**Animation in Processing**

**Carson Foster**

Now we’ll be looking at how animation works in Processing. Spoiler: you’ve already looked at some examples animation.

How Animation Works

In Processing, animation works by changing what is displayed during each draw function. That’s it. We’ve already seen examples of this when we looked at interaction and changed what was displayed using the keyboard and mouse.

Traditional Animation

This being said, you can also do the very traditional kind of animation, in which you simply display a slightly different image each frame. This gives the illusion of movement and change. Let’s look at an example utility class that deals with these types of animations.

This class will represent picture animations, and expects each picture to be a different image, with a common stem and extension, but numbered differently. This class will allow a maximum of 10000 frames/images total, with the first being labeled as “0000”. For example, with a stem of “animationImage” and an extension of “png”, the class will expect the first image of the animation to be named “animationImage0000.png”, the second to be “animationImage0001.png”, and so on.



First, I import ArrayList, since there will be an unknown number of PImages. There are three instance variables for this class: one holds the images that the class displays, another keeps track of the current frame number, and the last tells whether or not we should start the animation from the beginning once it has ended. Then, I have declared a utility method that formats an integer into the 4 digit, prefixed by zeroes format.



The first constructor for this class allows you to load an unknown number of image frames. Since the maximum number of frames this program allows is 10000, we know that there is less than that number of image files. Once our temporary variable is null, we know that the loadImage has failed, and that there are no more images. The next constructor optimizes the data structure for a known number of image frames, as we can pass this value to our ArrayList to ensure that the backing array is set at the exact right size and never resizes. Otherwise, this constructor is very similar to the previous one. In both constructors, we initialize the frame variable to -1, as we increment at the beginning of each nextImage call. Speaking of which, this function simply returns the next frame in the animation, appropriately handling both looping and non-looping cases. Finally, the class provides another method to allow users to jump to a specific frame in the animation.

Here’s an example of how this class would be used:



For this example, the 4 frames would be called “tile0000.png”, “tile0001.png”, “tile0002.png”, and “tile0003.png”. I have also manually decreased the frame rate of this example, as there are only 4 frames to be displayed.

A Button Class

Now, we’ll go through, step by step, a simple button ecosystem that I created, which you can use as well.



Here, I have defined two interfaces: the first is a functional interface that represents a method that will draw the button later on, and the second is the general button interface. This specifies that all buttons will update and draw themselves, and they can be queried as to whether they are hovered over or pressed currently. Notice that there is no mention of event listeners anywhere; it is not necessary to include, but if you feel the need to modify the following classes, it is a relatively simple addition. “It is left to the reader as an exercise” to make the implementation.

Next, I created an abstract button class that acts as an adapter between the interface and the following concrete classes. No matter what type of button, each button is a rectangle with coordinates, width, and height. Each button will also have Boolean variables that represent whether the button is currently pressed or hovered. This leads to a trivial implementation of two of the required methods. The update method has pretty clear logic, and it serves to update the status of the two Boolean variables. Finally, we have left the drawing of the button up to the subclasses.



Our first type of button will be the “image button”. This is by no means the standard name, if one such name exists, of this type of button. An “image button” has one image per state: pressed, hovered, and neither/normal. The corresponding image is displayed as each state occurs.



The implementation of the drawing method is trivial for this type of button, as we simply display the appropriate picture for the current state. The constructor simply initializes the instance variables.



This is what I call the “drawn button”. This type of button is essentially passed three methods, one for each possible state. The drawing method is similar to the drawing method of the “image button”, and comparably trivial. Let’s now look at an example of both types of buttons.





Because of the old Java version used in the PDE, I have been forced to resort to using anonymous classes. If you, reader, end up using this code in pure Java, please use lambdas. The example methods here simply change the color of the stroke of a rectangle, but you can do so much more.