

3 Interaction

“Always remember that this whole thing was started with a dream and a mouse.”

—Walt Disney

“The quality of the imagination is to flow and not to freeze.”

—Ralph Waldo Emerson

In this chapter:

- The “flow” of a program.
- The meaning behind **setup()** and **draw()**.
- Mouse interaction.
- Your first “dynamic” *Processing* program.
- Handling events, such as mouse clicks and key presses.

3.1 Go with the flow.

If you have ever played a computer game, interacted with a digital art installation, or watched a screensaver at three in the morning, you have probably given very little thought to the fact that the software that runs these experiences happens over a *period of time*. The game starts, you save princess so-and-so from the evil lord who-zee-ma-whats-it, achieve a high score, and the game ends.

What I want to focus on in this chapter is that very “flow” over time. A game begins with a set of initial conditions: you name your character, you start with a score of zero, and you start on level one. Let’s think of this part as the program’s *SETUP*. After these conditions are initialized, you begin to play the game. At every instant, the computer checks what you are doing with the mouse, calculates all the appropriate behaviors for the game characters, and updates the screen to render all the game graphics. This cycle of calculating and drawing happens over and over again, ideally 30 or more times per second for a smooth animation. Let’s think of this part as the program’s *DRAW*.

This concept is crucial to our ability to move beyond static designs (as in Chapter 2) with *Processing*.

Step 1. Set starting conditions for the program one time.

Step 2. Do something over and over and over and over (and over ...) again until the program quits.

Consider how you might go about running a race.

Step 1. Put on your sneakers and stretch. Just do this once, OK?

Step 2. Put your right foot forward, then your left foot. Repeat this over and over as fast as you can.

Step 3. After 26 miles, quit.

Exercise 3-1: In English, write out the “flow” for a simple computer game, such as Pong. If you are not familiar with Pong, visit: <http://en.wikipedia.org/wiki/Pong>.



3.2 Our Good Friends, *setup()* and *draw()*

Now that we are good and exhausted from running marathons in order to better learn programming, we can take this newfound knowledge and apply it to our first “dynamic” *Processing* sketch. Unlike Chapter 2’s static examples, this program will draw to the screen continuously (i.e., until the user quits). This is accomplished by writing two “blocks of code” *setup()* and *draw()*. Technically speaking *setup()* and *draw()* are functions. We will get into a longer discussion of writing our own functions in a later chapter; for now, we understand them to be two sections where we write code.

What is a block of code?

A block of code is any code enclosed within curly brackets.

```
{
  A block of code
}
```

Blocks of code can be nested within each other, too.

```
{
  A block of code
  {
    A block inside a block of code
  }
}
```

This is an important construct as it allows us to separate and manage our code as individual pieces of a larger puzzle. A programming convention is to indent the lines of code within each block to make the code more readable. *Processing* will do this for you via the Auto-Format option (Tools → Auto-Format).

Blocks of code will reveal themselves to be crucial in developing more complex logic, in terms of *variables*, *conditionals*, *iteration*, *objects*, and *functions*, as discussed in future chapters. For now, we only need to look at two simple blocks: *setup()* and *draw()*.

Let's look at what will surely be strange-looking syntax for *setup()* and *draw()*. See Figure 3.1.

```

void setup() {
  // Initialization code goes here
}

void draw() {
  // Code that runs forever goes here
}

```

fig. 3.1

Admittedly, there is a lot of stuff in Figure 3.1 that we are not entirely ready to learn about. We have covered that the curly brackets indicate the beginning and end of a “block of code,” but why are there parentheses after “setup” and “draw”? Oh, and, my goodness, what is this “void” all about? It makes me feel sad inside! For now, we have to decide to feel comfortable with not knowing everything all at once, and that these important pieces of syntax will start to make sense in future chapters as more concepts are revealed.

For now, the key is to focus on how Figure 3.1’s structures control the flow of our program. This is shown in Figure 3.2.

```

void setup() {
  // Step 1a
  // Step 1b
  // Step 1c
}

void draw() {
  // Step 2a
  // Step 2b
}

```

fig. 3.2

How does it work? When we run the program, it will follow our instructions precisely, executing the steps in *setup()* first, and then move on to the steps in *draw()*. The order ends up being something like:

1a, 1b, 1c, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b ...

Now, we can rewrite the Zoog example as a dynamic sketch. See Example 3–1.

Example 3-1: Zoog as dynamic sketch

```

void setup() {
  // Set the size of the window
  size(200,200);
}

void draw() {
  // Draw a white background
  background(255);

  // Set CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(150);
  rect(100,100,20,100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(100,70,60,60);

  // Draw Zoog's eyes
  fill(0);
  ellipse(81,70,16,32);
  ellipse(119,70,16,32);

  // Draw Zoog's legs
  stroke(0);
  line(90,150,80,160);
  line(110,150,120,160);
}

```

setup() runs first one time. **size()** should always be first line of **setup()** since *Processing* will not be able to do anything before the window size if specified.

draw() loops continuously until you close the sketch window.

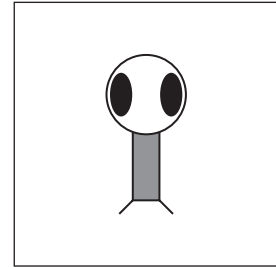


fig. 3.3

Take the code from Example 3-1 and run it in *Processing*. Strange, right? You will notice that nothing in the window changes. This looks identical to a *static* sketch! What is going on? All this discussion for nothing?

Well, if we examine the code, we will notice that nothing in the **draw()** function *varies*. Each time through the loop, the program cycles through the code and executes the identical instructions. So, yes, the program is running over time redrawing the window, but it looks static to us since it draws the same thing each time!



Exercise 3-2: Redo the drawing you created at the end of Chapter 2 as a dynamic program. Even though it will look the same, feel good about your accomplishment!

3.3 Variation with the Mouse

Consider this. What if, instead of typing a number into one of the drawing functions, you could type “the mouse’s *X* location” or “the mouse’s *Y* location.”

```
line(the mouse's X location, the mouse's Y location, 100, 100);
```

In fact, you can, only instead of the more descriptive language, you must use the keywords *mouseX* and *mouseY*, indicating the horizontal or vertical position of the mouse cursor.

Example 3-2: *mouseX* and *mouseY*

```
void setup() {
  size(200,200);
}

void draw() {
  background(255);

  // Body
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(mouseX,mouseY,50,50);
}
```

Try moving **background()** to **setup()** and see the difference! (Exercise 3–3)

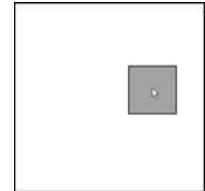
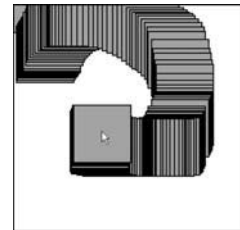


fig. 3.4

mouseX is a keyword that the sketch replaces with the horizontal position of the mouse.
mouseY is a keyword that the sketch replaces with the vertical position of the mouse.



*Exercise 3–3: Explain why we see a trail of rectangles if we move **background()** to **setup()**, leaving it out of **draw()**.*



An Invisible Line of Code

If you are following the logic of **setup()** and **draw()** closely, you might arrive at an interesting question: *When does Processing actually display the shapes in the window? When do the new pixels appear?*

On first glance, one might assume the display is updated for every line of code that includes a drawing function. If this were the case, however, we would see the shapes appear onscreen one at a time. This would happen so fast that we would hardly notice each shape appearing individually. However, when the window is erased every time *background()* is called, a somewhat unfortunate and unpleasant result would occur: flicker.

Processing solves this problem by updating the window only at the end of every cycle through *draw()*. It is as if there were an invisible line of code that renders the window at the end of the *draw()* function.

```
void draw() {
  // All of your code
  // Update Display Window -- invisible line of code we don't see
}
```

This process is known as *double-buffering* and, in a lower-level environment, you may find that you have to implement it yourself. Again, we take the time to thank *Processing* for making our introduction to programming friendlier and simpler by taking care of this for us.

We could push this idea a bit further and create an example where a more complex pattern (multiple shapes and colors) is controlled by *mouseX* and *mouseY* position. For example, we can rewrite Zoog to follow the mouse. Note that Zoog's body is located at the exact location of the mouse (*mouseX*, *mouseY*), however, other parts of Zoog's body are drawn relative to the mouse. Zoog's head, for example, is located at (*mouseX*, *mouseY-30*). The following example only moves Zoog's body and head, as shown in Figure 3.5.

Example 3-3: Zoog as dynamic sketch with variation

```
void setup() {
  size(200,200); // Set the size of the window
  smooth();
}

void draw() {
  background(255); // Draw a white background

  // Set ellipses and rects to CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(mouseX,mouseY,20,100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(mouseX,mouseY-30,60,60);
```

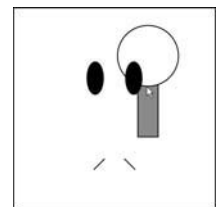



fig. 3.5

Zoog's body is drawn at the location (*mouseX*, *mouseY*).

Zoog's head is drawn above the body at the location (*mouseX*, *mouseY-30*).


```
// Draw Zoog's eyes
fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);

// Draw Zoog's legs
stroke(0);
line(90,150,80,160);
line(110,150,120,160);
}
```

 *Exercise 3-4: Complete Zoog so that the rest of its body moves with the mouse.*

```
// Draw Zoog's eyes
fill(0);
ellipse(_____, _____, 16, 32);
ellipse(_____, _____, 16, 32);

// Draw Zoog's legs
stroke(0);
line(_____, _____, _____, _____);
line(_____, _____, _____, _____);
```

 *Exercise 3-5: Recode your design so that shapes respond to the mouse (by varying color and location).*

In addition to *mouseX* and *mouseY*, you can also use *pmouseX* and *pmouseY*. These two keywords stand for the “previous” *mouseX* and *mouseY* locations, that is, where the mouse was the last time we cycled through *draw()*. This allows for some interesting interaction possibilities. For example, let’s consider what happens if we draw a line from the previous mouse location to the current mouse location, as illustrated in the diagram in Figure 3.6.

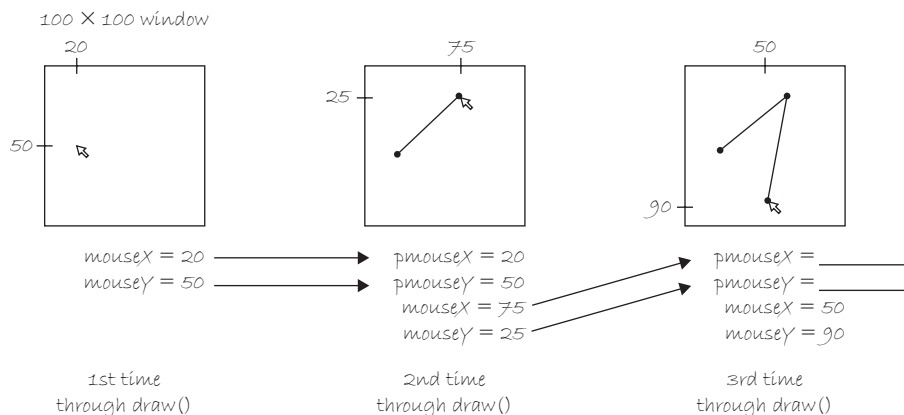


fig. 3.6



Exercise 3-6: Fill in the blank in Figure 3.6.

By connecting the previous mouse location to the current mouse location with a line each time through `draw()`, we are able to render a continuous line that follows the mouse. See Figure 3.7.

Example 3-4: Drawing a continuous line

```
void setup() {
  size(200,200);
  background(255);
  smooth();
}

void draw() {
  stroke(0);
  line (pmouseX,pmouseY,mouseX,mouseY);
}
```

Draw a line from previous mouse location to current mouse location.

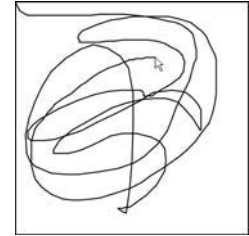


fig. 3.7

*Exercise 3-7: The formula for calculating the speed of the mouse's horizontal motion is the absolute value of the difference between **mouseX** and **pmouseX**. The absolute value of a number is defined as that number without its sign:*



- The absolute value of -2 is 2.
- The absolute value of 2 is 2.

*In Processing, we can get the absolute value of the number by placing it inside the **abs()** function, that is,*

- `abs(-5) → 5`

The speed at which the mouse is moving is therefore:

- `abs(mouseX-pmouseX)`

*Update Exercise 3-7 so that the faster the user moves the mouse, the wider the drawn line. Hint: look up **strokeWeight()** in the Processing reference.*

```
stroke(255);
_____ ( _____ );
line (pmouseX,pmouseY,mouseX,mouseY);
```



3.4 Mouse Clicks and Key Presses

We are well on our way to creating dynamic, interactive *Processing* sketches through the use the *setup()* and *draw()* framework and the *mouseX* and *mouseY* keywords. A crucial form of interaction, however, is missing—clicking the mouse!

In order to learn how to have something happen when the mouse is clicked, we need to return to the flow of our program. We know *setup()* happens once and *draw()* loops forever. When does a mouse click occur? Mouse presses (and key presses) as considered *events* in *Processing*. If we want something to happen (such as “the background color changes to red”) when the mouse is clicked, we need to add a third block of code to handle this event.

This event “function” will tell the program what code to execute when an event occurs. As with *setup()*, the code will occur once and only once. That is, once and only once for each occurrence of the event. An event, such as a mouse click, can happen multiple times of course!

These are the two new functions we need:

- *mousePressed()*—Handles mouse clicks.
- *keyPressed()*—Handles key presses.

The following example uses both event functions, adding squares whenever the mouse is pressed and clearing the background whenever a key is pressed.

Example 3-5: *mousePressed()* and *keyPressed()*

```
void setup() {
  size(200,200);
  background(255);
}

void draw() {

}

void mousePressed() {
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(mouseX,mouseY,16,16);
}

void keyPressed() {
  background(255);
}
```

Nothing happens in *draw()* in this example!

Whenever a user clicks the mouse the code written inside *mousePressed()* is executed.

Whenever a user presses a key the code written inside *keyPressed()* is executed.

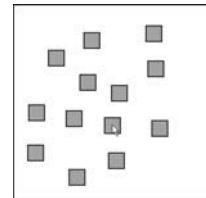


fig. 3.8

In Example 3-5, we have four functions that describe the program’s flow. The program starts in *setup()* where the size and background are initialized. It continues into *draw()*, looping endlessly. Since *draw()* contains no code, the window will remain blank. However, we have added two new functions: *mousePressed()* and

keyPressed(). The code inside these functions sits and waits. When the user clicks the mouse (or presses a key), it springs into action, executing the enclosed block of instructions once and only once.



Exercise 3-8: Add “background(255);” to the draw() function. Why does the program stop working?

We are now ready to bring all of these elements together for Zoog.

- Zoog’s entire body will follow the mouse.
- Zoog’s eye color will be determined by mouse location.
- Zoog’s legs will be drawn from the previous mouse location to the current mouse location.
- When the mouse is clicked, a message will be displayed in the message window: “Take me to your leader!”

Note the addition in Example 3-6 of the function *frameRate()*. *frameRate()*, which requires an integer between 1 and 60, enforces the speed at which *Processing* will cycle through *draw()*. *frameRate(30)*, for example, means 30 frames per second, a traditional speed for computer animation. If you do not include *frameRate()*, *Processing* will attempt to run the sketch at 60 frames per second. Since computers run at different speeds, *frameRate()* is used to make sure that your sketch is consistent across multiple computers.

This frame rate is just a maximum, however. If your sketch has to draw one million rectangles, it may take a long time to finish the draw cycle and run at a slower speed.

Example 3-6: Interactive Zoog

```
void setup() {
  // Set the size of the window

  size(200,200);
  smooth();
  frameRate(30);
}

void draw() {
  // Draw a black background
  background(255);

  // Set ellipses and rects to CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(mouseX,mouseY,20,100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(mouseX,mouseY-30,60,60);
```

The frame rate is set to 30 frames per second.

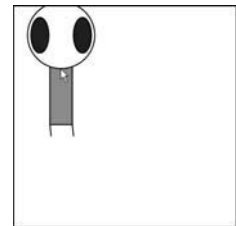


fig. 3.9

```
// Draw Zoog's eyes
fill(mouseX,0,mouseY);
ellipse(mouseX-19,mouseY-30,16,32);
ellipse(mouseX+19,mouseY-30,16,32);

// Draw Zoog's legs
stroke(0);
line(mouseX-10,mouseY+50,pmouseX-10,pmouseY+60);
line(mouseX+10,mouseY+50,pmouseX+10,pmouseY+60);
}

void mousePressed() {
  println("Take me to your leader!");
}
```

The eye color is determined by the mouse location.

The legs are drawn according to the mouse location *and the previous mouse location*.