CS 116x Winter 2015
Craig S. Kaplan

Assignment 06: Recursion and Randomness

Due date: Wednesday, 11 March, 12:00pm

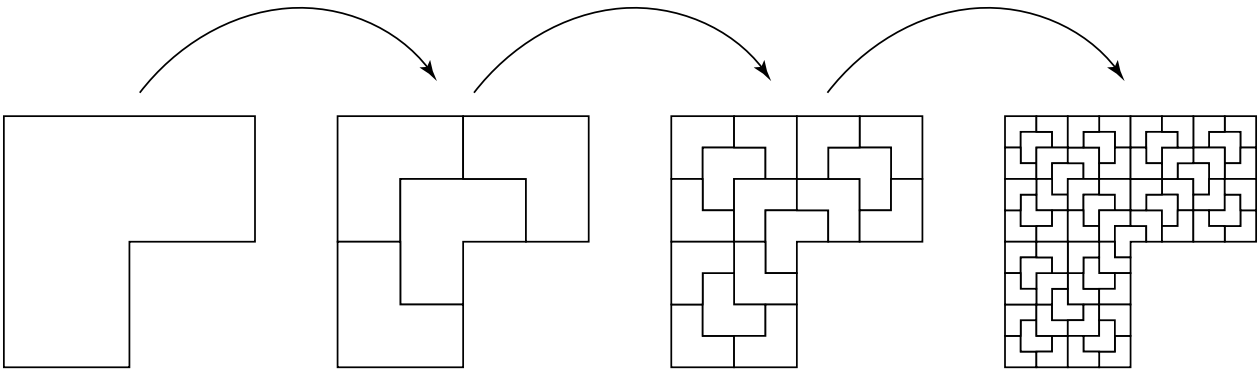
Question 1: Substitution tilings

In a substitution tiling, there is a fixed set of distinct tile shapes (a single shape in the examples in this assignment). There's also a rule that tells you how to replace each tile shape with a set of smaller copies of the same tiles. You can use these rules to lay out as many tiles as you want: start by drawing a single tile that's big enough to cover the region you want to fill, and apply the substitution rules repeatedly until you've got lots of small tiles. If you're interested, you can find many, many amazing examples of substitution tilings online at The Tiling Encyclopedia.

It turns out that (some of) these tilings can be drawn in a very natural way using the same kinds of techniques we used to draw common fractals like the Sierpiński Gasket: we write a recursive function. The base case of the function draws a single tile; the recursive case divides the tile into smaller copies of itself, each one represented via a recursive call.

Your goal of this assignment is to write a sketch that draws a substitution tiling. Two tilings are presented below. **You do not need to complete both sketches!** You can pick either one and just submit that sketch. If you wish, you can complete both and receive some bonus marks.

The Chair Tiling



A "chair" tile is a shape that looks like three squares glued together (in the picture above, it's drawn on its side to match Processing's coordinates). A chair can be subdivided into four chairs, each of which is exactly half the width and height of the original. This process can be continued to produce a tiling.

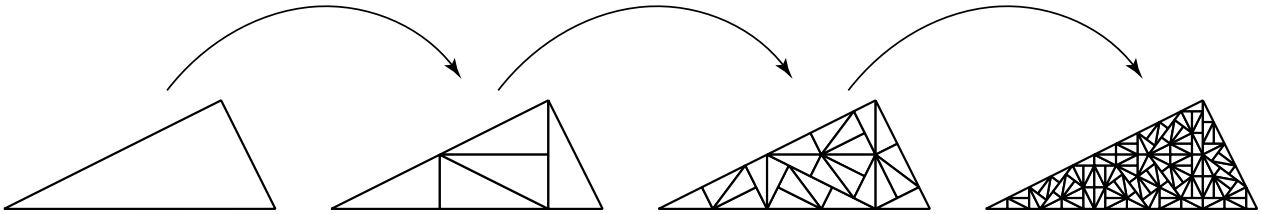
Using the starter code provided in `A06chair.zip`, write a single recursive function to draw the chair tiling elaborated to any desired number of levels. The tiles will fill up exactly three quarters of the sketch window. The provided `draw()` function translates and scales the world so that the initial chair is assumed to fit inside a unit (1×1) square. With that in mind, you can proceed as follows:

- First, write the base case. At Level 0, all you need to do is draw a single copy of the chair. Use the functions `beginShape()`, `vertex()`, and `endShape()`. Even if you leave the recursive case completely blank, you should already by able to test the base: set the number of recursive levels to zero and make sure you get a picture like the leftmost one above.

- Next, write the recursive case. For this tiling, follow the method used to draw the Sierpiński Gasket in conjunction with geometric context (see the `ContextGasket` sample sketch). Use four distinct nested geometric contexts. Within each context, include a single recursive call to the function to draw the tiling at the next smaller level. The only new bit is to figure out the right sequences of transformations (translations, rotations and scales) to move the four recursive chairs into position.

You can (and should) complete this question by writing a single recursive function and calling it from `draw()`. You don't need any global variables or other helpers.

The Pinwheel Tiling

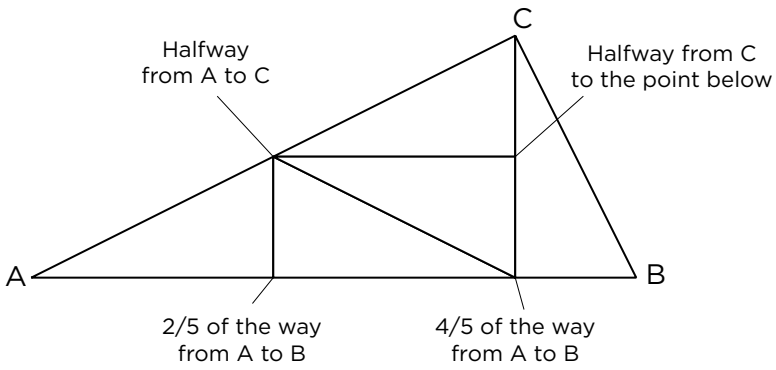


In the Pinwheel Tiling, a single triangle with very specific dimensions can be subdivided into five smaller copies of the same triangle. Again, this process can be repeated to create a tiling of the plane (with some very weird properties that I won't get into here).

Using the starter code provided in `A06pinwheel.zip`, write a single recursive function to draw the Pinwheel tiling elaborated to any desired number of levels. The sketch window will contain a single large triangular arrangement of small tiles, as in the drawings above. For this tiling, don't use geometric context. Instead, use an approach similar to the explicit manipulation of triangle corners as in the `CornerGasket` sample sketch. The recursive function will consume seven arguments: the current recursive level, and the x and y coordinates of the three corners of the current triangle. The base case just draws the triangle. The recursive case computes the necessary additional vertices of the smaller subdivided triangles and makes five recursive calls. You'll need to compute the coordinates correctly and pass them to the recursive calls in the right order to make sure all small triangles are identical to each other.

As before, write a single recursive function and call it from `draw()`.

You'll need to compute the coordinates of four new points relative to each recursive triangle, like the three points in the Gasket example. This diagram explains where those points live:



You can use the built-in `lerp()` function to compute the coordinates from the x and y coordinates for A, B and C, which will be passed in as arguments to the function. To do so, use `lerp()` separately on the x and y values. Don't assume that triangle ABC is oriented as in the diagram—triangles will point every which way in the final tiling.

What to submit: On LEARN, you should submit a sketch entitled either `A06chair` or `A06pinwheel`, as appropriate. Submit the entire sketch folder. If you wish, you can submit both, but you'll only receive bonus marks if they're both completely correct—please don't submit two half-broken sketches hoping that they'll add up to full marks. It's better to pick one approach (the one you find more intuitive) and focus on making it work perfectly.

Question 2: Archipelago

In this exercise you will use the built-in `noise()` function to construct something that looks like a map of an imaginary archipelago. Because of the coherence of `noise()`, it will be possible to pan around and explore a conceptually infinite expanse of islands. A correct solution must display a map

like the one above, and permit panning in the style of Google Maps or the sketches we experimented with during Module 06 lectures.

At some level, this sketch will resemble the `TenPrintScroll` and `Truchet` examples that were used as part of Module 06: you will iterate over every cell in a grid and use `noise()` to generate "random" numbers. In this case, the cells are individual pixels: you need to iterate over every pixel in the sketch window. When the noise value is less than some threshold, you assign that pixel to land and colour it green; when it's greater, you assign the pixel to water and colour it blue.

Manipulating the individual pixels in a sketch window follows a standard programming idiom, based on calling `loadPixels()`, writing to the sketch's `pixels[]` array, and calling `updatePixels()`. Examples of doing this can be found in Processing's online documentation for these functions.

With that in mind, you can proceed as follows:

1. Create a new sketch with a 640×480 window.
2. Create a `draw()` function. The function should use the idiom mentioned above to write to every pixel in the sketch's `pixels[]` array. Use nested loops over y and x, so that you can call `noise()` with the pixel's two-dimensional coordinates. As a test, write a colour to every pixel that shows that you're obtaining noise values correctly—for example, map noise to hue or brightness.
3. Now define a constant scaling factor that's multiplied into x and y when computing noise values, as in done in `TenPrintNoise`. This scaling factor will allow you to decide on the granularity of your archipelago (i.e., lots of tiny islands versus a few larger ones). Choose an aesthetically pleasing value; we won't vary scale dynamically in this sketch.
4. Turn the noise visualization into an imaginary map. Define a trehshold value between 0 and 1. Check each noise value against this threshold; assign smaller values to land and larger values to water. Choose appropriate colours for each, and choose a threshold value that gives the appearance of islands in a sea.
5. Finally, add panning. You can easily borrow ideas from sketches shown in class. Define two global translation variables that are used to offset coordinates passed to `noise()`, and write a short `mouseDragged()` function that changes those global variables. That should be all you need.

A complete implementation can be written in about 35 lines of code, not including comments. That's not a required target, just a suggestion that the solution need not be very complicated. Note that you don't any geometric context functions to complete this sketch.

Keep in mind that the `noise()` function is far from perfect. Don't be surprised if, some of the time, your archipelago has some symmetries in it, as if half the islands are a mirror reflection of the other half. You don't have to try to fix that, though it may be possible to use the `noiseDetail()` function to do so.

If you're interested, there are many possible ways to enhance this simple sketch.

- Add cities as dots on the map. This is fairly tricky. You'll want to make sure the cities are on land, relatively sparse, and not clumped too closely together. It might be possible to use further layers of noise to decide on city locations, in which case the cities might even pan around coherently with the map.
- Add other land features. It may be possible to specialize land colours to add white for polar regions, beige for desert, etc.
- Add zooming. Adding ControlP5 buttons for zooming in and out is easy; making the zoom actually work relative to the centre of the sketch window is probably harder. Making the hypothetical city dots work consistently at multiple zoom levels is probably very difficult.
- Er, simulate global warming by animating the cutoff threshold between land and water.

What to submit: On LEARN, you should submit a sketch entitled `A06archipelago`. Submit the entire sketch folder.