Transform 1: Translate, Matrices

This unit introduces coordinate system transformations and explains how to control their scope.

Syntax introduced:
`translate(), pushMatrix(), popMatrix()`

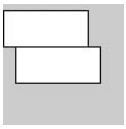
The coordinate system introduced in Shape 1 uses the upper-left corner of the display window as the origin with the x-coordinates increasing to the right and the y-coordinates increasing downward. This system can be modified with transformations. The coordinates can be translated, rotated, and scaled so shapes are drawn to the display window with a different position, orientation, and size.

Translation

The `translate()` function moves the origin from the upper-left corner of the screen to another location. It has two parameters. The first is the x-coordinate offset and the second is the y-coordinate offset:

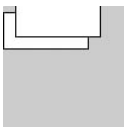
```
translate(x, y)
```

The values of the *x* and *y* parameters are added to any shapes drawn after the function is run. If 10 is used as the *x* parameter and 30 is used as the *y* parameter, a point drawn at coordinate (0,5) will instead be drawn at coordinate (10,35). Only elements drawn after the transformation are affected. The following examples show how this works.



16-01

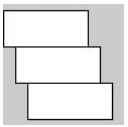
```
// The same rectangle is drawn, but only the second is
// affected by translate() because it is drawn after
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
```



16-02

```
// A negative number used as a parameter to translate()
// moves the coordinates in the opposite direction
rect(0, 5, 70, 30);
translate(10, -10);  // Shifts 10 pixels right and up
rect(0, 5, 70, 30);
```

The `translate()` function is additive. If `translate(10,30)` is run twice, all the elements drawn after will display with an x-offset of 20 and a y-offset of 60.



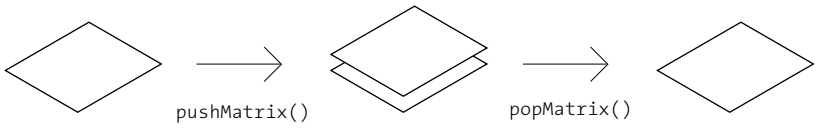
16-03

```
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts everything again for a total
rect(0, 5, 70, 30); // 20 pixels right and 60 down
```

Controlling transformations

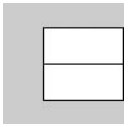
The transformation matrix is a set of numbers that defines how geometry is drawn to the screen. Transformation functions such as `translate()` alter the numbers in this matrix and cause the geometry to draw differently. In the previous examples, we saw how transformations accumulate as the program runs. The `pushMatrix()` function records the current state of all transformations so that a program can return to it later. To return to the previous state, use `popMatrix()`.

Think of each matrix as a sheet of paper with the current list of transformations (translate, rotate, scale) written on the surface. When a function such as `translate()` is run, it is added to the paper. To save the current matrix for later use, add a new sheet of paper to the top of the pile and copy the information from the sheet below. Any new changes are made to the top sheet of paper, preserving the numbers on the sheet(s) below. To return to a previous coordinate matrix, simply remove and discard the top sheet of paper to reveal the saved transformations below:



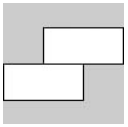
This is essentially how coordinate matrices are updated and stored, but more technical terms are used. Adding a sheet of paper is pushing, removing a sheet is popping and the pile of pages is called a stack. The `pushMatrix()` function is used to add a new coordinate matrix to the stack, and `popMatrix()` is used to remove one from the stack. Each `pushMatrix()` must have a corresponding `popMatrix()`. The function `pushMatrix()` cannot be used without `popMatrix()`, and vice versa.

Compare the two examples below. Both draw the same rectangles, but with different results. The second example employs `pushMatrix()` and `popMatrix()` to isolate the effects of the `translate()` function to apply only to the first rectangle. Because the other rectangle is drawn after the call to `popMatrix()` it draws from its x-coordinate without being affected by the translation.



```
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
rect(0, 50, 66, 30);
```

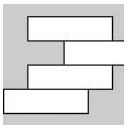
16-04



```
pushMatrix();
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
popMatrix();  // Remove the shift
// This shape is not affected by translate() because
// the transformation is isolated between the pushMatrix()
// and popMatrix()
rect(0, 50, 66, 30);
```

16-05

Embedding the `pushMatrix()` and `popMatrix()` functions can further control their range. In the following example, the first rectangle is affected by the first translation, the second rectangle is affected by the first and second translations, and the third rectangle is only affected by the first translation because the second translation is isolated with a `pushMatrix()` and `popMatrix()` pair. The fourth rectangle is not affected by any of the translations because the `popMatrix()` on the second-to-last line cancels the `pushMatrix()` on the first line.



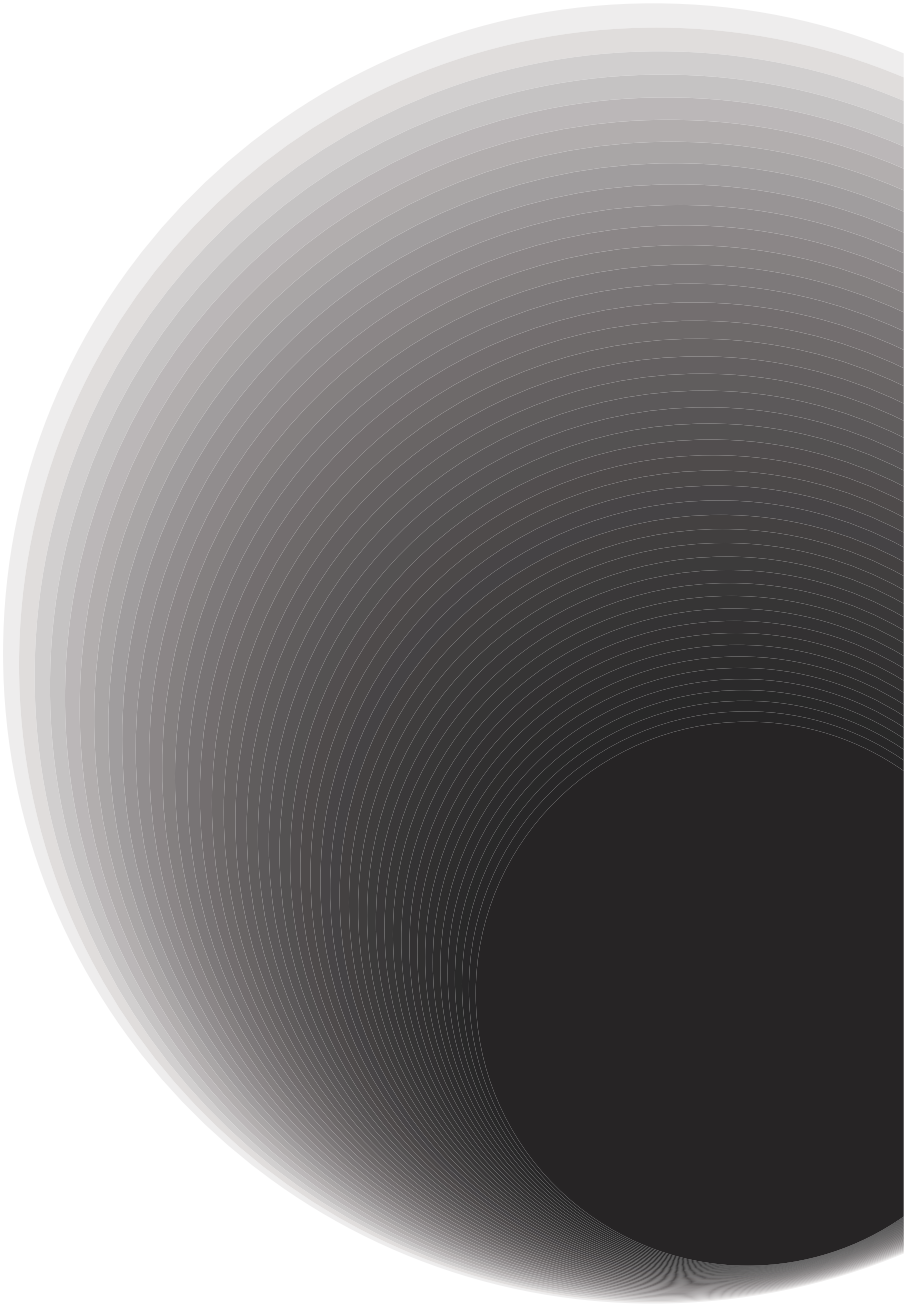
```
pushMatrix();
translate(20, 0);
rect(0, 10, 70, 20);  // Draws at (20, 30)
pushMatrix();
translate(30, 0);
rect(0, 30, 70, 20);  // Draws at (50, 30)
popMatrix();
rect(0, 50, 70, 20);  // Draws at (20, 50)
popMatrix();
rect(0, 70, 70, 20);  // Draws at (0, 70)
```

16-06

The transformation functions for rotating and scaling are introduced in Transform 2 (p. 137).

Exercises

1. *Use `translate()` to reposition a shape.*
2. *Use multiple translations to reposition a series of shapes.*
3. *Use `pushMatrix()` and `popMatrix()` to rearrange the composition from exercise 2.*



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.

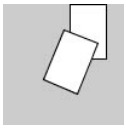
Rotation, Scaling

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)
```

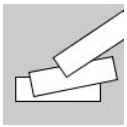
The rotate function assumes that the *angle* is specified in units of radians (p. 117). Shapes are always rotated around their position relative to the origin (0,0), and positive numbers rotate them in a clockwise direction.

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.



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

17-01



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

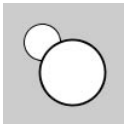
17-02

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:

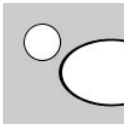
```
scale(size)
scale(xsize, ysize)
```

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.



17-03

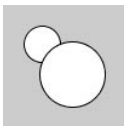
```
smooth();
ellipse(32, 32, 30, 30);
scale(1.8);
ellipse(32, 32, 30, 30);
```



17-04

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

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.



17-05

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

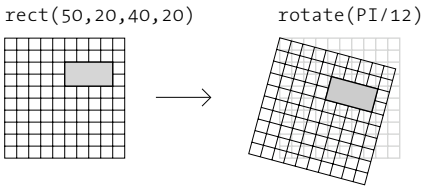
As with `translate()` and `rotate()`, the effects of each `scale()` accumulate each time the function is run.

17-06

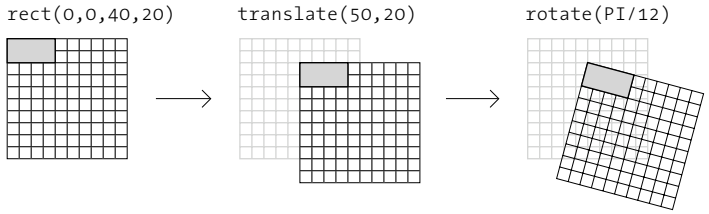
```
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
```

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:

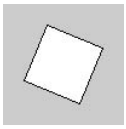


To rotate this shape around its upper-left corner, you must place that point at the coordinate (0,0). A translation is used to put the shape into the desired position in relation to the global coordinates. When the rotate function is run, the shape now orbits around its upper-left corner, the origin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



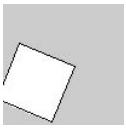
There are two ways to think about transformations. One method is to view the coordinate system as modified and the coordinates for shapes as converted to the new coordinate system. For example, if the coordinate system is rotated 30°, the coordinates of any shape drawn to the screen are converted into 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 :



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

17-07



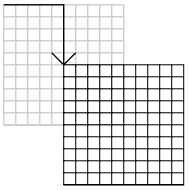
```
rotate(PI/8);
translate(width/2, height/2);
rect(-25, -25, 50, 50);
```

17-08

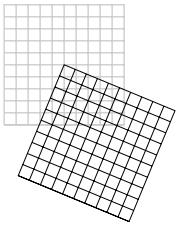
Code 17-07 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom

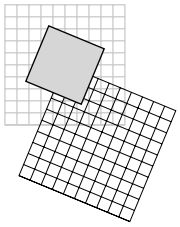
Translate



Rotate

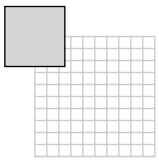


Draw rectangle

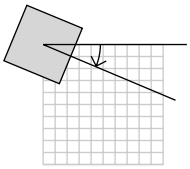


Shape view
Reading the code from bottom to top

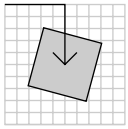
Draw rectangle



Rotate



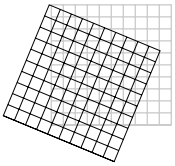
Translate



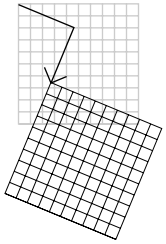
Code 17-08 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom

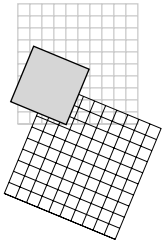
Rotate



Translate

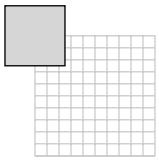


Draw rectangle

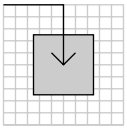


Shape view
Reading the code from bottom to top

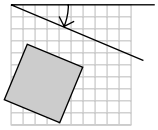
Draw rectangle



Translate

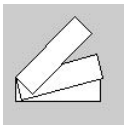


Rotate



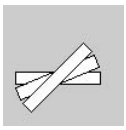
Transformation combinations
The order in which transformations occur in a program affects how they combine. For example, a `rotate()` after a `translate()` will have a different effect than the opposite. These diagrams present two ways to think about the transformations in codes 17-06 and 17-07.

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



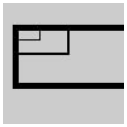
17-09

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



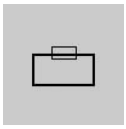
17-10

```
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);
}
```

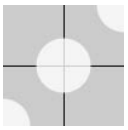
17-14

```
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

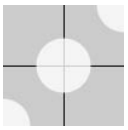
The default position of the coordinate origin (0,0) is the upper-left corner of the display window, the x-coordinate numbers increase to the right, the y-coordinates increase from the top, and each coordinate maps directly to a pixel position. The transformation functions can change these defaults to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