Transform 2: Rotate, Scale

This unit introduces the transformation functions for rotating and scaling and explains how to combine the functions to control the effect.

Syntax introduced:
`rotate(), scale()`

The transformation functions are powerful ways to modify the geometry displayed to the screen. It's simple to use one, but combining them requires a greater understanding of how they work. The order in which transformation functions are run can radically change the way they affect the coordinates.

Rotation, Scaling

The `rotate()` function rotates the coordinate system so that shapes can be drawn to the screen at an angle. It has one parameter that sets the amount of the rotation as an angle:

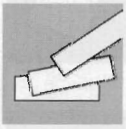
```
rotate(angle)
```

The rotate function assumes that the *angle* is specified in units of radians (p. 117). Shapes are always rotated around their position relative to the origin (0,0), and positive numbers rotate them in a clockwise direction.

As with all transformations, the effects of rotation are cumulative. If there is a rotation of π/4 radians and another of π/4 radians, objects drawn afterward will be rotated π/2 radians. The following examples show the most basic use of the `rotate()` function.

17-01

```
smooth();
rect(55, 0, 30, 45);
rotate(PI/8);
rect(55, 0, 30, 45);
```



17-02

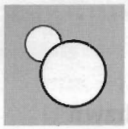
```
smooth();
rect(10, 60, 70, 20);
rotate(-PI/16);
rect(10, 60, 70, 20);
rotate(-PI/8);
rect(10, 60, 70, 20);
```

These examples make it clear that rotating objects around the origin has limitations. To rotate an object at a different position, it's necessary to use `translate()` followed by `rotate()`. This is explained in the next section, "Combining transformations."

The `scale()` function magnifies the coordinate system so that shapes are drawn larger. It has one or two parameters to set the amount of increase or decrease:

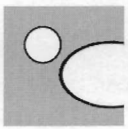
```
scale(size)
scale(xsize, ysize)
```

The version with one parameter scales shapes in all dimensions, and the version with two parameters can scale the x-dimension separately from the y-dimension. The parameters to scale are defined in terms of percentages expressed as decimals. Examples of decimal percentages are 2.0 for 200%, 1.5 for 150%, and 0.5 for 50%. The following examples show the most basic use of the `scale()` function.



```
smooth();
ellipse(32, 32, 30, 30);
scale(1.8);
ellipse(32, 32, 30, 30);
```

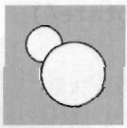
17-0



```
smooth();
ellipse(32, 32, 30, 30);
scale(2.8, 1.8);
ellipse(32, 32, 30, 30);
```

17-0

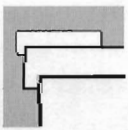
As the previous examples show, the stroke weight is also affected by `scale()`. To keep the same stroke weight and scale a shape, divide the parameter of the `strokeWeight()` function by the scale value.



```
float s = 1.8;
smooth();
ellipse(32, 32, 30, 30);
scale(s);
strokeWeight(1.0 / s);
ellipse(32, 32, 30, 30);
```

17-0

As with `translate()` and `rotate()`, the effects of each `scale()` accumulate each time the function is run.

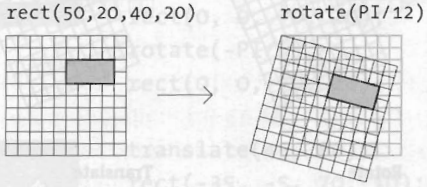


```
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
```

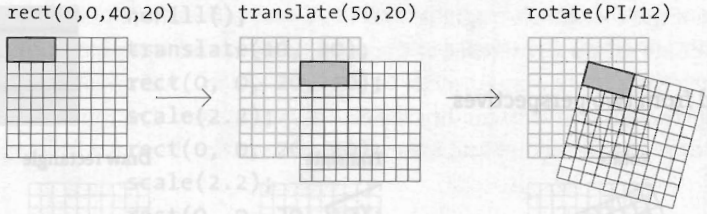
17-0

Combining transformations

When shapes are drawn to the screen, the `transform()`, `rotate()`, and `scale()` functions affect them in relation to the origin. For example, rotating a rectangle at coordinate (50,20) will cause the shape to orbit around the origin and not around its center or corner as you might expect:

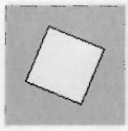


To rotate this shape around its upper-left corner, you must place that point at the coordinate (0,0). A translation is used to put the shape into the desired position in relation to the global coordinates. When the rotate function is run, the shape now orbits around its upper-left corner, the origin of its local coordinate system:



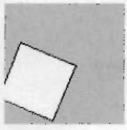
There are two ways to think about transformations. One method is to view the coordinate system as modified and the coordinates for shapes as converted to the new coordinate system. For example, if the coordinate system is rotated 30°, the coordinates of any shape drawn to the screen are converted into this modified system and displayed with a 30° tilt. The other school of thought applies the transformations directly to the shapes. In this same example, the shape itself is perceived to be rotated 30°.

The order in which transformations are made affects the results. The following two examples have the same lines of code, but the order of the `translate()` and `rotate()` functions is reversed :



17-07

```
translate(width/2, height/2);
rotate(PI/8);
rect(-25, -25, 50, 50);
```

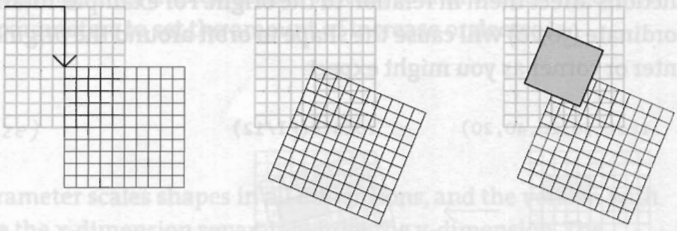


17-08

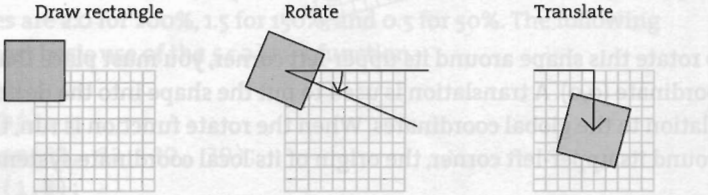
```
rotate(PI/8);
translate(width/2, height/2);
rect(-25, -25, 50, 50);
```

Code 17-07 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom

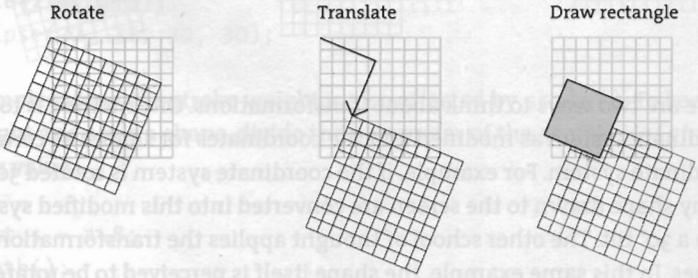


Shape view
Reading the code from bottom to top

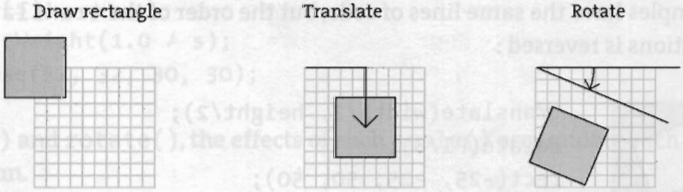


Code 17-08 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom



Shape view
Reading the code from bottom to top



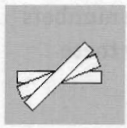
Transformation combinations

The order in which transformations occur in a program affects how they combine. For example, a `rotate()` after a `translate()` will have a different effect than the reverse. These diagrams present two ways to think about the transformations in codes 17-06 and 17-07.

These simple examples demonstrate the potential in combining transformations but also make clear that transformations require thought and planning. More combined examples follow:

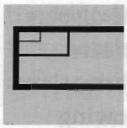
17-09

```
translate(10, 60);
rect(0, 0, 70, 20);
rotate(-PI/12);
rect(0, 0, 70, 20);
rotate(-PI/6);
rect(0, 0, 70, 20);
```



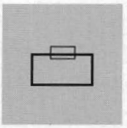
17-10

```
translate(45, 60);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
```



17-11

```
noFill();
translate(10, 20);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
```



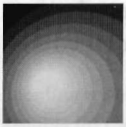
17-12

```
noFill();
translate(50, 30);
rect(-10, 5, 20, 10);
scale(2.5);
rect(-10, 5, 20, 10);
```

The effects of the transformation functions accumulate throughout the program, and these effects can be magnified with a `for` structure.

17-13

```
background(0);
smooth();
stroke(255, 120);
translate(66, 33);              // Set initial offset
for (int i = 0; i < 18; i++) {  // 18 repetitions
  strokeWeight(i);              // Increase stroke weight
  rotate(PI/12);                // Accumulate the rotation
  line(0, 0, 55, 0);
}
```



17-14

```
background(0);
smooth();
noStroke();
fill(255, 48);
translate(33, 66);                  // Set initial offset
for (int i = 0; i < 12; i++) {      // 12 repetitions
  scale(1.2);                       // Accumulate the scaling
  ellipse(4, 2, 20, 20);
}
```

Working with these examples will be more helpful than reading the explanation over and over. Try these examples inside Processing and make modifications to the numbers used and the sequence of translate, rotate, and scale to develop a sense of how these functions work.

New coordinates

The default position of the coordinate origin (0,0) is the upper-left corner of the display window, the x-coordinate numbers increase to the right, the y-coordinates increase from the top, and each coordinate maps directly to a pixel position. The transformation functions can change these defaults to modify the coordinate system. The following examples move the origin to the center and lower-left corner of the display window and modify the scale.

17-15

```
// Shift the origin (0,0) to the center
size(100, 100);
translate(width/2, height/2);
line(-width/2, 0, width/2, 0);     // Draw x-axis
line(0, -height/2, 0, height/2);   // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45);  // Draw at the origin
ellipse(-width/2, height/2, 45, 45);
ellipse(width/2, -height/2, 45, 45);
```

The `translate()` and `scale()` functions can combine to change the range of values. In the following example, the right edge of the screen is mapped to the x-coordinate of 1.0, the left edge to the x-coordinate -1.0, the top edge to the y-coordinate 1.0, and the bottom edge to the y-coordinate -1.0. This system will always scale to fit the entire display window. Run this program, but change the parameters to `size()` to see it work.

17-16

```
// Shift the origin (0,0) to the center
// and resizes the coordinate system
size(100, 100);
scale(width/2, height/2);
translate(1.0, 1.0);
strokeWeight(1.0/width);
line(-1, 0, 1, 0);  // Draw x-axis
line(0, -1, 0, 1);  // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 0.9, 0.9);  // Draw at the origin
ellipse(-1.0, 1.0, 0.9, 0.9);
ellipse(1.0, -1.0, 0.9, 0.9);
```

The `translate()` and `scale()` functions can be combined to put the origin in the lower-left corner of the screen. This is the coordinate system used by Adobe Illustrator and PostScript. Scaling the y-axis by -1 causes the y-coordinates to increment in the opposite direction. This can be useful when converting a program written using this coordinate system into Processing, rather than converting the y-coordinate of every point.

17-17

```
// Shift the origin (0,0) to the lower-left corner
size(100, 100);
translate(0, height);
scale(1.0, -1.0);
line(0, 1, width, 1);     // Draw x-axis
line(0, 1, 0, height );  // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45);     // Draw at the origin
ellipse(width/2, height/2, 45, 45);
ellipse(width, height, 45, 45);
```

Exercises

1. *Use `rotate()` to change the orientation of a shape.*
2. *Use `scale()` with a `for` structure to scale a shape multiple times.*
3. *Combine `translate()` and `rotate()` to rotate a shape around its own center.*