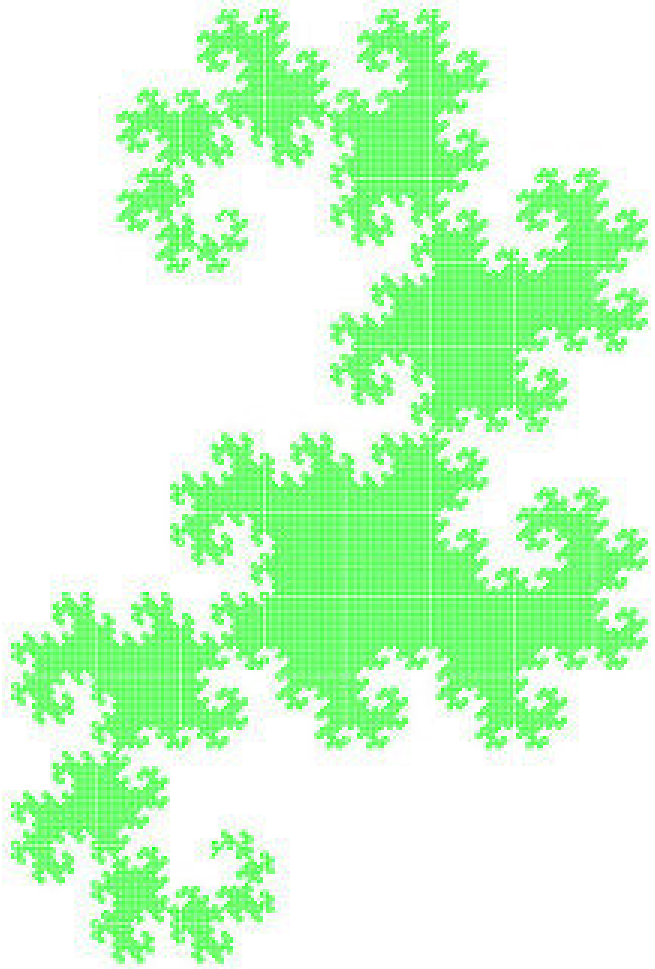
Lindenmayer Systems - A Python Adventure

BY ANDREAS WILHELM (HTTP://WWW.BORNAGEEK.COM) MAY 29, 2013



In my last article about Langton's Ant and Monty's Python (/general/development/2013/05/24/langtons-ant-and-montys-python.html), I talked about how to simulate Langton's Ant using Python's Turtle library

(http://docs.python.org/3.0/library/turtle.html). The turtle library reminded me of another interesting application for such a library, that I got to know in a lecture about formal systems, the Lindenmayer Systems.

Introduction

A Lindenmayer system (or L-system) is a formal grammar, which consists of an Alphabet *A*, a set of projection rules *P*, a starting axiom *S* and a set of terminals that could be translated into geometric structures. The basic idea of Lindenmayer systems is the successively rewriting of a string by replacing each nonterminal (right side of projection rules) with a new substring (left side of projection rules), which could consist of any combination of terminals and non-terminals. This refinement process makes it possible to generate very complex geometrical objects by applying simple rules to the starting axiom.

The basic assumptions for this rewriting systems have been laid by Noam Chomsky (http://en.wikipedia.org/wiki/Noam_Chomsky) with his work on formal grammars in 1957. Based on his research results Aristid Lindenmayer (http://en.wikipedia.org/wiki/Aristid_Lindenmayer)

introduced a new type of such rewriting systems in 1968, called Lindenmayer systems. The new aspect of this procedure was that the projection rules are applied in parallel, whereas Chomsky's method applies the rules sequentially. This new methology makes it possible to simulate cell divisions in cellular organisms, so it is now possible to simulate the growth of plants.

Alongside the simulation of plant growth, the modelling of fractals is an interesting application of Lindenmayer systems. A fractal is a mathematical construct of self-similar patterns. That means that the pattern is repeated in a smaller scale within itself. Popular examples of such fractals are Mandelbrot and Julia sets, Koch's Snowflake, the Hilbert Curve, the Dragon Curve, the Peano Curve, the Sierpinski Curve and the Sierpinski Triangle.

Many of this fractals could be approximated as a sequence of line segments. At this point, the Lindenmayer systems come into play, as the terminals of the underlying grammar are translated to a drawing operation.

A first example

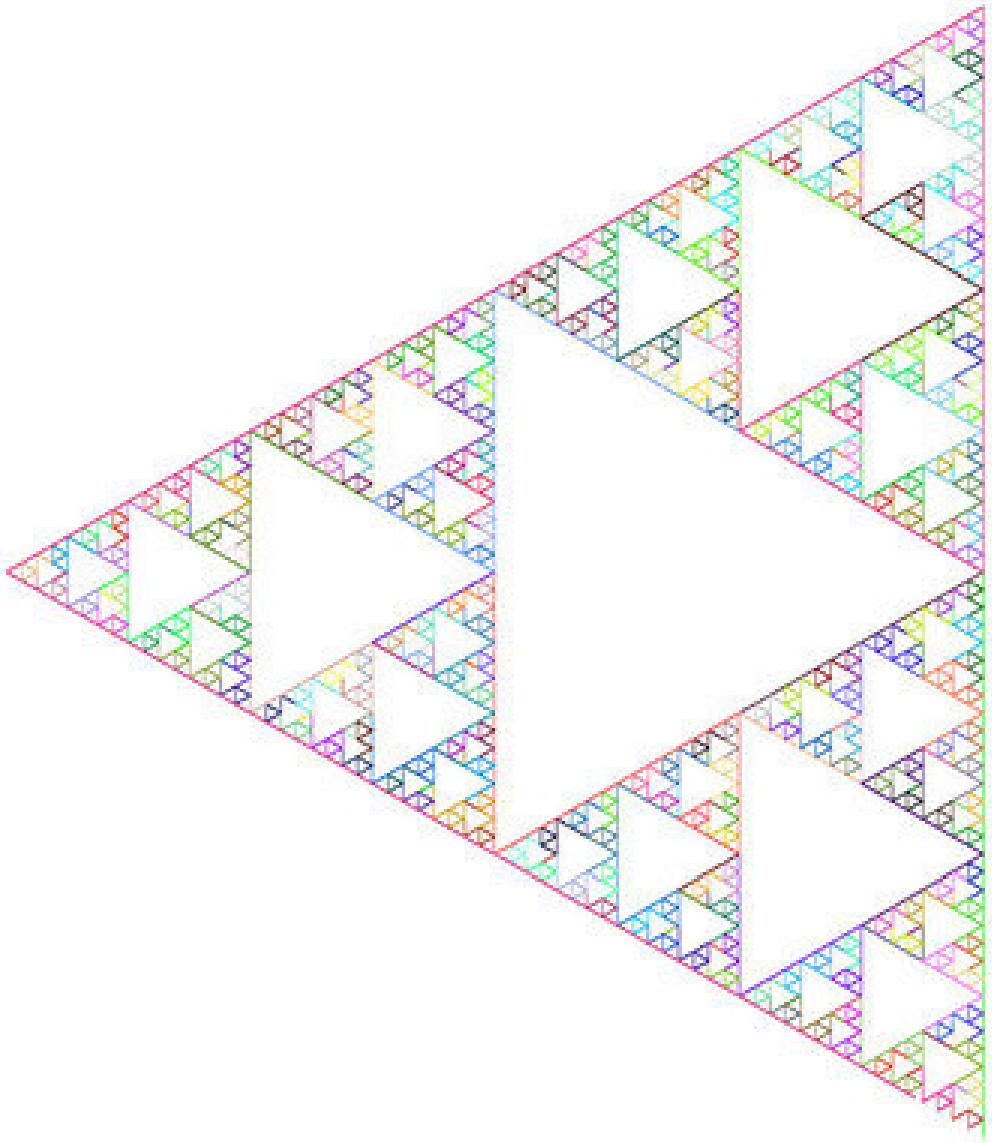
To get a better understanding of how this works, lets take a look at the grammar of the Sierpinksi Triangle, which is defined by the following L-system grammar:

A{'-', '+', 'F', 'X'}
S'FXF--FF--FF'
P{'X' -> '--FXF++FXF++FXF--', 'F' -> 'FF'}

In order to generate the Sierpinksi Triangle, the projection rules *P* are iterativly applied to the given Axiom *A*. That means that we replace all instances of ‘X’ within ‘FXF–FF–FF’ with ‘–FXF++FXF++FXF–’ and all instances of ‘F’ with ‘FF’, which will refine the pattern so that you could then see the first level of self-similar subpatterns. Below you see the resulting strings after applying the projection rules two times:

IFXF--FF--FF
IIFF--FXF++FXF++FXF--FF--FFFF--FFFF
IIIFFFF--FF--FXF++FXF++FXF--FF++FF--FXF++FXF++FXF--FF++FF--FXF++FXF++FXF--FF--FFFF--FFFFFFFF--FFFFFFFF

Geometric interpretation



You see that the length of the string is growing exponentially with each iteration, which means that the complexity of our fractal increases very fast. That is really great, but it is currently just a string. Where is the triangle? Now the turtle library comes into play. The strings produced by an Lindenmayer system could be interpreted as the movement of a turtle, which draws the fractal while moving.

This kind of graphical interpretation using a turtle was introduced by Przemyslaw Prusinkiewicz (http://en.wikipedia.org/wiki/Przemys%C5%82aw_Prusinkiewicz) in 1989. It could be thought of as a finite state machine, where the turtle is a triple *(x, y, a)* that contains the current position *(x,y)* and the current viewing direction as angle *a*. The step size *s* and the rotation angle *a* are fixed in this scenario. Given *a* and *s* the output produced by iterativly applying the projection rules could be processed symbol by symbol. The turtle then responses to the commands encoded with this chars:

FMove one step forward while drawing
fMove one step forward without drawing
ZMove a half step forward while drawing
zMove a half step forward without drawing

+Rotate clockwise by fixed angle *a*

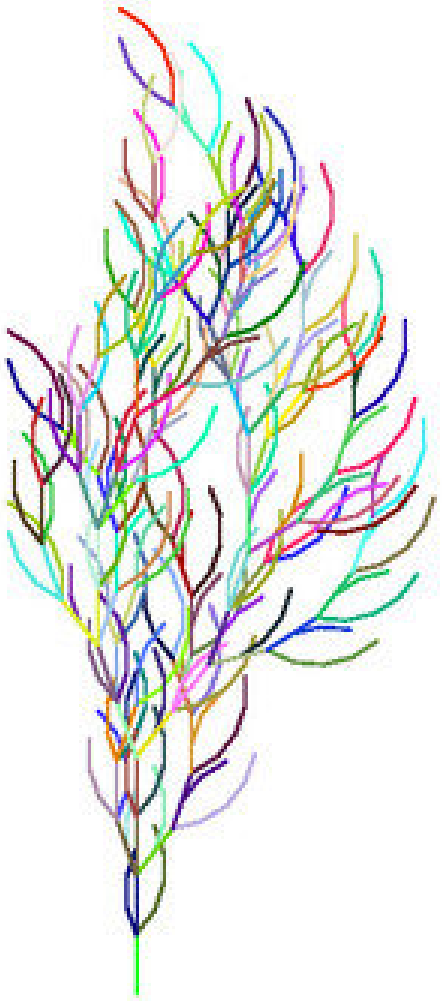
-Rotate counterclockwise by fixed angle *a*

+Rotate randomly clockwise or counterclockwise by fixed angle *a*

cChoose a random pen color

All non-terminals are ignored by the turtle. The above list of terminals enables us to create a geometrical interpretation of the result string generated by the Lindenmayer system for the Sierpinski triangle, which you see above. In this case I used an angle of 60 degrees and a step size of 1.

Branching Lindenmayer Systems



Using the list of terminals above you are able to draw simple fractals that consist just of a single line. To be able to create and simulate even more complex structures, Lindenmayer introduced a branching notation that

could be used to manage different turtle states on a stack. The idea is very simple. If the result string generated by the Lindenmayer system contains an opening bracket (‘[’), the current orientation and position as well as the current pen color are pushed to the stack. If later on a closing bracket (‘]’) is read, the last state of the turtle is recovered. That means that the turtle jumps back to its old position and rotates to the old direction, while the pen color changes to the old one.

This branching mechanism is oftentimes used when modelling plants, like grass, trees or algae using Lindenmayer systems. An example of such a system and the resulting image after 3 iterations, based on a line length of 2 and an angle of 17, you could see below:

A{'-', '+', 'F', 'X', '[', ']', 'c'}
S'F'
P{'F' -> 'FF +c[+F -F -F -F]-c[-F +F +F]'}

Conditional projection rules

To increase the copmplexity even more, you could add some conditions to the projection rules. That means that there could be different rules for one non-terminal, which are chosen based on the context of the non-terminal. The notation to represent this is very simple. If you would like to say that an ‘X’ is only mapped to ‘F-F’ if it is preceded by ‘FF’, you would write ‘FF<x’ -=””> ‘F-F’. The rule for the right-hand side of the non-terminal ‘X’ looks the same. So you would write ‘X>FF’, if the ‘X’ should be followed by ‘FF’ and you would write ‘FFFF', if the 'X' should be covered by 'FF' on the right and the left side. This way you could define arbitrary complex grammars, like the one below:

__A__{'-', '+', 'F', 'X', '[', ']'}
__S__'FX'
__P__{'F' -> 'FF', 'X>[' -> '+F+FF', ']<x' -=""> 'F-F', 'X' -> '[+F-FX]', '-F<x' -=""> 'FXFFXF'}</x'></x'>

This example also shows how to use a normal rule as fallback, so if no other rule matches this rule could be used.

I do not have a working example for such rules yet, but you could imagine that there are very interesting applications for this notation. Consider for example the modelling of plants. Right now you are just able to create plants that consists of branches. Sure you could model some leaves and blossoms, but they will be placed everywhere, also on the trunk. Cherry-trees in full blossom do not have blossoms on the trunk. To avoid this behavior you could use conditional rules, so the blossoms are just added under special conditions.

Implementation

As I previously mentioned I implemented Lindenmayer systems using Python and its turtle library. However, I did not like to hard code the axiom, the rules, the step size, the rotation angle and all the other stuff for every single grammar. So I decided to write a small parser that parses a given *.txt file that contains all settings and a grammar needed to create an image from a given Lindenmayer system. Therefore I used Python’s pyparsing library (http://pyparsing.wikispaces.com/), which I recently compared to some other lexical analysis methods in my article

Lexical Analysis using Python (/general/development/2013/05/10/lexical-analysis-using-python.html). I was very impressed because it was so simple.

In addition to some basic settings like the window size, the drawing offset, the line thickness and the background color, the *.txt file holds the definition of the grammar that implements the Lindenmayer system. This grammar could include the following terminals:

FMove forward a step - draws a line
fMove forward a step - draws no line
ZMove forward a half step - draws a line
zMove forward a half step - draws no line
xRandom turn left or right
+Turn left
-Turn righta
[and]Push and pop the current state used for branches
< and >Conditional mappings
(and) Terminal attributes like color, line width or length
#[0-9a-f]{6}Color definition
'[0-360]Angle definition

![:num:]Line width as floating factor or as pixel length
|[:num:]Line length as floating factor or as pixel length

Some of the terminals listed above are parsed but not yet handled by my implementation. This means, that a grammar that contains this terminals will work, but not utilize them. An example of such terminals are “(“ and “)”, which could be used to add parameters to a terminal. This will allow you to define a custom angle, color or line thickness, to enhance your models.

Below you find a working example of an Lindenmayer *.txt file, that includes all possible parameters and the grammar used to generate Sierpinski’s Triangle:

```
# This is a simple lindenmeyer project file
# Author: Andreas Wilhelm
# Project: Sierpinski's Triangle

# Image positioning and dimensions
Dimensions : 1920, 1080
Position : 200, -400

# Number of iterations
Iterations : 6

# Default angle
Angle : '60 # Default line length Linelength : 7 # Default line wi
```

In order to execute the script and process this Lindenmayer system, you just have to execute the following command within a terminal:

```
python lindenmayer.py -i sierpinskisTriangle.txt
```

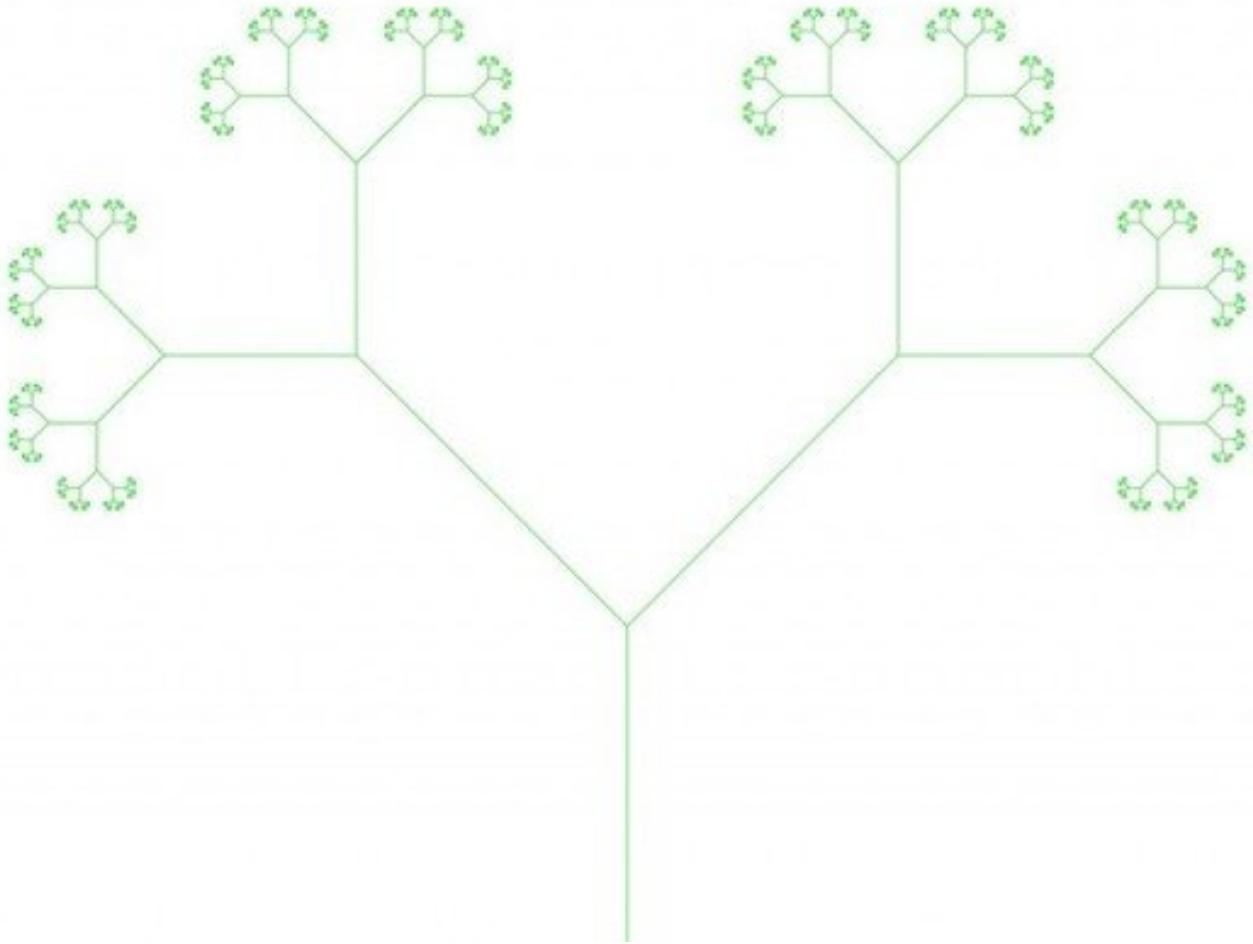
If you would like to save the resulting image as postscript file, you could use the *-o* parameter:

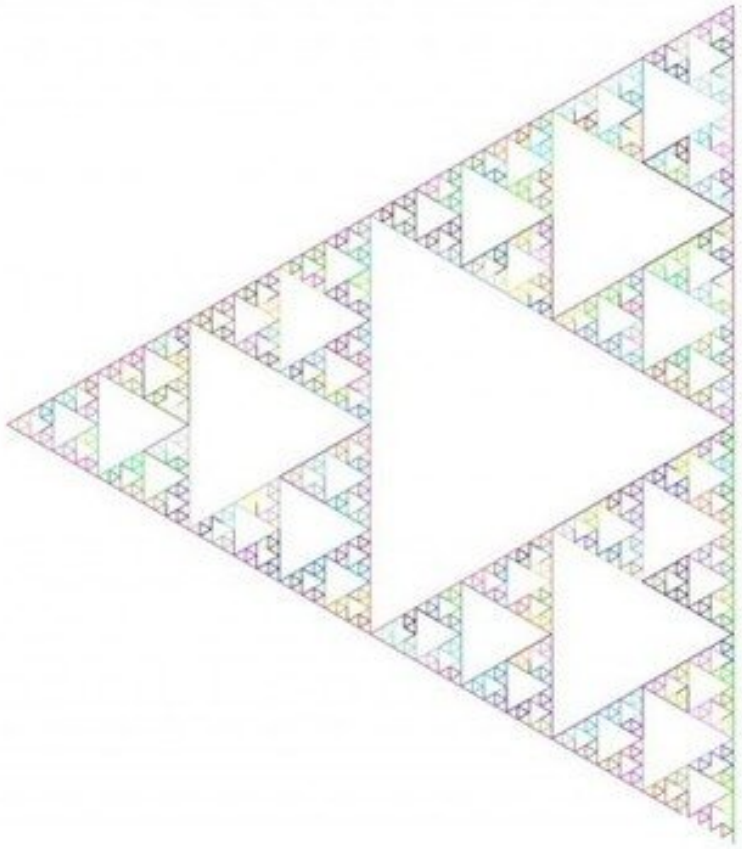
```
python lindenmayer.py -i sierpinskisTriangle.txt -o sierpinskisTri
```

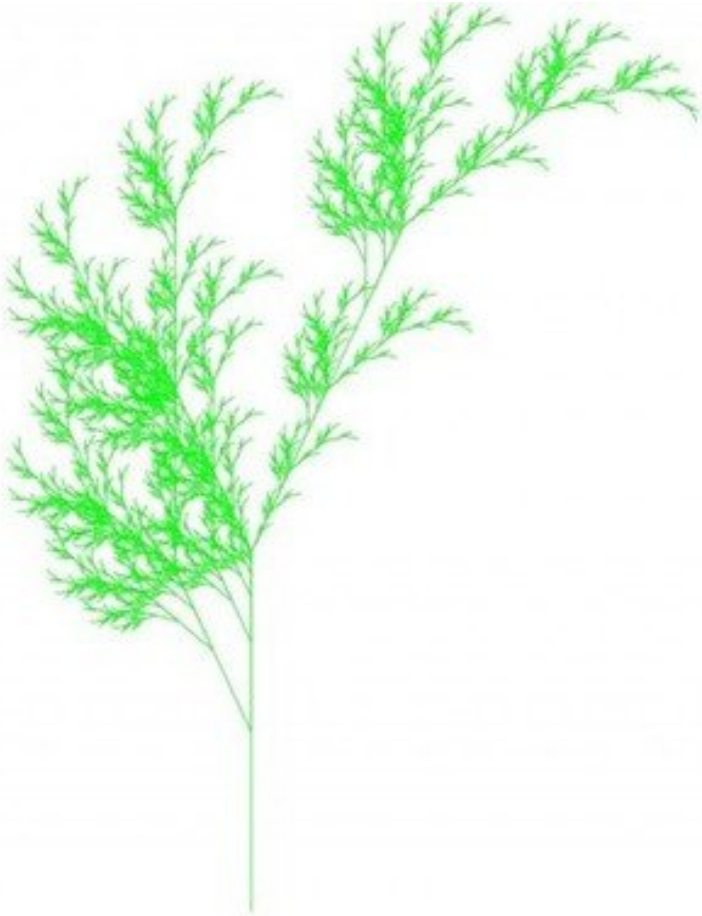
That’s it. The script will read in your Lindenmayer configuration file, parse it and open a window in which you could follow the animated drawing process. If the scipt finished the generation of the system, the image is saved (if this was requested) and the window stays open until you close it by pressing “Q” or clicking the “x” button in the upper left corner of the window.

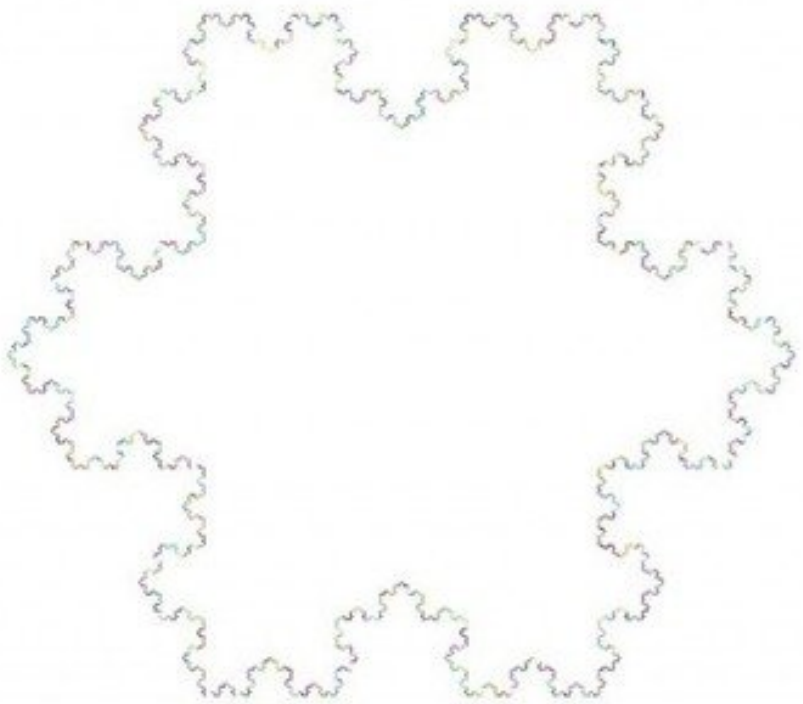
You could find the complete code of this project and some example Lindenmayer files at https://github.com/Bornageek/python (https://github.com/bornageek/python/blob/master/lindenmayer.py). Feel free to play with it and extend it, but please do not forget to share it with me and the others!

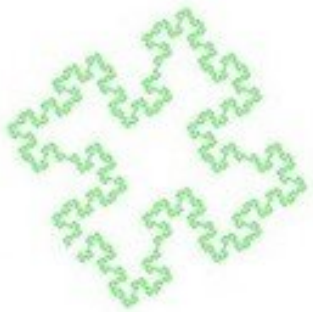
Gallery

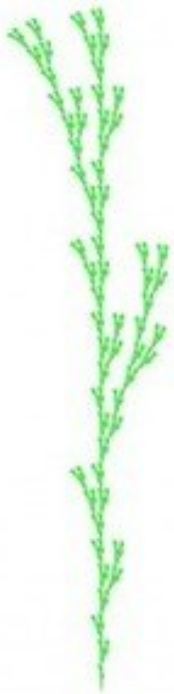


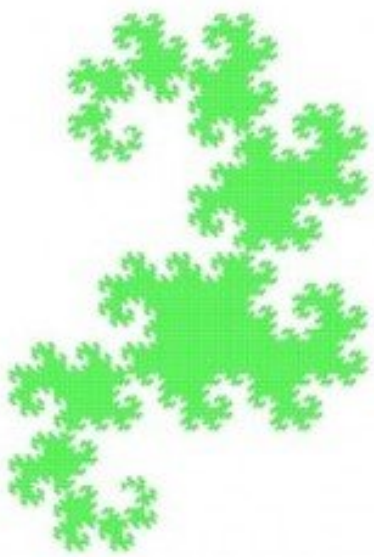












Future Work

Since this is just a very basic implementation and it only implements 2D Lindenmayer systems, I will extend this implementation over the next few weeks.

This extentions will include the possibilty to add some parameters to non-terminals to specify a custom step width and line thickness as well as a pen color and custom rotation angle. This makes it possible to fabricate more naturally appearing plants, which are not colored randomly, but based on conditional projection rules. Furthermore it will be possible to add leaves and blossoms using such rules.

Moreover, I will implement a second script using Python, that will handle the generation and visualisation of three-dimensional Lindenmayer systems. Because an other library than the turtle library have to be used for this purpose, I will also implement a new functionality to export the resulting images in different file formats.

Last but not least, I would like to over a webservice, so you and others could experiment with Lindenmayer systems online, but this will be covered in another post.

I hope you enjoyed this little Python adventure and I am looking forward to read your feedback. Stay tuned to read about my progress.

Until next time - happy modelling!

Phidelux (https://plus.google.com/u/0/109348079906250336178) is a Computer Science MSc. interested in hardware hacking, embedded Linux, compilers, etc.

Follow: ["Twitter (https://twitter.com/phidelux)", "GitHub (https://github.com/phidelux)", "RSS Feed (/feed.xml)", "E-Mail (mailto:blog@bornageek.com)"]