DM 536 Introduction to Programming

Fall 2013 Project (Part 1)

”Fractals and the Beauty of Nature"

By
Mads Petersen

Specifications

For this project there are 3 tasks that have to be solved.

1. Prepareration – understanding FractalWorld.py, specifically the new fd() function.
2. Sierpinski triangle – Writing a python program for drawing a Sierpinski triangle
3. Binary tree – Writing a python program for drawing a Binary Tree
4. The optional task of drawing a fern.

I won't go into to much more detail with the specification, for more detail see http://www.imada.sdu.dk/~petersk/DM536/ → Project description #1

FractalWorld.py

I don't think it would be very meaningful to write alot about the design and implementation of FractalWorld, since I wasn't responsible for it. Im just using it, so I'll just jump into the code, and how it differs from TurtleWorld.
Code is written in *italic* and my comments begin with #.

The new forward function.

def fd(self,dist=1, width=1, color=None):

#definition of the function with the parameters it takes, so if we pass a new turtle bob and 100, like so: "fd(bob,100)" to move our turtle forward 100, width will default to 1 and color will default to none

x, y = self.x, self.y

#makes to new variables called x and y, and assigns bob.x to x and bob.y to y, bob.x is the turtles' current x-coordinate in the world and bob.y is the y-coordinate

p1 = [x, y] #makes a new variable called p1 and assigns it the list[x,y] which is the two variables created above

p2 = self.polar(x, y, dist, self.heading) #makes a variable p2 and assigns it bob.polar(0,0,100,0),self.heading is the total amount that bob has been turned, left turns are positive numbers(unless you pass a negative nubmer, but then it is acutally a right turn) and right turns are negative. So if we turn bob left 90, his heading is 90 and if we then turn him right 90, his heading is 0. His initial heading is 0.

self.x, self.y = p2 #assigns bob.x to p2[0]=100 and bob.y to p2[1]=0(see below how these values are obtained)

if self.pen: #self.pen is true if bob has the pen down and is drawing and false if bob has the pen up.

if not color: #if color has no value, this resolves to true.

color = self.pen_color #color is set to bob.pen_color(default is blue) if no color was passed as argument with the fuction call to fd.

self.world.canvas.line([p1, p2], fill=color, width=width) #draws a line on the canvas in the world

bob belongs to going from p1 to p2, of the color=color(default blue) and with a width = witdth(default 1).

self.redraw() #updates the canvas with changes.

The polar fuction used in fd looks like this:

def polar(self, x, y, r, theta): #defination with parameters, in the case from above x = 0, y = 0, radius = 100 and theta = 0

*rad = theta * math.pi/180* #assings theta multiplied by pi from the math module divided by 180 to the variable rad

s = math.sin(rad) #assigns sin(0) = 0 to s

c = math.cos(rad) #assigns cos(0) = 1 to c

*return [x + r * c, y + r * s]* #returns the list containing at posistion 0 x+r*c(0+100*1=100) and at posistion 1 y+r*s(0+100*0=0), that is [100,0]

The old foward fuction took only 2 parameters, self and dist. So 2 new parameters have been added, making it possible to control the width of the lines and their color.

It is also easy to setup the delay when drawing, and the width and height of the canvas when creating a fractal world.

Sierpinski Triangle

Design

From the beginning of this project I had the idea that I would try and make the algortithm work on very low depth first, i.e. 1, and then go up from there and try and adjust the code based on what happend as I went up in depth(or is it down ?). So, using the code from the koch.py file as a template, I thought i would need first one movement forward, then a turn 120 degrees left, then forward again, then turning 120 degrees, then forward, and then a turn again. To put in some pseudo code using the fucntions from FractalWorld:

1. fd(sidelength)
2. leftturn(120)
3. fd(sidelength)
4. leftturn(120
5. fd(sidelength)
6. leftturn(120

This should give me a triangle provided the sidelength is the same everytime. So I put this into python code an went forward from there.

Implementation

So after converting the above to python code I had this:

```
def koch(t, s, depth):
  if depth <= 0:
    fd(t, s)
  else:
    koch(t, s/3, depth-1)
    lt(t,120)
    koch(t, s/3, depth-1)
    rt(t,120)
    koch(t, s/3, depth-1)
    lt(t,120)
```

After testing this code through depths 1-3(see testing section). I got the exact same as depth 2 and depth 3, that's obviously a problem, now when I slowed down the drawing speed by setting the delay = 1, I could see that the turtle just drew the trinangles on top of each other. So it seemed I needed it to move in between drawing triangles.

To solve this problem I considered inserting another recursive call, since that does end up moving the turtle in the base-case, but then I'd get alot of turns and other moves as well, and that wasn't really necessary. So what I did was I copied the base-case's move command(i.e. fd(t,s)) in after the last leftturn, this was only done because it was easy and I wanted something to test. Now the results of this was pretty amazing, and a bit unexpected.

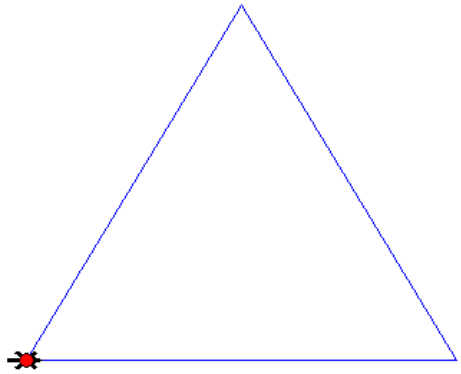
Now I was drawing Sierpinski triangles, almost anyway, as can be seen from the tests, the sides of the smaller triangles where too short, but as I looked at the code it was obvious why, in the koch-code the recursive calls are made with s/3, as argument, I needed triangles of sidelength equal to half the original, so I changed the recursive calls to s/2 instead, and tested again with depth 3.

The result was very pleasing, it seemed to work, almost, perfectly with any depth, with one exception, the line to the right, at the end of the drawing. This seemed like it was the beginning of the next depth of triangles, and was obviously a result of the last fd(t,s) call, because it wasn't there in any of the previous versions of my code. So I inserted a pu(t) call above this, and a pd(t) call above the base-case fd(t,s) call, since all the last forward call really do is move the turtle in posistion to draw the next batch of three triangles. Then the code was tested again with depth 5.

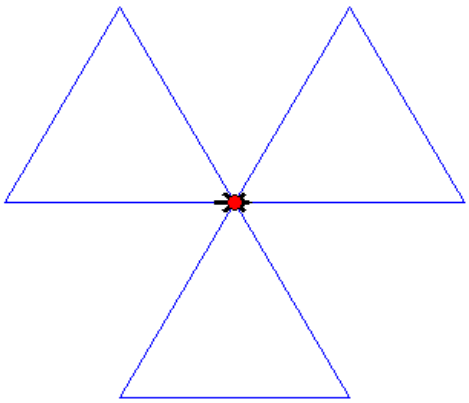
Perfect result.

Testing

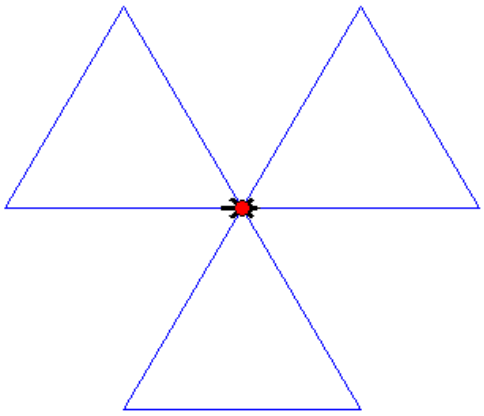
Depth 1 triangle with the v.01 code.



Depth 2 triangle with the v.01 code.



Depth 3 triangle with the v.01 code.

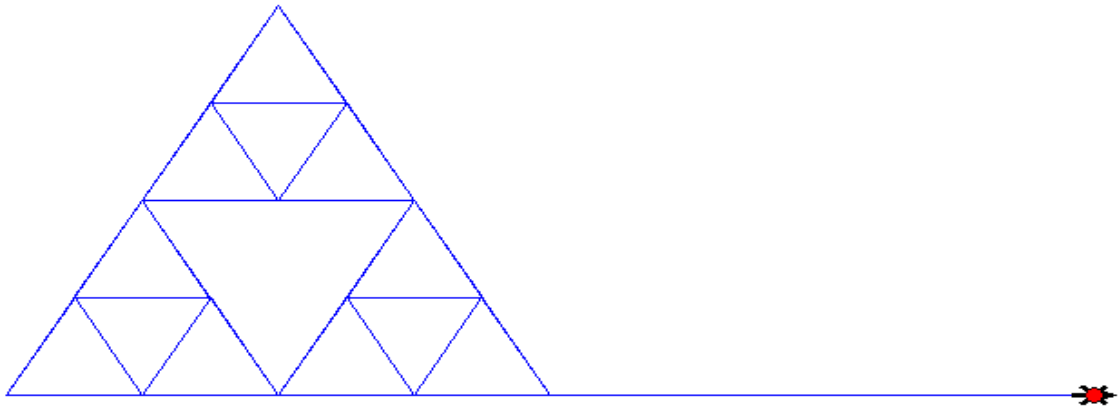


Depth 1 triangle with the v.02 code.

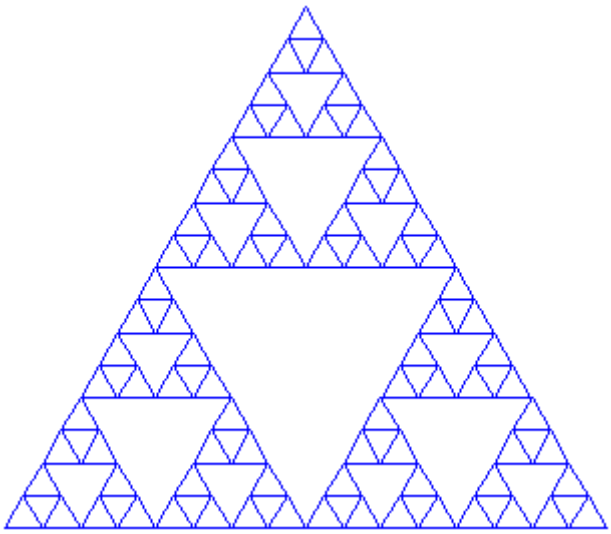
Depth 2 triangle with the v.02 code.

Depth 3 triangle with the v.02 code.

Depth 3 triangle with the v.03 code.



Depth 5 triangle with the v.04 code.



Binary Tree

Design

As with the Sierpinski Triangle, here I wanted to start out with something that worked out in low depth first, and then go from there. So I started out with trying to make the stem of the tree, and 2 brances. Here my idea was to first go forward, turn left, go forward, turn 180, go forward, turn left, go forward. And hereby have a stem with 2 branches, in psuedo code:

1. fd(t,s)
2. lt(t,60)
3. fd(t,s/2)
4. lt(t,180)
5. fd(t,s/2)
6. lt(t,60)
7. fd(t,s/2)

Implementation

I turned the initial psuedo-code to python code, and ended up with this:

```
def tree(f,s,d):
  if d == 0:
    fd(f,s)
  else:
    tree(f,s/2,d-1)
    lt(f,60)
    tree(f,s/2,d-1)
    lt(f,180)
    tree(f,s/2,d-1)
    lt(f,60)
    tree(f,s/2,d-1)
```

I then tested it with depths 0-1-2. Tests 0-1 looked like what i wanted, but when i got to depth 2, well things got a bit weird. I then started tinkering with the code and testing rapidly, i'm not going to list all the changes and all the results, since they are numerous and not much different, until i tried to make all the recursive calls use the same length, i.e. I made them all use 's/2' because it required less changes then making them all use 's'. And then i got a more interesting result at depth 2. Now i got something that sort of looked like a binary tree, only it had another stem between each branching, and of course the branches was the same length as the stem.

I tried testing this with depths 3-6 and could see, when i turned the delay up a bit, more problems. And i couldn't really make heads or tails of what to change to get the result i wanted, so i turned to the internet, and found this site:
http://interactivepython.org/courselib/static/pythonds/Recursion/graphical.html

here were examples of both a Sierpinski Triangle and a Binary Tree, it wasn't done in exactly the same way i was trying to do it, but converting this code into something that i could use, i got this:

```
def tree(f,s,d):
  if d == 0:
    pass
  else:
    fd(f,s,)
    rt(f,40)
    tree(f,s/1.5,d-1)
    lt(f,80)
    tree(f,s/1.5,d-1)
    rt(f,40)
    fd(f,-s)
```

Now i tested this up through the depths, I'll only include the last test i did, because it worked just fine. This way of doing it draws the entire right side of the tree first, then it goes back up through the depths and adds branching as it moves back down the right side. Also the base-case is just used to stop the process, not to acutally draw anything.

To obtain the differences in in width between layers, width = d was added to the fd() call.
Since the depth of the top-most branches are the smallest and the depth of the stem is the greatest this gives a good effect.

And to give different colors to the last two layers this was added to the beginnging of the function:

```
  colors = "Darkgreen"
  if d<3:
    colors = "Lightgreen"
```

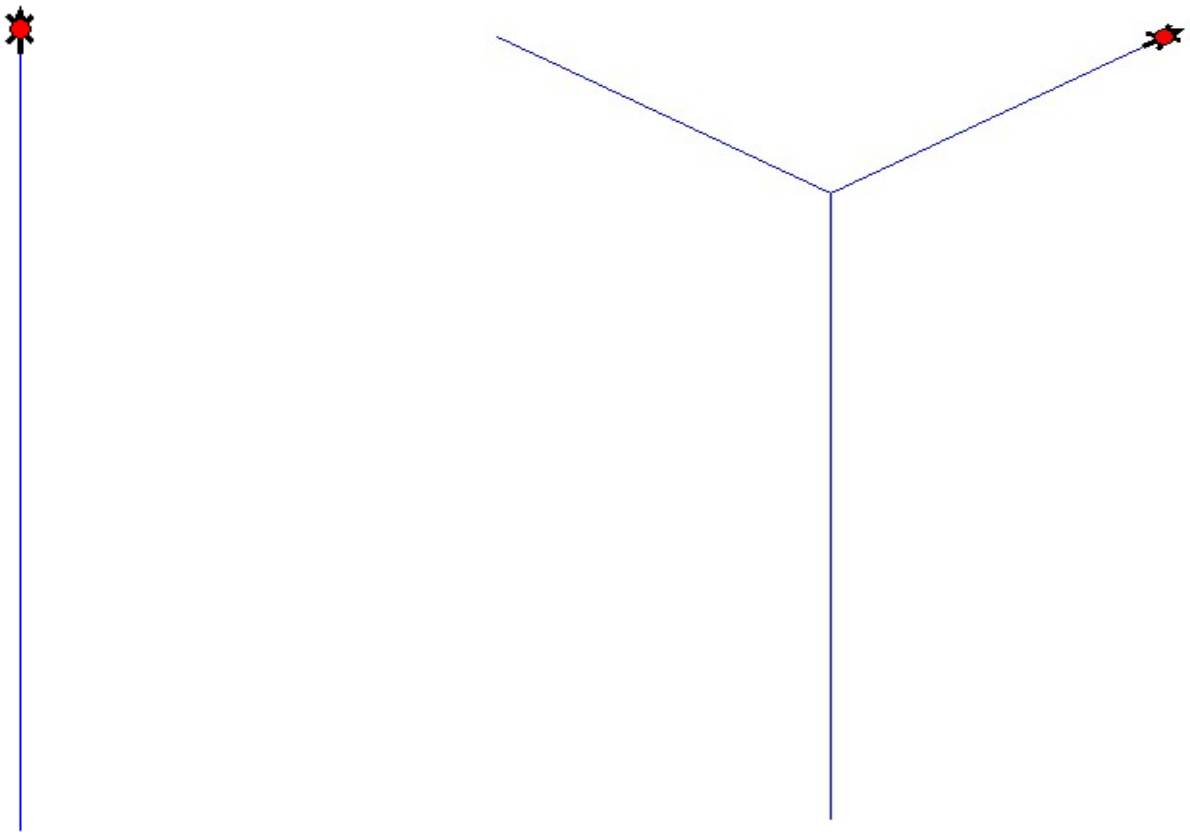
and added color=colors to the fd()call

These two changes where then tested at depth 12.

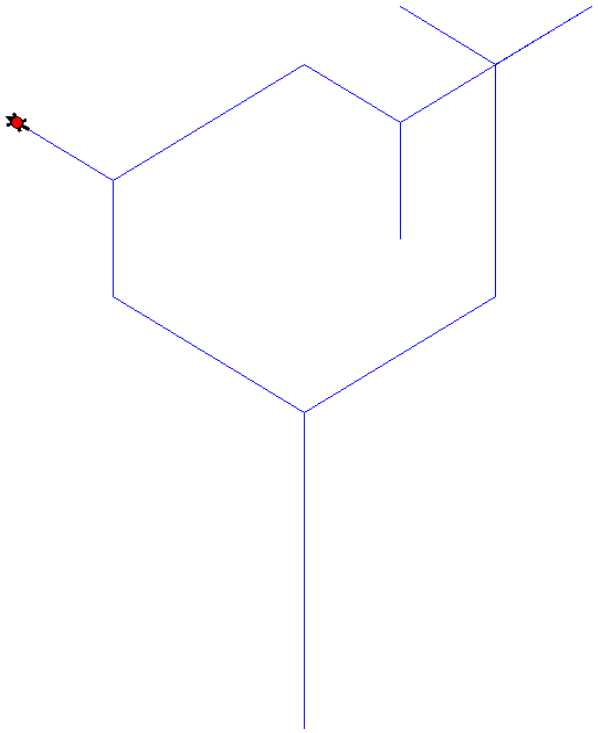
This gave an excellent end result.

Testing

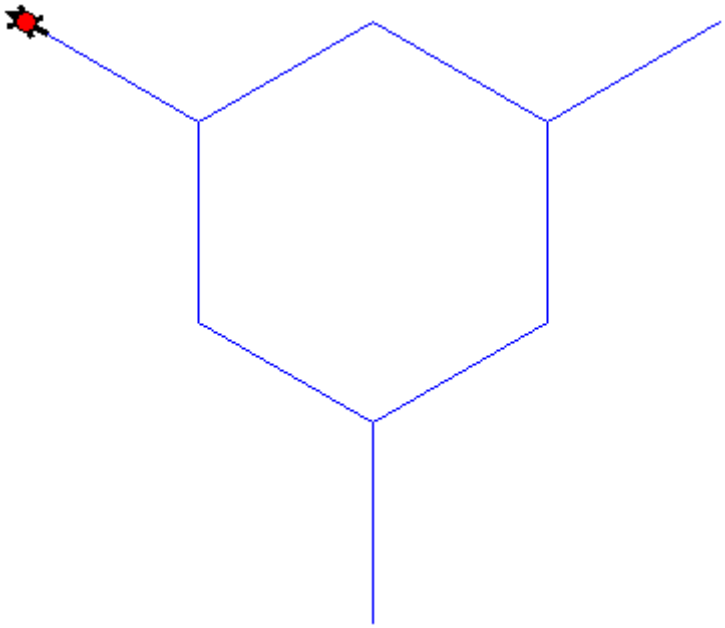
v.01 depth 0 and depth 1:



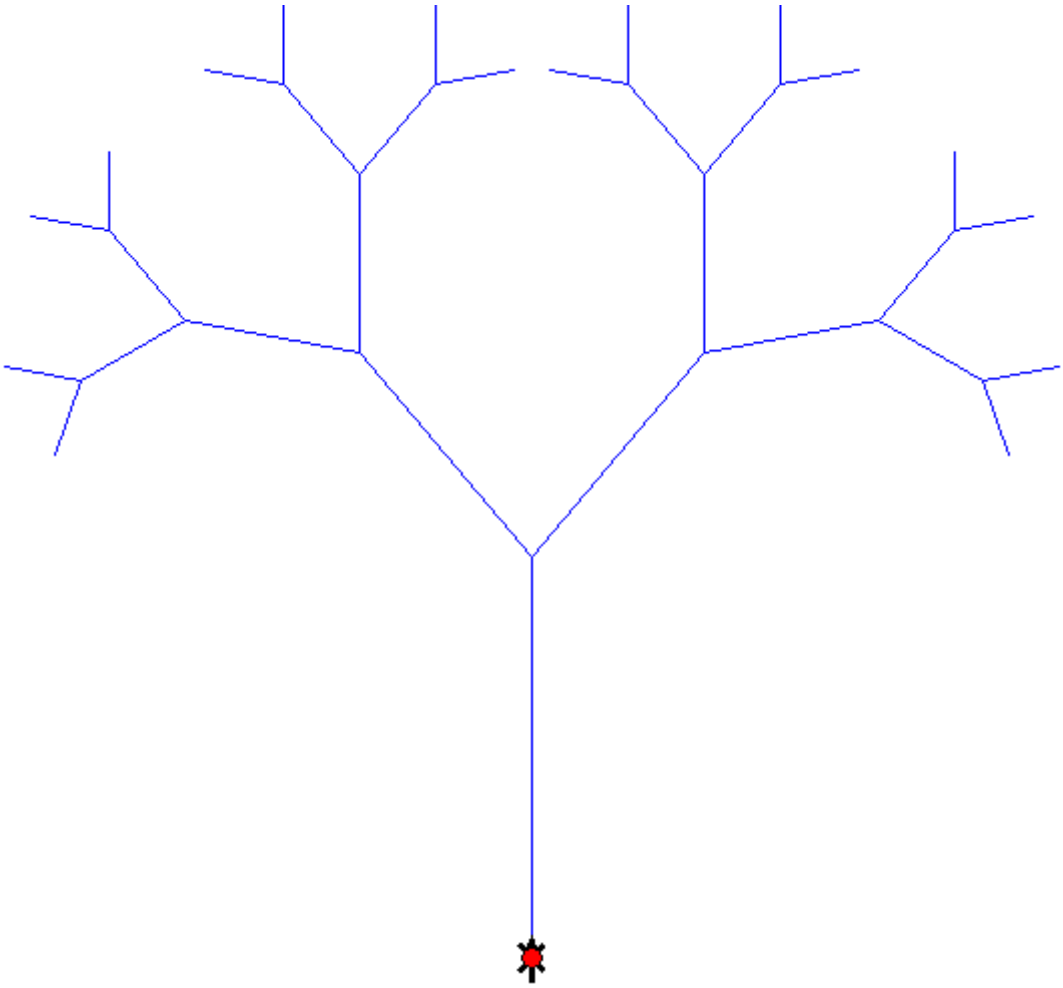
v.01 depth 2:



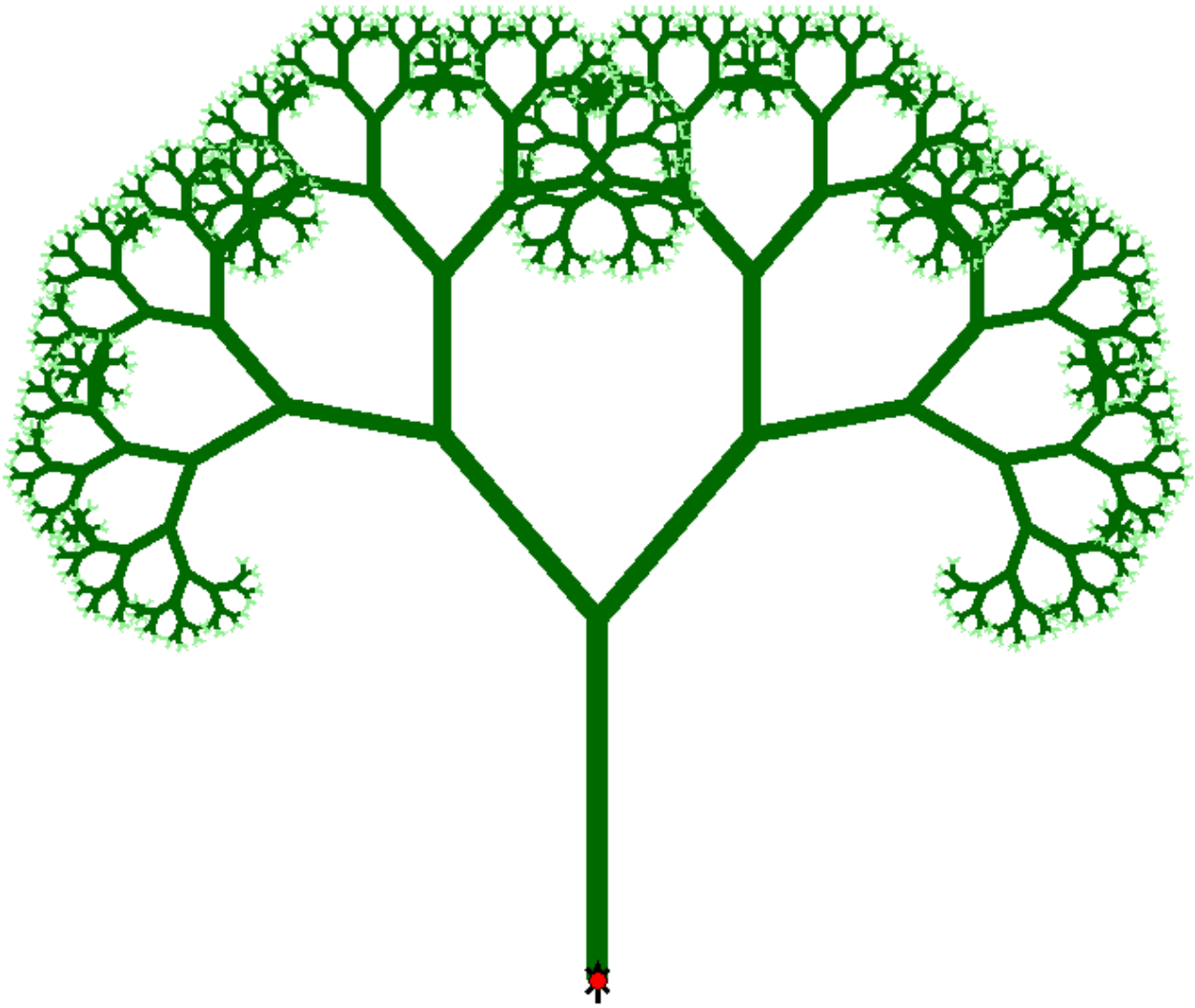
v.02 depth 2:



v.03 depth 5:



v.04 depth 12:



Conclusion

The Sierpinski Triangle was hugely satisfiying, I managed to solve it without much internet help, if any, just using some stackdiagrams and tests, tho it was also quite a stroke of luck that the first thing i decide to test works out so well. The binary tree tho gave me alot of problems, and i couldn't just see my way out of it with the tools I had. If I hadn't found the site I did, that had an example of code for a binary tree, i would probably have spent 10 times longer trying to solve it. I find that it is EXTREMELY hard for me to predict how recursive code like this behaves ahead of time, i.e. Before testing it, therefore I had great difficulty designing the code. That is also why i started out with simple code at low depths, because if i try to predict it at higher depths I completely lose sight of it, and if I try and do stack diagrams for it before testing, they get massive and it takes a very long time. Therefore I try to only do stackdiagrams when i don't understand why the codes behaves as it does in a test. This helped out alot when doing triangles. Not so much with the Binary tree, probably because i had the wrong apporoach from the beginning. I also find it diffucult to see when the code in the base-case is useful and when it is not. For instance the change between Sierpinski and Binary tree, where the code in the base-case is deleted, if I hadn't had a sample code to look at it would no doubt have taken me ages to figure out.

Appendix

Source code for the final versions of the above assignments.

Sierpinski Triangle:

```
from mine.FractalWorld import *

height1=1000

width1=1500

w = FractalWorld(width=width1,height=height1,delay=0) #creates our world, with some arguments for height, width and delay

f = Fractal(draw=False) #creates a Fractal, which is the same as a turtle in turtleworld, except here itself isn't drawn.

f.x = -(width1/2)+50 # sets the newly created Fractal's x-posistion to 50 pixels to the right of the left border of the window.

f.y = -(height1/2)+50 # sets the Fractal's y-posistion to 50 pixels above the lower border of the window


def sierp(f,s,d):
  """
  Draws a Sierpinski Triangle with a sidelength of s/2 of depth d with the fractal
  or turtle f.
  """
  if d == 0:
    pd(f)
    fd(f,s)
  else:
    sierp(f,s/2,d-1)
    lt(f,120)
    sierp(f,s/2,d-1)
    lt(f,120)
    sierp(f,s/2,d-1)
    lt(f,120)
    pu(f)
    fd(f,s)
```

Binary Tree

```
from mine.FractalWorld import *
height1=1000
width1=1500
w = FractalWorld(width=width1,height=height1,delay=0) #creates our world, with some arguments for height, width and delay
f = Fractal(draw=False) #creates a Fractal, which is the same as a turtle in turtleworld, except here itself isn't drawn.
f.y = -(height1/2)+50 # sets the Fractal's y-posistion to 50 pixels above the lower border of the window.
f.heading = 90 # turns the fractal to face "north" on the screen.


def tree(f,s,d):
  colors = "Darkgreen"
  if d<3: #makes the last two layers of branches lightgreen
    colors = "Lightgreen"
  if d == 0:
    pass
  else:
    pd(f)
    fd(f,s,width=d,color=colors)
    rt(f,40)
    tree(f,s/1.5,d-1)
    lt(f,80)
    tree(f,s/1.5,d-1)
    rt(f,40)
    pu(f)
    fd(f,-s)
```