CS 116x Winter 2015
Craig S. Kaplan

Module 06

Procedural Content Generation

Topics

- **Recursion as an extension of hierarchical modelling**
- **Simple fractals**
- **The use of randomness as a design tool, controlling randomness in code**
- **Emergent design from simple rules**

Readings

- *Getting Started*, Pages 97–99
- *Learning Processing*, Sections 13.3–13.6, 13.10
- *Nature of Code*, Chapters 7, 8

Introduction

In this module, I want to collect together a range of topics that demonstrate the usefulness of programming to create designs that would be difficult or impossible to draw by hand. This is one place where the power of computation really shines through: the computer becomes an artistic medium with its own distinct aesthetic.

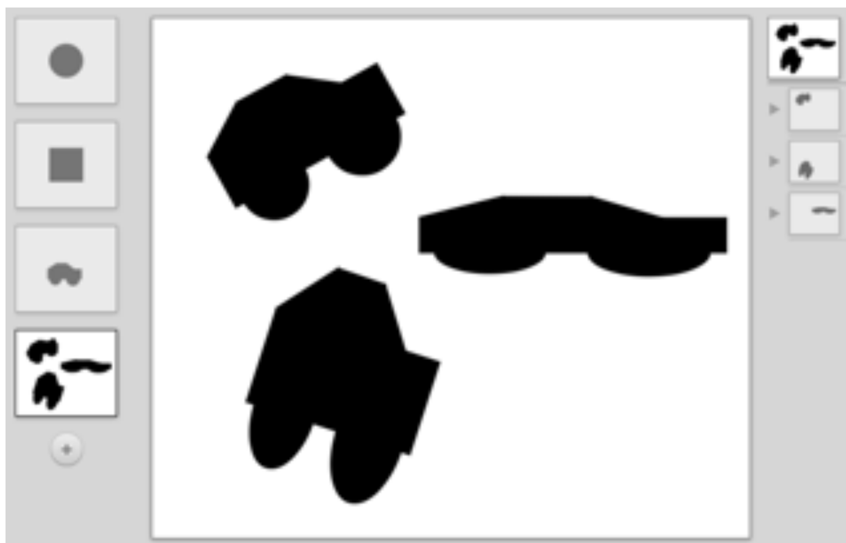
Recursion

Recursion is an incredibly powerful tool across much of mathematics and computer science, at both the conceptual and practical levels. All too often, it also proves to be a stumbling block for people new to

> The ultimate trip into the twisted world of recursion is the legendary mind-bending book *Gödel, Escher, Bach: an Eternal Golden Braid* by Douglas Hofstadter.

programming. We won't spend a long time studying recursion in this course (unlike in CS 115 and CS 135, where recursion is everywhere). My main goal is to communicate some intuition for what recursion is and why it's so useful, particularly in the kinds of visual programming tasks we pursue. Of course, you will also practice writing recursive code in the lab and on the assignment.

Let's begin at an intuitive level, by using a drawing program called Recursive Drawing (recursivedrawing.com). This is a bare-bones tool lacking in many basic features, but there's one special thing that it does *very* well, as we'll see.



It doesn't take much experimentation to realize that there's an immediate connection between this simple drawing interface and the kind of hierarchical modelling we explored in the previous module. We can treat each of the shapes in the "library" (the left sidebar) as a kind of "function" in a hypothetical programming language. When you drag shape A from the library into a some new design B, you are effectively setting up a geometric context and calling the A function as part of writing a B function. For example, the composition above contains three cars, each of which relies on a previously defined "car" function. If we modify the underlying car, all three instances are immediately affected.

So far, that's just a demonstration of hierarchical modelling as in Module 05. But this piece of software has a remarkable superpower: you can drag a shape out of the library onto *itself*! What does that even mean? Well, try it: create a new shape and drag a circle or square onto it. Now drag another copy of the shape you're creating onto the main canvas. Try adjusting the position, scale and rotation of the main shape and of any copies that also appear on the canvas (it's safer to scale *down*, not up). Now step back from the computer and mediate on what you've seen. Can you tell yourself a convincing story that accounts for this behaviour? This tool demonstrates the essence of *recursion*.

If you're comfortable with the idea of recursion as embodied here, the next step is to ask how the same ideas might find their way into code. If each shape in the Recursive Drawing library is equivalent to a function, then a shape that incorporates a copy of itself ought to correspond to a function that calls itself. And that's exactly what we usually mean when we speak of programming a recursive function:

A *recursive function* is a function that calls itself.

(More generally, a recursive function might be part of a longer chain, e.g., `A()` calls `B()` and `B()` calls `A()`, or `A()` calls `B()` which calls `C()` which calls `A()`, etc. But we'll avoid these more convoluted forms of recursion in this course.

That seems simple enough to express in code. Let's try something like this as part of a longer Processing sketch.

```
void makeDrawing()
{
  ellipse( 0, 0, 150, 150 );

  pushMatrix();
  translate( 130, -20 );
  scale( 0.6 );
  makeDrawing();
  popMatrix();
}
```

This function certainly seems to have the right structure, but unfortunately it's fatally flawed. The problem is that there's nothing to tell the function when to *stop*. Processing will attempt to compose a drawing from an infinite sequence of ever smaller drawings, and this infinite regress will eventually consume all of some resource in the computer, crashing the sketch.

In practice, this program will crash very quickly because Processing permits a relatively small number (32) of nested calls to `pushMatrix()`.

We avoid infinite regress with some sort of stopping condition, usually called a *base case*. Every recursive function must have a base case, a way to execute the body of the function without ever making a recursive call. For the makeDrawing() function above, the easiest way to add a base case is to keep track of the *level* of the recursion: how deep are we in a nested sequence of recursive calls? Typically, we count down:

- The first time we call the recursive function we pass in the total number of levels we want to use.
- Every time we make a recursive call, we pass in the next smaller number of levels.
- The function checks if the current level is zero, and if so it does something trivial.

We might then arrive at code like this:

```
void makeDrawing( int levs )
{
  ellipse( 0, 0, 150, 150 );

  if( levs > 0 ) {
    pushMatrix();
    translate( 130, -20 );
    scale( 0.6 );
    makeDrawing( levs - 1 );
    popMatrix();
  }
}
```

Note that `levs` can't be a global variable, it must be an argument to the function. Every call to the function now operates "at level *n*" for some *n*; a level-*n* drawing is made out of an ellipse, combined with a level-(*n*-1) drawing. Think of it as a collapsed form of this much more verbose code:

```
void makeDrawing_0()
{
  ellipse( 0, 0, 150, 150 );
}

void makeDrawing_1()
{
  ellipse( 0, 0, 150, 150 );
  pushMatrix();
  translate( 130, -20 );
  scale( 0.6 );
  makeDrawing_0();
  popMatrix();
}

void makeDrawing_2()
{
  ellipse( 0, 0, 150, 150 );
  pushMatrix();
  translate( 130, -20 );
  scale( 0.6 );
  makeDrawing_1();
  popMatrix();
}

void makeDrawing_3()
{
  ellipse( 0, 0, 150, 150 );
  pushMatrix();
  translate( 130, -20 );
  scale( 0.6 );
  makeDrawing_2();
  popMatrix();
}

void makeDrawing_4()
{
  ellipse( 0, 0, 150, 150 );
  pushMatrix();
  translate( 130, -20 );
  scale( 0.6 );
  makeDrawing_3();
  popMatrix();
}

// ...and so on...
```

The idea that the level goes down by one in every recursive call represents an important principle: the recursive call must get a little bit closer to the base case (because if it doesn't, the program will never

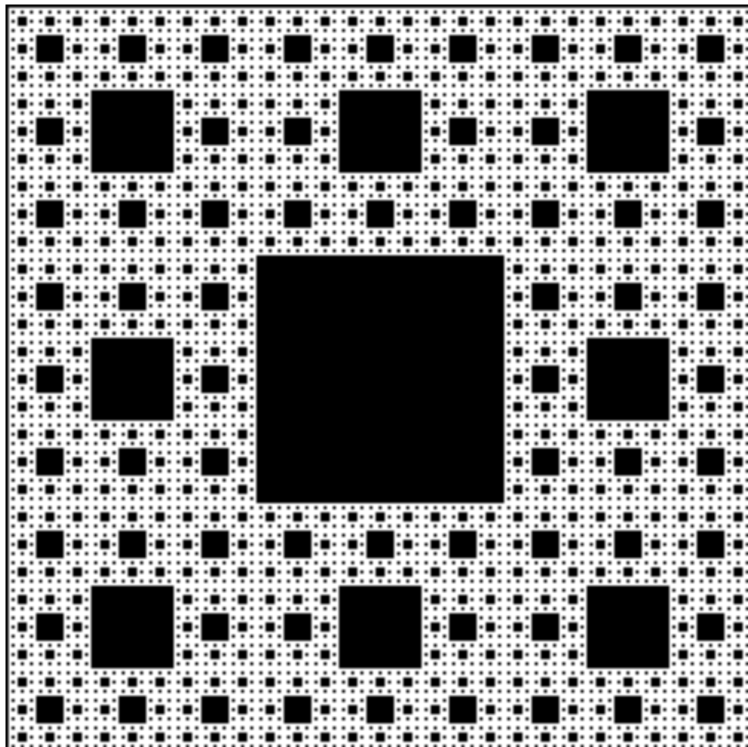
finish). Always make sure your recursive calls are "making progress". Another way to structure this program would be to keep track of how large the circles will be on the screen, and stop the recursion when they get too small.

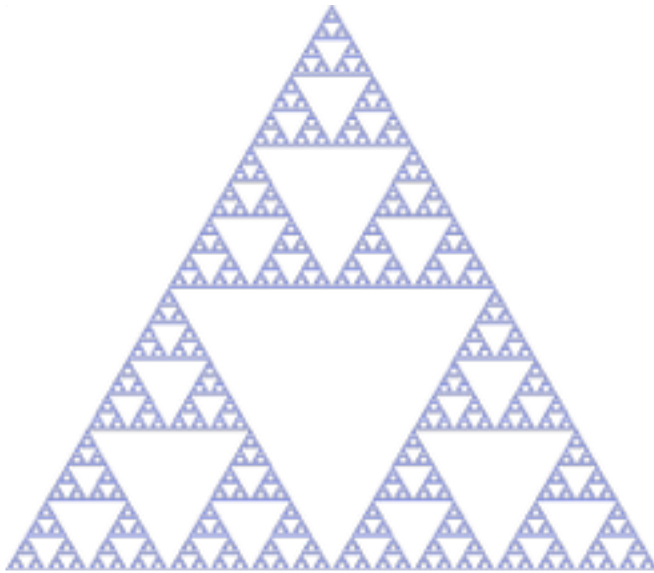
Putting together what we've learned, we arrive at these general guidelines for writing recursive functions:

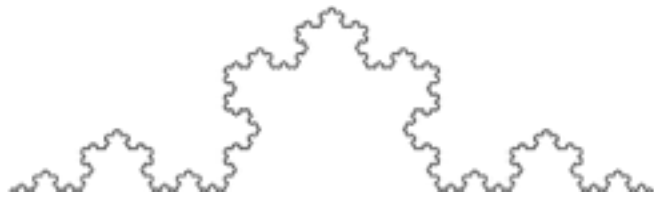
In a recursive function...

- **The function body will contain at least one call to the function itself.**
- **The recursive calls will be to "simpler" instances of the problem.**
- **There will be a base case in which no further recursion happens.**

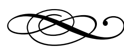
Computer scientists love to write code that draws self-similar structures like these fractals. Some simple examples are the Sierpinski Carpet, Sierpinski Triangle, and the Koch Curve. (Images below from Wikipedia.)







There are numerous other examples of abstract mathematical designs like these. They can sometimes be seen intruding into popular culture, though they tend to remain within the province of admirers of overt mathematical form. They might occasionally provide an interesting basis for more freeform design, though.



Randomness

Obviously we've already encountered randomness through the built-in `random()` function, and used it many times in this course. But in the context of this module, we should explore the nature of this randomness in a bit more detail, and also consider alternatives.

Let's begin with a classic example of random computer-based design, in the form of a short

program written on the Commodore 64 in the early 1980s:

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

Don't bother trying to run this program, unless you're using the C64 BASIC programming language.

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 do something similar to the Blizzard question in Assignment 05. We compute an array of random numbers up front, and refer back to those numbers every time we draw. (Exercise: do it!) But there's a better way, one that doesn't rely on storing any explicit random numbers. The built-in function `randomSeed()` takes an integer as input and resets Processing's random number generator based on that "seed". For any given seed, any sequence of calls to `random()` will return the same sequence of answers. So we can fix the sketch above simply by forcing the same seed at the start of `draw()`.

Example sketch: TenPrint

Why does this work? It's a mathematical fact that may surprise others. Most random number generators produce numbers that aren't "random" at all, merely

unpredictable. They're usually called *pseudorandom*. Pseudorandom numbers are perfectly fine as a source of chaos for art and design purposes, but it's very dangerous to assume 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.

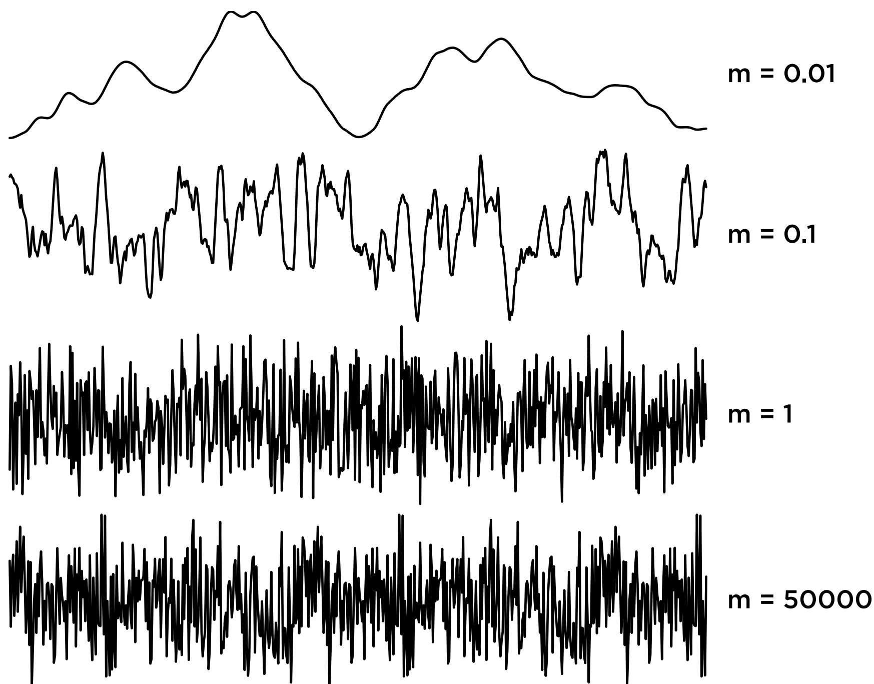
So far this sketch works fine, but what if we want to *scroll* the random maze? We can restart the sequence of random numbers, but we can't translate it or otherwise extend it to cover new grid cells as they enter the screen. What we really need is some way to conceptually "attach" random numbers to 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. That number never changes—it's permanently associated with the input number.

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 useful range in which you can use this function.

```
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 end by exploring 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.

```
float[] xs;
float[] ys;

boolean checkIntersection(
  float x1, float y1, float x2, float y2 )
{
  float d = dist( x1, y1, x2, y2 );
  return d <= 80;
}

boolean maybeAddCircle( float x, float y )
{
  for( int idx = 0; idx < xs.length; ++idx ) {
    if( checkIntersection(
        x, y, xs[idx], ys[idx] ) ) {
      return false;
    }
  }

  xs = append( xs, x );
  ys = append( ys, y );
  return true;
}

void setup()
{
  size( 500, 500 );
  stroke( 0 );
  colorMode( HSB, 100 );

  xs = new float[0];
  ys = new float[0];
}

void draw()
{
  float x = random( width );
  float y = random( height );
  if( maybeAddCircle( x, y ) ) {
    ellipse( x, y, 80, 80 );
  }
}
```

We can make a few additional extensions to this sketch (not shown here) to 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.

Example sketch: Foam

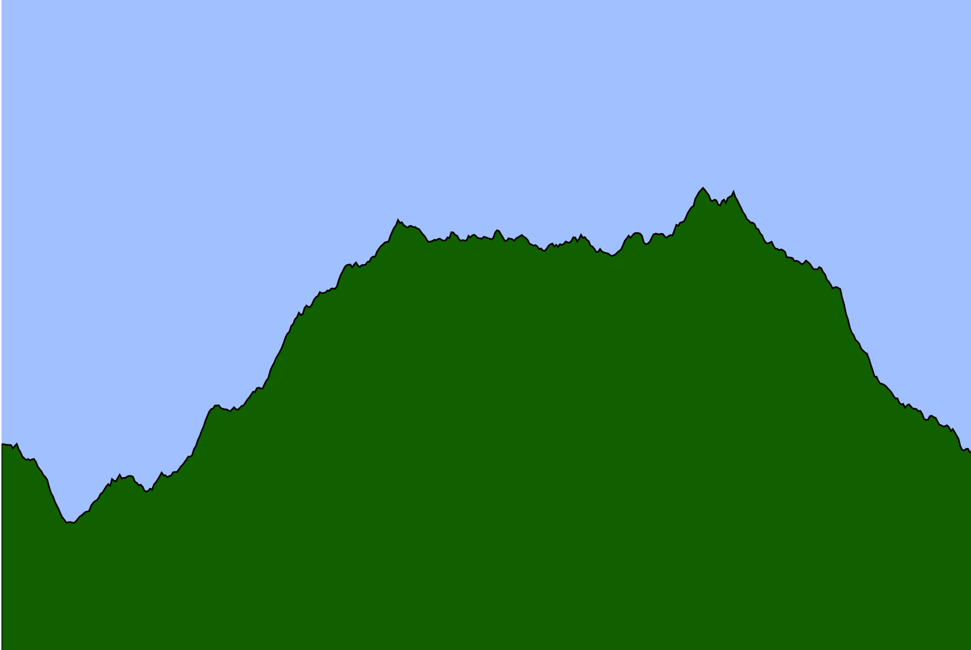
This sketch suggests a general framework for creating fractal-like structures. We place objects wherever they fit. If nothing fits, make the objects smaller. Repeat for as long as desired. This technique is explored in a paper by Dunham and Shier, and in some 2D and 3D examples by Paul Bourke. On the 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.

Example sketch: Mountains

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 recursive 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.