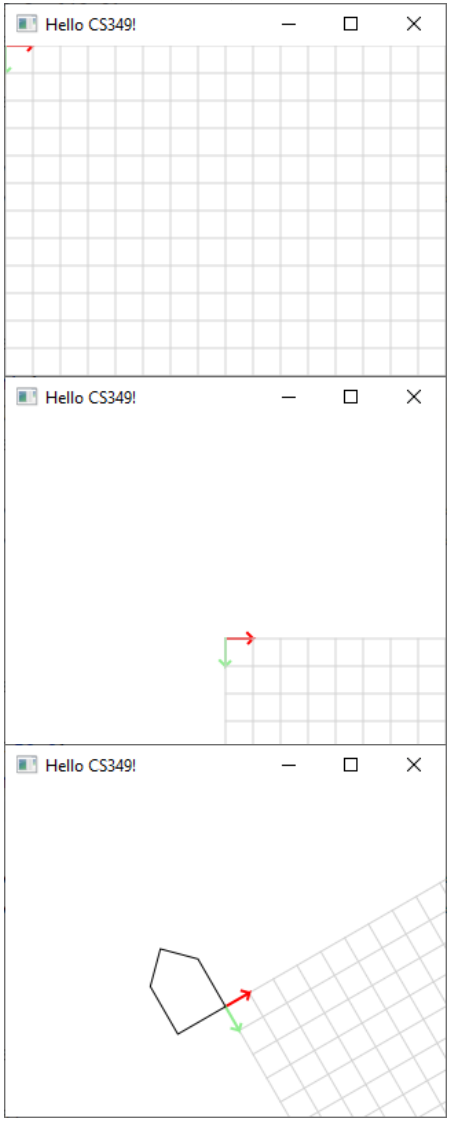

```
rotate(-30.0)
translate(160.0, 160.0)
strokePolygon(
    listOf(...).toDoubleArray(),
    listOf(...).toDoubleArray(),
    5)
```



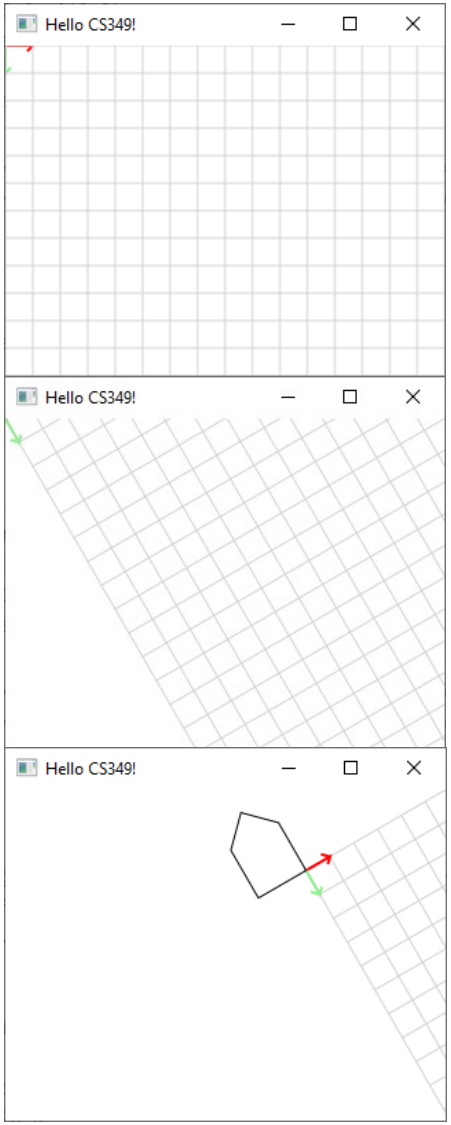

Graphics/07.HouseDemo

# Combining Transformations: GC Order

```
translate(160.0, 160.0)
rotate(-30.0)
```




```
rotate(-30.0)
translate(160.0, 160.0)
```



## Combining Transformations

We could create transformation equations, but

$$v''' : \begin{cases} x''' = (2x)\cos(-30) - (1y)\sin(-30) + 6 \\ y''' = (2x)\sin(-30) + (1y)\cos(-30) + 7 \end{cases}$$

is hard to express and implement in code.

Matrices are a better way to express multiple transformations:

- always a fixed size regardless of number of transformations
- easy to transform many vertices in code (i.e., matrix multiplication)
- easy to parallelize code (i.e., can perform calculations on GPU)

## Goal: Matrix Representation

Represent a 2D transformation as a matrix: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

Represent each vertex as a 2 x 1 column vector: $\begin{pmatrix} x \\ y \end{pmatrix}$

Multiply matrixes to apply the transformation and get new vertex:

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

# Matrix Representation

Transformations can be combined by matrix multiplication

- Transformations are associative: we can multiply them together

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

This creates a single **transformation matrix**

- represents *the result of multiple individual transformations*
- can be easily and quickly applied to vertices in multiple Shapes
- can be loaded into the GPU for performance

# Matrix Representations – Naïve Approach

Scale:

$$\begin{aligned} x' &= x \times s_x \\ y' &= y \times s_y \end{aligned} \Leftrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Rotation :

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

Translation:

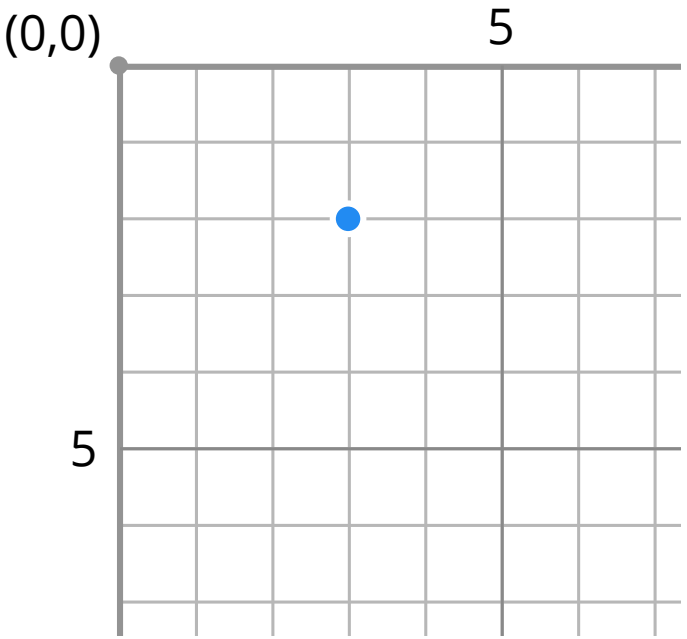
$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned} \Leftrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & t_x/y \\ t_y/x & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

# Matrix Representation – Using Homogeneous Coordinates

Adding a 3rd component $w$ to coordinates: if $w$ is $0$, the coordinates represent a vector, otherwise it is a vertex.

Dividing $x$ and $y$ by $w$ yields the corresponding Cartesian point:

$[x, y, w]$ represents a point at Cartesian location $[x/w, y/w]$




$$\begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 7.5 \\ 5.0 \\ 2.5 \end{bmatrix} \ldots$$

Cartesian coordinates

homogeneous coordinates

There are infinite homogenous coordinates representing the same vertex in Cartesian coordinates.

## Homogeneous Matrix Representation

Each homogenous vertex is a $3\times1$ column matrix with $w = 1$:

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

As a result, $2\times2$ transformation matrixes will not work:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = ???$$

Instead, we use $3\times3$ transformation matrixes:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax & by & c \\ dx & ey & f \\ gx & hy & i \end{bmatrix}$$

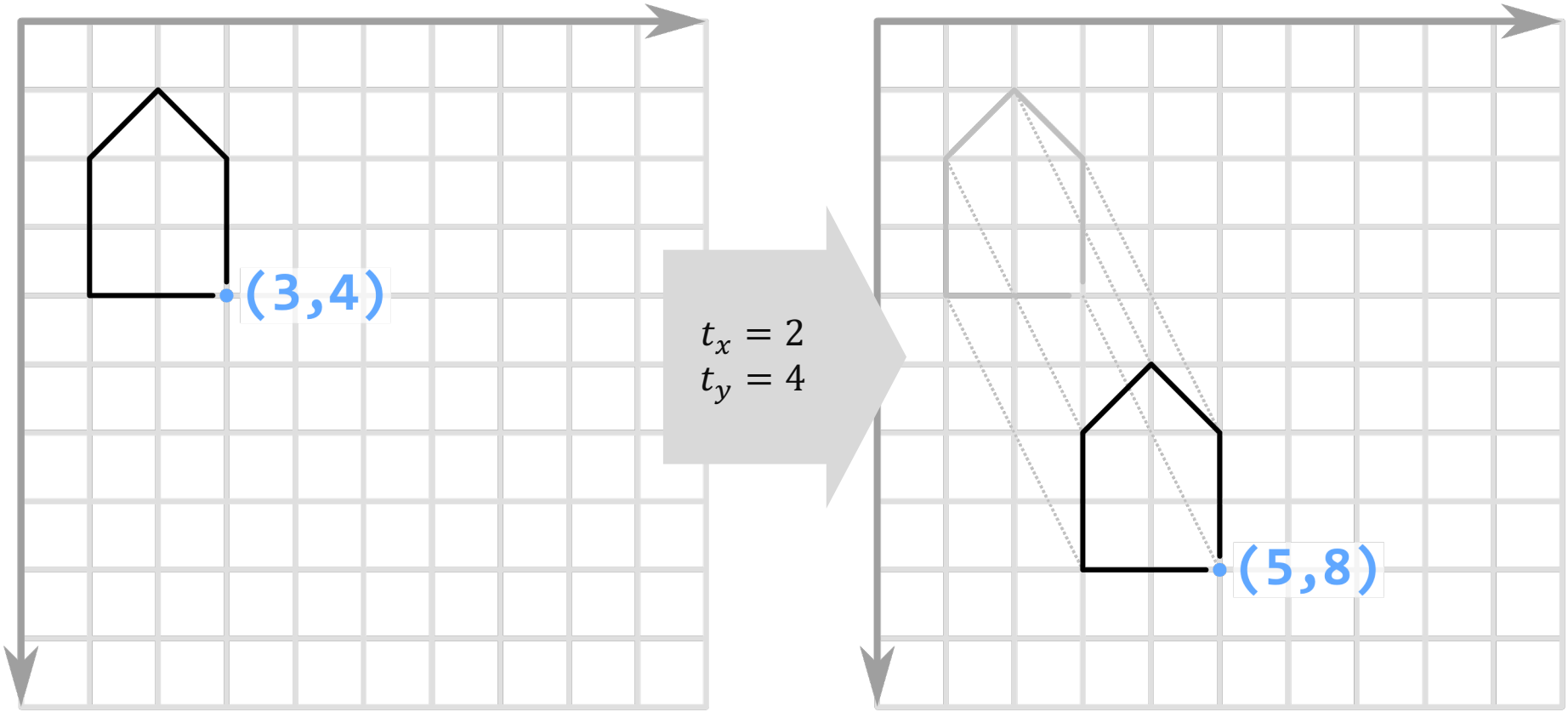
## Homogeneous Matrix Representation: Translation

Now we can represent 2D translation with a $3\times3$ matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

# Homogeneous Matrix Representation: Translation

Translation adds the scalars $t_x$, $t_y$ to each vertex $v$.




$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \\ 1 \end{bmatrix}$$

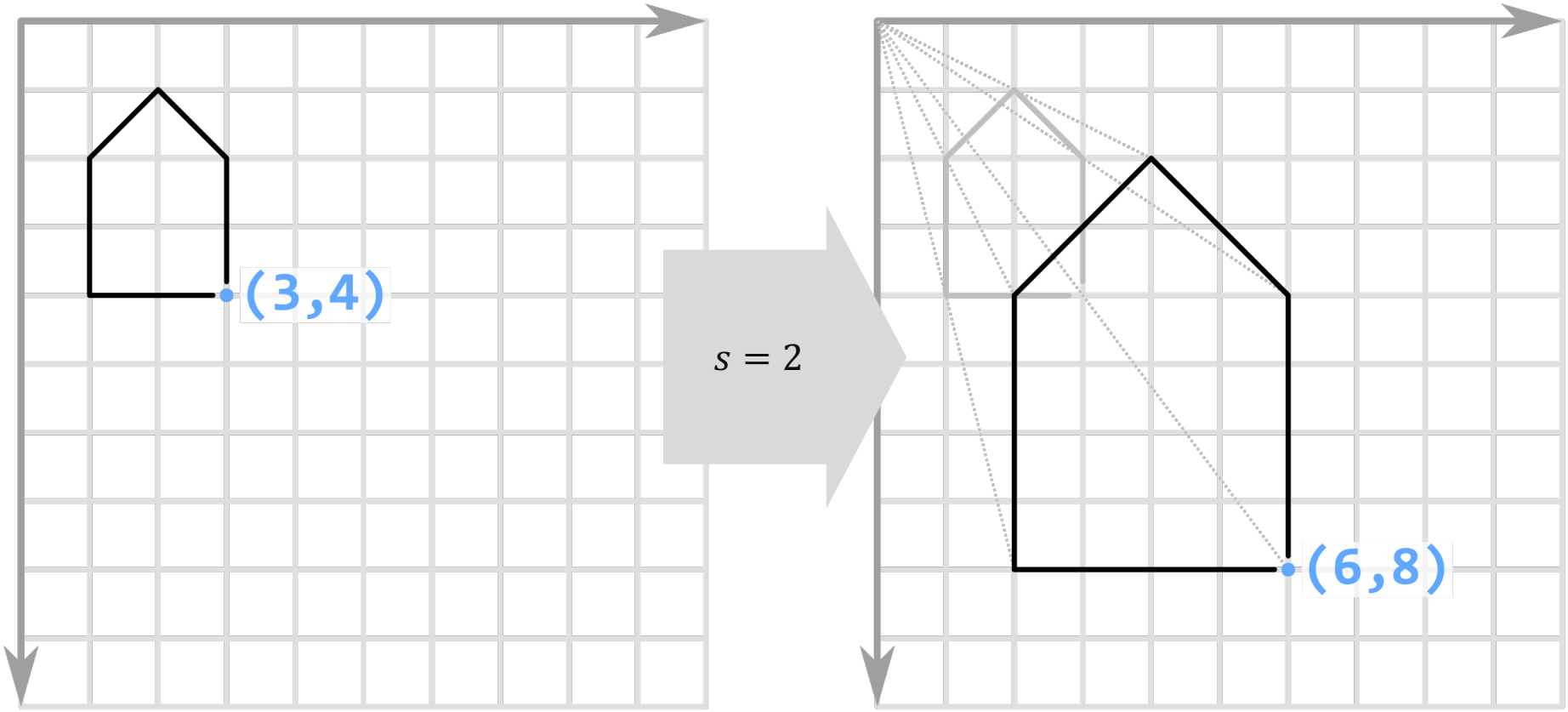
## Homogeneous Matrix Representation: Scale

Now we can represent 2D scale with a $3\times3$ matrix:

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

# Homogeneous Matrix Representation: Scale

Multiply each point coordinate by the same scalar.




$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \\ 1 \end{bmatrix}$$

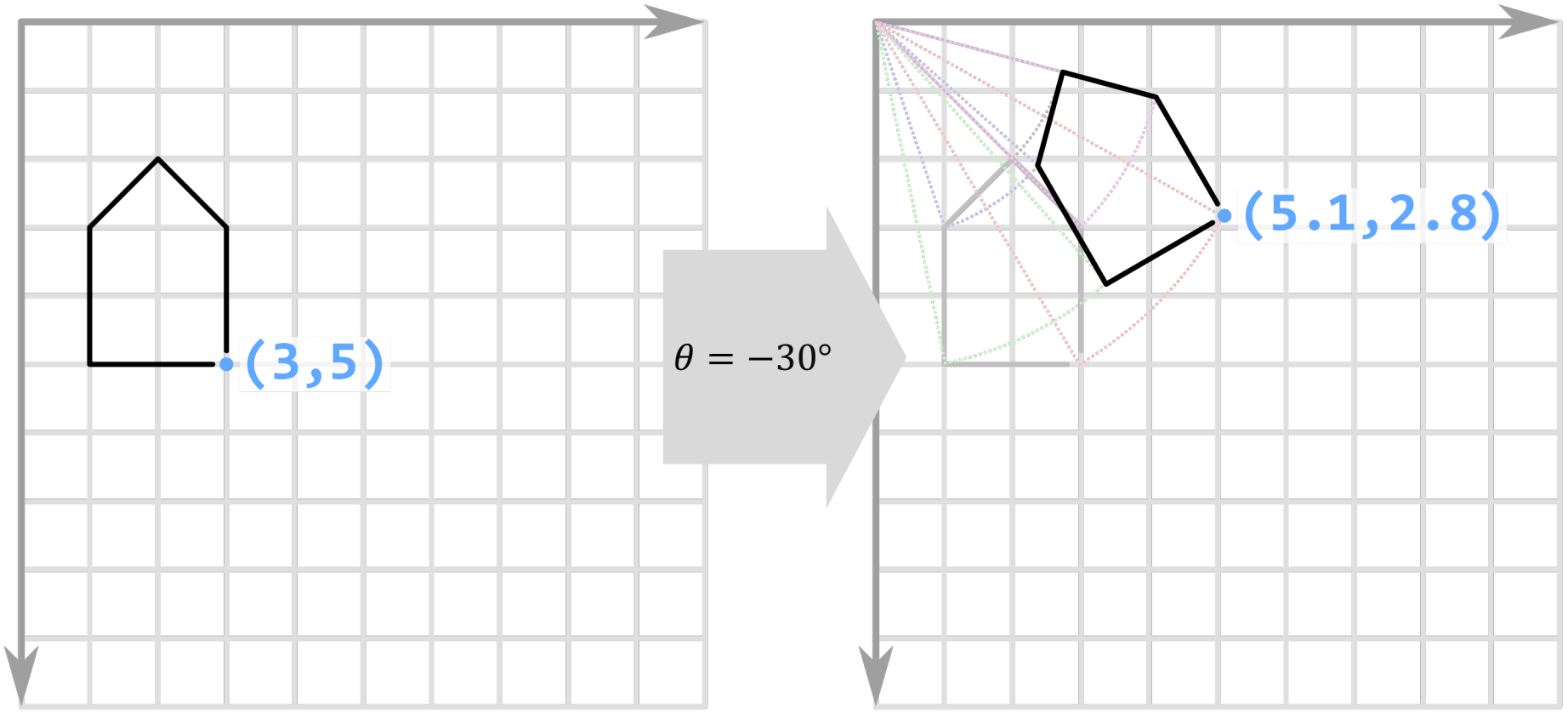
## Homogeneous Matrix Representation: Rotation

Now we can represent 2D rotation with a $3\times3$ matrix:

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

# Rotation

Each new point coordinate component is a function of an angle $\theta$ and both of the previous point coordinates.




$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0.87 & 0.5 & 0 \\ 0.5 & 0.87 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 5.1 \\ 2.8 \\ 1 \end{bmatrix}$$

## Transformation Matrices

*Translate*

$$\boldsymbol{T}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

*Scale*

$$\boldsymbol{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

*Rotate*

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

# Affine Transformation Matrix: Concatenation

These 3x3 matrices are examples of an **Affine Transformation Matrix.** They can express any combination of translation, rotation, and scaling:

Individual transformations can be combined using matrix multiplication, which is often called transformation concatenation.

The original vector $v$ is first translated, then rotated, and finally scaled:

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

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# Affine Transformation Matrix: Concatenation

Matrix multiplication is not commutative: if $A$ and B are matrices, “Not Commutative” means:

$$A \times B \neq B \times A$$

The order in which transformations are concatenated can change the result:

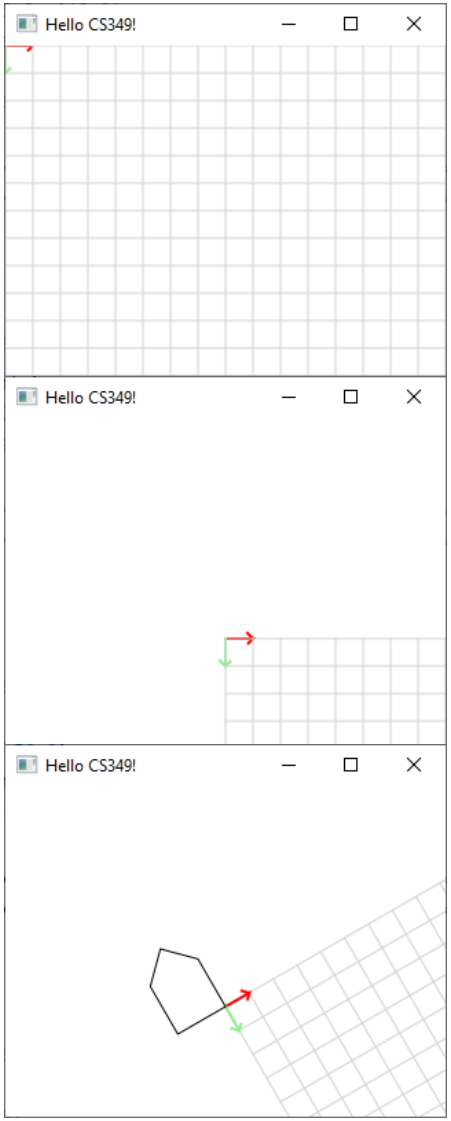
$$T(2,3) \cdot S(4,5) = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 2 \\ 0 & 5 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

whereas:

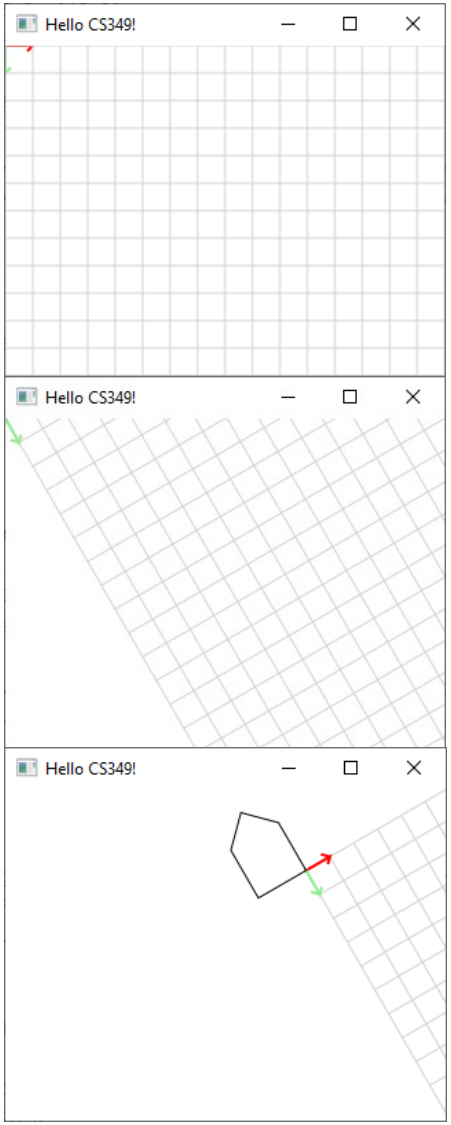
$$S(4,5) \cdot T(2,3) = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 8 \\ 0 & 5 & 15 \\ 0 & 0 & 1 \end{bmatrix}$$

# Affine Transformation Matrix: Concatenation

```
translate(160.0, 160.0)
rotate(-30.0)
```



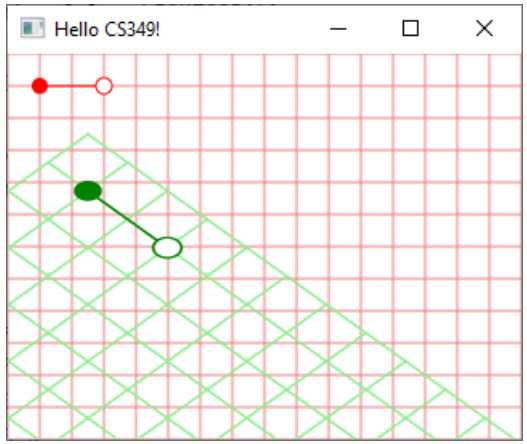

```
rotate(-30.0)
translate(160.0, 160.0)
```



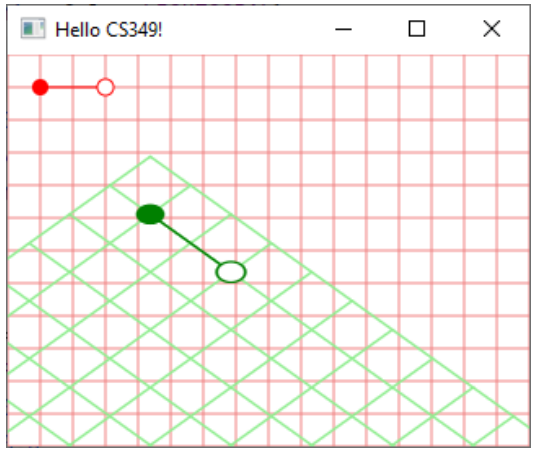
# Affine Transformation Matrix: Order Examples

```
[S]cale(1.75, 1.25); [R]otate(45.0); [T]ranslate(50.0, 50.0)
```

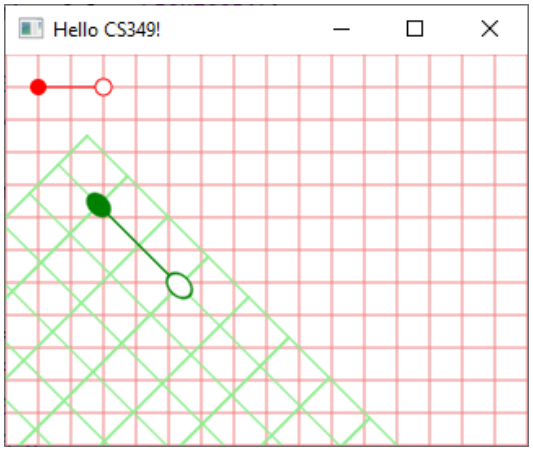
TSR



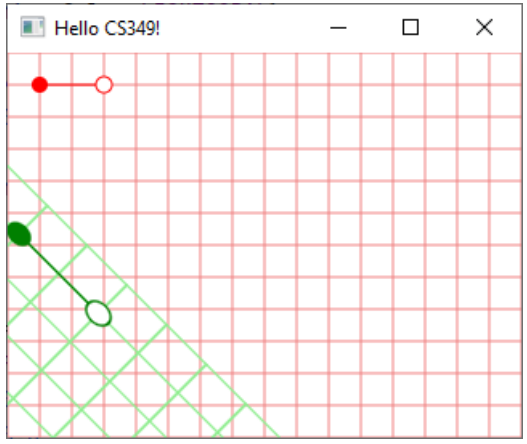

STR



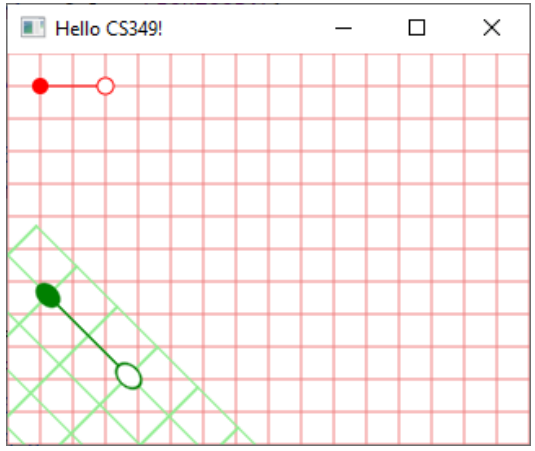

TRS



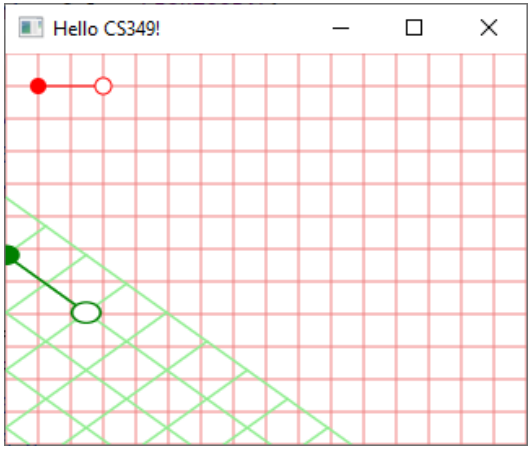

RTS




RST




SRT



# Cumulative Transformations

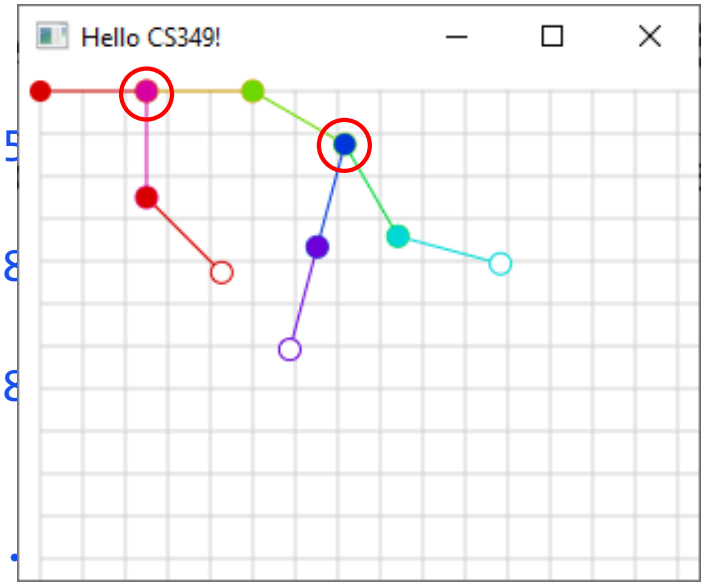
It can be useful to draw shapes relative to the current transform:

```
canvas.graphicsContext2D.apply {
    translate(10.0, 10.0)
    drawGrid(this, Color.LIGHTGRAY)
    drawHandle(this, Color.hsb(0.0, 1.0, 0.85))
    translate(50.0, 0.0)
    drawHandle(this, Color.hsb(60.0, 1.0, 0.85)
    translate(50.0, 0.0)
    rotate(30.0)
    drawHandle(this, Color.hsb(120.0, 1.0, 0.85
    translate(50.0, 0.0)
    rotate(30.0)
    drawHandle(this, Color.hsb(180.0, 1.0, 0.85
    translate(50.0, 0.0)
    rotate(-45.0)
    drawHandle(this, Color.hsb(270.0, 1.0, 0.85
    translate(50.0, 0.0)
    rotate(135.0)
    drawHandle(this, Color.hsb(360.0, 1.0, 0.85
}
```


Graphics/06.CumulativeDemo

# Saving and Restoring Transformation State

It can be useful to save the graphics context at some point, then restore is again, e.g., building complex geometries:




```
canvas.graphicsContext2D.apply {
    drawHandle(this, Color.hsb(0.0, 1.0, 0.85
    translate(50.0, 0.0)
    save()
    drawHandle(this, Color.hsb(45.0, 1.0, 0.8
    translate(50.0, 0.0)
    rotate(30.0)
    drawHandle(this, Color.hsb(90.0, 1.0, 0.8
    translate(50.0, 0.0)
    save()
    rotate(30.0)
    drawHandle(this, Color.hsb(135.0, 1.0, 0.
    translate(50.0, 0.0)
    rotate(-45.0)
    drawHandle(this, Color.hsb(180.0, 1.0, 0.85))
    restore()
    rotate(75.0)
    drawHandle(this, Color.hsb(225.0, 1.0, 0.85))
    translate(50.0, 0.0)
    drawHandle(this, Color.hsb(270.0, 1.0, 0.85))
    restore()
    rotate(90.0)
    drawHandle(this, Color.hsb(315.0, 1.0, 0.85))
    translate(50.0, 0.0)
    rotate(-45.0)
    drawHandle(this, Color.hsb(360.0, 1.0, 0.85))
}
```

## Saving and Restoring Transformation State

It can be useful to save the graphics context at some point, then restore is again, e.g., for additional drawings:

```
fun drawHandle(gc: GraphicsContext, col: Color) {
    gc.save()
    gc.stroke = col
    gc.fill = col
    gc.strokeLine(0.0, 0.0, 50.0, 0.0)
    gc.fillOval(-5.0, -5.0, 10.0, 10.0)
    gc.fill = Color.WHITE
    gc.fillOval(45.0, -5.0, 10.0, 10.0)
    gc.strokeOval(45.0, -5.0, 10.0, 10.0)
    gc.restore()
}
```

# End of the Chapter

- Graphics pipeline
- Transformations
  - Which one exists
  - How does the math work (conceptually)
- Affine Transformation Matrices
  - Why we use them

Any further questions?