CS 106 Winter 2016
Craig S. Kaplan

Module 07
Recursion and fractals
Topics

- Recursion as an extension of hierarchical modelling
- Simple fractals

Readings

- *Learning Processing*, Section 13.11
- *Nature of Code*, Chapter 8

Introduction

One place where programming becomes a powerful tool for visual expression is in the creation of content that would be too difficult to draw by hand. You saw a bit of *generative design* in CS 105 as an example of this sort of phenomenon, and hopefully a few of the examples in CS 106 have demonstrated the utility of programming. In this module we'll look at recursion as a programming technique and its application in creating designs like fractals.

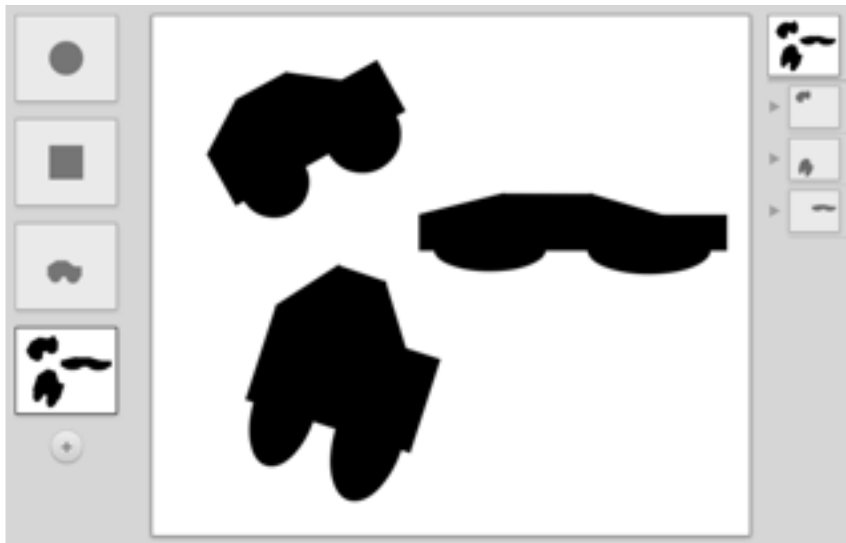
Recursion

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 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

> 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.

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 an online 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 (and more abstract) 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...
```

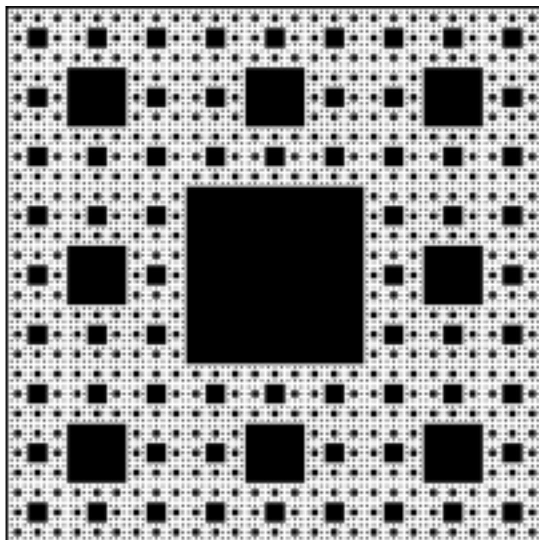
The idea that the level goes down by one in every recursive call represents an important principle: the recursive call must get a little bit closer to the base case (because if it doesn't, the program will never finish). Always make sure your recursive calls are "making progress". Another way to structure 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.

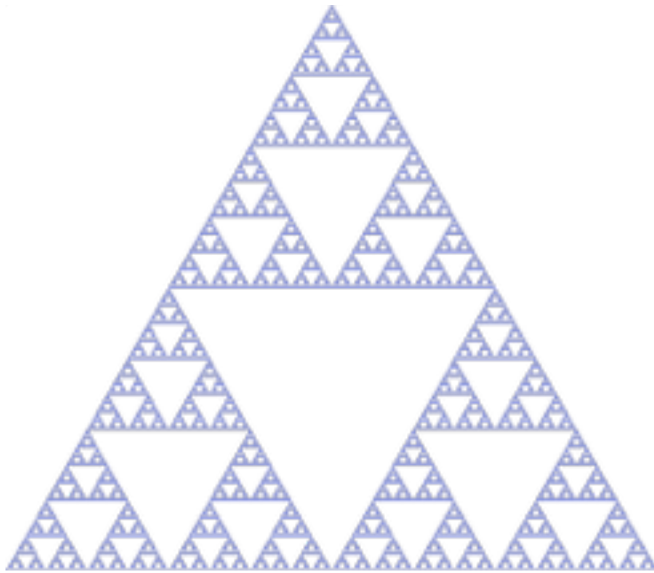
Putting together what we've learned, we arrive at these general guidelines for writing recursive functions:

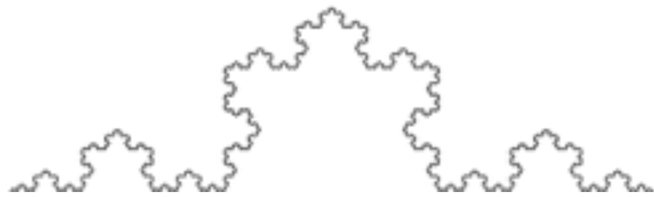
In a recursive function...

- **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.