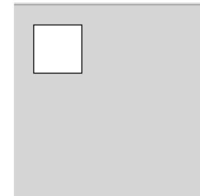
Lab Exercise 4: Drawing and Redrawing

Goal: To write a Processing program **yourself** – the steps are given – and to practice using variables and assignment statements. These are the primitive parts of most programming, meaning that you will use them daily in this class. (You'll use the concepts everywhere, too!)

All you need to do is follow the steps, doing what they say.



Step 1: **Open** the Processing development page, and **set** the size to 200x200. **Draw** a 50x50 rectangle with upper left corner at position 20, 20. **Run** it. **Save** it. This is what you should have so far.

Patter: Your program has two statements, and is a *static* program that draws a single figure. Next we turn it into an active program, because we want it to do something.

Step 2: **Revise** your program to be active by adding the `void setup( ) { ... }` and `void draw( ) { ... }` blocks. The setup block should contain the `size( )` statement as its first statement and the draw block should contain the `rect( )` statement. **Run** it, getting the same result. **Save** it.

Patter: Remember that size is ALWAYS the first thing in `setup( )`. Always. So when we add more stuff to `setup( )` below, don't put it in front of size!

Your program will now have six lines (unless you've added blanks), because we generally write methods like setup on two lines: the first line is everything up to and including the open brace (`{`) for the body, and the matching closing body brace (`}`) goes on its own line, as in

```
void setup( ) {
}
```

This is only custom, but you MUST follow it in CSE120. Computers are completely content to have the text all packed together, leaving spaces only between words, but people hate it. Placing the braces in these standard positions lets people see where the code begins and ends. Hence, the custom.

Notice that when your cursor is near a parenthesis or brace, Processing highlights the other matching one.

Step 3: **Dump** the line around the box using the `noStroke()` operation. Set a nice gray background color, say, 64, 64, 64. Of course, both of these operations go into `setup( )`. **Run** it. **Save** it. This is what you should have so far:

Patter: Recall from the discussion of color, that black, gray and white are produced by setting the RGB values to be the same. So, to get our gray background we can write background(64, 64, 64), but if we don't specify all three, as in background(64), the language assumes we mean the same value for all three.

Step 4: **Declare** four integer variables at the start of your program, that is, *before* setup( ). Two are x and y, which should both be initialized to 20; two are over and down, which should be initialized to 50. When we say "initialized to" we mean, setting them to a value in the declaration, as in

```
int x=20;
```

Replace the constants in your rectangle specification with these four variables: x and y will be the corner position, over will be the width and down will be the height of the rectangle. **Run** it. **Save** it. There should be no change from step 3.

Patter: The point of changing the constants of the rectangle specification to variables is so that we can make them vary. By varying the position of the rectangle, we can make it appear to move, which is what we start next.

Step 5: To refresh the screen – that is, draw a new version of the design repeatedly – place a frameRate(30) command in the setup( ) block of the program. This tells how fast the draw( ) block should be run, i.e. 30 times per second. Then, as the rectangle is repositioned, it will be displayed. (A frame rate of 30 is fast enough to fool our eyes into seeing motion from a sequence of stills; it is standard.)

Continuing, we next think about which position variable to increase to make the box look like it is moving DOWN the window: x or y? Obviously, y because larger values are further down in the window. So, **add** a statement to "increment y by 2" after drawing the box. ("Increment" means make it larger, of course, which is adding.) **Run** it. **Save** it. This is what you should have now:

Patter: Is that what you want? No. You want to see a square falling. What's happening? (I have purposely introduced an error into this lab, because programming involves a lot of trying to do something and not having it quite work right; this is an example.) The kind of debugging we're about to do is typical, so think it through with me.

Since everything has been going along fine so far, the problem must be with our last change. Is the problem with the frameRate( ) or the increment statement? Both or neither? Pretty obviously, the frameRate( ) is OK, because the image does keep changing and that is what it is supposed to do. Is it the increment statement? Pretty obviously, it's

(c) 2011-2012 Lawrence Snyder

OK, too, because the box *is* moving down the window. It just doesn't look much like a square. And that's odd, because from our instruction

```
rect( x, y, over, down);
```

and the fact that over and down have been initialized to the same value, 50, it should be a square. What if we were drawing a square on top of the old square(s)? What would that look like? Like what we just produced! So, we have a theory of what's wrong. We are overwriting the old square. How do we get a new square each time? We need to erase the old one, and the way to do that is to specify the background again. Ah, ha! We only specify the background once. So, let's try moving the background command into the `draw( )` block, at its start. And when we do that, this is what we get:

Step 6: The amount that we incremented `y` by determines the speed of the drop, of course, because we move 2 pixels with each frame (recall there are 30 per second), ours moves down at 60 pixels per second. **Play** with the increment amount. *It's a good time to try alternatives.*

Patter: It is possible to slow this way down by reducing the `frameRate( )`, or using fractional increments, like 0.5. But, to do that, you need to declare `y` to be a `float` because 0.5 is a decimal number. Check it out!

Step 7: Next we make the square turn green as it drops. (No one knows why.) Recall that we change color by setting the fill statement. The default is `fill(255,255,255)`, which obviously fills with white. To get it to progressively turn green, we need to move progressively to 0, 255, 0. We do that be subtracting from the red (the first 255) and blue (the last 255) values. So, **declare** a new integer variable, `change` and initialize it to 0. **Add** a fill statement before drawing the rectangle, and for the first and last arguments **subtract** `change`. Finally, after the increment of `y`, **increment** change by 2, too. When you're finished, your program will look like this:

Patter: Where does the box go when it disappears off of the bottom of the screen? It just keeps going. The coordinates extend (effectively) to infinity, so as long as the program keeps running, the square drops. Would it be possible for the square to reverse? That is, start off the screen and move up? Sure. It's a computer, anything is possible.

Step 8: **Modify** your program so that the square starts below the window and moves up, turning green as it goes. As a hint, keep in mind that the rectangle was set in its original position by how we initialized the `x` and `y` variables. Obviously, a different initialization is needed. And keep in mind that rather than referring to the window height, for example, as 200, refer to it as `height`. That way things will be more flexible. Your result should look like this:

(c) 2011-2012 Lawrence Snyder

Wrap Up

You have written an entire Processing program. Although it was a simple program for you to write, the computer needs millions of instructions to perform the operations that you specify. So, it may look simple, but you've already commanded the computer to do a lot of work.

To Turn In

Rename your `.pde` file with your name and turn it in to the class drop box.