Probability & non-uniform distributions

Remember when you first started programming here? Perhaps you wanted to draw a lot of circles on the screen. So you said to yourself: “Oh, I know. I’ll draw all these circles at random locations, with random sizes and random colours.” In a computer graphics system, it’s often easiest to seed a system with randomness. In these lessons, however, we’re looking to build systems modelled on what we see in nature. Defaulting to randomness is not a particularly thoughtful solution to a design problem—particularly the kind of problem that involves creating an organic or natural-looking simulation.

With a few tricks, we can change the way we use `random()` to produce “non-uniform” distributions of random numbers. This will come in handy throughout this course as we look at a number of different scenarios. When we examine genetic algorithms, for example, we’ll need a methodology for performing “selection”—which members of our population should be selected to pass their DNA to the next generation? Remember the concept of survival of the fittest? Let’s say we have a population of monkeys evolving. Not every monkey will have an equal chance of reproducing. To simulate Darwinian evolution, we can’t simply pick two random monkeys to be parents. We need the more “fit” ones to be more likely to be chosen. We need to define the “probability of the fittest.” For example, a particularly fast and strong monkey might have a 90% chance of procreating, while a weaker one has only a 10% chance.

Let’s pause here and take a look at probability’s basic principles. First we’ll examine single event probability, i.e. the likelihood that a given event will occur.

If you have a system with a certain number of possible outcomes, the probability of the occurrence of a given event equals the number of outcomes that qualify as that event divided by the total number of all possible outcomes. A coin toss is a simple example—it has only two possible outcomes, heads or tails. There is only one way to flip heads. The probability that the coin will turn up heads, therefore, is one divided by two: 1/2 or 50%.

Take a deck of fifty-two cards. The probability of drawing an ace from that deck is:

number of aces / number of cards = 4 / 52 = 0.077 = ~ 8%

The probability of drawing a diamond is:

number of diamonds / number of cards = 13 / 52 = 0.25 = 25%

We can also calculate the probability of multiple events occurring in sequence. To do this, we simply multiply the individual probabilities of each event.

The probability of a coin turning up heads three times in a row is:

(1/2) * (1/2) * (1/2) = 1/8 (or 0.125)

…meaning that a coin will turn up heads three times in a row one out of eight times (each "time" being three tosses).

Want to review probability before continuing? Study compound events and dependent probability.

There are a couple of ways in which we can use the `random()` function with probability in code. One technique is to fill an array with a selection of numbers—some of which are repeated—then choose random numbers from that array and generate events based on those choices.

Running this code will produce a 40% chance of printing the value 1, a 20% chance of printing 2, and a 40% chance of printing 3.

We can also ask for a random number (let’s make it simple and just consider random decimal values between 0 and 1) and allow an event to occur only if our random number is within a certain range. Check out the example below, and keep clicking restart until the randomly picked number is finally less than the threshold:

This method can also be applied to multiple outcomes. Let’s say that Outcome A has a 60% chance of happening, Outcome B, a 10% chance, and Outcome C, a 30% chance. We implement this in code by picking a random number and seeing into what range it falls.

- *between 0.00 and 0.60 (60%) –> Outcome A*
- *between 0.60 and 0.70 (10%) –> Outcome B*
- *between 0.70 and 1.00 (30%) –> Outcome C__*

Click the restart button to see how often you get different outcomes:

We could use the above methodology to create a random walker that tends to move to the right. Here is an example of a `Walker` with the following probabilities:

- *chance of moving up: 20%*
- *chance of moving down: 20%*
- *chance of moving left: 20%*
- *chance of moving right: 40%*