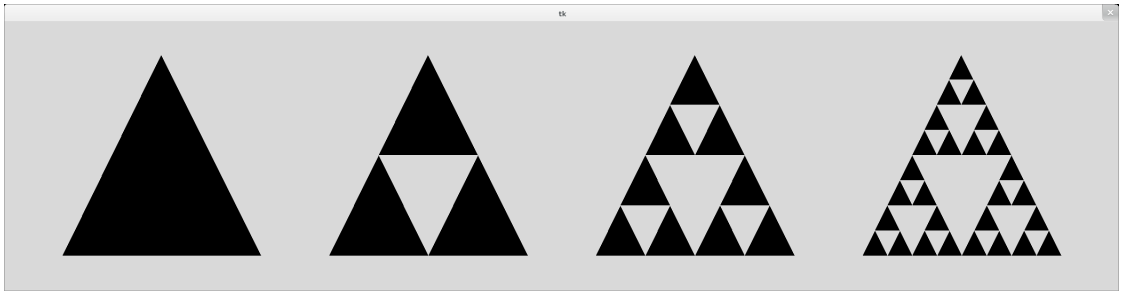
15-110: Principles of Computing

Assignment #4: Re-Re-Recursive-sive-sive Fractals (Due July 26, 2011)

1. Sierpinski Triangles

A fun application of recursion is in generating **fractals**. Fractals are shapes or images that are generated by repeated application of an equation or a set of rules on said shape or image. The fascinating thing about fractals is their abundance in nature and the wide variety of remarkable images and patterns one could generate with only a simple set of rules. In this assignment, you will be drawing two-dimensional fractals.



Sierpinski's Triangle (up to depth 3)

You all have probably seen Sierpinski's Triangle before. It is an equilateral triangle that is divided into four (4) identical regions where the middle region is removed. Then the pattern is repeated for each of the three (3) smaller equilateral triangles at the next depth level, and then their subsections in the depth after that (recursion!), and so on per depth level. In this section, you will be implementing two different recursive methods of drawing Sierpinski's Triangle: the *constructive* method, and the *chaos game* method.

(a) Constructive Method

In this section, you will be *constructing* the fractal, or creating the triangles that make up the fractal (as opposed to deleting the middle triangles). We will start out with definitions.

(a.1) Midpoint

A two-dimensional `coordinate` is defined to be a tuple (2-ple) of floats.

```
>>> print "Example point: ", (3.0, 5.0)
Example point: (3.0, 5.0)
```

The *midpoint* of two (2) two-dimensional coordinates a and b is defined to be:

$$\begin{aligned} a &= (a_x, a_y) \\ b &= (b_x, b_y) \\ midpoint(a, b) &= \left(\frac{a_x + b_x}{2}, \frac{a_y + b_y}{2} \right) \end{aligned}$$

With this in mind, you will implement the following function:

```
def midpoint(pointA, pointB):
```

where `midpoint(pointA, pointB)` will return the midpoint of the two (2) two-dimensional coordinates a and b. For the rest of this document, M_{ab} will represent the midpoint of the two points a and b.

(a.2) Drawing the Triangles

A `triangle` is defined to be a triple (3-ple) of points. So, in Python, they will look like:

```
>>> print "Example triangle: ", ((1.0, 2.0), (0.0, 0.0), (2.0, 0.0))
Example triangle: ((1.0, 2.0), (0.0, 0.0), (2.0, 0.0))
```

Given this definition of a triangle, write the function

```
def drawTriangle(triangle):
```

The function `drawTriangle` takes in a triangle named `triangle` and draws the triangle on the global canvas widget. As described in the Background section, you will access the canvas widget through the global widget dictionary, and use the `create_polygon` function to draw the triangle onto the globally available canvas. Please see **Section 2** for more information on how to implement this function.
This function will not have any meaningful return values.

(a.3) Constructing the Fractal

In the constructive method, you will be sub-dividing an equilateral triangle into four (4) equal sub-triangles. And then you will sub-divide those triangles into four (4) sub-sections again, and again, and again, until you hit the maximum depth limit. The fractal can go to an infinite depth but after a few levels, the triangles become smaller than the pixels on your screen.

For our Sierpinski Triangle fractal, we will only care about three (3) of the four (4) sub-triangles. When the recursion depth is increased by one (1), one would sub-divide every triangle into three (3) smaller triangles.

With this definition of a triangle and `drawTriangles` and the recursive method of dividing the Sierpinski triangle, implement the following **recursive** function:

```
def sierpinski(currentDepth, triangle):
```

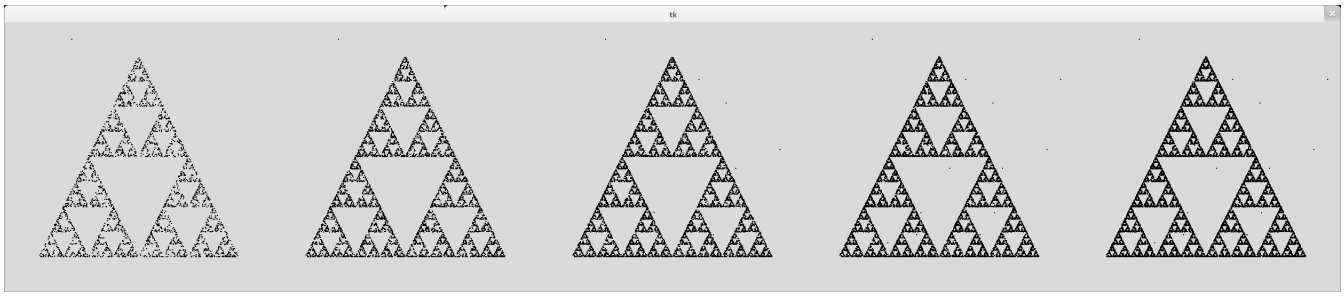
The function starts with one triangle given as the `triangle` argument. From that triangle, it will divide the triangle and recurse on the smaller triangles up to the integer depth `currentDepth`. When `currentDepth` is zero (0), then the function should draw the given triangle and return. When `sierpinski` is called with depth one (1), it would draw exactly three (3) triangles, and nine (9) triangles at depth two (2).
This function will not have any meaningful return values.

WARNING: You should not have any loops in your code for `sierpinski`. If there are any `for` or `while` loops or global variables/state in your code for this function, you will get zero (0) points for this function. You have been warned.

(b) Chaos Game Method The second way of rendering a Sierpinski Triangle fractal is by something known as the **chaos game** method. The algorithm is as follows:

Given three vertices of the initial triangle a, b, and c:

1. Start with a random point p
2. Pick a random vertex from a, b, or c
3. Plot the midpoint m between p and the chosen vertex
4. Go back to step 2 where m is the new p



A progression of the chaos game method at 3000 points per iteration.

(b.1) Chaos

Implement the following functions:

```
def chaos(points, start): # Recursive function
def drawPixel(point):
```

Like in the `sierpinski` function, `points` is an int that represents the number of points that should be drawn. Unlike the `sierpinski` function though, the `start` parameter is not a triangle; it is a single coordinate, the coordinate p in step 1 from the given algorithm. Your job is to write a **recursive** function that executes one cycle of the algorithm and plots the *midpoint* computed in Step 3 using the function `drawPixel` before making the recursive call. There should be no loops in the code of this function.

The function `drawPixel` is like the `drawTriangle` function except that it draws a single pixel onto the globally available canvas at the coordinates given to it as the `point` argument. A **pixel** by definition is the smallest possible discrete unit that can be drawn on a computer screen. For the purposes of this lab, you will draw every pixel as a *rectangle* of length 1. Please see **Section 2** for more information on how to implement this function.

Protip: You can access a random vertex of the triangle from the staff-provided function, `randomVertex()`. When called, it will return exactly one (1) two-dimensional coordinate.

2. Implementing the GUI

(a) Background information

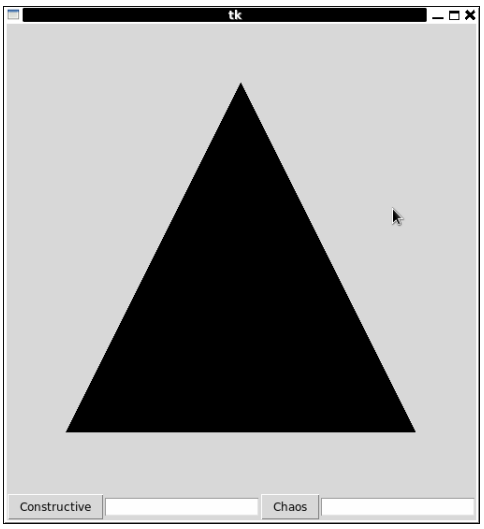
In this assignment, we will be using a new type of design structure for our Tkinter-related code, aka. our GUI. There is a global dictionary available called `widgetDict`. This dictionary is where you will be storing all of your Tkinter widgets, like your buttons and canvas. In the following function,

```
def run():
```

you will initialize `widgetDict`, your root and widgets within this function, and then store all of these widgets inside `widgetDict`. This way, you will be able to access the canvas and widgets from the functions that need access to the canvas and widgets such `drawTriangle` and `drawPixel`. In case you cannot remember, you can store and access a widget `foo` into `widgetDict` with the name "bar" with the statement:

```
widgetDict["bar"] = foo
```

Your `run` function is where you will be doing all of the setup of the GUI. You will initialize the Tk root and canvas, and you will create two buttons and two text entry boxes.



Example screenshot of the GUI

The provided skeleton file starts the `run` function with a call to `initTriangle`. It is a function that will initialize the original triangle based on the window size and the desired triangle size, and make the coordinates of the three vertices available as global variables `ax`, `ay`, `bx`, `by`, `cx`, and `cy`.

(b) Wrapper Functions

Write the following two functions:

```
def sierpinskiWrapper():
def chaosWrapper():
```

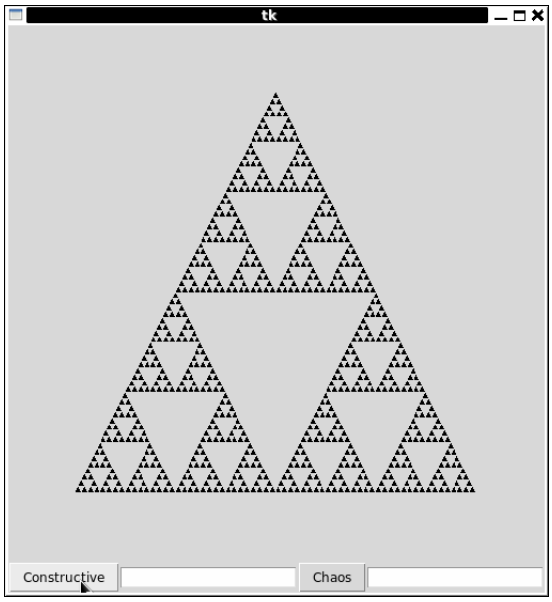
When the user clicks on a button, one of these two functions should be called. The "Constructive" button should call `sierpinskiWrapper` and the "Chaos" button should call `chaosWrapper`. These two functions should wipe the canvas of all triangles and pixels (achieved with the canvas widget's `delete(ALL)` command), and then get the input from the entry box directly to the right of the button that was pressed.

If this input *string* is empty, then the wrapper functions should just wipe the canvas and end. Otherwise, the input string should be checked to see if it is an integer and converted to an int, and then checked to see if it is within acceptable parameters.

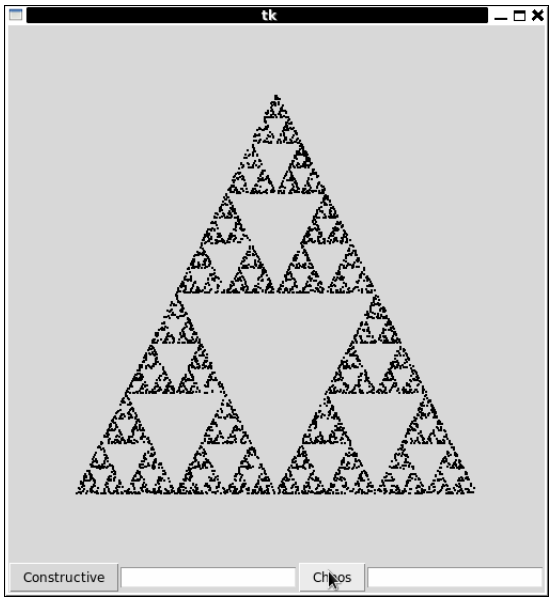
- The program should draw a Sierpinski triangle using the constructive method if and only if the input is between 0 and 9 (inclusive), and end otherwise.
- The program should draw a Sierpinski triangle using the Chaos game method if and only if the input is between 1 and the value of the global constant `maxDepth`, and end otherwise.

Should there be more than one word in the entry box (e.g. "5 golden rings"), the input is considered valid if the first word (e.g. "5") is a valid integer that fits the restrictions given above. Upon receiving bad input, your wrapper functions should only wipe the canvas and entry box, and then return silently without errors.

Also, whenever a button is pressed, the relevant entry box's text should be erased.



Example screenshot of constructive method



Example screenshot of chaos game method

Note how there is no text in the entry boxes.

3. Grading

Your grade in this assignment will be distributed as follows:

- `midpoint` (5%)
- `drawTriangle` (5%)
- `sierpinski` (20%)
- `drawPixel` (5%)
- `chaos` (20%)
- `sierpinskiWrapper` (10%)
- `chaosWrapper` (10%)
- `run` (15%)
- Style (10%)

Your code will be graded on correctness, correct behaviour of the GUI, and robustness (ability to not crash on malformed input into entry boxes). Individual functions will receive full points if they are correct and follow the specifications.

The Tk window should not launch when your Python source is loaded on the command line with the command `python -i lab5.py` on a machine on Andrew Linux (cluster computers, unix.andrew shells, etc.), and should launch only when the `run()` command is issued in the Python shell. It should also launch after the Tk window was closed and the `run()` command is issued again in the same Python shell (and again, ad infinitum).

Upon receiving bad input, your code should not throw an exception or crash or fail. Each crash caused by our test suite will result in a deduction of correctness points.

In the case of infinite loops (recursive or otherwise), and other such errors in your code, the functions responsible will lose correctness points for each and every error evaluated on a case-by-case basis by the TAs. Submissions are expected to compile without error and failure to do so will result in loss of full style points at the very least along with the expected correctness deductions.

This point distribution is not guaranteed to be final and will be subject to change up to the point in time when they are graded. Any changes made to the rubric later in time will be reflected in this specification so check back often.

(b) Testing

The skeleton source file distributed to students does not contain any tests. Student written tests will not contribute to the grade whatsoever other than the side-effect of well-written tests leading to well-written code but students are recommended to start writing their own tests and to try to reason about their code to determine possible edge cases that need to be tested.

(c) Style

Student-written code will also be graded on style according to the style guidelines provided during lectures and recitations. Well documented code with meaningful variable names, proper indenting, and clean design are some of the factors that lead to good coding style. One-letter or otherwise poor variable names, lack of comments and documentation, import statements within functions or other places that is not the top of the source file, and unnecessary global variable usage are among factors that will lead to loss of style points.

A practice that will not actually affect the style portion of the grade but is highly recommended is keeping lines of code to within 80 characters per line (or 120, or some similarly reasonable number). The goal of learning good coding style is to be able to write good, clean code that is not messy and can be easily read and understood by another peer of a different skill level as yourself. A large persistent problem that plagues the industry even today is the existence of extremely large codebases whose size easily grows past thousands or millions of lines of code that is written poorly and hard to understand and maintain (and fix). It is the belief of the course staff that good coding style and discipline learned early on in a programmer's career can go a long way.

(d) Helper Functions + Globals

- `windowSize` is the size constant for your Tk window. You may change it when testing your code for different results and images.
- `triSize` is the *height* of the triangle. You may change it when testing your code for different results.
- `maxDepth` is the maximum recursion depth that your program can attempt before getting a stack overflow.
- `initTriangle` is a function that computes the initial triangle's vertices' xy-coordinates and saves them as global variables `ax`, `ay`, `bx`, `by`, `cx`, `cy`.
- `randomVertex` is a function that returns a random vertex of either `(ax, ay)`, `(bx, by)`, or `(cx, cy)`

(Extra) Cool Things To Try.

- Try modifying your Chaos game button to allow you to not delete the already-drawn chaos game method Sierpinski triangle on the canvas!
- Try modifying your program to use more than one colour in some sort of pattern while drawing triangles!
- Sierpinski has other fractals named after him. Try to write a fractal generator (in a separate file) for Sierpinski's Carpet or Sierpinski's Hexagon (or pentagons)!
- There are also other simple and interesting fractals such as the T-Square and Mandelbrot Set fractals!