CS 106 Winter 2016
Craig S. Kaplan

Module 08

Randomness and noise

Topics

- **The use of randomness as a design tool, controlling randomness in code**
- **Emergent design from simple rules**

Readings

- *Learning Processing*, Sections 13.3–13.6

Introduction

Artists have long relied on sources of randomness as part of the design process. In music, Mozart has his musical dice and John Cage placed notes based on the imperfections he saw on an empty page. Of course, randomness also plays a large role in visual art, with Jackson Pollock as just one prime example.

Throughout this course, we've used the built-in function `random()` in an informal way to get live behaviour that we didn't explicitly design in detail, tying ourselves in to this long tradition. In order to maximize our ability to get interesting results in the presence of randomness, we need to obtain greater control over the way that we generate and use random numbers. We'll talk not just about the `random()` function, but also about `randomSeed()` and `noise()`.

Random numbers aren't

Mathematicians argue strenuously over what makes a number "random"—it turns out to be a deep question. We won't spend time on that, but it's important to understand that the "random" numbers that Processing gives you are *not truly random*. In fact, they're completely predetermined! Once you discover this fact, it becomes possible to control the way random numbers are generated in a useful way.

Truly random numbers are hard to make—you need something that behaves in an unpredictable way according to the laws of physics, like a lump of radioactive material. In lieu of that, we use simple mathematical formulas that give the *illusion* of randomness. They produce numbers in a pattern, but the pattern is too hard for us to discern. Such functions are called *pseudo-random number generators*. An example might look like this:

```
int a = 8121;
int c = 28411;
int m = 134456;
int seed = 0;

int myRandom()
{
  seed = (a*seed + c) % m;
  return seed / 1024;
}
```

You don't need to understand the choice of numbers above, or even the mathematical formula in the body of the function. The important observations are:

- Each time you call the function `myRandom()` it will give you back some unpredictable positive integer (between 0 and around 130). The number is different each time because it depends on the variable `seed`, which is overwritten after every call.
- There's no actual randomness in this function. For example, every time I restart my sketch and print out the first ten values I get from `myRandom()`, I see the same ten numbers.

It turns out that this is very close to the way the built-in function `random()` really works. It's a completely deterministic mathematical formula, one that will ultimately produce numbers in a fixed (but hard to predict) pattern.

But wait, you cry, this isn't the way that Processing works! Consider a simple sketch:

```
println( random( 1 ) );
```

Running this sketch multiple times will produce different outputs! In that case, where's the pattern?

Well, the pattern you get is determined by a value called a *seed*. In the sketch above, if I manually change the value of the `seed` variable I'll jump into the random numbers at a different point in the pattern, but the pattern itself will be the same thereafter.

We have the same power in Processing, via the built-in function `randomSeed()`. That function takes any integer as input, and initializes the pseudo-random number generator using that integer. The actual number you pass in is basically irrelevant, except:

- Different seeds will drop you in at different points in the pattern
- If you reset Processing with the same seed, you'll get the same pattern of random numbers afterwards

With that in mind, Processing does something useful for you—when your sketch starts, it sets the random seed to a value based on the current time, making the numbers seem "more random". But you can override that default behaviour at any time by using `randomSeed()`.

Pseudo-random numbers are perfectly fine as a source of chaos for art and design purposes, but it's very dangerous to assume blindly that they're truly random. Cryptographic systems based on pseudorandom numbers are easier to hack. Gambling machines that use pseudorandom numbers without

care can be beaten. When real randomness matters, there are better sources and better algorithms, though it's always a hard problem.

Slashes and backslashes

Let's start to look at using randomness in context, with the following amazingly simple (non-Processing) program:

```
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

Don't bother trying to run this program, unless you're using the BASIC programming language on a Commodore 64.

This program loops forever, randomly printing forward and backward slashes. The result looks a bit like a random maze, though it isn't truly a maze (there are loops and closed-off passages). It may seem innocuous, but this program is the subject of an entire book on computer art, written by a group that includes Casey Reas, one of the creators of Processing.

It isn't too hard to translate this into a Processing sketch, though it's better to use more than one line of code.

```
void setup()
{
  size( 600, 400 );
  strokeCap( ROUND );
  strokeWeight( 7 );
  stroke( 0 );
  noFill();

  background( 255 );

  for( int y = 0; y < height; y += 20 ) {
    for( int x = 0; x < width; x += 20 ) {
      if( random(1) <= 0.5 ) {
        line( x, y, x + 20, y + 20 );
      } else {
        line( x + 20, y, x, y + 20 );
      }
    }
  }
}
```

Notice that we get a different design every time we run the sketch. That's good: it means that our random numbers keep changing, like we would hope.

But that lack of repetition can also be a liability. What if we want to redraw the frame in the same (or almost the same) way? If we do so naively, we get different random decisions, and the pattern changes chaotically.

```
void setup()
{
  size( 600, 400 );
  strokeCap( ROUND );
  strokeWeight( 7 );
  stroke( 0 );
  noFill();
}

void draw()
{
  background( 255 );

  for( int y = 0; y < height; y += 20 ) {
    for( int x = 0; x < width; x += 20 ) {
      if( random(1) <= 0.5 ) {
        line( x, y, x + 20, y + 20 );
      } else {
        line( x + 20, y, x, y + 20 );
      }
    }
  }
}
```

One way to resolve this chaos is to make all the random numbers you need once up front, and set them aside. For example, we might create a 2D array of random numbers, and always refer back to them when drawing the slashes and backslashes. (Exercise: do it!) But there's a much more elegant approach: just use the built-in function `randomSeed()` at the start of each frame to re-initialize the random number generator to produce the same sequence of "coin flips".

Example sketch: TenPrint

Note the use of the constant 0.5 in the code above. When we say `random(1)`, we get an unpredictable floating-point number that's at least 0 and less than 1. What would happen if we chose a number less than 0.5? If we chose one higher? What if we made the number change over space or time?

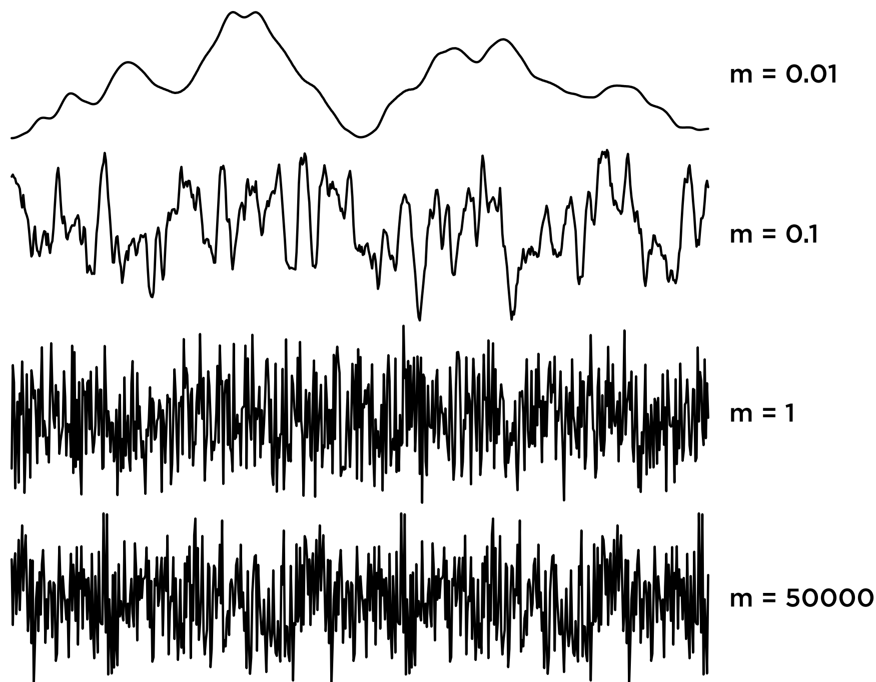
So far this sketch works fine, but what if we want to *scroll* the random maze? We can restart the sequence of random numbers, but we can't translate it or otherwise extend it to cover new grid cells as they enter the screen. What we really need is some way to conceptually "attach" random numbers to every point in an ambient field in space. That would let us develop a theoretically infinite grid of slashes and backslashes, and simply show a small fragment of it in every frame.

That's one possible use of the built-in `noise()` function. This function can be passed one, two, or three floating-point numbers as parameters. In one dimension, the noise function produces an unpredictable number for every input. But that number never changes—it's permanently associated with the input number. This behaviour is very different from `random()`! The numbers passed in to `noise()` are like *coordinates*, as if you're looking up a number in a map. Every time you look up the same coordinates, you get the same answer. But multiple calls to `random()` will produce a sequence of distinct values.

By default, the noise() function has interesting statistical properties. At very fine scales (i.e., when you zoom in on it), it changes slowly and looks very smooth. As you zoom out is gets more chaotic, but at some point the function runs out of randomness and starts repeating. Still, there's a wide range in which you can use this function productively.

```
float m = 0.01; // We'll vary this

void setup()
{
  size( 600, 200 );
  noFill();
  background( 255 );
  beginShape();
  for( int idx = 0; idx < width; ++idx ) {
    vertex( idx, noise(idx*m)*200 );
  }
  endShape();
}
```



For the random `10 PRINT` sketches, we can use the two-parameter version of the `noise()` function to assign random orientations to every line in the plane, whether or not we actually draw it. Then, we can even add direct manipulation and use the mouse to explore a conceptually infinite random pattern.

Example sketch: TenPrintNoise

Example sketch: TenPrintManip

Example sketch: Truchet

Example sketch: QBert

Combining fractals and randomness

Let's explore two examples that combine fractals and randomness.

Diminishing circles

It's easy to write a Processing sketch that places a set of circles completely at random. Things get more interesting when ask that the circles not intersect. We need to maintain explicit arrays for the centres and radii of the circles, and use code to check whether a new circle intersects any existing one before drawing it.

Example sketch: DiminishingCircles

We also gradually diminish the radius of the circles we're trying to add. Circles never fill up the plane completely, so as the radius goes down we always eventually find places to fit new circles.

This sketch suggests a general framework for creating fractal-like structures. We place objects wherever they fit. If nothing fits, make the objects smaller. Repeat for as long as desired. This technique is explored in a paper by Dunham and Shier, and in some 2D and 3D examples by Paul Bourke. On the more mathematical side, fractals like the Apollonian Gasket are a kind idealized version of this circle fractal.

Mountains

A standard technique for generating fractal mountain ranges is called *midpoint displacement*. Given some lines that make up a mountain range, we divide each line in half and randomly move the midpoint up or down. The amount of displacement in Y is proportional to the distance between the line's endpoints in X, so that we add finer details as we work at smaller scales. It's easiest to use recursion to generate a 2D fractal mountain range.
In addition to the number of levels remaining in the recursion, each recursive call takes four `float` parameters that describe the current line segment to

“mountainify”. The base case simply draws the line segment. The recursive case computes the midpoint of the line segment (i.e., the averages of the X and Y coordinates). It generates a random displacement and moves the Y coordinate of the midpoint up or down by that distance, scaled by the width of the segment (the difference between the X values of its endpoints) and a global scaling factor. Then it recursively draws two sub-mountains, one based on the left sub-segment and one on the right sub-segment. These could be drawn using lots of calls to `line()`, though a more elegant approach is to use `beginShape()`, `endShape()`, and `vertex()`.

This same technique adapts naturally to 3D, though the recursion is more complicated because we have to express the connectivity between every mountain point and its neighbours in a 2D grid.

Example sketch: Mountains