**Basic Drawing in Processing**

**Carson Foster**

Drawing Basic Shapes

The Point

 The most basic ‘shape’ that you can draw is a point. Yes, a single point. Processing’s syntax for drawing a point is quite simple:



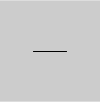
The two arguments represent the x and y coordinates of the point.



In this example, the point function creates a single point in the middle of the screen. The mechanism pictured here is quite useful: after the size function is used, the variable names width and height can be used to refer to the width and height of the drawing space, respectively.

The Line

From the humble point, we now arrive at the line: a connection between two points. In Processing, the syntax to draw a line is the same as the previously discussed representation: the coordinates for the start point, and then the coordinates for the end point, as you can see here:





In the following example, a line is drawn in the center third of the drawing space.



The Triangle

The line had two points, so can you make a guess as to what sort of shape we’ll be discussing now? That’s right, very observant of you. The triangle is our next shape, and is represented as you would expect:



These three points define the triangle, in the way you would imagine.



As you can see, this command draws a nice isosceles triangle in the middle of the drawing area.

The Square

As the next step in the logical progression of shapes, we next look at the syntax for drawing a square. The three necessary arguments to draw a square in Processing are the coordinates of the upper left-hand point of the square, and then a number that is both the width and height of the square.



In the following example, we’ll draw a square at the bottom right of the screen, and its area will be one fourth of the sketch’s area.

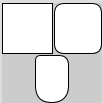


The Rectangle

The more general form of the square, the rectangle is the next shape we’ll cover. The rectangle has a couple of different sets of arguments you can use. All of them include the basic representation of the rectangle that we discussed: the coordinates of the top-left corner, and then the width and the height. The first set of arguments is just these. The other two specify a radius/radii for drawing rounded rectangles. The second set specifies a single radius that it applies to all corners. The last set of arguments needs the 4 regular arguments, and 4 arguments representing the separate radii of each corner, starting at the top left and moving clockwise.



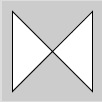
As you can see, there are a couple options to choose from when drawing a rectangle. The following example will show all of them.



I’ve spaced the rectangles out a little bit to make sure that their outlines are visible. Take note of how the different corner radii appear.

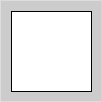
The Quadrilateral

Next, we’ll discuss the more general form of the rectangle: the quadrilateral. As you now definitely suspect, a quadrilateral is specified by four points in Processing. These points must be provided in clockwise or counterclockwise order.



The same points can construct different shapes if they are provided in a different order:



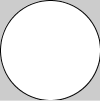
This creates a bowtie shape, while the following will create a rectangle:



The Circle

We’ll next be discussing the curved shapes, so we’ll start with the simplest, the circle. A circle in Processing is described in a similar way to the common representation we have already looked at.



The coordinates still refer to the center of the circle, but the third argument, instead of the radius, represents both the width and the height of the circle (the diameter).

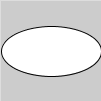


In this example, a circle is inscribed in the center of the sketch window.

The Ellipse

The ellipse is the more general form of the circle, and we’ll describe an ellipse using the same four parameters as a rectangle: coordinates (but this time of the center), and width and height.



Be mindful that the x and y coordinates describe the **center** of the ellipse. In some other languages, ellipses are represented with the same arguments, but the x and y coordinates represent the upper-left corner of the bounding rectangle of the ellipse (the rectangle that could be drawn surrounding the ellipse). This is not the case in Processing.

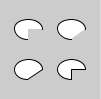


Pay attention to how the arguments affect the actual ellipse shown in this example.

The Arc

More complicated than our other two curved shapes, we next speak about the arc, which is essentially part of an ellipse. The syntax for creating is arc is as follows:



The first four arguments correspond to the same arguments for an ellipse. The start and stop arguments are angles (in radians) to start and stop the drawing of the ellipse, in clockwise order. This means that π/2 is pointing down, and that the start argument must be smaller than the stop argument, so sometimes negative values will be in order. The last argument is an optional mode, which can be one of 3 values: PIE, CHORD, or OPEN. See the example to find out how each one works.



The arcs proceed in clockwise order, so the top left arc is the first arc function call, and the bottom left arc is the third arc function call.

The Bezier Curve

The Bezier curve is a type of curve commonly used in computer graphics. I’ll only provide a brief introduction, but there are plenty of online resources if you would like to research further. In Processing, the command we’ll be using only uses cubic Bezier curves, which is flexible enough for basic applications. Essentially, you can represent a cubic Bezier curve with four points:



The first point and the last point are anchor points: the Bezier curve starts at the first point and ends at the last point. The second and third points are called control points. They usually don’t lie on the actual Bezier curve, but they direct the curve instead. What this means, in short, is that the Bezier curve goes, from the starting anchor, in the direction of the first control point, and approaches the ending anchor from the direction of the second control point.



This example shows the Bezier curve, as well as the lines between the starting anchor and first control point and the ending anchor and second control point. As you can see, the line leaves the starting anchor in the direction of the first control point and enters the ending anchor from the direction of the second control point. The two unfamiliar functions relate to color and will be discussed shortly.

Irregular, Closed Polygons

Now, we’ll discuss how to create irregular, closed polygons, which are shapes made up of line segments that do not necessarily have to be the same length and connect the end and beginning. The functions discussed here have more applicable uses, but they are too advanced for the scope of this series. The syntax we will use is as follows:



To start, call the beginShape function. To end the shape, call the endShape(CLOSE) function. In between these too functions, make any number of calls to vertex, which takes a point as its arguments. As the names imply, each call to vertex adds a vertex (corner) to the polygon.



As you can see, this sequence of function calls draws an arrowhead-like shape to the sketching area.

Color, Stroke, and Fill

Recall that the normal representation of color is with either 3 or 4 numbers, representing the red, green, blue, and optionally opacity/alpha. In Processing, you have the choice of either using variables to store colors, or to specify them on the spot. To create a primitive of the color datatype, using the following syntax:



The last two options are familiar enough: they let you create a color by specifying the three/four values we have already discussed. Next, let’s touch on what this grey parameter is. Essentially, it’s an integer between 0 and 255, but for the greyscale, where 0 is completely black, and 255 is completely white. Therefore, you can specify a greyscale color, with an optional opacity, very quickly.

To store this value, you declare and initialize the color variable the same way you would in Java. Observe in the following example:



This is all well and good, but how do we use our color variables, or immediately specify colors for use? This is what we’ll discuss next, and the three following functions have the same syntax:



This closely resembles the multiple forms of the color function, but also includes two more options: which are passing a color variable/value, and passing a color with an alpha value.

Background

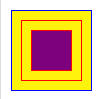
The background function simply sets the background of the sketch area to a specified color, instead of the grey that you’ve seen in most of the examples so far. The default background is this grey color.

Stroke

Stroke is what we call the outline of a shape, which is usually black. In our previous examples from this chapter, the stroke is the black border surrounding the white shape. As you can see, the default stroke is black. However, you can change this by calling the stroke function, which changes the stroke color for all of the shapes resulting from the following function calls (in the same program), until you change the stroke again.

Fill

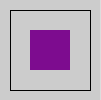
The fill is the color of the inside of the shape, which is bounded by the stroke. This is by default white, and the white you see in previous examples is the fill of those shapes. Similarly, calling the fill function changes the fill color for all following shapes in that program, until you change the fill again.



In the above example, take note of when the stroke and fill of each box changes.

Additional Manipulation of Fill and Stroke

There are a couple of other, pretty simple functions that manipulate fill and stroke. First, we’ll discuss the noFill and noStroke functions, which just turn off the fill and stroke, respectively. These functions take no arguments, so just look at the example.



As you can see, the outermost square has no fill, the innermost square has no stroke, and the square in between them has neither, meaning that it is completely invisible.

The remaining functions only apply to stroke, and they change the style and size of the outline. First, the strokeWeight function changes the size of the stroke. It takes one argument, the new weight/size of the stroke, in pixels.



Next, the strokeCap function changes the appearance of the ends of a stroke. You can specify one of three different cap modes: SQUARE, PROJECT, or ROUND. ROUND is the default mode.



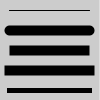
Finally, the strokeJoin function changes how the strokes of line segments are connected. You can specify one of three join modes to this function: BEVEL, ROUND, or MITER. MITER is the default join mode.



Take a look at how the different joints look in the following example:



Finally, look at the different stroke weights and stroke caps:



Transformations

Now we’ll talk about the same three transformations we looked at last lesson, as well as two utility functions. First, let’s take a look at how transformations work in Processing. Similar to stroke and fill, a transformation is applied to all shapes made by functions called after that transformation. In addition, transformations are additive, which means that they combine with each other. For example, if you translate 30 pixels left, draw a square, and translate 20 pixels left, you end up translated 50 pixels left. This tells us that transformations are not ‘reset’ before the next transformation. So then, how do we return to our original coordinate system (remember, we transform the coordinate system/axes, not shapes)? Well, you can’t, unless you want to explicitly perform the reverse transformations to bring the coordinate system back to the default. However, you can ensure that transformations you make do not affect the rest of the program. This is accomplished by the two aforementioned utility functions, which are called pushMatrix and popMatrix.

**

Any transformations made in between these two function calls will not affect any following code. After the popMatrix function call, the coordinate system goes back to whatever it was before the pushMatrix call. Therefore, we can use these functions contain all of the changes we make to the coordinate system, without affecting any other code. However, if you have applied a transformation prior to the pushMatrix call, that transformation will still apply to all of the code inside the two functions.

 Before we start talking about the actual transformations again, take note of the following example. We’ll be modifying it to show how the transformations appear in Processing.



Translation

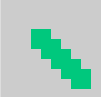
Recall from the last lesson that a translation can be represented with two numbers, each one corresponding to how much the axes are moving in one of the two dimensions. This is the same representation that Processing uses:



Here, look at an example:



This doesn’t seem very useful at all. Couldn’t you just change the coordinates of the square yourself? Yes, yes, you could. However, translation one real use is that it simplifies moving things in for loops. Instead of playing with addition and subtraction for each coordinate, all you have to do is translate the amount you need and draw the exact same shape. Since transformations are additive, this is very handy. Observe:



See how easy that was?

Rotation

A rotation is represented with one number, an angle, in radians, that tells how far clockwise the coordinate system should be rotated.



Are you ready for the example already? Here:





Is this what you wanted? No? Oh. Remember, the coordinate system is rotated, which means that the shape is not rotated in place. In order to achieve this effect, use a translation before your rotation, like so:

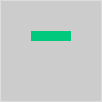


The square is now once more, located at (30, 30).

Scaling

Finally, scaling in Processing works the exact same way as in our past lesson. Processing uses the scale factor instead of a percentage number.



As you can see, you can either provide a single scale factor, or a separate scale factor for each of the dimensions. Here’s an example:



Notice how we use the same trick from rotation to make sure that the final shape ends up where we want it to (at (30, 30)).