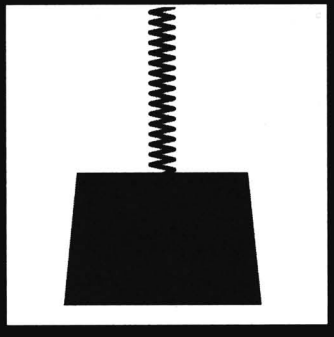
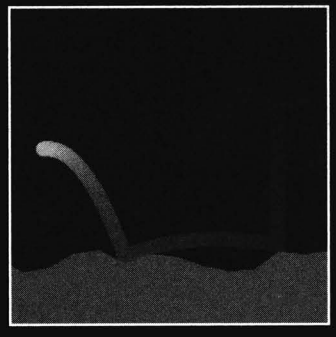
11 **MOTION**





Let the fun begin! I suspect many of you have been waiting patiently (or not so patiently) for this chapter. It took a certain restraint on my part to not start flying pixels around the screen back in Chapter 1. Animation and motion design is usually what gets my students hooked on coding—and even to embrace trigonometry. This chapter, we'll explore all kinds of neat motion (including some trig), from deflecting to bouncing to easing to springing. You'll even create an asteroid shower and learn how to code all sorts of interesting collisions.

Before we dive right into the code, though, I want to discuss very briefly how computer animation works in general, and also some different strategies employed to implement it.

Animation basics

A computer monitor is an animation machine, continuously refreshing the pixels on the screen at the monitor's refresh rate. Obviously, this is not terribly engaging animation, but the fact that animation can happen in front of your eyes and be wholly undetectable is significant. Our brains are wired this way as well, as we perceive a persistent unflickering visual field, in spite of the fact that our eyes are continuously receiving new data. In animation, we exploit this phenomenon by moving data in front of the viewer's eyes at certain rates, tricking the brain into seeing smooth, continuous motion. To move this data, the computer needs to start a process that changes pixel color values over time. From a computer animator's standpoint, this could simply include dragging a shape to two different places on the screen using some timeline interface, such as found in applications like Flash, Director, Final Cut Pro, After Effects, and LightWave. Often, these high-end products employ a keyframing model, where the user sets the beginning keyframe and ending keyframe of a simple animation, and the computer generates the in-between frames. What these cool, very high-level applications don't reveal is how the actual time sequencing is handled internally by the computer—usually you just hit a play button.

Computers with a single processor execute a single instruction at a time, but very quickly (over a billion times per second). However, in a multitasking environment like we're all used to working in, we need to do more than one thing at a time. For example, I may be running five programs at the same time, each with certain automated functions occurring behind the scenes. I may also set a rendering in a 3D program and work on something else while it's completing in the background. Obviously, I wouldn't want all these different processes to be put in a line, or queue, and have to wait for each to be completed before the next task begins. Instead, the operating system splits the executing commands up so that the individual processes can be weaved together, sharing the available processing time. Since the computer can do so many things in a second, we perceive simultaneous processing.

We often refer to these individual processes as **threads**. Normally, we won't need to worry about low-level stuff like threads in Processing, as the application handles it behind the scenes. However, it's helpful to have a little understanding of how threads work, especially since threads are critical for animation.