Art and Algorithmics

A World of Procedural Trees

Morgan Saville

August 13, 2021

Contents

1 Introduction

It is uncontroversial to state that a visual artwork's value is not solely dependent on the aesthetic properties of the piece. A part – and in my opinion a very large part – of the beauty of an artwork comes from how it was created. Some have used this line of reasoning to argue artwork generated using a computer is of inherently low value, as it does not require the artist to have any technical artistic skill. I however believe that the beauty of procedurally generated art stems from the construction of the algorithms used in the process. While the programmer may need not have any talent whatsoever when it comes to drawing or painting, their art may still be of great value if they are able to make the space and time complexities of their algorithms dance with one another in just the right manner to manifest the programmer's vision efficiently and on request, with versatility and elegance.

This book will serve as an introduction the field of Algorithmic Botany. This is the science of modelling plant growth and biological processes. However, the aim of this book is not to gain a scientifically rigorous understanding of how plants grow. Instead we simply want to use the techniques we learn to create various styles of plant-based artwork.

We will explore different programming styles and how one might decide on which style to use. We will discuss the trade-off between computation time and memory usage which is omnipresent throughout the field of Computer Science. The code examples I provide will be written in the programming language Python version 3.7.5 (which can be downloaded from `https://www.python.org/`) because as programming languages go, it is particularly readable and friendly to beginners. We will try to implement as many algorithms as we can from scratch, so as to better understand where the beauty comes from in these pieces.

Our code will use a Python library named PyGame (`https://www.pygame.org/`) to handle all of the graphics. This as well as every other library we require can be installed via PyPI, the Python Package Index (`https://pypi.org/`). We will now briefly go over how to initialise PyGame to show a black window, as this will be the starting point for most of the visual artwork we create in later chapters.

The first step is to import the `pygame` module into our script as well as the `sys` module. We also need to import all the variables from the `pygame.locals` package.

```
import pygame
import sys
from pygame.locals import *
```

In case you are wondering why the first two lines are not combined into a single statement: `import pygame, sys` which would work exactly the same, the PEP 8 style guide for imports discourages this[1]. However, this is a question of style not functionality, so I'd recommend you use whichever you prefer. The next step is to initialise PyGame with

```
pygame.init()
```

Now we can create create the screen object. This is an instance of PyGame's `Surface` class. Instances of this class represent images, and we can draw to them using many of PyGame's functions. Instead of using the `Surface` constructor we will use the function `pygame.display.set_mode` because this particular surface object is special, as is it the main surface for the graphical window. The parameters we pass to it are a tuple containing the window width and height in pixels, followed by a mode. This mode is an integer which specifies whether the window is a fixed size, resizable or fullscreen. We will use mode 0 which is a fixed size window for now, but this can be customized. For now we will use a size of 640 x 360.

```
width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)
```

We will now create what is known as the *game loop*. Each iteration of this loop will be one frame (i.e. the PyGame window will be updated once per iteration). As well as drawing to the screen, on each frame we will process all of the events (key presses, mouse clicks, etc.) which happened on the previous frame. This allows us to keep our programs responsive. The way we do this is by iterating over each object in the list of events returned by the function `pygame.event.get`. For each one, check its type and react accordingly. For now, we only care about the event of the user trying to close the window. The `type` attribute of this event object is equal to the value of the `QUIT` variable imported from `pygame.locals`. If this event occurs, quit pygame using the `pygame.quit` function and close the program using the `sys.exit` function.

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
```

After the so-called *event loop*, we would draw to the window. For now we will ignore this step because we don't have anything in particular we would want to draw. After this, we would call `pygame.display.update()` to update the window. Our script now looks like this:

[1] https://www.python.org/dev/peps/pep-0008/#imports

```python
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # Here is where we draw to the screen

    pygame.display.update()
```

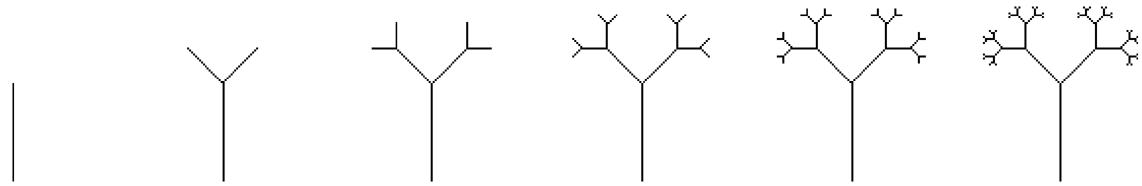
We will refer to this code as the *PyGame base code* as it will be used in most of our examples, and is in general a good place to start when making procedural art in Python.

2 Algorithmic Botany

In this chapter we will explore algorithms which generate plants and plant-like images. This is a vibrant field rife with opportunities for customisation and creativity.

2.1 Binary Fractal Tree

It goes without saying that there are many ways to generate simulations of trees algorithmically, but a binary fractal tree is one of the most simple. We start with a branch, which is a line segment in space. Each branch will have two child branches (branches whose bases are at its tip). These child branches will be a certain proportion of the length of their parent, and will be offset from their parent by some angle. The two child branches each have two child branches of their own and so on and so on.



Above is an example of the progression of such a tree where the length of each successive child branch is scaled by a factor of 0.5 from that of its parent and each child branch is rotated by $\frac{\pi}{4}$ radians from its parent. Note that in future, we will always measure angles in radians as it makes the geometry easier.

The name "Binary Fractal Tree" comes from the fact that each branch has exactly two children (hence binary) and the tree is self-similar (i.e. each branch is a smaller copy of the tree, hence fractal).

We will generate this image in two different ways – a procedural way and an object oriented way, and we will talk about the advantages of both. The difference between the two is that procedural programming tends to solve a problem from the top down. We will use simple data structures such as lists to solve our problem step by step. On the other hand, object oriented programs use complex data structures such as classes and instances of those classes. These can have their own attributes and functions, and so different parts of the problem can be delegated to these different objects to handle.

Starting with the procedural solution, we require the PyGame base code to get started.

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # Here is where we draw to the screen

    pygame.display.update()
```

Our code will work as follows: The state of a branch will be encoded as a list containing four values. The first two are the x and y coordinates of the tip of the branch. The third is the anticlockwise angle between the tip of the branch and the line starting at its base and extending to the right. The fourth value in the list is an integer representing how many children the branch currently has. We will store lists of this sort in a stack (since Python does not have a built-in stack structure, we will just use another list). To start with, this stack will contain two lists describing the base and tip of the first branch respectively.

We also need to define our variables for the length multiplier and the angle by which each child should vary from its parent. In addition we will define the initial length of the branch (this variable will change as the tree is drawn so that the branches are different lengths), as well as the number of layers we want our tree to have. We will define this all above the game loop. Since we will be using trigonometric functions, don't forget to `import math` into the script.

```
lengthMultiplier = 0.5
childAngle = math.pi/4
layerCount = 3

length = 100

stack = [[width/2, height, 0, 1],
```

```
            [width/2, height-length, math.pi/2, 0]]
```

We also need to fill the screen with white (RGB 255, 255, 255). We will do this before the game loop so that the background does not overwrite the tree drawing each frame.

```
screen.fill((255, 255, 255))
```

Now within the game loop we will use a `while` loop to iterate until the stack only has one list in it. Within this loop, we need to get the last two elements from the stack such that we can draw a line between the branches they represent. For now we'll make the line black (RGB 0, 0, 0) and give it a thickness of 1.

```
    while len(stack) > 1:
        base, tip = stack[-2:]
        pygame.draw.line(screen,
                         (0, 0, 0),
                         (base[0], base[1]),
                         (tip[0], tip[1]),
                         1)
```

Now we need to check whether or not to give this branch a child. The first condition to check is whether the number of branches already in the stack is less than or equal to `layerCount`, because if so we have room to give the current branch more children. In this case, we multiply the `length` variable by the value of `lengthMultiplier` because we are moving up the tree and the branches should be changing length.

We can calculate the angle of the new branch by taking the angle of the previous branch (stored in `tip[2]`) and either add or subtract the variable `childAngle`. We need to subtract if the previous branch has no current children (`tip[3] == 0`) and we need to add it if the previous branch already has one child (`tip[3] == 1`). This is how we achieve the effect of the first child branching off to the left and the second branching to the right. A more concise way to write this would be `newAngle = tip[2] - childAngle + (2*childAngle*tip[3])` which will achieve the same effect.

```
        if len(stack) <= layerCount:
            length *= lengthMultiplier
            newAngle = tip[2]-childAngle + (2*childAngle*tip[3])
```

Using these values we can use basic trigonometry to calculate the position of the tip of the new branch.

```
            newTip = (tip[0] + math.cos(newAngle) * length,
                      tip[1] - math.sin(newAngle) * length)
```

The reason why we subtract the sin term rather than adding it is that in most graphical programs including PyGame, the y coordinate increases downwards, whereas in regular mathematical graphs, the y coordinate increases upwards. This is an important distinction to keep in mind, as it is often much easier to plan out the geometry of your desired artwork in terms of mathematical graphs.

Once we have calculated the position of the tip of the new branch, we increment the value of `tip[3]` to show that the previous branch now has one more child, and then we push the new branch to the stack. To do the latter, we must construct the list which represents this new branch. While this could be written as `[newTip[0], newTip[1], newAngle, 0]`, it is more concise to write it as `[*newTip, newAngle, 0]`.

```
            tip[3] += 1
            stack.append([*newTip, newAngle, 0])
```

The stack loop now looks like this:

```
    while len(stack) > 1:
        base, tip = stack[-2:]
        pygame.draw.line(screen,
                         (0, 0, 0),
                         (base[0], base[1]),
                         (tip[0], tip[1]),
                         1)

        if len(stack) <= layerCount:
            length *= lengthMultiplier
            newAngle = tip[2]-childAngle + (2*childAngle*tip[3])

            newTip = (tip[0] + math.cos(newAngle) * length,
                      tip[1] - math.sin(newAngle) * length)

            tip[3] += 1
            stack.append([*newTip, newAngle, 0])
```

We now need to add an `else` case to the `if` statement. If there is no room to add another child branch, we need to pop the last branch off the stack (note that this does not mean that this branch is no longer in the tree, as it will have already been drawn in a previous iteration of the stack loop, but rather that it is no longer on the path we are currently drawing). We then need to continue to pop branches from the stack until we find one with fewer than 2 children. Once this happens we can continue with the next iteration of the loop. Each time we pop a branch from the stack, we need to divide the `length` variable by the value of `lengthMultiplier`, as we are moving back down the tree.

```
        else:
            stack = stack[:-1]
            length /= lengthMultiplier
            while stack[-1][3] == 2:
                stack = stack[:-1]
                length /= lengthMultiplier
```

This is the entire program:

```
import pygame
import sys
import math
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

lengthMultiplier = 0.5
childAngle = math.pi/4
layerCount = 5

length = 100

stack = [[width/2, height, 0, 1],
         [width/2, height-length, math.pi/2, 0]]

screen.fill((255, 255, 255))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    while len(stack) > 1:
        base, tip = stack[-2:]
```

```
            pygame.draw.line(screen,
                             (0, 0, 0),
                             (base[0], base[1]),
                             (tip[0], tip[1]),
                             1)

            if len(stack) <= layerCount:
                length *= lengthMultiplier
                newAngle = tip[2]-childAngle + (2*childAngle*tip[3])

                newTip = (tip[0] + math.cos(newAngle) * length,
                          tip[1] - math.sin(newAngle) * length)

                tip[3] += 1
                stack.append([*newTip, newAngle, 0])

            else:
                stack = stack[:-1]
                length /= lengthMultiplier
                while stack[-1][3] == 2:
                    stack = stack[:-1]
                    length /= lengthMultiplier

        pygame.display.update()
```

Naturally you can modify the parameters of the program – `lengthMultiplier`, `childAngle`, `layerCount` – to customise the result. Below are some examples of various combinations.

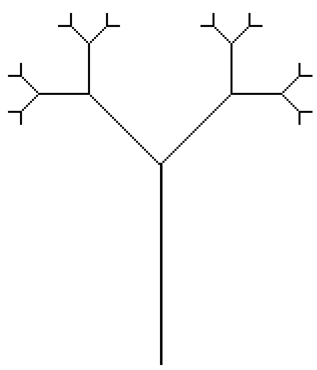


Figure 2.1: `lengthMultiplier`=0.5, `childAngle`=$\frac{\pi}{4}$, `layerCount`=5

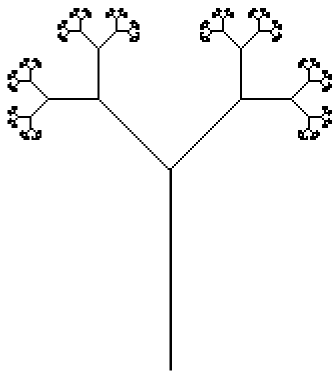


Figure 2.2: `lengthMultiplier`=0.5, `childAngle`=$\frac{\pi}{4}$, `layerCount`=10

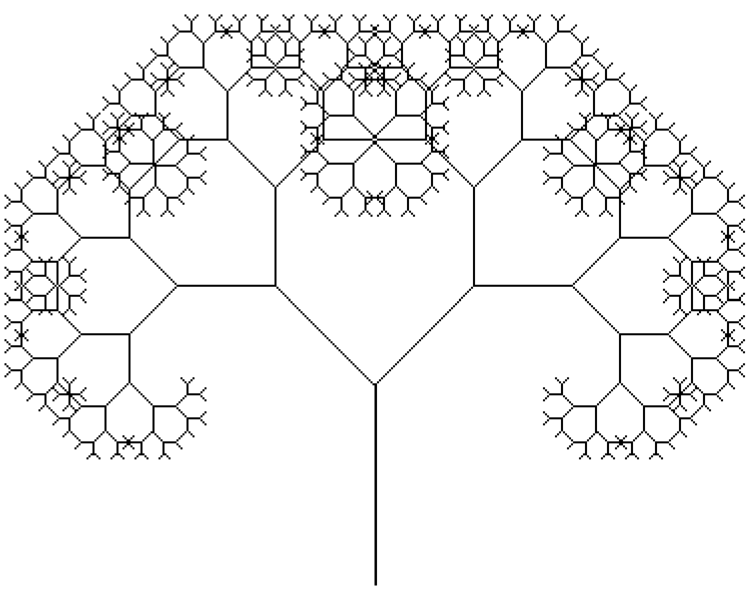


Figure 2.3: `lengthMultiplier`=0.7, `childAngle`=$\frac{\pi}{4}$, `layerCount`=10

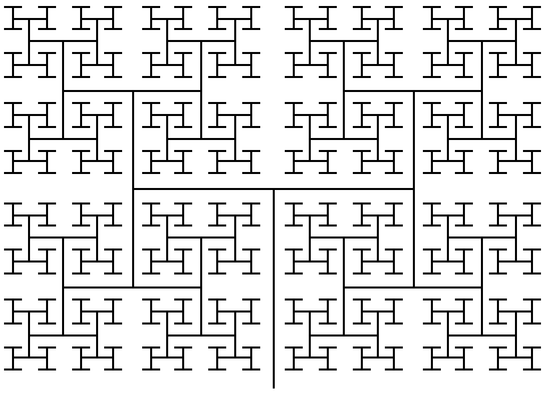


Figure 2.4: `lengthMultiplier`=0.7, `childAngle`=$\frac{\pi}{2}$, `layerCount`=10

As you can see, these parameters alone are enough to make some pretty interesting shapes. However, we can do much better. Unfortunately the procedural method we are using to draw the tree leaves us with little room to be creative. Let's do it again using object oriented programming to give us more freedom.

2.1.1 Object Oriented Method

We'll start with the PyGame base code.

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # Here is where we draw to the screen

    pygame.display.update()
```

Let's create a class named `Branch`. Our tree object will be an instance of this class. Each `Branch` object will have several parameters: a list named `children` which will contain other `Branch` objects, an int called `depth` which represents how many descendants the branch can have (i.e. the root branch might have `depth=2`, each of its children would have `depth=1`, each of their children would have `depth=0` and those branches can have no more children), an angle and a length. The latter three will be passed in via the constructor and the `children` list will be initially left empty.

```
class Branch:
    def __init__(self, depth, angle, length):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length
```

Now we need to populate the `children` list as long as `depth > 0`. However, since child branches need to have their lengths scaled by some amount, that amount needs to be accessible by the `Branch` object. We will achieve this by passing to the `Branch` constructor an instance of another class named `Gene`. Objects of the `Gene` class will contain information about the length multiplier and the angle by which each branch should deviate from its parent.

```
class Gene:
    def __init__(self, lengthMultiplier, childAngle):
        self.lengthMultiplier = lengthMultiplier
        self.childAngle = childAngle

class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length
```

Now we can go back to the `Branch` constructor and check whether `depth > 0` and if so, add two new `Branch` objects to `children`.

```
class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length
```

```

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene)

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene)

            self.children += [child1, child2]
```

There are two things to note here. The first is that the last line of this snippet does the equivalent of two `self.children.append` function calls, but is more succinct. The more important thing to note is that the same `Gene` object is being passed from parent branch to child branch, but this need not be the case – there is room for mutation, which we will implement later.

For now we will focus on adding a function to the `Branch` object which will handle drawing the branch to the screen. We will call this function `draw`. It will need to take as parameters a PyGame `Surface` object on which to draw as well as a base position from which to draw (which is the same as the tip of its parent branch). Then we simply need to calculate the position of the tip of the branch using the same trigonometry as before (so remember to `import math`) and draw a line between the base and the tip. We will again use a black line with thickness 1.

```
    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)
```

However, we don't want to have to call this function on each branch individually. Instead we can use recursion. Each time we draw any branch, it should recursively draw all its children onto the same screen, and using the parent branch's tip as the child branch's base.

```
    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

```

```
        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)

        for child in self.children:
            child.draw(screen, tip)
```

We're done for now with the Branch class and we will revisit it later. For now let's create an instance of it right before the game loop. First we will create a Gene object to pass to it.

```
treeGene = Gene(0.5, math.pi/4)

tree = Branch(5, math.pi/2, 100, treeGene)
```

Let's discuss the parameters we passed to the Branch constructor. The first parameter (for which we used 5) is equivalent to the layerCount variable from the procedural example; it tells the program how many layers the tree should have. The second parameter makes the first branch appear at a right angle to the horizontal (from which all angles are measured). The third parameter is the length of the first branch. Finally of course the last parameter is the gene object to be passed to each child branch.

To draw the tree, inside the game loop we fill the screen with white and then call the tree.draw function, passing it the screen object as well as the position of the bottom of the tree (we will use the centre of the bottom edge of the screen).

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))

    tree.draw(screen, (width/2, height))

    pygame.display.update()
```

Our code now looks like this:

```
import pygame
import sys
import math
from pygame.locals import *

class Gene:
    def __init__(self, lengthMultiplier, childAngle):
        self.lengthMultiplier = lengthMultiplier
        self.childAngle = childAngle

class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene)

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene)

            self.children += [child1, child2]

    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)

        for child in self.children:
            child.draw(screen, tip)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

treeGene = Gene(0.5, math.pi/4)

tree = Branch(5, math.pi/2, 100, treeGene)
```

```

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))

    tree.draw(screen, (width/2, height))

    pygame.display.update()
```

If you now run the program you should see the same results as with the procedural method.

Again, you can of course customise the parameters passed to the initial `Branch` constructor and to the `Gene` constructor but object oriented programming allows us to do so much more than this. The entire tree is now stored in memory, and so we can manipulate it. The first change we'll make is to implement gene mutation.

2.1.2 Gene Mutation

We will add a function in the `Gene` class named `mutate`. It will return a new `Gene` object whose attributes are slightly different from the original. The mutated attribute will be normally distributed, so we need to import a library capable of generating random (or psuedorandom) normally distributed numbers. The library NumPy can do this. It can be installed with PyPI. We can add `from numpy.random import normal` to the preamble of our program, and then add our function to the `Gene` class.

```
class Gene:
    def __init__(self, lengthMultiplier, childAngle, stddev):
        self.lengthMultiplier = lengthMultiplier
        self.childAngle = childAngle
        self.stddev = stddev

    def mutate(self):
        return Gene(abs(self.lengthMultiplier * normal(1, self.stddev)),
                    abs(self.childAngle * normal(1, self.stddev)),
                    self.stddev)
```

As you can see, we have modified the constructor to accept an additional parameter, `stddev`, the standard deviation of the normal distribution we will use to mutate the parameters. The mutated attributes are multiplied by a random number with mean 1 and standard deviation equal to the `stddev` variable passed to the `Gene` constructor. We then take the absolute value of the mutated attribute such that it can never be negative. This avoids issues such as branches growing backwards due to negative length. If you like, you can also randomly mutate the `stddev` attribute of the new gene, but I find that the result looks nicer if you don't.

Next we need to go back to where we create the `treeGene` object and pass in a standard deviation. In essence you can consider this to be a proportion of the initial value. For now we will use 0.1 but again this is customisable.

```
treeGene = Gene(0.5, math.pi/4, 0.1)
```

Lastly we need to modify the `Branch` constructor such that instead of passing the `gene` object to the children branches upon creation, we instead pass the object returned by `gene.mutate`.

```
class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            self.children += [child1, child2]
```

Our code now looks like this:

```
import pygame
import sys
import math
from numpy.random import normal
from pygame.locals import *

class Gene:
```

```
    def __init__(self, lengthMultiplier, childAngle, stddev):
        self.lengthMultiplier = lengthMultiplier
        self.childAngle = childAngle
        self.stddev = stddev

    def mutate(self):
        return Gene(abs(self.lengthMultiplier * normal(1, self.stddev)),
                    abs(self.childAngle * normal(1, self.stddev)),
                    self.stddev)

class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            self.children += [child1, child2]

    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)

        for child in self.children:
            child.draw(screen, tip)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

treeGene = Gene(0.5, math.pi/4, 0.1)

tree = Branch(5, math.pi/2, 100, treeGene)
```

```

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))

    tree.draw(screen, (width/2, height))

    pygame.display.update()
```

Below are some examples of some randomised trees created by the above script.

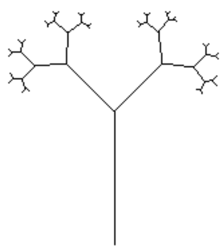


Figure 2.5: Randomised Tree with Initial Length Multiplier 0.5, Child Angle $\frac{\pi}{4}$ and Standard Deviation 0.1 (As in the Script Shown Above)

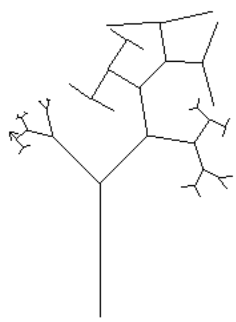


Figure 2.6: Randomised Tree with Initial Length Multiplier 0.5, Child Angle $\frac{\pi}{4}$ and Standard Deviation 0.3

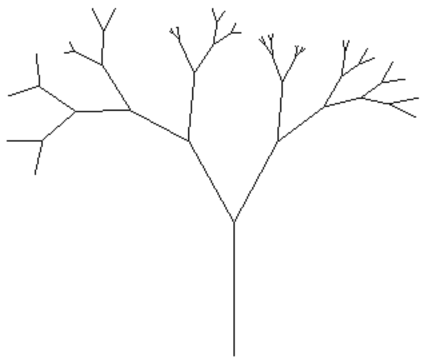


Figure 2.7: Randomised Tree with Initial Length Multiplier 0.7, Child Angle 0.5 and Standard Deviation 0.2

Personally, I'm a fan of the combination of parameters in Figure 2.7 – an initial length multiplier of 0.7, a Child Angle of 0.5 and a Standard Deviation of 0.2. In our code this corresponds to the line:

```
treeGene = Gene(0.7, 0.5, 0.2)
```

2.1.3 Adding Colour

Next we're going to add some colour (note that for style reasons I will spell this "*color*" within the code). As you can imagine, there are many ways to add colour to a stick-figure tree. The method we will use here involves drawing the branches as brown lines and the leaves as coloured diamonds. Of course any polygon will work and will require almost identical code, but I like the look of diamonds.

The first step is to detect which branches are actually leaves, and draw them as diamonds. The detection turns out to be quite simple – a branch is a leaf if and only if it has no child branches. We will test for this condition in the `Branch.draw` function. If the length of the `children` attribute of the branch is non-zero, then we draw the branch as a regular straight line. Otherwise, we draw it as a diamond. This check will look like the following:

```
    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

```

```
        if len(self.children):
            pygame.draw.line(screen,
                             (0, 0, 0),
                             base,
                             tip,
                             1)
        else:
            #Draw as diamond

        for child in self.children:
            child.draw(screen, tip)
```

Note that in Python, non-zero numbers evaluate to the boolean `True` and zero evaluates to the boolean `False`. Because of this, the line `if len(self.children):` is equivalent to writing `if len(self.children) != 0:`.

The question then arises: how do we draw a diamond? Luckily, PyGame has a handy `pygame.draw.polygon` function which requires a list of points. We already have two of the points – the base and the tip. We will call the straight line segment between these the major axis of the diamond. The minor axis is the perpendicular bisector to the major axis (i.e. a line segment which shares a midpoint with the major axis and is at a right angle to it). The length of the minor axis is up to our discretion, but I like to use the length of the major axis multiplied by the reciprocal of the Golden Ratio, $\frac{1}{\phi} = \frac{2}{1+\sqrt{5}} \approx 0.61803398875$.

We can work out the gradient of the major axis by calculating the change in y coordinate over the change in x coordinate from the base to the tip: `M=(tip[1]-base[1])/(tip[0]-base[0])`. Geometry tells us therefore that the gradient of the minor axis is `m=-1/M`. We can simplify this to one line: `m=(base[0]-tip[0])/(tip[1]-base[1])`. This is the change in y per unit change in x along the minor axis. We can work out the magnitude of such a step using Pythagoras' Theorem: `a = (m**2 + 1)**0.5`. Therefore a step of length `1/a` in the x direction and `m/a` in the y direction will correspond to a step of magnitude 1 along the minor axis.

We can work out the length of the major axis again using Pythagoras' Theorem and multiplying by $\frac{1}{2\phi}$ gives us half of the length of the minor axis. We can therefore set a variable `semiMinorAxisLength = majorAxisLength * oneOverTwoPhi`.

We can calculate the shared midpoint by taking the mean x and y coordinates of the base and tip points, and then use all of the information above to calculate the two other vertices of our diamond as follows:

```
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) ** 0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi
```

```
            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            m = (base[0]-tip[0])/(tip[1]-base[1])
            a = (m**2 + 1)**0.5

            vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                       midpoint[1] + semiMinorAxisLength*m/a)

            vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                       midpoint[1] - semiMinorAxisLength*m/a)
```

This works in almost all cases. It fails however when the tip is directly horizontally aligned with the base. In this case, the minor axis has an infinite gradient and we will get a division by zero error. We can therefore add in a case to check this.

```
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) ** 0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi

            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            if (tip[1] == base[1]):
                vertex1 = (midpoint[0],
                           midpoint[1] + semiMinorAxisLength)

                vertex2 = (midpoint[0],
                           midpoint[1] - semiMinorAxisLength)

            else:
                m = (base[0]-tip[0])/(tip[1]-base[1])
                a = (m**2 + 1)**0.5

                vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                           midpoint[1] + semiMinorAxisLength*m/a)

                vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                           midpoint[1] - semiMinorAxisLength*m/a)
```

Now we can draw the diamond using `pygame.draw.polygon`. The parameters it accepts are a `pygame.Surface` object to which to draw, a colour (for which we will use black as the outline of the leaf) and then a tuple of points. We will use `(base, vertex1, tip, vertex2, base)`.

We pass the base twice so that the polygon is closed. The final parameter is a line thickness, for which we will pass a value of 1. A value of 0 would mean that the polygon is filled. The `Branch.draw` function now looks like this:

```
    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        if len(self.children):
            pygame.draw.line(screen,
                                 (0, 0, 0),
                                 base,
                                 tip,
                                 1)
        else:
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) ** 0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi

            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            if (tip[1] == base[1]):
                vertex1 = (midpoint[0],
                           midpoint[1] + semiMinorAxisLength)

                vertex2 = (midpoint[0],
                           midpoint[1] - semiMinorAxisLength)

            else:
                m = (base[0]-tip[0])/(tip[1]-base[1])
                a = (m**2 + 1)**0.5

                vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                           midpoint[1] + semiMinorAxisLength*m/a)

                vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                           midpoint[1] - semiMinorAxisLength*m/a)

            pygame.draw.polygon(screen,
                                    (0, 0, 0),
                                    (base, vertex1, tip, vertex2, base),
                                    1)

        for child in self.children:
            child.draw(screen, tip)
```

If we now run the program we see the diamonds render perfectly.

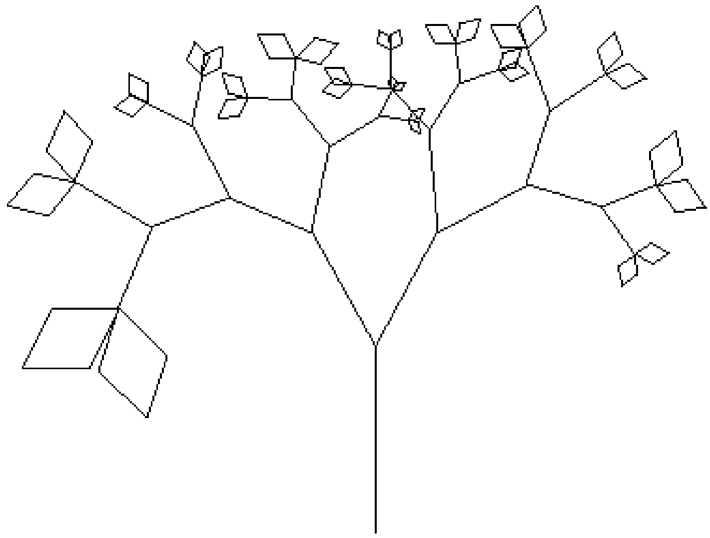


Figure 2.8: A Randomised Binary Fractal Tree with the Leaves Drawn as Diamonds

Now that we have leaves with an area, we can get to work on colouring them. The obvious choice would be to colour them green, but this is boring and unoriginal. Instead we will make each leaf a random colour.

Colours are usually stored in RGB (red, green, blue) format in graphical programs, so we might think that a good way to select a colour for our leaf is to pick a random value for each of red, green and blue. The problem with this is that most colours are incredibly ugly. Most of the leaves would end up being a brownish grey and the tree would not look as vibrant as we want. Instead we will use the HSV (hue, saturation, value) colour space. The saturation and value will be maxed out at 100 and we will pick a uniform random value between 0 (inclusive) and 360 (non-inclusive) for the hue. We will therefore get a very vibrant and saturated colour at a random angle through the colour wheel. Luckily, PyGame has built-in functionality to convert colours from HSV to RGB. We need to use the `numpy.random.randint` function to generate uniformly distributed random integers, so don't forget to modify your `import` statement to:

```
from numpy.random import normal, randint
```

We will now go back to the draw function after the vertices have been calculated, but before the polygon is drawn. We will pick a random value for the hue. Then, we will create a `pygame.Color` object, passing the values `0, 0, 0` to the constructor. These are RGB values, so the colour represented by the object will be black at first. We can then access the `hsva` attribute of the object and set it equal to our new tuple, `(hue, 100, 100, 100)`. This will change the colour to our random pretty one. The "*a*" in "`hsva`" stands for "*alpha*" and refers to opacity. By setting it to its maximum (100), we make the leaf fully opaque, though this as well as the saturation and value are parameters with which you can mess around until you find something you like.

Finally we duplicate the call to `pygame.draw.polygon`. We modify the first call such that our `pygame.Color` object is passed in place of the tuple `(0, 0, 0)`, and we change the final parameter to 0 so that the polygon is filled in with the colour. The second call will then draw the black outline on top like before, but this is of course optional. At this point, the `Branch.draw` function looks like this:

```
    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        if len(self.children):
            pygame.draw.line(screen,
                             (0, 0, 0),
                             base,
                             tip,
                             1)
        else:
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) **
            ↪  0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi

            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            if (tip[1] == base[1]):
                vertex1 = (midpoint[0],
                           midpoint[1] + semiMinorAxisLength)

                vertex2 = (midpoint[0],
                           midpoint[1] - semiMinorAxisLength)

            else:
                m = (base[0]-tip[0])/(tip[1]-base[1])
                a = (m**2 + 1)**0.5

                vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                           midpoint[1] + semiMinorAxisLength*m/a)

                vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                           midpoint[1] - semiMinorAxisLength*m/a)

            hue = randint(0, 360)
            leafColor = pygame.Color(0, 0, 0)
            leafColor.hsva = (hue, 100, 100, 100)
```

```

            pygame.draw.polygon(screen,
                                leafColor,
                                (base, vertex1, tip, vertex2, base),
                                0)

            pygame.draw.polygon(screen,
                                (0, 0, 0),
                                (base, vertex1, tip, vertex2, base),
                                1)
```

If we run the code now, we would find something close to the desired result except that the colours change rapidly and randomly every frame. While the leaf colours changing over time may be desirable for some projects, it will be easier to manipulate and control the colour of each leaf if it is stored as an attribute of the `Branch` object itself. To achieve this we need to modify the `Branch` constructor such that the objects have an attribute named `leafColor`. By default we initialise it to `None` because it does not apply to every branch, only those with no children. Next, we go back to the `Branch.draw` function and change it so that this colour is only generated only if the existing colour is `None`. This way it will only be generated the first time the tree is rendered, and then will remain constant on all subsequent renderings. The `Branch` class now looks like this:

```
class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length
        self.leafColor = None

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            self.children += [child1, child2]

    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        if len(self.children):
```

```
            pygame.draw.line(screen,
                             (0, 0, 0),
                             base,
                             tip,
                             1)
        else:
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) ** 0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi

            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            if (tip[1] == base[1]):
                vertex1 = (midpoint[0],
                           midpoint[1] + semiMinorAxisLength)

                vertex2 = (midpoint[0],
                           midpoint[1] - semiMinorAxisLength)

            else:
                m = (base[0]-tip[0])/(tip[1]-base[1])
                a = (m**2 + 1)**0.5

                vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                           midpoint[1] + semiMinorAxisLength*m/a)

                vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                           midpoint[1] - semiMinorAxisLength*m/a)

            if self.leafColor is None:
                hue = randint(0, 360)
                self.leafColor = pygame.Color(0, 0, 0)
                self.leafColor.hsva = (hue, 100, 100, 100)

            pygame.draw.polygon(screen,
                                self.leafColor,
                                (base, vertex1, tip, vertex2, base),
                                0)

            pygame.draw.polygon(screen,
                                (0, 0, 0),
                                (base, vertex1, tip, vertex2, base),
                                1)

        for child in self.children:
            child.draw(screen, tip)
```

Now we need to make the non-leaf branches look pretty. The way I like to do this is to change their colour to brown (I picked RGB 77, 0, 27 but this is up to your own taste) and make them slightly thicker (I used thickness 3). This means going back to the top of the `Branch.draw` function and modifying the call to `pygame.draw.line` so that it looks like this:

```
            pygame.draw.line(screen,
                             (77, 0, 27),
                             base,
                             tip,
                             3)
```

Our code now looks like this:

```
import pygame
import sys
import math
from numpy.random import normal, randint
from pygame.locals import *

class Gene:
    def __init__(self, lengthMultiplier, childAngle, stddev):
        self.lengthMultiplier = lengthMultiplier
        self.childAngle = childAngle
        self.stddev = stddev

    def mutate(self):
        return Gene(abs(self.lengthMultiplier * normal(1, self.stddev)),
                    abs(self.childAngle * normal(1, self.stddev)),
                    self.stddev)

class Branch:
    def __init__(self, depth, angle, length, gene):
        self.children = []
        self.depth = depth
        self.angle = angle
        self.length = length
        self.leafColor = None

        if self.depth > 0:
            child1 = Branch(depth-1,
                            angle - gene.childAngle,
                            length * gene.lengthMultiplier,
                            gene.mutate())

            child2 = Branch(depth-1,
                            angle + gene.childAngle,
```

```
                              length * gene.lengthMultiplier,
                              gene.mutate())

            self.children += [child1, child2]

    def draw(self, screen, base):
        tip = [base[0] + math.cos(self.angle) * self.length,
               base[1] - math.sin(self.angle) * self.length]

        if len(self.children):
            pygame.draw.line(screen,
                             (77, 0, 27),
                             base,
                             tip,
                             3)
        else:
            oneOverTwoPhi = 1/(1 + 5 ** 0.5)

            majorAxisLength = ((tip[0]-base[0])**2 + (tip[1]-base[1])**2) **
            ↪ 0.5
            semiMinorAxisLength = majorAxisLength * oneOverTwoPhi

            midpoint = ((base[0] + tip[0])/2,
                        (base[1] + tip[1])/2)

            if (tip[1] == base[1]):
                vertex1 = (midpoint[0],
                           midpoint[1] + semiMinorAxisLength)

                vertex2 = (midpoint[0],
                           midpoint[1] - semiMinorAxisLength)

            else:
                m = (base[0]-tip[0])/(tip[1]-base[1])
                a = (m**2 + 1)**0.5

                vertex1 = (midpoint[0] + semiMinorAxisLength/a,
                           midpoint[1] + semiMinorAxisLength*m/a)

                vertex2 = (midpoint[0] - semiMinorAxisLength/a,
                           midpoint[1] - semiMinorAxisLength*m/a)

            if self.leafColor is None:
                hue = randint(0, 360)
                self.leafColor = pygame.Color(0, 0, 0)
                self.leafColor.hsva = (hue, 100, 100, 100)

            pygame.draw.polygon(screen,
                                self.leafColor,
                                (base, vertex1, tip, vertex2, base),
                                0)
```

```python

                pygame.draw.polygon(screen,
                                    (0, 0, 0),
                                    (base, vertex1, tip, vertex2, base),
                                    1)

        for child in self.children:
            child.draw(screen, tip)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

treeGene = Gene(0.7, 0.5, 0.2)

tree = Branch(5, math.pi/2, 100, treeGene)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))

    tree.draw(screen, (width/2, height))

    pygame.display.update()
```

Below are some examples of what this program might generate.

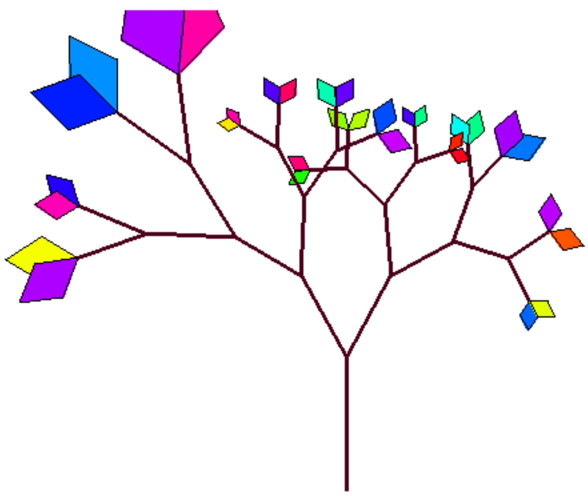


Figure 2.9: A Randomised Binary Tree with Coloured Diamond Leaves

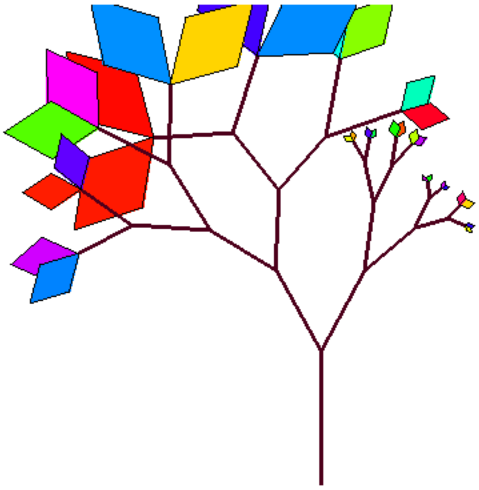


Figure 2.10: Another Randomised Binary Tree with Coloured Diamond Leaves

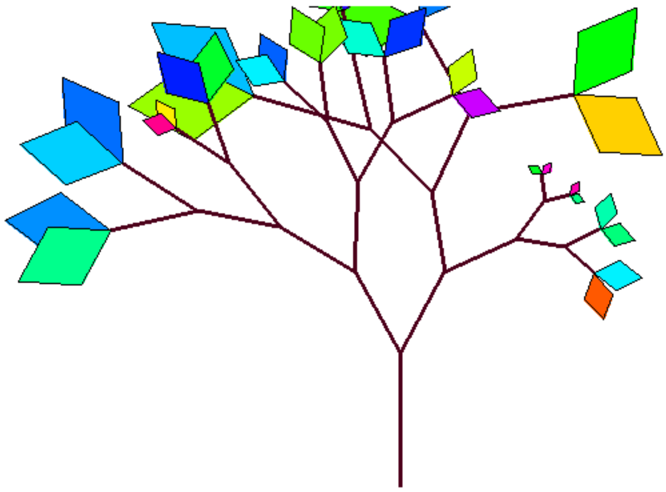


Figure 2.11: Yet Another Randomised Binary Tree with Coloured Diamond Leaves

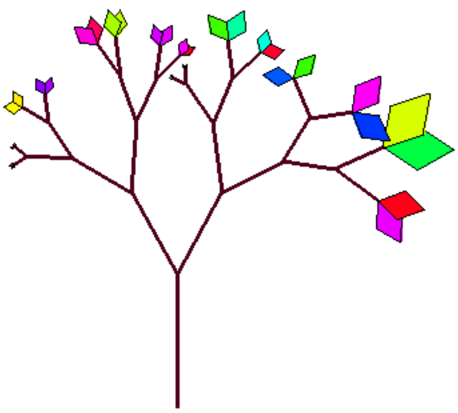


Figure 2.12: One More Randomised Binary Tree with Coloured Diamond Leaves

Note that all of the above examples have identical parameters. They resulted from the same exact code being run multiple times. The variation arises purely from the random elements we included in the logic.

Hopefully now you can see the advantages of the object-oriented approach as opposed to the procedural way of generating and drawing this tree. The use of complex data structures (classes in particular) allow us to have more descriptive variables instead of having that confusing stack of lists which the procedural method called for. OOP (object oriented programming) also allows us to make changes to the tree in between generating it and drawing it. For example, we would be able to vary the angles of the branches over time creating the illusion of the tree blowing in the wind.

So when would a procedural approach be more appropriate? The answer to this is usually to do with memory constraints. If you don't have much memory available to you (e.g. if the simulation is being run on a microprocessor or a Raspberry Pi), it may be impractical to store hundreds or even thousands of class instances in memory. Though this takes more computation time, it may instead make more sense to draw the tree as you generate it, discarding the now irrelevant data as you go.

Making the program in an object oriented manner may nevertheless be the best solution if you are unsure exactly what you want your end product to look like. It allows you the freedom to experiment with modifications such as:

- Making the child branches thinner than their parents
- Giving the leaves some random alpha (opacity) value
- Making the colour of the leaves change over time
- Varying the number of children each branch has
- Making the child branches sprout from some proportion along the parent branch, rather than always from the tip.

Some of which may well be impossible or at least incredibly inefficient with a procedural approach.

2.2 L-System Trees

An L-system is a formal grammar system which works on the character level. Developed in 1968 by biologist and botanist Aristid Lindenmayer at the University of Utrecht, L-systems are used to model natural growth. In particular, Lindenmayer used them to model the growth of plants, and this is what we will use them for here.

L-systems work by taking a string and replacing instances of certain characters with a new string. The details of exactly which characters to replace with what string is determined by a ruleset. The initial string is called an “axiom” and each available character for which a rule applies is called a “variable”. A character for which no rule applies is called a “constant”.

For example, if we start with an axiom of “A” and our ruleset contains a single rule which states that “A” should be replaced with “ABA”, this is what would happen if we iterate this ruleset multiple times.

axiom: A

$A \longrightarrow ABA$

Iteration 1: ABA
Iteration 2: $ABABABA$
Iteration 3: $ABABABABABABABA$
Iteration 4: $ABABABABABABABABABABABABABABABA$

Now suppose we start with the same axiom and with an additional rule which replaces “B” with “C”.

axiom: A

$A \longrightarrow ABA$
$B \longrightarrow C$

Iteration 1: ABA
Iteration 2: $ABACABA$
Iteration 3: $ABACABACABACABA$
Iteration 4: $ABACABACABACABACABACABACABACABA$

We will now get to work implementing L-systems in Python.

2.2.1 Implementing the L-System

Starting with a blank script we will create a class called `LSystem`. Its constructor will accept a string named `axiom` and a dictionary named `ruleset`. The keys of the `ruleset` dictionary are the variables of the LSystem – "A", "B", etc. – and the corresponding values represent what this variable is to be replaced with – "ABA", "C", etc. We will then assign these parameters to attributes of the class instance.

```
class LSystem:
    def __init__(self, axiom, ruleset):
        self.axiom = axiom
        self.ruleset = ruleset
```

We now need to add a function named `generate` which accepts a number `n` of iterations and will return a string which corresponds to the $\mathtt{n}^{\text{th}}$ iteration of the ruleset. We could do this iteratively with a `for` loop but instead we will implement a recursive algorithm for reasons which will become apparent later. We can quickly handle the degenerate case. If `n` is equal to zero (or equivalently if `not n` evaluates to `True`) then simply return the axiom. Otherwise we need to fetch the $(\mathtt{n}-1)^{\text{th}}$ iteration so that we can iterate the ruleset again.

```
    def generate(self, n):
        if not n:
            return self.axiom

        previous = self.generate(n-1)
```

We can construct the result in a string named `new` which we will initially make blank. We then iterate over each character in `previous` and if the character is one of the keys in the `ruleset` dictionary, then we append the corresponding value to the `new` string. If not, then simply append the character. Once this is done, we can return `new` and we are done. Our `LSystem` class looks like this:

```
class LSystem:
    def __init__(self, axiom, ruleset):
        self.axiom = axiom
        self.ruleset = ruleset

    def generate(self, n):
        if not n:
            return self.axiom

        previous = self.generate(n-1)
        new = ""
        for char in previous:
            if char in self.ruleset:
                new += self.ruleset[char]
            else:
```

```
                new += char
        return new
```

We can now test it out using the parameters from earlier.

```
ruleset = {"A": "ABA",
           "B": "C"}

axiom = "A"

lsystem = LSystem(axiom, ruleset)

print(lsystem.generate(4))
```

As we expect, we get the output

```
ABACABACABACABACABACABACABACABA
```

So why did we use recursion rather than a loop? Recursion allows us to use a technique called *memoization* (the fact that this is exclusively a programming term warrants the Americanised spelling). If you haven't come across memoization or function decorators before, the concept may be a little tricky so we'll leave our L-system code for a while and discuss a simpler example.

2.2.2 Memoization

Let's take a recursive function for calculating Fibonacci numbers[1]. To summarise, the first two Fibonacci numbers are equal to 1 and any other Fibonacci number is equal to the sum of the previous two. Below is a simple recursive algorithm to generate the n^{th} Fibonacci number, as well as a loop to output the first 10 values.

```
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

for i in range(10):
    print(fibonacci(i))
```

[1] Sequence A000045 in the On-Line Encyclopedia of Integer Sequences (`https://oeis.org/A000045`)

As we'd expect we see the output:

```
1
1
2
3
5
8
13
21
34
55
```

The only problem is that it's pretty slow. We can use the `time` module to time how long it takes to generate the 40th Fibonacci number.

```
import time

def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

startTime = time.time()

print(fibonacci(40))

elapsed = time.time() - startTime

print(elapsed)
```

Though this value will obviously vary from computer to computer, the result I got was nearly 40 seconds which is frankly ridiculous.

```
165580141
39.46642065048218
```

Luckily there is something we can do about it. In order to work out `fibonacci(40)` for example, the program will recursively work out `fibonacci(39)` and `fibonacci(38)`. Then the call to `fibonacci(39)` will in turn call `fibonacci(38)` and `fibonacci(37)` and so on. However this means that `fibonacci(38)` is being called twice. In fact every input except 39 and 40 will be evaluated more than once and the problem grows exponentially as the inputs go to zero. As such, if we have some cache – a dictionary which maps values of `n` to the corresponding value of `fibonacci(n)` – and we store values in this cache whenever we calculate them, then the recursion can be cut short as soon as we need to calculate any of those values again. Here is how we would implement this in code:

```
import time

cache = {}
def fibonacci(n):
    if n < 2:
        return 1
    if n not in cache:
        cache[n] = fibonacci(n-1) + fibonacci(n-2)
    return cache[n]

startTime = time.time()

print(fibonacci(40))

elapsed = time.time() - startTime

print(elapsed)
```

Now the program finishes in under one 219$^{\text{th}}$ of a second.

```
165580141
0.0045545101165771484
```

That's more than a 99.988% reduction in execution time. Even though the specific times will vary it should be clear that this is an incredible improvement. This is the main concept of memoization: Cache inputs to the function alongside the corresponding output so that if the same input is passed to the function again, we don't need to recalculate it.

The solution above is rather inelegant though as it uses global variables and is rather intrusive to the implementation of the function. Suppose we had many such recursive functions and we wanted to memoize each of them. We would need multiple caches and we would need to modify the code of each of the functions to use those caches which is inconvenient. Instead what we can do is create some process by which a function is converted into a memoized version of itself.

In mathematics, a process which turns a number into another number is called a function (e.g. $f(x) = x^2$) and a process which turns a function into another function is called an operator (e.g. $g(x) = \frac{d}{dx}f(x)$). In keeping with this, I will refer to the process of memoizing a function as an operator although in Python terms, it is simply a function whose input and output are both functions. We want to end up with some code of the form:

```
fibonacci = memoize(fibonacci)
```

This is the part which can be a little difficult to wrap your head around if you've never seen anything like this before so be sure you understand the significance of this line before moving on. We are passing the `fibonacci` function itself to the `memoize` operator which will return a new function which is a memoized version of `fibonacci`, and we replace the reference to the original function which this new one.

We can now implement the `memoize` operator which accepts as its parameter a function, `func`, to memoize. Within this operator definition there will be a local cache. There will also be a nested function named `helper` which accepts a parameter `n`. It will first check whether `n` is not in the cache. If so, calculate `func(n)` and store the result in the cache. Then outside the conditional, return the cached output corresponding to the input `n`. The `memoize` operator should then return this `helper` function.

```python
def memoize(func):
    cache = {}
    def helper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]
    return helper
```

That's it. If we add this to our original code, we get this:

```python
def memoize(func):
    cache = {}
    def helper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]
    return helper

def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

fibonacci = memoize(fibonacci)

print(fibonacci(100))
```

Here we've included a little test at the bottom to show that you can just call the function like normal and we get the expected output incredibly quickly:

```
573147844013817084101
```

There is one final adjustment to make. The line

```python
fibonacci = memoize(fibonacci)
```

is very ugly and confusing. I would argue that it obscures away how the code works. All we are doing here is wrapping the `fibonacci` function inside the `helper` function created by `memoize`. This is called "*decorating* the `fibonacci` function with `memoize`". Another problem with the above snippet is that it is sometimes ambiguous where in the code to put it. If the function belonged to some class, would you call this line after the class definition or for each instance of the class? Perhaps it would be called in the constructor. Each of these approaches have their flaws. Luckily, Python has a built in piece of syntax which allows you to instead add what is called a "`decorator`" right above the function definition. It looks like this:

```python
@memoize
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

This way we don't have to handle the headache of replacing the reference to the function with a memoized version, we just tell the Python interpreter to decorate the function with `memoize`. Our entire code now looks like this:

```python
def memoize(func):
    cache = {}
    def helper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]
    return helper

@memoize
def fibonacci(n):
    if n < 2:
```

```python
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

That's memoization. In this example, it reduced the time complexity of the `fibonacci` function from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$. This is the difference between the function being intractable and it being tractable. Memoization makes this function computationally usable.

It does however take up more space in memory, as the results of each function call need to be stored in the cache even after the call is finished.

Furthermore, memoization doesn't always work. In fact it only works with functions with the following properties:

- The function always returns the same output for a given input, no matter when or how many times it is called.
- The function only produces an output and doesn't carry out any important procedure. If the function has already been called with the given input, the `helper` function ensures that it isn't called again, and so any procedure carried out by the function won't happen.

2.2.3 Variadic Memoization

There is also a small problem with the particular implementation we have used. Namely, it only works on functions with a single input. Since lists can't be used as a key for a dictionary in Python, we would need to use a custom `Cache` class instead. It will have as its attributes a list named `keys` and a list named `vals`. The idea is that the key at index `i` corresponds to the value at index `i`.

```python
class Cache:
    def __init__(self):
        self.keys = []
        self.vals = []
```

Since this is a custom class not a dictionary, we can no longer simply get and set values using square bracket indexing like we're used to. Luckily there is a way to make this happen. Python classes have so called "*dunder*" functions (short for "double-underscore", named so because the function names both start and end with two underscores). These tell the class object how to behave when they are used with certain syntax e.g. if one is added to another using the + operator, etc.

We can overload three of these dunder functions – `__getitem__`, `__setitem__` and `__contains__`. The first of these three determines how the object should react when a value is accessed from it via square brackets (`cache[key]`) and so we can overload it like this:

```
    def __getitem__(self, key):
        return self.vals[self.keys.index(key)]
```

This simply returns the value which is at the same index in the `vals` list as the argument is in the `keys` list. The second dunder to overload determines how the object should set a value using square brackets (`cache[key] = value`). As such we can overload it like this:

```
    def __setitem__(self, key, val):
        if key in self.keys:
            self.vals[self.keys.index(key)] = val
        else:
            self.keys.append(key)
            self.vals.append(val)
```

If the key is already in the `keys` list, set the corresponding value from the `vals` list to the parameter. Otherwise, add the key and the value to the end of their respective lists. Finally, we need to overload the dunder which dictates how to determine whether or not the key is in the cache using the `in` keyword (`key in cache`). This is easy to implement as we just need to return whether or not the key is in the object's `keys` list.

```
    def __contains__(self, key):
        return key in self.keys
```

Our finished `Cache` class now looks like this:

```
class Cache:
    def __init__(self):
        self.keys = []
        self.vals = []

    def __getitem__(self, key):
        return self.vals[self.keys.index(key)]

    def __setitem__(self, key, val):
        if key in self.keys:
            self.vals[self.keys.index(key)] = val
```

```
        else:
            self.keys.append(key)
            self.vals.append(val)

    def __contains__(self, key):
        return key in self.keys
```

Now we need to modify the `memoize` operator to work with this class. The first step is to make it so that the `cache` variable defined by the function is an instance of the `Cache` class rather than just a dictionary.

```
def memoize(func):
    cache = Cache()
```

The next step is to modify the `helper` function to accept a variable number of arguments and keyword arguments. To do this we can use the `*args` and `**kwargs` syntax.

```
    def helper(*args, **kwargs):
```

This means that `args` will be a list containing all of the positional arguments passed to the function and `kwargs` will be a dictionary mapping keyword argument names to the values passed in for them. This makes `helper` a "*variadic*" or "*varargs*" function – i.e. a function with a variable number of parameters. We will then create a list with two elements – `args` followed by `kwargs` – which will be used as the key for our cache. In the event where we need to call the memoized function, we pass `*args, **kwargs` to it as its parameters.

```
    def helper(*args, **kwargs):
        params = [args, kwargs]
        if params not in cache:
            cache[params] = func(*args, **kwargs)
        return cache[params]
```

Our final variadic memoization code looks like this:

```
class Cache:
    def __init__(self):
        self.keys = []
        self.vals = []
```

```

    def __getitem__(self, key):
        return self.vals[self.keys.index(key)]

    def __setitem__(self, key, val):
        if key in self.keys:
            self.vals[self.keys.index(key)] = val
        else:
            self.keys.append(key)
            self.vals.append(val)

    def __contains__(self, key):
        return key in self.keys

def memoize(func):
    cache = Cache()
    def helper(*args, **kwargs):
        params = [args, kwargs]
        if params not in cache:
            cache[params] = func(*args, **kwargs)
        return cache[params]
    return helper
```

So how does this apply to our L-system? We left off with the code looking like this:

```
class LSystem:
    def __init__(self, axiom, ruleset):
        self.axiom = axiom
        self.ruleset = ruleset

    def generate(self, n):
        if not n:
            return self.axiom

        previous = self.generate(n-1)
        new = ""
        for char in previous:
            if char in self.ruleset:
                new += self.ruleset[char]
            else:
                new += char
        return new
```

The `generate` function is recursive, so if we want to generate multiple versions of the L-system (perhaps with different numbers of iterations) then it would be efficient to memoize the `generate` function. Since the function accepts multiple parameters (`self` and `n`) we have to use the variadic implementation. This means our code now looks like this:

```
class Cache:
    def __init__(self):
        self.keys = []
        self.vals = []

    def __getitem__(self, key):
        return self.vals[self.keys.index(key)]

    def __setitem__(self, key, val):
        if key in self.keys:
            self.vals[self.keys.index(key)] = val
        else:
            self.keys.append(key)
            self.vals.append(val)

    def __contains__(self, key):
        return key in self.keys

def memoize(func):
    cache = Cache()
    def helper(*args, **kwargs):
        params = [args, kwargs]
        if params not in cache:
            cache[params] = func(*args, **kwargs)
        return cache[params]
    return helper

class LSystem:
    def __init__(self, axiom, ruleset):
        self.axiom = axiom
        self.ruleset = ruleset

    @memoize
    def generate(self, n):
        if not n:
            return self.axiom

        previous = self.generate(n-1)
        new = ""
        for char in previous:
            if char in self.ruleset:
                new += self.ruleset[char]
            else:
                new += char
        return new
```

2.2.4 Constructing Images From L-Systems

The L-system generates a string, so how do we use it to generate an image of a tree? Consider the L-System with variables 0 and 1, and constants [and]. Its axiom is 0 and its rules are:

$$0 \longrightarrow 1[0]0$$
$$1 \longrightarrow 11$$

We can construct such an L-system with our code, and output for example the 4^{th} recursion.

```
axiom = "0"
ruleset = {"0": "1[0]0",
           "1": "11"}

lsystem = LSystem(axiom, ruleset)

print(lsystem.generate(4))
```

The output we get is:

```
11111111[1111[11[1[0]0]1[0]0]11[1[0]0]1
[0]0]1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0
```

If we interpret this string as a series of instructions – 0 or 1 means move forwards, [means push your current position and angle to a stack and then rotate anticlockwise 45°, and] means pop the position and angle from the stack and then rotate clockwise by 45°– and follow those instructions one by one, we produce an image. We will do this procedurally at first and then switch over to an object oriented approach.

First let's save the code for the `LSystem` (i.e. the `Cache` class, the `memoize` function and the `LSystem` class) into a file. It can be called whatever you want, but I'm going to call it "lsystem.py". Then, in another script in the same directory, we can import the `LSystem` class as well as the `math` module. This, as well as the PyGame base code gives us the following script:

```
import pygame
import sys
import math
from pygame.locals import *
```

```python
from lsystem import LSystem

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # Here is where we draw to the screen

    pygame.display.update()
```

Before the game loop we will create the L-system we described earlier and store the 4$^{\text{th}}$ iteration as a string called `recipe`. I like to use this name because I think of this string as a list of instructions which we follow in order to cook up an image. We can also create a function named `draw` which accepts as its parameters a PyGame surface on which to draw, a recipe string, a starting angle and a starting position. This function needs to have the stack we will use to store positions and angles (which will initially be an empty list) and current x and y coordinates (which will initially be defined by the parameter of the function).

```python
axiom = "0"
ruleset = {"0": "1[0]0",
           "1": "11"}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

def draw(screen, recipe, angle, base):
    stack = []
    x, y = base
```

Within `draw`, we now need to iterate over each character in `recipe`. We will call the current character `instruction`. If `instruction` is either `"0"` or `"1"` (equivalent to asking whether `instruction` is contained by the list `["0", "1"]`) then we calculate the new position based off the x and y coordinates and the angle. It will be a straight line of length let's say 20 pixels from the current x and y coordinate.

```python
    for instruction in recipe:
        if instruction in ["0", "1"]:
            newPoint = (x + math.cos(angle) * 20,
```

```
                                y - math.sin(angle) * 20)
```

Again note that the sin term is negative, because decreasing y coordinates in pixel space means moving upwards. We now draw a line between the old point and the new. We will make this line black with thickness 1 for now. We then set `x` and `y` to their new values.

```
                pygame.draw.line(screen,
                                 (0, 0, 0),
                                 (x, y),
                                 newPoint,
                                 1)

                x, y = newPoint
```

Now, if `instruction` is equal to `"["`, then we push to the end of the stack a list containing the current x and y coordinates and the current angle. We then increment the angle by $\frac{\pi}{4}$ radians (45°).

```
            elif instruction == "[":
                stack.append([x, y, angle])
                angle += math.pi/4
```

Finally if `instruction` is equal to `"]"`, then we retrieve the x and y coordinates and the angle from the end of the stack, remove them from the stack, and then decrease the angle by $\frac{\pi}{4}$ radians.

```
            elif instruction == "]":
                x, y, angle = stack[-1]
                stack = stack[:-1]
                angle -= math.pi/4
```

Our draw function now looks like this:

```
def draw(screen, recipe, angle, base):
    stack = []
    x, y = base

    for instruction in recipe:
        if instruction in ["0", "1"]:
```

```
            newPoint = (x + math.cos(angle) * 20,
                        y - math.sin(angle) * 20)

            pygame.draw.line(screen,
                             (0, 0, 0),
                             (x, y),
                             newPoint,
                             1)

            x, y = newPoint

        elif instruction == "[":
            stack.append([x, y, angle])
            angle += math.pi/4

        elif instruction == "]":
            x, y, angle = stack[-1]
            stack = stack[:-1]
            angle -= math.pi/4
```

Inside the game loop and after the event loop we can fill the screen with white and then call the `draw` function passing in as its parameters the screen, the recipe, a starting angle of $\frac{\pi}{2}$ (at right angles to the horizontal) and a base position of the centre of the bottom edge of the screen. Our script now looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

axiom = "0"
ruleset = {"0": "1[0]0",
           "1": "11"}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

def draw(screen, recipe, angle, base):
    stack = []
    x, y = base

    for instruction in recipe:
```

```
        if instruction in ["0", "1"]:
            newPoint = (x + math.cos(angle) * 20,
                        y - math.sin(angle) * 20)

            pygame.draw.line(screen,
                             (0, 0, 0),
                             (x, y),
                             newPoint,
                             1)

            x, y = newPoint

        elif instruction == "[":
            stack.append([x, y, angle])
            angle += math.pi/4

        elif instruction == "]":
            x, y, angle = stack[-1]
            stack = stack[:-1]
            angle -= math.pi/4

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    draw(screen, recipe, math.pi/2, (width/2, height))

    pygame.display.update()
```

When we run the code, we see this familiar result.

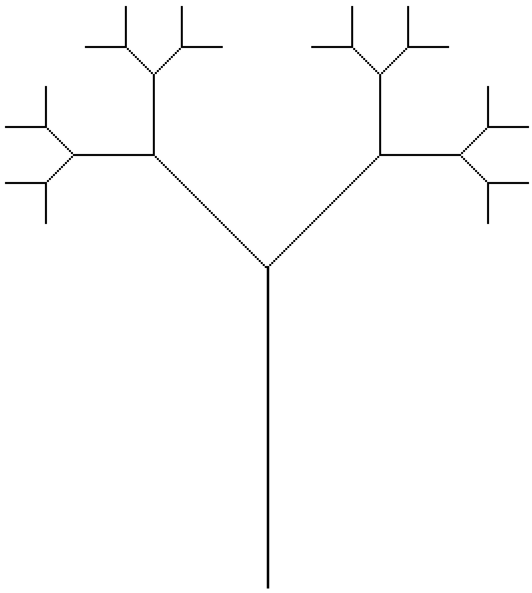


Figure 2.13: An L-System Binary Fractal Tree

Don't worry though, we haven't just spent this long recreating the same image as in the previous section. Let's say instead we replace our L-system with one whose axiom is X and whose rules are:

$$\begin{aligned} X &\longrightarrow F+[[X]-X]-F[-FX]+X \\ F &\longrightarrow FF \end{aligned}$$

Here, F means move forwards, X means do nothing, $+$ means increase the angle by 25°, $-$ means decrease the angle by that same amount, [means push the position and angle to the stack, and] means pop the position and angle from the stack. We can modify our code to obey those instructions such that it looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "X"
ruleset = {"X": "F+[[X]-X]-F[-FX]+X",
           "F": "FF"}

```

```
lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(5)

def draw(screen, recipe, angle, base):
    stack = []
    x, y = base

    for instruction in recipe:
        if instruction == "F":
            newPoint = (x + math.cos(angle) * 10,
                        y - math.sin(angle) * 10)

            pygame.draw.line(screen,
                             (0, 0, 0),
                             (x, y),
                             newPoint,
                             1)

            x, y = newPoint

        elif instruction == "+":
            angle += math.radians(25)

        elif instruction == "-":
            angle -= math.radians(25)

        elif instruction == "[":
            stack.append([x, y, angle])

        elif instruction == "]":
            x, y, angle = stack[-1]
            stack = stack[:-1]

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    draw(screen, recipe, math.pi/2, (width/2, height))

    pygame.display.update()
```

Take note of the following things:

- We have changed the screen size to 1600×900 so that the entire image can fit on the screen.

- We are now using the 5th iteration of the L-system rather than the 4th.
- We used the function `math.radians` to convert 25°into radians.

When we run this script, here is the result.

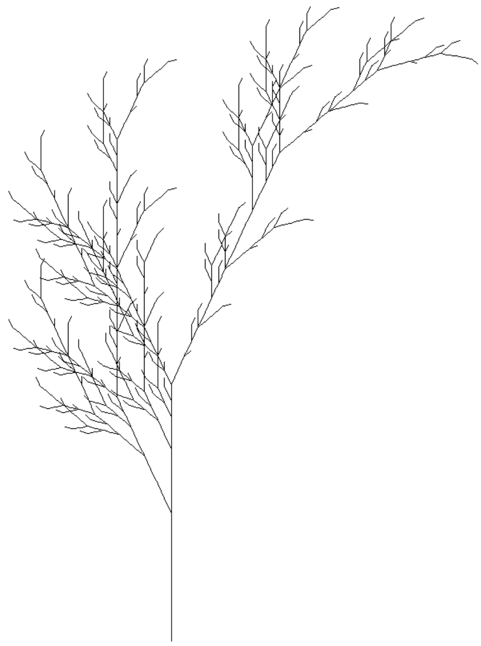


Figure 2.14: L-System Fractal Plant

As you can see, the ruleset and axiom of the L-system really make a difference to how the image looks. As such we will make a few changes to our code so that it is easier to draw the output of an arbitrary L-system, without having to modify the `draw` function each time.

2.2.5 Generalising the Drawing Function

The first change we need to make is to how the x and y coordinates, the angle and the stack are stored. Instead of these being separate variables local to the `draw` function, we will instead encode them each as attributes of the same instance of a class `State`, named as such because instances of the class represent the state of the pen in the drawing. This class is fairly simple to create.

```
class State:
    def __init__(self, x, y, angle):
        self.x = x
        self.y = y
        self.angle = angle
```

```
        self.stack = []
```

Note that we do not need to pass the stack to the constructor because we will always want to initialise it as empty. We will now create several functions – not as part of the `State` class but simply as global parts of the script. The functions we will create are called `forward`, `rotate`, `push` and `pop`. Feel free to create more of these as you see fit but for now we will only create these ones as they are usually the only actions which an L-system drawing requires. Each of these functions will accept a `State` object as a parameter as well as sometimes some additional parameters. The idea is that the function will modify the attributes of the `State` instance as necessary.

We will start with `forward`. In addition to a `State` instance it will accept a number `distance` which represents the distance to move.

```
def forward(state, distance):
    state.x += math.cos(state.angle) * distance
    state.y -= math.sin(state.angle) * distance
    return True
```

The reason why we return `True` at the end is to indicate that after completing this operation, a line should be drawn from the previous position to the current position. If we had returned `False`, then this would indicate not to draw a line.

We will now implement the `rotate` function, which is quite simple.

```
def rotate(state, angle):
    state.angle += angle
    return False
```

Now we can add `push`.

```
def push(state):
    state.stack.append([state.x, state.y, state.angle])
    return False
```

Finally we can implement `pop`.

```
def pop(state):
    state.x, state.y, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]
    return False
```

Now we can create a data structure which maps a certain character in the recipe with a list of those actions. In the last example, F should correspond to a call to `forward(state, 10)` and so on. At first it may seem like such a data structure could simply be a dictionary where the keys are characters and the corresponding values are lists of these four functions. However, this does not work. Imagine we set up the data structure like this:

```
actions = {"F": [forward],
           "+": [rotate],
           "-": [rotate],
           "[": [push],
           "]": [pop]}
```

This seems at first like it would make sense, but it turns out that it doesn't contain enough information. For example if we wanted F to mean move forward 10 pixels like we had before, we would need to pass both the `State` instance *and* the value 10 to the `forward` function. However, this call is actually going to come from within the `draw` function, where the state is known but not the desired length. Similarly this data structure does not contain enough information to differentiate between the action corresponding to $+$ and that to $-$. Luckily Python has some handy syntax to help us overcome this problem.

Python's `lambda` keyword allows us to create functions inline. To use a contrived example, if we wanted to create a function named `greeting` which returned `"Hello, my name is "` followed by the input to the function, the standard way to achieve that would be as follows:

```
def greeting(x):
    return "Hello, my name is "+x
```

However, an equivalent thing to write would be:

```
greeting = lambda x: "Hello, my name is "+x
```

This syntax is very handy in the context of our L-system because it allows us to do something like this:

```
moveForwardByTen = lambda state: forward(state, 10)
```

Where now `moveForwardByTen` is a function with a single parameter, `state`, and when it is called it makes a nested call to `forward`, passing in an additional parameter (10), and returns whatever value is returned thereby. Instead of assigning this to a variable, we can instead place it directly into the `actions` data structure such that it now looks like this:

```
actions = {"F": [lambda state: forward(state, 10)],
           "+": [lambda state: rotate(state, math.radians(25))],
           "-": [lambda state: rotate(state, -math.radians(25))],
           "[": [push],
           "]": [pop]}
```

Note that we do not have to come up with `lambda` wrappers for `push` or `pop` because those do not take any additional parameters. Now each character is mapped to a list of functions which only take a single parameter – `state` – and output a boolean value which indicates whether or not to draw a line.

Now we need to redesign our `draw` function to work with this `actions` dictionary. This unfortunately requires rewriting the entire function from scratch. We will make it accept an additional parameter, `actions`, which is of the form of the aforementioned dictionary. It will then create a `State` instance from its parameters.

```
def draw(screen, recipe, actions, angle, base):
    state = State(*base, angle)
```

Note that the syntax here (`*base, angle`) is equivalent to writing (`base[0], base[1], angle`) but is more succinct. We will now iterate over every character, `instruction`, in the string `recipe`. If `instruction` is one of the keys in the `actions` dictionary, then iterate over each element in the corresponding list (each of which will be a function which requires a single parameter, `state`) and invoke it, passing `state` to it. The `draw` function now looks like this:

```
def draw(screen, recipe, actions, angle, base):
    state = State(*base, angle)

    for instruction in recipe:
        if instruction in actions:
            for action in actions[instruction]:
                action(state)
```

This is not quite enough though as we still need to draw some lines. To do this, for each instruction which is indeed a key in the `actions` dictionary, before iterating over its actions we need to store the current x and y coordinates in case we need to draw a line from them. We can also create a boolean variable named `drawLine` which at first will be set to `False`. Then, as we go through each action, if any of them return `True`, then we set `drawLine` to `True`. Another way of stating this is that we set `drawLine = drawLine or action(state)`. This means that if either `drawLine` is already `True` or if the call to `action(state)` returns `True`, then we set `drawLine` to `True`. Otherwise, it stays `False`. An equivalent but even more succinct way of expressing this is `drawLine |= action(state)`. After this iteration over all the actions, if `drawLine` is `True`, then we draw a line between the old position and the current one. The `draw` function now looks like this:

```
def draw(screen, recipe, actions, angle, base):
    state = State(*base, angle)

    for instruction in recipe:
        if instruction in actions:
            oldPoint = (state.x, state.y)
            drawLine = False

            for action in actions[instruction]:
                drawLine |= action(state)

            if drawLine:
                pygame.draw.line(screen,
                             (0, 0, 0),
                             oldPoint,
                             (state.x, state.y),
                             1)
```

The next change is a simple one. Inside the game loop where we call `draw`, we need to modify the call to also pass in the `actions` dictionary.

```
    draw(screen, recipe, actions, math.pi/2, (width/2, height))
```

Our script now looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem
```

```

class State:
    def __init__(self, x, y, angle):
        self.x = x
        self.y = y
        self.angle = angle
        self.stack = []


def forward(state, distance):
    state.x += math.cos(state.angle) * distance
    state.y -= math.sin(state.angle) * distance
    return True

def rotate(state, angle):
    state.angle += angle
    return False

def push(state):
    state.stack.append([state.x, state.y, state.angle])
    return False

def pop(state):
    state.x, state.y, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]
    return False

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "X"
ruleset = {"X": "F+[[X]-X]-F[-FX]+X",
           "F": "FF"}

actions = {"F": [lambda state: forward(state, 10)],
           "+": [lambda state: rotate(state, math.radians(25))],
           "-": [lambda state: rotate(state, -math.radians(25))],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(5)

def draw(screen, recipe, actions, angle, base):
    state = State(*base, angle)

    for instruction in recipe:
        if instruction in actions:
            oldPoint = (state.x, state.y)
```

```
                drawLine = False

                for action in actions[instruction]:
                    drawLine |= action(state)

                if drawLine:
                    pygame.draw.line(screen,
                                     (0, 0, 0),
                                     oldPoint,
                                     (state.x, state.y),
                                     1)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    draw(screen, recipe, actions, math.pi/2, (width/2, height))

    pygame.display.update()
```

If we run it, we see the same result as before.

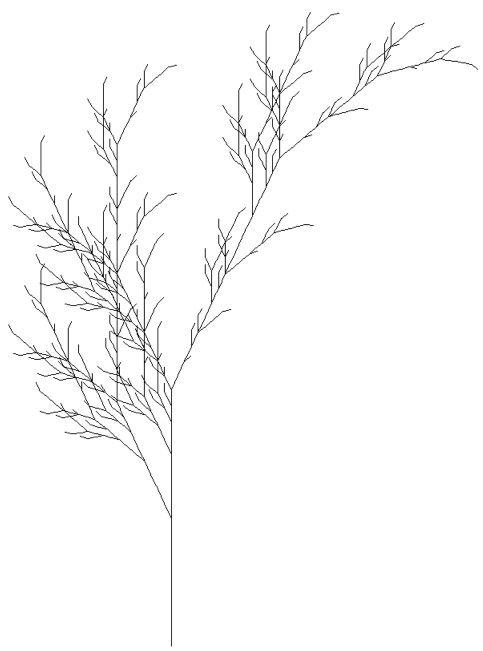


Figure 2.15: L-System Fractal Plant using the Generalised Procedure

Though the output looks the same, we made this change for a very good reason. Namely, so that it is much easier to modify the structure of the L-system. For example, if we want to go back to the binary fractal tree L-system, we only need to change the `axiom`, `ruleset` and `actions` variables to

```
axiom = "0"
ruleset = {"0": "1[0]0",
           "1": "11"}

actions = {"0": [lambda state: forward(state, 10)],
           "1": [lambda state: forward(state, 10)],
           "[": [push,
                 lambda state: rotate(state, math.pi/4)],
           "]": [pop,
                 lambda state: rotate(state, -math.pi/4)]}
```

If we run the program now, we will see a binary tree, without having to modify the `draw` function.

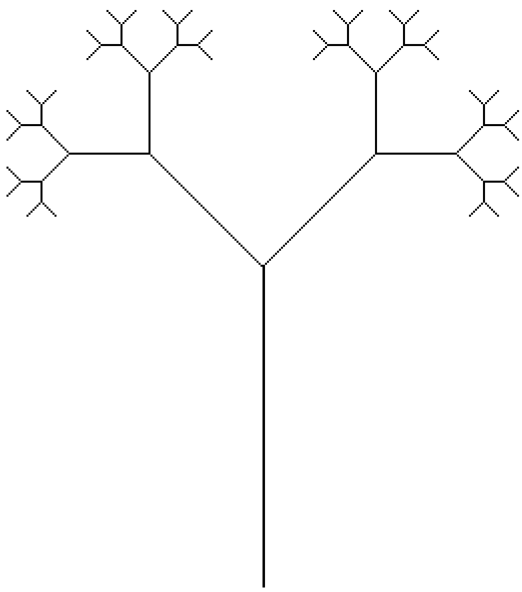


Figure 2.16: An L-System Binary Fractal Tree using the Generalised Procedure

Furthermore we can easily upgrade to a new (and my personal favourite) L-system – specifically one whose axiom is F and which has a single rule:

$$F \longrightarrow FF + [+F - F - F] - [-F + F + F]$$

Here, F means move forwards 15 pixels, $+$ and $-$ mean rotate $\frac{\pi}{6}$ radians clockwise and anticlockwise respectively, [means push and] means pop. We simply need to replace the appropriate variables with:

```
axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [lambda state: forward(state, 15)],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}
```

We will also use the 4th iteration of the L-system rather than the 5th

```
recipe = lsystem.generate(4)
```

Our code now looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem

class State:
    def __init__(self, x, y, angle):
        self.x = x
        self.y = y
        self.angle = angle
        self.stack = []


def forward(state, distance):
    state.x += math.cos(state.angle) * distance
    state.y -= math.sin(state.angle) * distance
    return True

def rotate(state, angle):
    state.angle += angle
    return False

def push(state):
    state.stack.append([state.x, state.y, state.angle])
    return False

def pop(state):
    state.x, state.y, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]
```

```python
    return False

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [lambda state: forward(state, 15)],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

def draw(screen, recipe, actions, angle, base):
    state = State(*base, angle)

    for instruction in recipe:
        if instruction in actions:
            oldPoint = (state.x, state.y)
            drawLine = False

            for action in actions[instruction]:
                drawLine |= action(state)

            if drawLine:
                pygame.draw.line(screen,
                                 (0, 0, 0),
                                 oldPoint,
                                 (state.x, state.y),
                                 1)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    draw(screen, recipe, actions, math.pi/2, (width/2, height))

    pygame.display.update()
```

If we run it, the result looks like this:

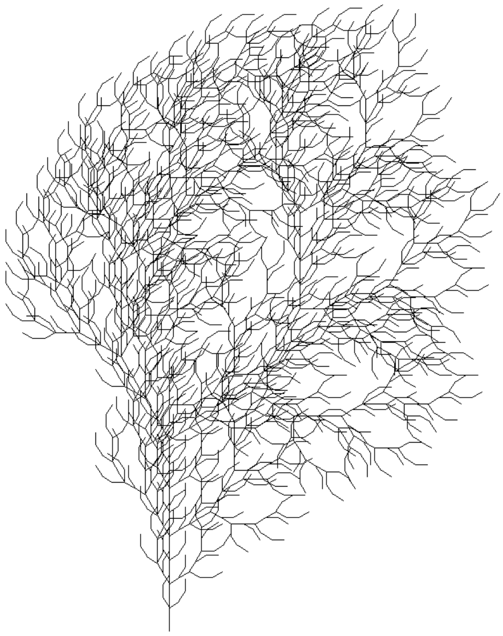


Figure 2.17: An L-System Fractal Tree

For most of the remainder of this section, we are going to be working with this particular L-system. However, I hope that the way we have implemented both the creation and the drawing of the L-systems makes it easy to experiment and be creative with exactly which L-system you want to use.

The next change I want to make to this image is to add a splash of colour. However, before doing this it makes sense to restructure the program to be more object oriented.

2.2.6 Object Oriented Method

Though it may seem that the program is already object oriented (we use the `LSystem` and `State` classes amongst others), the process for converting the recipe into an image is still procedural. This is to say that the entire finished tree is never stored as a manipulable object in memory. The first step in fixing this is to define a `Branch` class. We will define this underneath the definition for the `State` class. The constructor will accept the angle this branch makes with the horizontal, and will store this as an attribute. Much like the binary fractal tree we implemented in the previous section, each `Branch` instance will have a list of its children which by default will be empty.

```
class Branch:
    def __init__(self, angle):
        self.angle = angle
        self.children = []
```

We can also define another class named Tree. The constructor will accept as parameters the recipe string, the actions dictionary, an initial angle from the horizontal and a base position. For now, all the constructor will do is set the base position as its own attribute.

```
class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
```

We now need to modify the State class so that instead of having x and y attributes, it has a branch attribute which holds an instance of the Branch class.

```
class State:
    def __init__(self, branch, angle):
        self.branch = branch
        self.angle = angle
        self.stack = []
```

Now we need to modify the push and pop functions to reflect this change. We can also remove the return statements because they will no longer be necessary.

```
def push(state):
    state.stack.append([state.branch, state.angle])

def pop(state):
    state.branch, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]
```

We can also remove the return statement from rotate.

```
def rotate(state, angle):
    state.angle += angle
```

The forward function is a little harder to modify. Instead of working out new x and y co-ordinates, we can simply create a new Branch object. Its angle will be the current angle (state.angle). Once we've created this object, we can add it to the current branch's list of children (state.branch.children) and then set the current branch to the new branch. As such the forward function no longer requires a distance parameter.

```python
def forward(state):
    newBranch = Branch(state.angle)
    state.branch.children.append(newBranch)
    state.branch = newBranch
```

This is why we don't need the `return` statements anymore: If a new branch needs to be added (which is the OOP equivalent of a line being drawn) then this is handled by the function itself, not whatever called it.

Now we can return to the `Tree` constructor. This will be very similar to the `draw` function. First we set `self.root` equal to a new `Branch` object whose angle is equal to that which was passed to the constructor. We will also create a `State` object whose branch is equal to the root branch we've just created and whose angle is also that which was passed to the `Tree` constructor.

```python
class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle)
        self.state = State(self.root, angle)
```

Now, like in `draw` we iterate over each character, `instruction`, in the `recipe` string. If `instruction` is one of the keys of the `actions` dictionary, iterate over each element of the corresponding list of actions and invoke each one, passing it the `state` variable. The `Tree` class now looks like this:

```python
class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle)
        self.state = State(self.root, angle)

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)
```

We can now delete the `draw` function. Instead, after defining the `recipe` string before the game loop, we can construct a `Tree` object. We can pass the `recipe` string, the `actions` dictionary, $\frac{\pi}{2}$ and the coordinates for the centre of the bottom edge of the screen to the constructor.

```
tree = Tree(recipe, actions, math.pi/2, (width/2, height))
```

Don't forget that since we modified the `forward` function to only accept one parameter, we must change the reference to it inside `actions`.

```
actions = {"F": [forward],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}
```

We must now go back to the `Branch` class and give it its own `draw` function. As parameters, it will accept a PyGame surface on which to draw, a base position, and a length of line to draw. It will then draw a black line of thickness 1 from this base position to the tip. It will then recursively call the `draw` function of each of its children, passing them its tip to use as their base. The `Branch` class now looks like this:

```
class Branch:
    def __init__(self, angle):
        self.angle = angle
        self.children = []

    def draw(self, screen, base, length):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)

        for child in self.children:
            child.draw(screen, tip, length)
```

Similarly we must give the `Tree` class a `draw` function. This will accept as its parameters a PyGame surface and a branch length. It will call the `draw` function of its root branch. The `Tree` class now looks like this:

```
class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
```

```
        self.root = Branch(angle)
        self.state = State(self.root, angle)

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)

    def draw(self, screen, length):
        self.root.draw(screen, self.base, length)
```

Finally we can go into the game loop and instead of calling `draw` which we deleted, we call `tree.draw`, passing it the screen and a branch length of let's say 15 pixels.

```
    tree.draw(screen, 15)
```

The code is now fully object oriented and the `tree` object is totally constructed and stored in memory before it starts to be drawn. The code looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem

class State:
    def __init__(self, branch, angle):
        self.branch = branch
        self.angle = angle
        self.stack = []

class Branch:
    def __init__(self, angle):
        self.angle = angle
        self.children = []

    def draw(self, screen, base, length):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        pygame.draw.line(screen,
                         (0, 0, 0),
                         base,
                         tip,
                         1)
```

```

        for child in self.children:
            child.draw(screen, tip, length)

class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle)
        self.state = State(self.root, angle)

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)

    def draw(self, screen, length):
        self.root.draw(screen, self.base, length)

def forward(state):
    newBranch = Branch(state.angle)
    state.branch.children.append(newBranch)
    state.branch = newBranch

def rotate(state, angle):
    state.angle += angle

def push(state):
    state.stack.append([state.branch, state.angle])

def pop(state):
    state.branch, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [forward],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

tree = Tree(recipe, actions, math.pi/2, (width/2, height))
```

```python

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.draw(screen, 15)

    pygame.display.update()
```

If we run it, we see the same result as before:

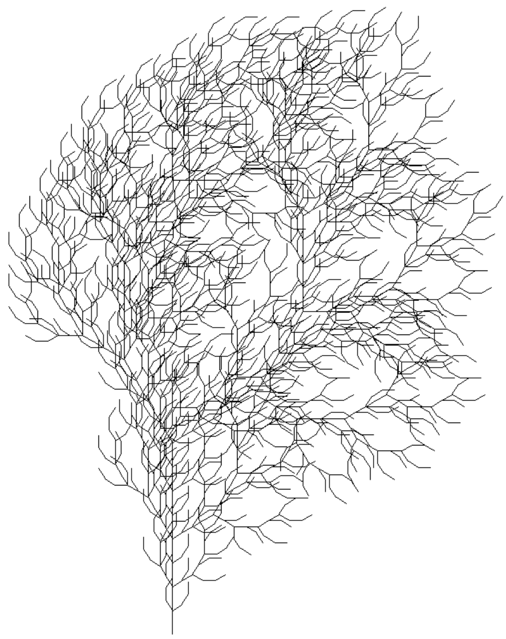


Figure 2.18: An L-System Fractal Tree using the Object Oriented Method

.

2.2.7 Adding a Depth-Based Colour Gradient

It is now much easier to add some colour. We could again draw the leaves as coloured diamonds and draw the branches as brown lines again but I prefer a different method. First we need to know for a given branch how far it is away from the root. To achieve this, we can add another attribute to the `Branch` class named `depth`. This will be passed in via the constructor.

```
    def __init__(self, angle, depth):
        self.angle = angle
        self.depth = depth
        self.children = []
```

Next we need to go to the Tree constructor and pass in 0 as the depth for the root branch.

```
        self.root = Branch(angle, 0)
```

Now we need to go to the forward function. When the new branch is created, pass it the current branch's depth+1.

```
def forward(state):
    newBranch = Branch(state.angle, state.branch.depth+1)
    state.branch.children.append(newBranch)
    state.branch = newBranch
```

Now we need to add a function to the Branch class which tells us the greatest depth reached by any of its descendants. We will call this function max_depth. If the branch has no children, return its own depth. Otherwise, recursively return the greatest value of max_depth for any of its children. It is also helpful to memoize this function as we will be calling it every time the branch is drawn, and it is exponentially recursive. Memoization makes its time complexity linear rather than exponential.

```
    @memoize
    def max_depth(self):
        if len(self.children):
            maxDepth = 0
            for child in self.children:
                maxDepth = max(maxDepth, child.max_depth())
            return maxDepth
        else:
            return self.depth
```

Don't forget that we also need to import memoize from the lsystem module we created so modify your import statement to

```
from lsystem import LSystem, memoize
```

Now in the `Branch.draw` function we can define a variable `p=self.depth/self.max_depth()`. This value will range from 0 at the root to 1 at a leaf. Therefore, instead of drawing the line in black, we can draw it in the RGB colour $240p$, $240p$, $240p$. This will range from black at the root to light grey at the leaves. We will also set the branch thickness to `int(5-4*p)` so that the thickness is 5 at the root and 1 at the leaves. The `Branch` class now looks like this:

```
class Branch:
    def __init__(self, angle, depth):
        self.angle = angle
        self.depth = depth
        self.children = []

    def draw(self, screen, base, length):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        pygame.draw.line(screen,
                         (240*p, 240*p, 240*p),
                         base,
                         tip,
                         int(5-4*p))

        for child in self.children:
            child.draw(screen, tip, length)

    @memoize
    def max_depth(self):
        if len(self.children):
            maxDepth = 0
            for child in self.children:
                maxDepth = max(maxDepth, child.max_depth())
            return maxDepth
        else:
            return self.depth
```

When we run the code we get this result:

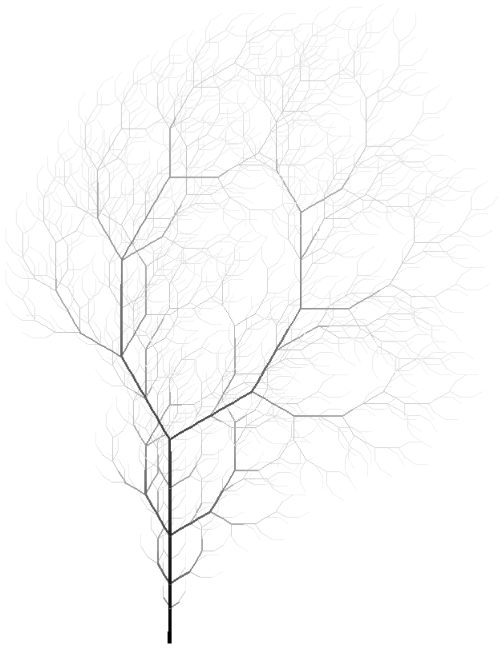


Figure 2.19: An L-System Fractal Tree with Depth Gradient

This already looks much better, but it's still too grey for my taste. Another thing we could do is pick a colour for the leaves and a colour for the root. We can linearly interpolate between the two colours in HSV colour space using `p` as the parameter.

We will therefore create a function named `lerp`. This is a common abbreviation in programming for Linear Interpolation. It will take three numbers as its parameters. The first two, `a` and `b`, are boundaries. The third one, `t`, is a proportion. The function will return the number which is that proportion of the way between the two boundaries.

```
def lerp(a, b, t):
    return a + (b-a)*t
```

Now we will create a second function called `lerp_color` which takes similarly named parameters. However, now `a` and `b` are tuples with 4 elements each (hue, saturation, value and alpha), and we need to return a tuple whose elements are each linearly interpolated with parameter `t`. One way to write this would be:

```
def lerp_color(a, b, t):
    return (lerp(a[0], b[0], t),
            lerp(a[1], b[1], t),
            lerp(a[2], b[2], t),
            lerp(a[3], b[3], t))
```

However, I find it more satisfying and concise to use Python's inline `for` loop syntax.

```
def lerp_color(a, b, t):
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))
```

Now we can go back to the `Branch.draw` function and add in the colours. We will pick a root colour. I like to use HSV 3°, 100%, 100% with a value of 100% for the alpha. For the leaf colour, I like to use HSV 108°, 100%, 100%, again with a value of 100% for the alpha. We then need to create a `pygame.Color` object and set it to black for now. We then set this object's `hsva` attribute with the value returned by linearly interpolating between the root and leaf colours with `p` as our parameter.

```
        rootColor = (3, 100, 27, 100)
        leafColor = (108, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)
```

We can then modify the call to `pygame.draw.line` so that it uses this colour. I also like to change the thickness from `int(5-4*p)` to `int(10-9*p)` because otherwise the leaves are too thin for the colour to really show.

```
        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))
```

Our code now looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem, memoize

def lerp(a, b, t):
    return a + (b-a)*t

def lerp_color(a, b, t):
```

```
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))

class State:
    def __init__(self, branch, angle):
        self.branch = branch
        self.angle = angle
        self.stack = []

class Branch:
    def __init__(self, angle, depth):
        self.angle = angle
        self.depth = depth
        self.children = []

    def draw(self, screen, base, length):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        rootColor = (3, 100, 27, 100)
        leafColor = (108, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)

        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))

        for child in self.children:
            child.draw(screen, tip, length)

    @memoize
    def max_depth(self):
        if len(self.children):
            maxDepth = 0
            for child in self.children:
                maxDepth = max(maxDepth, child.max_depth())
            return maxDepth
        else:
            return self.depth

class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle, 0)
        self.state = State(self.root, angle)

        for instruction in recipe:
```

```
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)

    def draw(self, screen, length):
        self.root.draw(screen, self.base, length)

def forward(state):
    newBranch = Branch(state.angle, state.branch.depth+1)
    state.branch.children.append(newBranch)
    state.branch = newBranch

def rotate(state, angle):
    state.angle += angle

def push(state):
    state.stack.append([state.branch, state.angle])

def pop(state):
    state.branch, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [forward],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

tree = Tree(recipe, actions, math.pi/2, (width/2, height))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.draw(screen, 15)

    pygame.display.update()
```

If we run the program, we will get the following result:

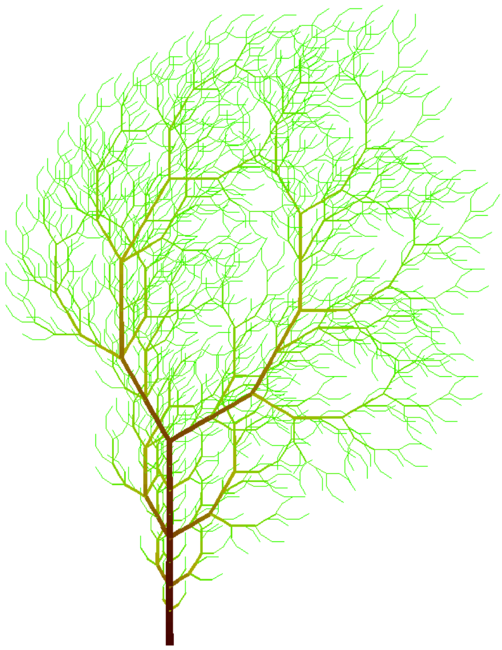


Figure 2.20: An L-System Fractal Tree with Coloured Depth Gradient

While we are certainly improving our tree, we can still do much better. Imagine that every branch had its own leaf colour, and that the branch's colour itself would be the linear interpolation between the shared root colour and its own leaf colour, using `p` as the parameter. The question is now how do we decide the colour for a given branch? We could pick a colour with 100% saturation and value but with a random hue, but if each branch's hue is totally random and independent from the hue of its neighbouring branches than the tree will look discordant and jagged. Instead we can use a *noise field* to vary the hue smoothly (but still randomly) throughout the tree.

To help visualise this, take a look at the following image:

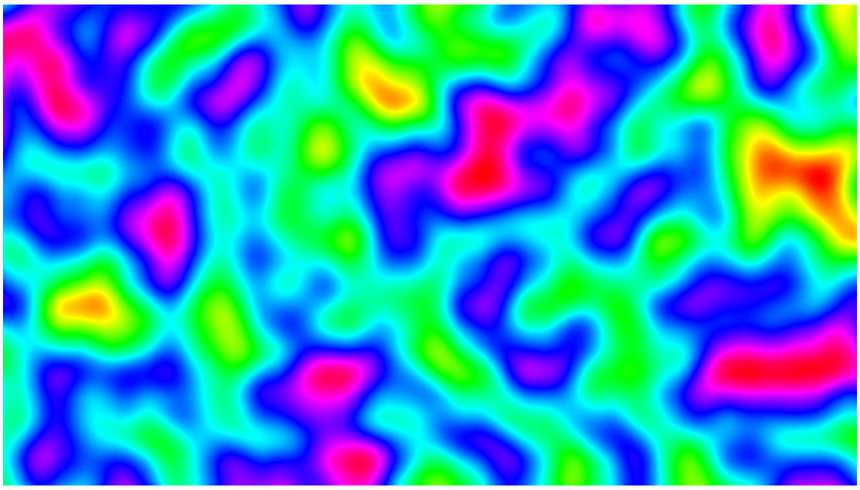


Figure 2.21: Hue Noise Field

You can think of that image as being able to map a pair of coordinates to a hue. As such we will refer to this as a *hue space*. If we treat the x and y coordinates of each branch as the corresponding point in the hue space, then we can assign each branch a random leaf colour while keeping nearby branches similar colours. To construct this hue space, we will use an algorithm called "Perlin noise" which we will implement in the abstract before applying it back to our L-system trees.

2.2.8 Perlin Noise

Perlin noise is an algorithm developed by Ken Perlin in 1983 as the fruit of his efforts to create more natural-looking computer-generated textures. This algorithm is so valuable to the world of computer-generated imagery that Perlin was awarded an Academy Award for Technical Achievement in 1997.

The algorithm allows us to create N-dimensional smooth noise fields, but the time complexity of calculating a noise value for a given set of coordinates is $\mathcal{O}(2^N)$. In 2001 Perlin presented his new algorithm called "Simplex noise" which worked in $\mathcal{O}(N^2)$ time instead, and has fewer directional artefacts. However, since we will only be using 2-dimensional noise, here we will only talk about Perlin noise, not simplex noise.

We want to create a function which takes in two parameters, an x and a y coordinate, and spits out a smoothly varying noise value. The first thing to note about Perlin noise is that if both the x and the y coordinates are integers, the noise value returned will be 0. From here on, we will refer to each of these points with integer coordinates as "*gridpoints*".

Each gridpoint is also assigned a random 2D vector which represents the gradient of the noise function at that point. The general idea is that for any point (x, y) which is not a gridpoint, we use the gradients of the function at the four surrounding gridpoints to work out the value of the noise function.

To start off let's get a new blank script and call it "*noise.py*". Inside this script we will need to import the `math` module as well as the `RandomState` class and the `randint` function from the `numpy.random` package.

```
import math
from numpy.random import RandomState, randint
```

Now we will create the class which will make up our noise field object. We will call the class `Perlin2D`. Its constructor will accept one keyword argument, `seed`, whose default value will be `None`. Within the constructor, if this value is indeed `None` (which means that no value was passed in via the constructor), set the attribute `self.seed` equal to a random integer between 0 and let's say 1,000,000 using the `randint` function. Otherwise, just set the attribute equal to the parameter.

```
class Perlin2D:
    def __init__(self, seed=None):
        if seed is None:
            self.seed = np.random.randint(1000000)
        else:
            self.seed = seed
```

This attribute will be used as the seed for the random number generator we use to create random gradient vectors for the gridpoints. So that we can have different instances of this class with different seeds, we will use an instance of NumPy's `RandomState` class and we will give it the `seed` attribute.

```
        self.rng = np.random.RandomState(self.seed)
```

We will store the gridpoint gradient vectors in a 2-dimensional dictionary (i.e. a dictionary whose values are also dictionaries). The keys of this dictionary will be integer y coordinates, and the corresponding values will be more dictionaries of which the keys are integer x coordinates and the values are tuples with two elements representing the components of the gradient vector. For example, if the gridpoint at coordinates (x, y) had a gradient vector with components (a, b) then the value at `gradients[y][x]` would equal the tuple `(a,b)`.

We will only populate this dictionary as and when we need to, so at first it will be empty. Our `Perlin2D` class now looks like this:

```
class Perlin2D:
```

```
    def __init__(self, seed=None):
        if seed is None:
            self.seed = np.random.randint(1000000)
        else:
            self.seed = seed
        self.rng = np.random.RandomState(self.seed)
        self.gradients = {}
```

Now we will create a function which, given integer x and y coordinates, will return the corresponding gradient function. If this hasn't been set yet, we set it to a random vector with unit length, and then return that. We will call the function `get_gradient`. We check whether the provided y coordinate is not a key in the `gradients` dictionary. In this case, set add this key to the dictionary alongside the value of another blank dictionary.

Next we check whether the x coordinate is not a key in the `gradients[y]` dictionary. In this case, we generate a uniformly distributed random angle between 0 and 2π using the `uniform` function of the `self.rng` object. We then use `math.cos` and `math.sin` to generate a tuple with the components of a random unit vector. We then store this at `gradients[y][x]`. Then we return the value stored there.

```
    def get_gradient(self, x, y):
        if y not in self.gradients:
            self.gradients[y] = {}
        if x not in self.gradients[y]:
            angle = self.rng.uniform(0, 2*math.pi)
            self.gradients[y][x] = (math.cos(angle),
                                    math.sin(angle))
        return self.gradients[y][x]
```

Now we need our main function which will calculate the noise values. We will call this function `noise`. It will accept two parameters, `x` and `y`. The first step is to round down both of these coordinates to integers called `i` and `j` respectively. If `x` and `y` are already integers (which is equivalent to asking whether `x==i and y==j`) then we return 0 as we've previously mentioned.

```
    def noise(self, x, y):
        i = int(x)
        j = int(y)

        if x == i and y == j:
            return 0
```

Otherwise, we need to get the gradient vectors associated with the four gridpoints surrounding (x, y). These will be at coordinates (i, j), $(i + 1, j)$, $(i, j + 1)$ and $(i + 1, j + 1)$. We will store these vectors in variables named `g00`, `g10`, `g01` and `g11` for reasons which are hopefully apparent.

```
        else:
            g00 = self.get_gradient(i  , j  )
            g10 = self.get_gradient(i+1, j  )
            g01 = self.get_gradient(i  , j+1)
            g11 = self.get_gradient(i+1, j+1)
```

We will now define variables `u=x-i` and `v=y-j` such that (u, v) represents the vector between (i, j) and (x, y). All of the following relationships therefore follow:

$$\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$
$$\begin{bmatrix} i+1 \\ j \end{bmatrix} + \begin{bmatrix} u-1 \\ v \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$
$$\begin{bmatrix} i \\ j+1 \end{bmatrix} + \begin{bmatrix} u \\ v-1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$
$$\begin{bmatrix} i+1 \\ j+1 \end{bmatrix} + \begin{bmatrix} u-1 \\ v-1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

As you can see, the vectors involving u and v are the vectors between the surrounding gridpoints and (x, y). We can take the dot product between these vectors and the corresponding gradient vectors. To do this we will quickly implement a global function called `dot` at the top of the script which performs the dot product on two tuples.

```
def dot(a, b):
    return (a[0]*b[0]) + (a[1]*b[1])
```

Now going back to our `noise` function we can perform the dot product to calculate the extent to which the vector between (x, y) and the gridpoints aligns with the gradient function at those gridpoints. We will call these values `n00`, `n10`, `n01` and `n11`.

```
            u = x-i
            v = y-j

            n00 = dot(g00, (u  , v  ))
            n10 = dot(g10, (u-1, v  ))
            n01 = dot(g01, (u  , v-1))
            n11 = dot(g11, (u-1, v-1))
```

The overall noise value will be a linear combination of these four variables. In order to know the ratio in which to combine them, we will use smoothed out values of `u` and `v`. We will smooth out these values by passing them through the following function:

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

Though this function may seem arbitrary, we use this specific function because it is the simplest polynomial which obeys the following rules:

- $f(0) = 0$
- $f(1) = 1$
- $f'(0) = 0$
- $f'(1) = 0$
- $f''(0) = 0$
- $f''(1) = 0$

These properties are what makes sure that the noise function has a well defined derivative and second derivative everywhere.

We therefore define the following variables:

```
            smoothU = 6*u**5 - 15*u**4 + 10*u**3
            smoothV = 6*v**5 - 15*v**4 + 10*v**3
```

As `u` and `v` vary from 0 to 1, so do these respective variables, except these variables do so non-linearly to ensure that there are no jagged edges in the noise field. We now combine `n00` and `n10` in the following ratio to compute the noise value for any point of the form (x, j) such that if `smoothU` is 0 then the result is `n00` and if `smoothU` is 1 then the result is `n10`:

```
            nx0 = n00 * (1-smoothU) + n10 * smoothU
```

We do a similar combination for points of the form $(x, j + 1)$.

```
            nx1 = n01 * (1-smoothU) + n11 * smoothU
```

Finally, we combine these two values in a similar ratio but instead using `smoothV` to get a noise value for the point (x, y)

```
            nxy = nx0 * (1-smoothV) + nx1 * smoothV
```

Finally, we return this value. Our entire script now looks like this:

```
import math
from numpy.random import RandomState, randint

def dot(a, b):
    return (a[0]*b[0]) + (a[1]*b[1])

class Perlin2D:
    def __init__(self, seed=None):
        if seed is None:
            self.seed = np.random.randint(1000000)
        else:
            self.seed = seed
        self.rng = np.random.RandomState(self.seed)
        self.gradients = {}

    def get_gradient(self, x, y):
        if y not in self.gradients:
            self.gradients[y] = {}
        if x not in self.gradients[y]:
            angle = self.rng.uniform(0, 2*math.pi)
            self.gradients[y][x] = (math.cos(angle),
                                    math.sin(angle))
        return self.gradients[y][x]

    def noise(self, x, y):
        i = int(x)
        j = int(y)

        if x == i and y == j:
            return 0
        else:
            g00 = self.get_gradient(i  , j  )
            g10 = self.get_gradient(i+1, j  )
            g01 = self.get_gradient(i  , j+1)
            g11 = self.get_gradient(i+1, j+1)

            u = x-i
            v = y-j

```

```
            n00 = dot(g00, (u  , v  ))
            n10 = dot(g10, (u-1, v  ))
            n01 = dot(g01, (u  , v-1))
            n11 = dot(g11, (u-1, v-1))

            smoothU = 6*u**5 - 15*u**4 + 10*u**3
            smoothV = 6*v**5 - 15*v**4 + 10*v**3

            nx0 = n00 * (1-smoothU) + n10 * smoothU
            nx1 = n01 * (1-smoothU) + n11 * smoothU

            nxy = nx0 * (1-smoothV) + nx1 * smoothV

            return nxy
```

We are now done implementing 2D Perlin noise. Given an x and a y coordinate, we can generate a smoothly varying noise value. The noise values tend to be quite concentrated in the range $[-0.6, 0.6]$ but these are not strict bounds.

Now we can go back to our L-system code and use this to generate colours.

2.2.9 Generating Colours with Perlin Noise

We left off with the code looking like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem, memoize

def lerp(a, b, t):
    return a + (b-a)*t

def lerp_color(a, b, t):
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))

class State:
    def __init__(self, branch, angle):
        self.branch = branch
        self.angle = angle
        self.stack = []

class Branch:
    def __init__(self, angle, depth):
```

```
        self.angle = angle
        self.depth = depth
        self.children = []

    def draw(self, screen, base, length):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        rootColor = (3, 100, 27, 100)
        leafColor = (108, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)

        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))

        for child in self.children:
            child.draw(screen, tip, length)

    @memoize
    def max_depth(self):
        if len(self.children):
            maxDepth = 0
            for child in self.children:
                maxDepth = max(maxDepth, child.max_depth())
            return maxDepth
        else:
            return self.depth

class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle, 0)
        self.state = State(self.root, angle)

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)

    def draw(self, screen, length):
        self.root.draw(screen, self.base, length)

def forward(state):
    newBranch = Branch(state.angle, state.branch.depth+1)
    state.branch.children.append(newBranch)
```

```
    state.branch = newBranch

def rotate(state, angle):
    state.angle += angle

def push(state):
    state.stack.append([state.branch, state.angle])

def pop(state):
    state.branch, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [forward],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

tree = Tree(recipe, actions, math.pi/2, (width/2, height))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.draw(screen, 15)

    pygame.display.update()
```

Now we need to import our Perlin noise field class:

```
from noise import Perlin2D
```

We now want to go to our `Tree` constructor and give add a noise field attribute named `noiseField`, setting it equal to a new instance of `Perlin2D`. Now the constructor looks like this:

```
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle, 0)
        self.state = State(self.root, angle)
        self.noiseField = Perlin2D()

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)
```

We can now modify the `Branch.draw` function to accept as a parameter the `Tree` object of which the branch is a part, and also to pass the same variable when it calls the `draw` function of its children.

```
class Branch:
    def __init__(self, angle, depth):
        self.angle = angle
        self.depth = depth
        self.children = []

    def draw(self, screen, base, length, tree):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        rootColor = (3, 100, 27, 100)
        leafColor = (108, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)

        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))

        for child in self.children:
            child.draw(screen, tip, length, tree)
```

We also need to modify the call to `root.draw` inside the `Tree.draw` function such that it passes in this variable:

```
    def draw(self, screen, length):
        self.root.draw(screen, self.base, length, self)
```

Finally, in `branch.draw` we can access the `noiseField` attribute of the tree object passed in as a parameter, and call its `noise` function. As the `x` and `y` coordinates, we can pass in the coordinates of `tip`, but scaled by some value. We will call this value `roughness` because the greater this number is, the more rapidly the noise field will fluctuate in response to changes in x and y. I like to set `roughness = 0.005`. The noise value we produce will be roughly between -0.6 and 0.6, so we can multiply the result by 600 to get a number which is roughly between -360 and 360. We then calculate the congruent value mod 360 (the remainder after dividing by 360) to get a smoothly varying number between 0 and 360. We will use this as the hue for our `leafColor` tuple. The `draw` function now looks like this:

```
    def draw(self, screen, base, length, tree):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        roughness = 0.005
        hue = (600*tree.noiseField.noise(tip[0]*roughness,
                                         tip[1]*roughness))%360

        rootColor = (3, 100, 27, 100)
        leafColor = (hue, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)

        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))

        for child in self.children:
            child.draw(screen, tip, length, tree)
```

As a whole, the script looks like this:

```
import pygame
import sys
import math
from pygame.locals import *
from lsystem import LSystem, memoize
from noise import Perlin2D

def lerp(a, b, t):
    return a + (b-a)*t

def lerp_color(a, b, t):
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))

class State:
    def __init__(self, branch, angle):
        self.branch = branch
        self.angle = angle
        self.stack = []

class Branch:
    def __init__(self, angle, depth):
        self.angle = angle
        self.depth = depth
        self.children = []

    def draw(self, screen, base, length, tree):
        tip = (base[0] + math.cos(self.angle) * length,
               base[1] - math.sin(self.angle) * length)

        p = self.depth/self.max_depth()

        roughness = 0.005
        hue = (600*tree.noiseField.noise(tip[0]*roughness,
                                         tip[1]*roughness))%360

        rootColor = (3, 100, 27, 100)
        leafColor = (hue, 100, 100, 100)
        color = pygame.Color(0, 0, 0)
        color.hsva = lerp_color(rootColor, leafColor, p)

        pygame.draw.line(screen,
                         color,
                         base,
                         tip,
                         int(10-9*p))

        for child in self.children:
            child.draw(screen, tip, length, tree)

    @memoize
    def max_depth(self):
```

```
        if len(self.children):
            maxDepth = 0
            for child in self.children:
                maxDepth = max(maxDepth, child.max_depth())
            return maxDepth
        else:
            return self.depth

class Tree:
    def __init__(self, recipe, actions, angle, base):
        self.base = base
        self.root = Branch(angle, 0)
        self.state = State(self.root, angle)
        self.noiseField = Perlin2D()

        for instruction in recipe:
            if instruction in actions:
                for action in actions[instruction]:
                    action(self.state)

    def draw(self, screen, length):
        self.root.draw(screen, self.base, length, self)

def forward(state):
    newBranch = Branch(state.angle, state.branch.depth+1)
    state.branch.children.append(newBranch)
    state.branch = newBranch

def rotate(state, angle):
    state.angle += angle

def push(state):
    state.stack.append([state.branch, state.angle])

def pop(state):
    state.branch, state.angle = state.stack[-1]
    state.stack = state.stack[:-1]

pygame.init()

width, height = (1600, 900)
screen = pygame.display.set_mode((width, height), 0)

axiom = "F"
ruleset = {"F": "FF+[+F-F-F]-[-F+F+F]"}

actions = {"F": [forward],
           "+": [lambda state: rotate(state, -math.pi/6)],
           "-": [lambda state: rotate(state, math.pi/6)],
           "[": [push],
           "]": [pop]}
```

```

lsystem = LSystem(axiom, ruleset)
recipe = lsystem.generate(4)

tree = Tree(recipe, actions, math.pi/2, (width/2, height))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.draw(screen, 15)

    pygame.display.update()
```

Below are some examples of what the result of this program might look like for different values of `roughness`.

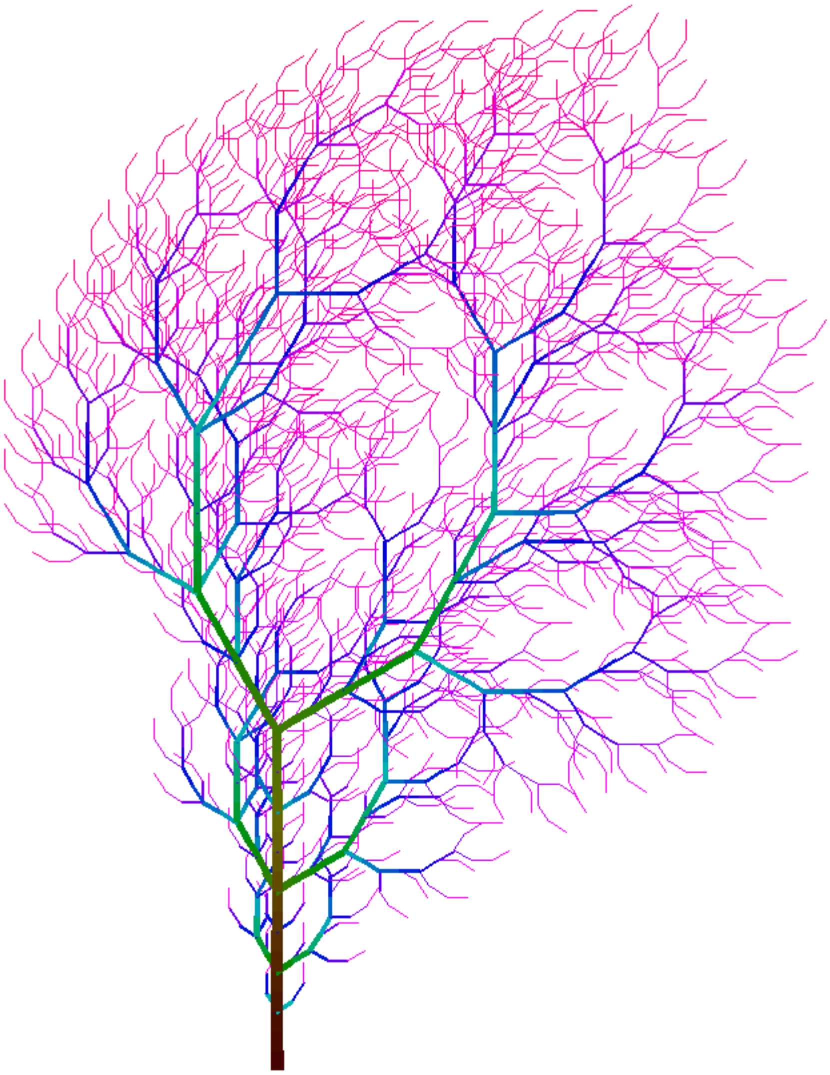


Figure 2.22: An L-System Fractal Tree with Perlin Hues, `roughness=0.0001`

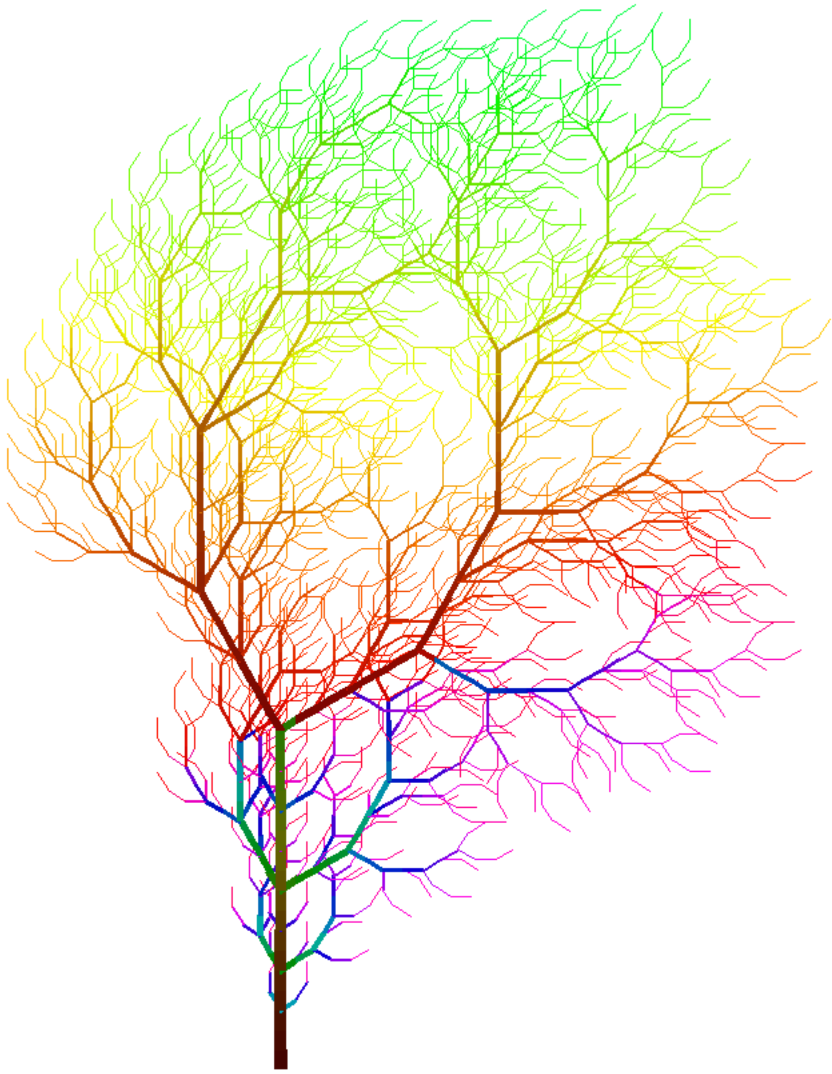


Figure 2.23: An L-System Fractal Tree with Perlin Hues, `roughness=0.0005`

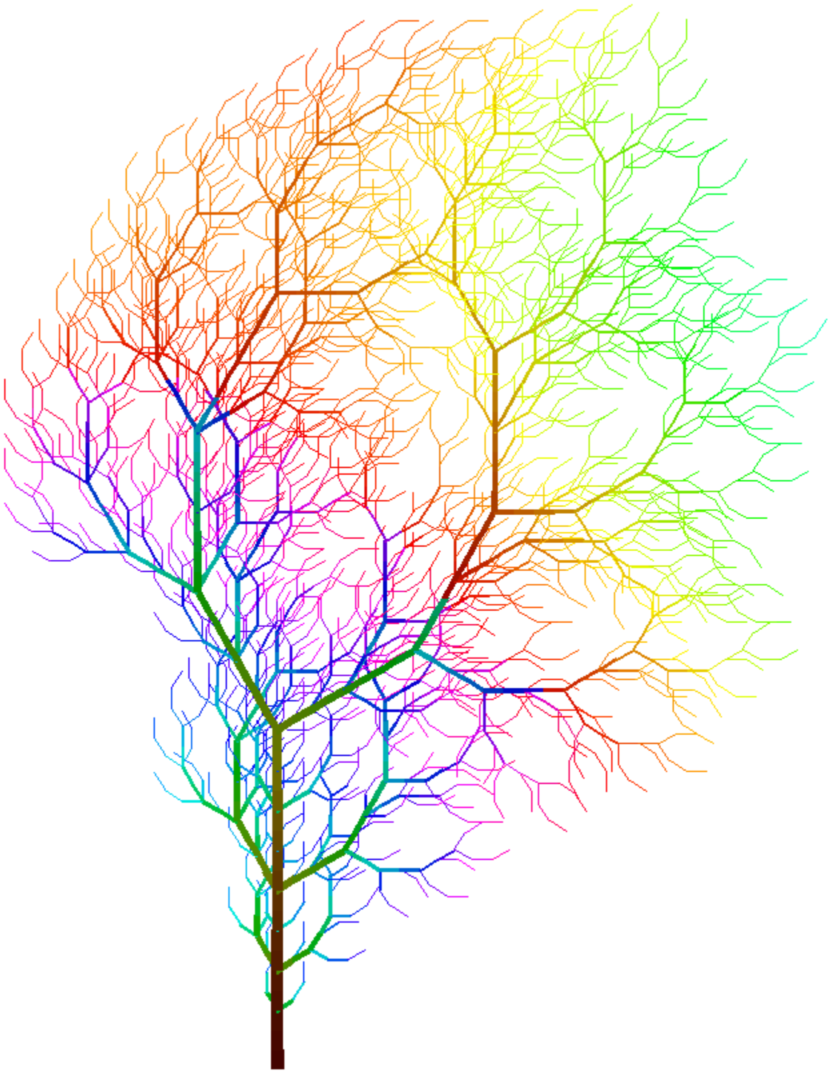


Figure 2.24: An L-System Fractal Tree with Perlin Hues, `roughness=0.001`

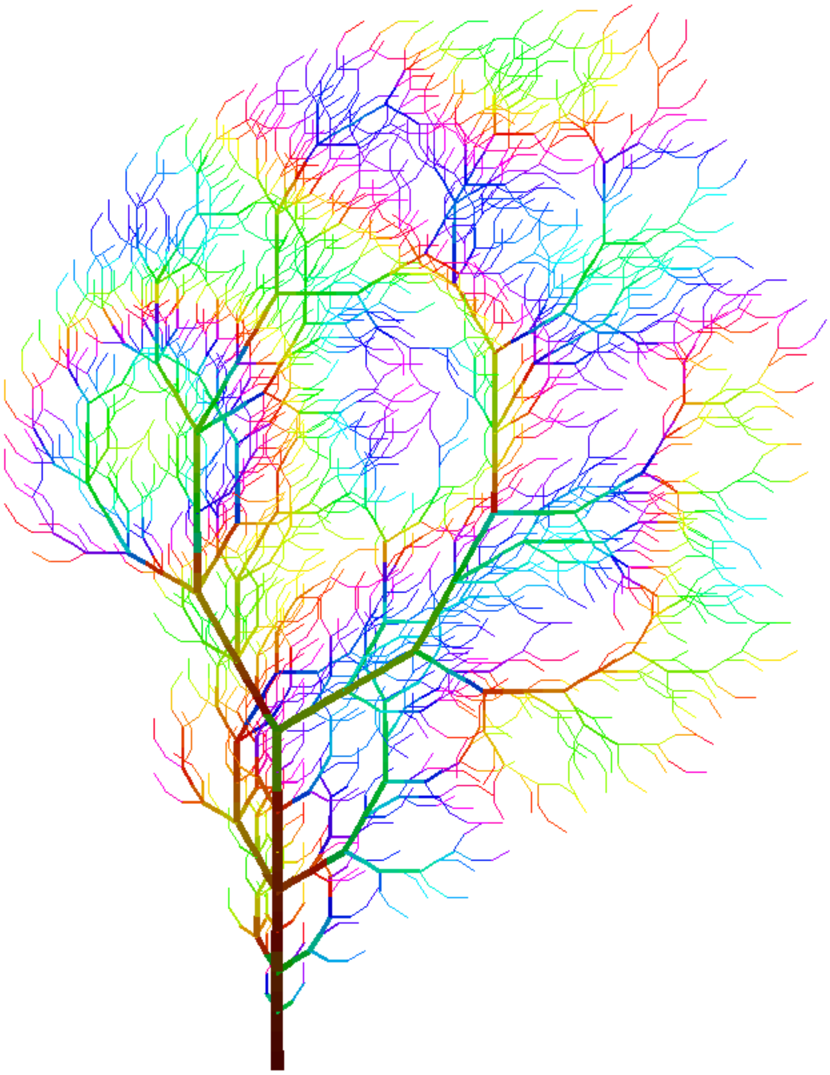


Figure 2.25: An L-System Fractal Tree with Perlin Hues, `roughness`=`0.005`

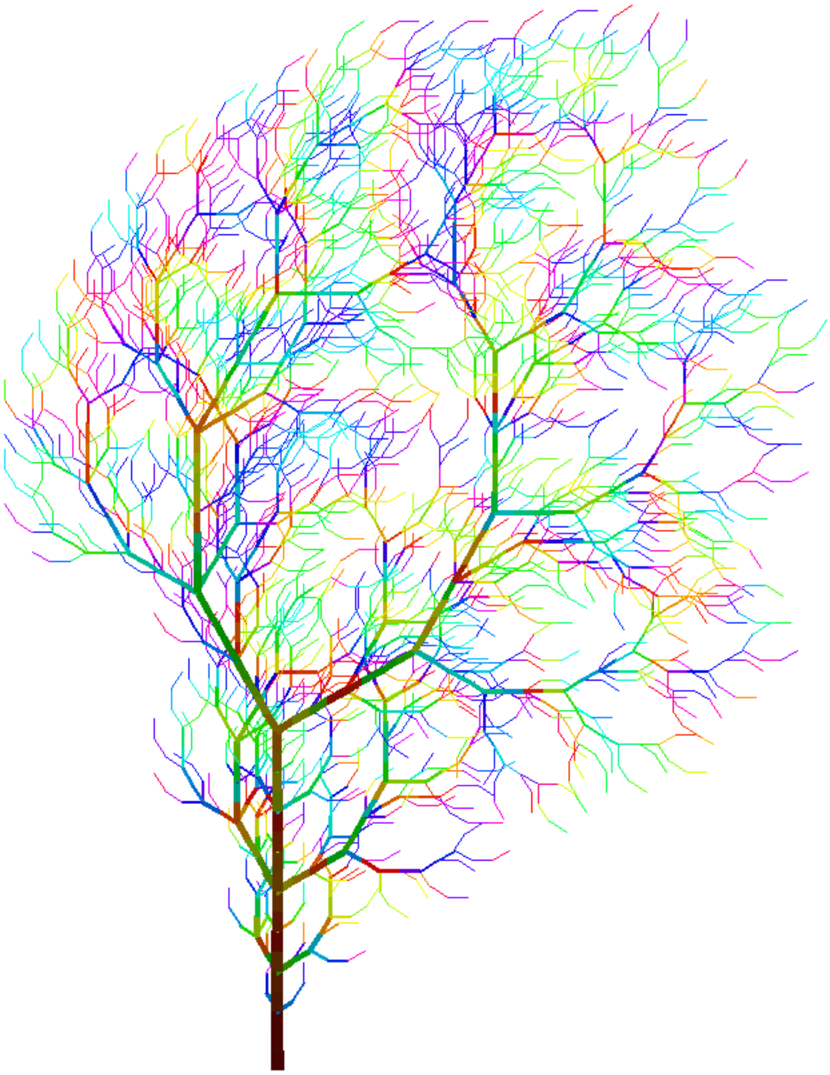


Figure 2.26: An L-System Fractal Tree with Perlin Hues, `roughness=0.01`

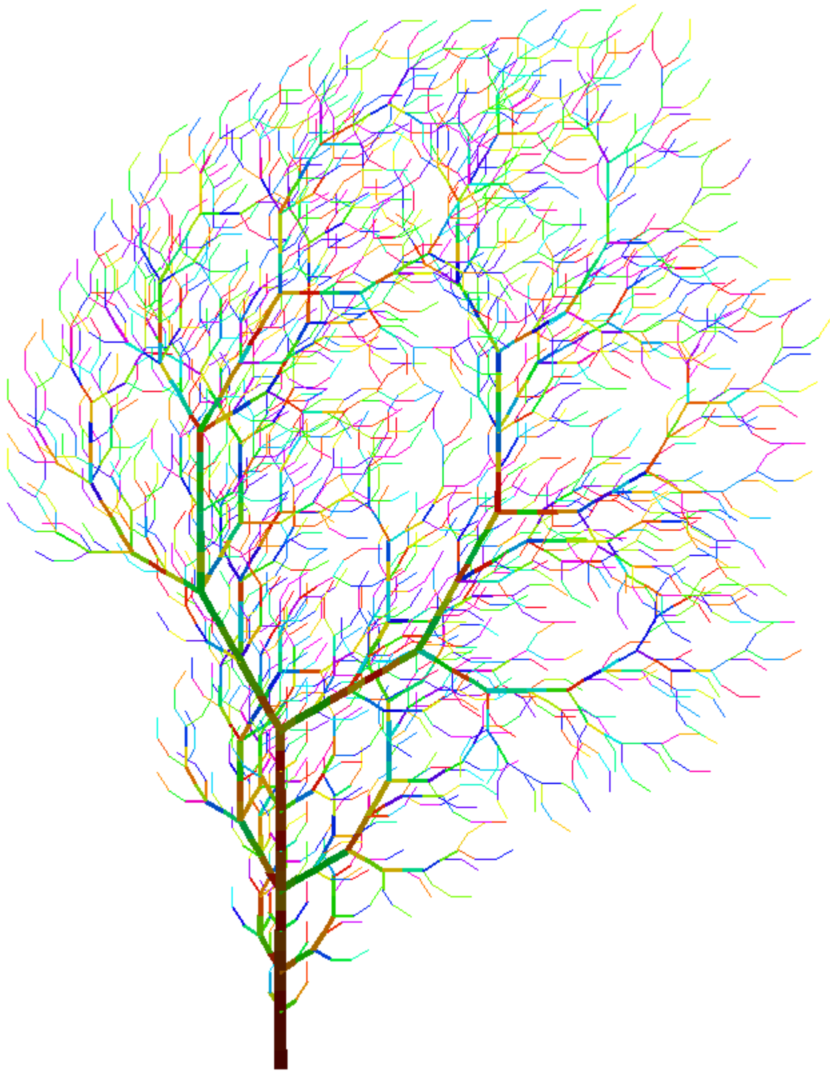


Figure 2.27: An L-System Fractal Tree with Perlin Hues, `roughness=0.05`

We can of course go back and change our `axiom`, `ruleset` and `actions` variables to construct different L-systems in this way. For example, going back to

```
axiom = "X"
ruleset = {"F": "FF",
           "X": "F+[[X]-X]-F[-FX]+X"}

actions = {"F": [forward],
           "+": [lambda state: rotate(state, math.radians(25))],
           "-": [lambda state: rotate(state, -math.radians(25))],
           "[": [push],
           "]": [pop]}

lsystem = LSystem(axiom, ruleset)
```

```
recipe = lsystem.generate(6)
```

Changing the branch length specified in the `tree.draw` function call in the game loop from 15 to 5 so that the entire image can fit on the screen, the result would be something like this:

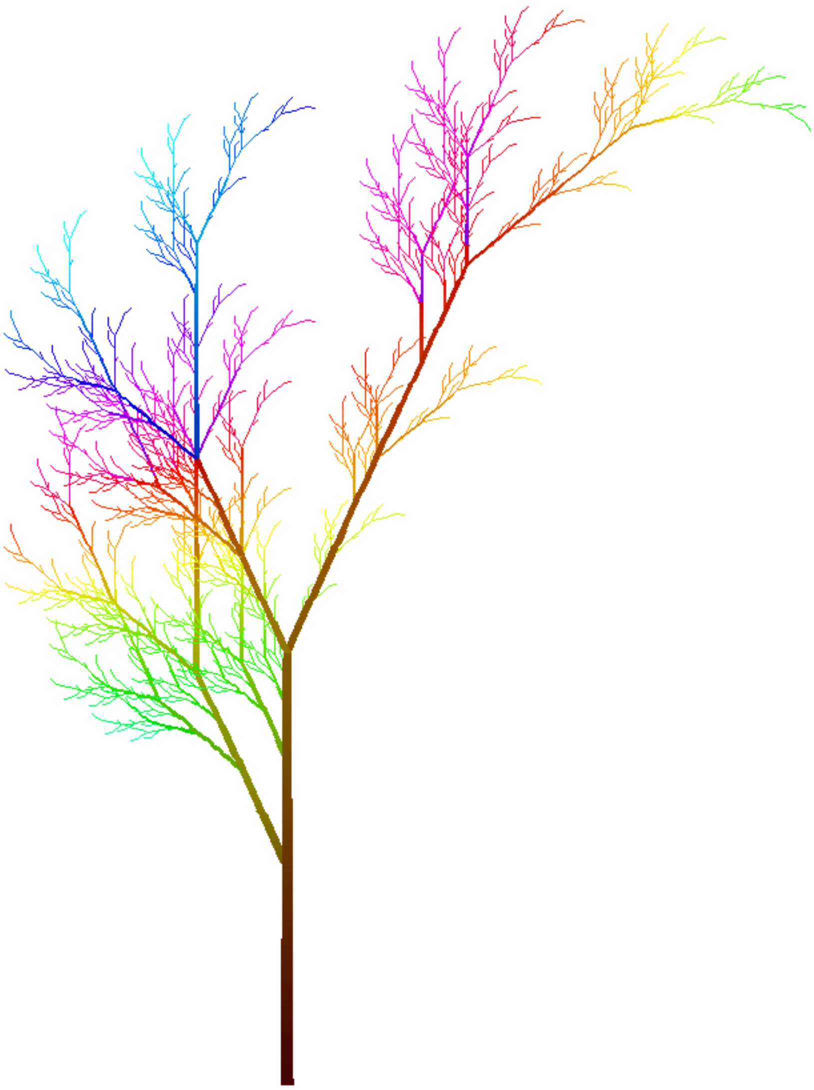


Figure 2.28: An L-System Fractal Plant with Perlin Hues, `roughness=0.001`

This is where we'll leave the L-system project. Here are some ideas for ways you might extend or improve the project:

- We pass the coordinates of `tip` to the noise field to generate a hue value. Instead the coordinates we pass to the field could be relative to the base of the tree. That way, if the entire tree was moving through space, the colours would stay the same.

- Implement an algorithm which will calculate the correct branch length to use such that the entire tree fits within a certain width and height.

- Vary the angles of each branch slightly over time (perhaps using another Perlin noise field) to create the illusion of the tree swaying in the wind.

- Use different L-systems. Many L-systems exist that could even draw 3-dimensional plant-like branching structures.

- Make the colours of the branches change over time, perhaps from green to orange to red, simulating the changing seasons.

- Animate the process of the tree growing from the root to the leaves.

- Non-linear interpolation between the root and leaf colours, or perhaps interpolation through a different colour space.

2.3 Space Colonising Trees

The next algorithm we will implement to generate trees is called "space colonisation". It is profoundly different from the L-system algorithm in two major ways:

- For a given L-system, the same recipe string and therefore a very similar shaped tree will be produced each time. Whereas, space colonisation gives us much more control over the exact shape of our tree. For example, we can generate square, circular and even heart shaped trees.

- With L-systems, we specify a point for the base of the tree and the rest of the tree grows outwards from there. With space colonisation, we also select a set of points in the vicinity of where we want the leaves to end up, and using pseudo-physics simulations, the tree will grow towards them. I say pseudo-physics because we will refer to these leaf points as exerting a "force" on the branches of the tree but this may not necessarily follow the actual laws of physics.

Before we can get started implementing the space colonisation algorithm itself, we need to do some housekeeping first. We need to create several helper classes and functions so that once we start to implement the actual growth of the tree, we'll be able to visualise what is actually going on.

2.3.1 Housekeeping

This algorithm is largely geometric, so the first thing we will do is create a `Vector` class. Its constructor will accept x and y components which it will assign as parameters.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

We will be doing an awful lot of arithmetic with these objects, so it'll make the rest of our code much cleaner if we overload some of this class's dunder functions. For now, the only ones we need to overload are the following (note that in the following examples, `v1` and `v2` are instances of the `Vector` class and `k` is a number):

- Addition: `__add__` so we can use `v1 + v2`.
- In-place addition: `__iadd__` so we can use `v1 += v2`.
- Subtraction: `__sub__` so we can use `v1 - v2`.
- Division: `__truediv__` so we can use `v1 / k`.
- In-place division: `__idiv__` so we can use `v1 /= k`.

If you extend this project beyond what we do in this chapter, you may need to overload some more dunders but for now these are all we need.

It is also important to note the distinction between the `__truediv__` dunder and the `__floordiv__` dunder. The former is for floating point (i.e. non-integer) division and refers to the / operator, and the latter is for rounding the result of the division down to the next integer and refers to the // operator.

It is fairly simple to overload these dunders and so here is what our `Vector` class looks like:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

```

```
    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        return self
```

We can now create our `Leaf` class. Leaves will be represented by points on the screen which exert forces on the branches of the tree, causing the tree to grow towards them. However, they are just for guidance, and the tree may or may not actually end up reaching them. The `Leaf` constructor will accept a `Vector` object containing its position as a parameter, and will assign this to an attribute

```
class Leaf:
    def __init__(self, pos):
        self.pos = pos
```

The entire point of object oriented programming is that we can delegate objects to be responsible for solving certain parts of the problem. We made the `Leaf` constructor require the position of the leaf, but I think it should be the leaf's responsibility to decide on a random position for itself. As such we will create a static class function called `Leaf.random`. The descriptor "static" here means that it should not be used as a function of a specific instance of the `Leaf` class, but as a function of the `Leaf` class as a whole. It does not accept `self` as one of its parameters, as it is not referring to any one instance.

We will make this function pick a random x and y coordinate within a rectangle and return a new `Leaf` instance at this point. The position and size of the rectangle will be specified via the parameters to the function: a vector, `pos`, which specifies the position of the top left corner of the rectangle, and a vector, `size`, which specifies the width and height. Since we need uniformly distributed random numbers, don't forget to add `from numpy.random import uniform` to the top of our script.

```
    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))
```

We could now for example write `leaf = Leaf.random(Vector(0, 0), Vector(100, 100))` and `leaf` would be equal to an instance of the `Leaf` class with random coordinates in the specified rectangle.

We can now start to make our `Branch` class. Its attributes will be its parent branch, its tip position vector, and its direction vector. These will be passed in via the constructor.

```
class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction
```

Now that we've begun to define leaves and branches, we can add another class, `Tree`. Its constructor will accept a base position vector for the first branch and a direction vector in which to start growing (we will call this vector `sprout` because it the direction in which the tree sprouts). We don't actually need to assign these parameters to attributes. Instead we simply use them to create a `Branch` object. We create an attribute named `branches` which is a list containing this object. Since this `Branch` object is the first branch in the tree, it does not have a parent branch, and so we pass in `None` as the first parameter to its constructor.

```
class Tree:
    def __init__(self, base, sprout):
        self.branches = [Branch(None, base, sprout)]
```

As well as this, we will create an attribute named `leaves` which is a list of `Leaf` objects which we will create using `Leaf.random`. We can create a list of let's say 250 such leaves using Python's inline `for` loop syntax. For now let's say (somewhat arbitrarily) that the top left of the rectangle will be at position $(270, 100)$ and the size will be 100×100.

```
class Tree:
    def __init__(self, base, sprout):
```

```
        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))
```

At this point, our code looks like this:

```
from numpy.random import uniform

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        return self

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))

class Branch:
```

```
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

class Tree:
    def __init__(self, base, sprout):
        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))
```

We probably want to start visualising this, and so we can add in the PyGame base code, giving us this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        return self
```

```

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

class Tree:
    def __init__(self, base, sprout):
        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    pygame.display.update()
```

Before the game loop we will create an instance of the `Tree` class. For the base, we will pass a vector corresponding to the middle of the bottom edge of the screen and as the sprouting direction, we will use the vector `Vector(0, -1)` which is directly up.

```
tree = Tree(Vector(width/2, height),
            Vector(0, -1))
```

We will now add a way of drawing what we've got so far to the screen. In `Leaf` we will add a `draw` function which accepts as a parameter a PyGame surface on which to draw and will draw a red dot on the screen at the leaf's position using `pygame.draw.circle`.

The problem is that `pygame.draw.circle` requires the position of the centre of the circle to be a tuple of integers, whereas we have the position of the leaf stored as one of our `Vector` objects. Luckily, we can simply add an attribute to the `Vector` class which stores a tuple of the components, rounded down to integers. We will call this attribute `roundTuple` and it will have to be set in the constructor but also reset in the dunders `__iadd__` and `__idiv__` as these modify the components of the object. Our Vector class now looks like this:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self
```

At last we can go back and add the function `Leaf.draw`.

```
    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
```

```
                                  2,
                                  0)
```

The parameters to the `pygame.draw.circle` function call are in order: the PyGame surface on which to draw, the colour (red is RGB 255, 0, 0), the position of the centre of the circle, the radius (2 pixels), and the thickness of the line (0 means filled).

For style reasons I like to define all of a class's non-static functions above all of its static ones but this is purely an aesthetic preference, not a functional one.

We should add a similar function `draw` to the `Branch` class. It should draw a straight line between its parent's tip and its own tip. We will make the line black and of thickness 1 for now. It is also important to note that we first need to check whether the parent branch is not set to `None`, because if so we cannot draw the branch.

```
    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)
```

Furthermore we can add a `draw` function to the `Tree` class. It will do nothing but call the `draw` function of each of the elements of its `branches` and `leaves` lists.

```
    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)
```

Next we go into the game loop and fill the screen with white, and then call `tree.draw(screen)` to draw the tree each frame.

```
    screen.fill((255, 255, 255))
    tree.draw(screen)
```

Our code now looks like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(topLeft, size):
```

```
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout):
        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.draw(screen)

```

```
        pygame.display.update()
```

If we run it, we should see a result like this:

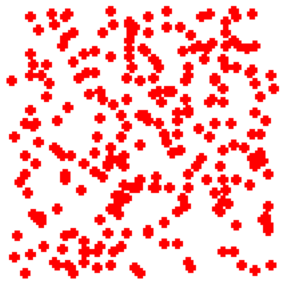


Figure 2.29: 250 Randomly Generated Leaf Points

This seems like a lot of work for a square full of dots, right? Well this has laid the groundwork for implementing the actual tree generation algorithm.

You might also be wondering why we can't see any branches. This is because the only branch we've created is the root of the tree. We set its parent to `None` and so it doesn't get drawn. However all other branches we generate will indeed be drawn.

2.3.2 Tree Growth

Once we have the leaves in place, the growth of the tree is characterised by two parameters – `minDist` and `maxDist`. The relevance of these variables will become apparent soon, but for now we will just make these parameters of the `Tree` class. More specifically, we will accept these parameters as keyword arguments of the constructor, square them, and store their squared values as attributes of the object. The `Tree` constructor now looks like this:

```
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))
```

As you can see, we have used 10 and 100 as the default values for `minDist` and `maxDist` respectively.

There are two stages of the space colonisation algorithm. I like to call them "*extension*" and "*explosion*". We will implement extension first.

Extension

At the moment the tree has one branch which is at the base of the tree, which might be quite far away from the leaves. We will therefore continue to add branches to the tree such that it grows in the direction of the `sprout` vector until a branch gets within `maxDist` of any of the leaves, at which point we will move to the "explosion" stage.

To achieve this we need to add a function to the `Branch` class called `extend`. This function will create a new branch, whose parent is `self`, whose tip is at `self.tip + self.direction` and whose direction is equal to `self.direction`. We will then return this new branch.

```
    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction)
```

There is something very important to note here: When we pass `self.direction` to the child branch, we are not passing in a new vector whose components are the same as `self.direction`. Instead we are passing a reference to the exact same object. This is an issue because it means that if we alter the components of the child branch's direction vector, it will also modify that of the parent because they are referring to the same instance of `Vector`. To remedy this, we will go to the `Vector` class and add a function called `copy`, which will return a *new* vector object whose components are the same as the original.

```
    def copy(self):
        return Vector(self.x, self.y)
```

We will now be able to modify the direction vectors of parents and children independently if we adjust the `Branch.extend` function to:

```
    def extend(self):
        return Branch(self,
```

```
                      self.tip + self.direction,
                      self.direction.copy())
```

Note now that we do not need to copy `self.tip + self.direction` because the dunder we overloaded for addition already returns the result in a new `Vector` object.

We can now go back to the `Tree` constructor and add a boolean attribute which keeps track of whether or not we are in the extension phase, which we will be by default.

```
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True
```

Now we can add a function `Tree.grow` which will first check whether or not we are in the extension phase, and if so, extend the last branch in the list of branches by calling its `extend` function. We can then append the new child `Branch` object returned thereby to the branch list.

```
    def grow(self):
        if self.extending:
            self.branches.append(self.branches[-1].extend())
```

Next we go to the game loop and right before we draw the tree to the screen with `tree.draw(screen)`, we insert a call to `tree.grow`.

```
    tree.grow()
```

The code now looks like this:

```
import pygame
import sys
```

```
import math
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def copy(self):
        return Vector(self.x, self.y)

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)
```

```
        return Leaf(Vector(x, y))

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            self.branches.append(self.branches[-1].extend())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

```

```
tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()
    tree.draw(screen)

    pygame.display.update()
```

If we run it now, we would see this:

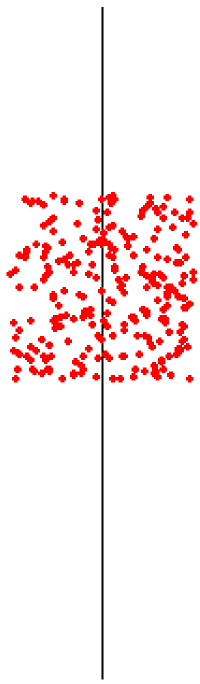


Figure 2.30: Unbounded Extension

This is close to what we want. The branches are indeed extending upwards, but we need them to stop when they get within `maxDist` of any of the leaves.

In most physics engines or games in which the program needs to check and compare the distances between many points, you will find that they rarely compare the distances directly. Instead they compare the distances squared (or the "square distances" as they are more commonly called). This is because in order to work out the distance between two points, Pythagoras'

Theorem tells you that you need to perform this calculation:

$$\text{distance} = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

Unfortunately the square root operation is incredibly slow computationally speaking and if you would need to perform it hundreds or thousands of times then it's best to avoid it altogether. We instead calculate

$$\text{distance}^2 = (\Delta x)^2 + (\Delta y)^2$$

Since x^2 is a strictly increasing function for positive values of x, the statement that the distance between two objects is less than `maxDist` is equivalent to stating that the square distance between them is less than `maxDist` squared. This is why we squared `maxDist` and `minDist` before storing them as attributes of the tree.

So, before extending the most recent branch, we need to iterate over each leaf and get the square distance between it and the branch. If this is less than `sqrMaxDist` then we can set the `extending` flag to `False` and move on to the explosion phase. We therefore modify `tree.grow` as follows:

```
    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = (leaf.pos.x-currentBranch.tip.x) ** 2 + (leaf.pos.y-currentBranch.tip.y) ** 2
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())
```

There are two major things to note here. The first is that if we find a leaf that is close enough to the current branch, we `break` out of the loop. This is because if we do find one, we don't need to keep checking the rest of the leaves.

The other thing to note is that we use Python's somewhat unusual `for`-`else` syntax. The `else` case of a loop gets run when the loop is finished running **if and only if the loop was not broken out of**. If the loop terminated naturally (i.e. for a `while` loop the condition becomes `False` or for a `for` loop the iteration is complete) then the `else` case will be triggered. If a `break` statement is used, the `else` case is skipped over. In this context, we use it to ensure that the `extend` function only gets called if we are still in the extension phase, and so the loop was never broken out of.

The last change we'll make to the extension logic is to reassign the task of working out the square distance between the two points. Since we will have to work out such a distance again in different parts of the code, it makes more sense to have `sqr_dist` be a function of the `Vector` class.

```
    def sqr_dist(self, other):
        return (self.x-other.x)**2 + (self.y-other.y)**2
```

We will therefore change our `Tree.grow` function to use this.

```
    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())
```

We are now done with the implementing the extension phase. Our code looks like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
```

```
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def copy(self):
        return Vector(self.x, self.y)

    def sqr_dist(self, other):
        return (self.x-other.x)**2 + (self.y-other.y)**2

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
```

```
                              self.parent.tip.roundTuple,
                              self.tip.roundTuple,
                              1)

class Tree:
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()
```

```
        tree.draw(screen)

        pygame.display.update()
```

If we run it, we see the following results:

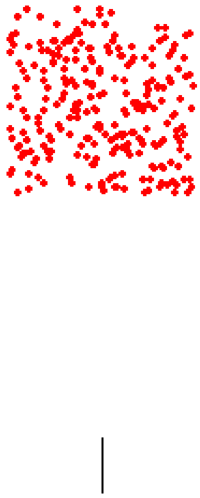


Figure 2.31: Extension at Frame 30

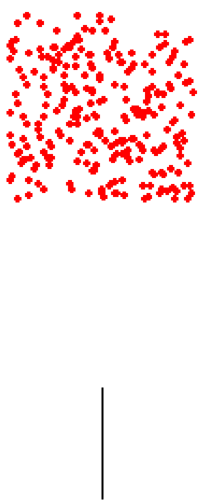


Figure 2.32: Extension at Frame 60

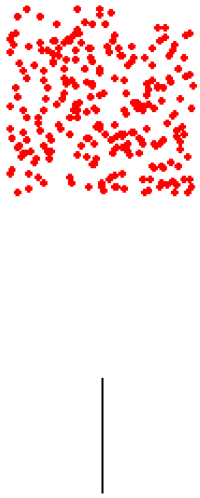


Figure 2.33: Extension at Frame 90

Once the branches get within 100 pixels of the nearest leaf, the extension process stops. We can now move on to implementing the explosion phase.

Explosion

The explosion phase works like this: For each leaf, we find the closest branch to it which is withing the range `minDist` to `maxDist`. If such a branch exists, we apply a "force" to it with a magnitude of 1 and a direction towards the leaf. After this has occurred for all leaves, each branch spawns a child in the direction of the mean average of the forces which have been applied to it this frame. The process then repeats. If any leaf has a branch which is within `minDist` of it, the leaf is then removed from the leaf list. The growth is finished once there is a frame in which no forces are applied.

To start off implementing this, we will give the `Leaf` class a function named `influence`. Its parameters will be a list of branches on which it is allowed to exert forces (if they are within the allowed distance range) and a value of `sqrMinDist` and `sqrMaxDist`. This function can return 1 of 3 things: If it manages to exert a force on a branch, it will return that branch. If it does not, it will return `None`. If it detects a branch which is within `minDist` of it, it will return `False` as an indicator that it should be removed from the leaf list.

The first thing this function will do is perform such a check. Iterate over each branch, and check whether the square distance between it and the branch is less than `sqrMinDist`. If so, return `False`.

```
    def influence(self, branches, sqrMinDist, sqrMaxDist):
```

```
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
```

In this loop we can also search for the closest branch which is less than or equal to `maxDist` away. We will keep track of this branch as well as the square distance to it in variables called `closest` and `closestSqrDist` respectively which we will define above the loop to be `None` and `-1` respectively. It doesn't actually matter what value we give `closestSqrDist` at first, because it will be either replaced by the distance to the first branch we find within the range if such a branch exists, or if no such branch exists, then its value will never be used.

We can then check inside the loop whether `sqrDist` is less than or equal to `sqrMaxDist`. If so, we then check whether either `closest` is equal to `None` (which would mean this is the first branch we have found within the range) or `sqrDist` is less than `closestSqrDist` (which would mean that this branch is closer than all the previous branches we've found in the range). In either of these cases, we want to set `closest` equal to the branch and `closestDist` equal to `sqrDist`.

```
    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        return closest
```

After the loop we return the value of `closest`. It will either be equal to the closest branch we found within the range, or `None` if there is no branch in the range.

However, before we return we can apply a force to the branch. Let's go back to the `Branch` constructor and add an attribute called `forces` which we will initialise as an empty list. This will contain vectors of all the forces applied to it in the current frame.

```
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction
```

```

        self.forces = []
```

Going back to `Leaf.influence`, we will check whether we found a branch and if so calculate the force to apply to it. This should be equal to a normalised vector with the same direction as the vector from the branch to the leaf. We can do this by subtracting the branch position vector from the leaf position vector, and then dividing by the distance between them (the square root of `closestSqrDist`). We then add this vector to the branch's `forces` list.

```
    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest
```

After this function has been called for each of the `Leaf` objects, each of the `Branch` objects will have a list of different force vectors applied by different leaves. We need to create a function, `Branch.grow` which can add these to the direction vector, and then normalise it (divide it by its own magnitude such that its new magnitude is 1)

```
    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude
```

We then need to reset the list of forces to empty for next time.

```
        self.forces = []
```

Finally we create a new `Branch` object whose parent is the current one, whose tip is at `self.tip + self.direction` and whose direction is a copy of `self.direction`. We then return this new instance.

```
    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())
```

Next we have to go back to `Tree.grow` and actually invoke these functions we've just created. We do so in an `else` case to the `if self.extending` statement, because we want this logic to only apply in the explosion phase. The first thing to do inside the `else` case is to iterate over each leaf and call its `influence` function.

However, we cannot do this in the normal way. We want to check the return value of this function call and, if it is `False`, then we want to remove the leaf from the list. Generally speaking it is problematic to remove elements from a list while you're iterating over it. This is because let's say we are at index 2 and we remove the current element. Normally the loop would move on to index 3, but the element which was at index 3 has now become the element at index 2 because the previous element at index 2 was removed, so by moving on to index 3 we skip an element.

Luckily we can get around this problem by iterating through the loop backwards. If the index is decreasing from the end of the loop, the only elements which might get shifted around when we remove an element will be the ones we've already dealt with.

Therefore instead of our index going from 0 (inclusive) to `len(self.leaves)` (exclusive) with a step of 1 each time, it will go from `len(self.leaves)-1` (inclusive) to -1 (exclusive) with a step of -1 each time. Our `else` case therefore looks like this:

```
        else:
            for i in range(len(self.leaves)-1, -1, -1):
```

```
                self.leaves[i].influence(self.branches,
                                         self.sqrMinDist,
                                         self.sqrMaxDist)
```

We can store the value returned by this function in a variable called `returned`, and if this is `False` we remove the leaf at index `i` from the list.

```
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].indluence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]
```

After this loop we need to iterate through through each element, `branch`, in the list of branches. If they have at least one force applied to them (or equivalently if `len(branch.forces)` evaluates to `True`), then call `branch.grow` and append the result to the list. We again want to iterate through the list backwards, because there is no point iterating over the new branches we have added to the list. Since we don't need the index of each branch, we can just iterate over a reversed version of the list, `self.branches[::-1]`.

```
            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())
```

We are now done implementing the core algorithm of space colonisation. Our code looks like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
```

```
                       self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def copy(self):
        return Vector(self.x, self.y)

    def sqr_dist(self, other):
        return (self.x-other.x)**2 + (self.y-other.y)**2

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest

    def draw(self, screen):
```

```
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        return Leaf(Vector(x, y))

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

        self.forces = []

    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2
```

```python

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
```

```
                pygame.quit()
                sys.exit()

        screen.fill((255, 255, 255))
        tree.grow()
        tree.draw(screen)

        pygame.display.update()
```

If we run the code, we will see the tree start to grow from the root, and once it gets within 100 pixels of the nearest leaf, it will branch off towards it. If the tree gets within 10 pixels of any leaf, the leaf will disappear.

Figure 2.34: Space Colonising Tree Growth at Frame 60

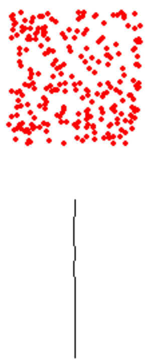


Figure 2.35: Space Colonising Tree Growth at Frame 120

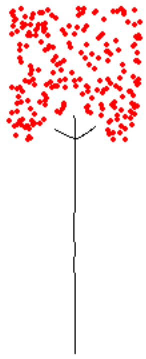


Figure 2.36: Space Colonising Tree Growth at Frame 180

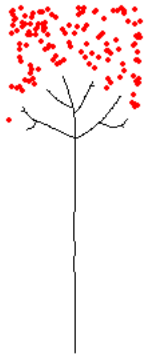


Figure 2.37: Space Colonising Tree Growth at Frame 210

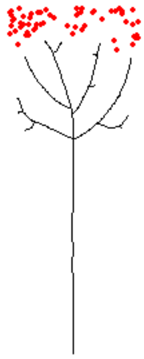


Figure 2.38: Space Colonising Tree Growth at Frame 240

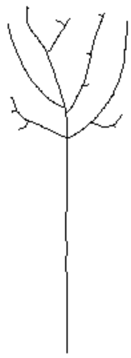


Figure 2.39: Space Colonising Tree Growth at Frame 270

2.3.3 Shaping the Trees

As I mentioned earlier, space colonisation gives us much greater control over the exact shape of the tree. We can control this by varying how the leaf points are selected. We selected random points within a square as our leaves, and so the tree grew into the shape of a square. Now we will see what the tree looks like if we make it a circle.

This is luckily quite an easy change to make. We just have to modify `Leaf.random`. Once we've generated `x` and `y`, we check whether the square distance between that point and the centre of the square is less than the square of half the width. If not, just try another random point by recursively calling the function.

```
    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        pos = Vector(x, y)
        center = topLeft+(size/2)

        if pos.sqr_dist(center) < (size.x/2) ** 2:
            return Leaf(pos)
        else:
            return Leaf.random(topLeft, size)
```

Our code now looks like this:

```
import pygame
import sys
import math
```

```python
from numpy.random import uniform
from pygame.locals import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def copy(self):
        return Vector(self.x, self.y)

    def sqr_dist(self, other):
        return (self.x-other.x)**2 + (self.y-other.y)**2

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
```

```
                    if closest is None or sqrDist < closestSqrDist:
                        closest = branch
                        closestSqrDist = sqrDist

            if closest is not None:
                force = (self.pos - closest.tip)/closestSqrDist**0.5
                closest.forces.append(force)

            return closest

        def draw(self, screen):
            pygame.draw.circle(screen,
                               (255, 0, 0),
                               self.pos.roundTuple,
                               2,
                               0)

        def random(topLeft, size):
            x = topLeft.x + uniform(size.x)
            y = topLeft.y + uniform(size.y)

            pos = Vector(x, y)
            center = topLeft+(size/2)

            if pos.sqr_dist(center) < (size.x/2) ** 2:
                return Leaf(pos)
            else:
                return Leaf.random(topLeft, size)

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

        self.forces = []

    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())
```

```
    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())

    def draw(self, screen):
```

```
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()

    tree.draw(screen)

    pygame.display.update()
```

Our tree now looks like this:

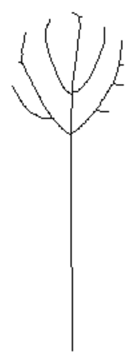


Figure 2.40: Circular Space Colonising Tree

Circles and squares are all well and good, but what if we wanted to generate points inside an arbitrary polygon? It turns out that this is quite difficult, but not impossible. The simplest way to achieve this is to generate a point inside a rectangle around the polygon, and then check whether this point is inside the polygon itself, and if not, try again.

Our code is getting a little long and messy now, so we're going to create a new script called "*geometry.py*". We will actually move our `Vector` class from the old script into here. We will then delete the `Vector` class from the old script and instead just add

```
from geometry import Vector
```

to the top. Going back to the new script, as well as the `Vector` class we will also have a class called `LineSegment`. Its constructor will accept two vectors `a` and `b` which are the point vectors of the two ends of the line segment.

```
class LineSegment:
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

We will also create a function called `orientation` which accepts three point vectors `a`, `b` and `c`. It will return an integer – 0 if the points are given in a clockwise order, 1 if the points are given in an anticlockwise order, or 2 if the points are all colinear (i.e. there is a straight line which passes through all 3 of them).

We can compute this value using slopes. We calculate the slope of the vector from `a` to `b` and of that from `b` to `c`. If the former is greater than the latter, the points are in clockwise order. If it is less, the points are anticlockwise. If they are equal, the points are colinear. Using some algebra we can say that this is equivalent to calculating:

$$s = (y_b - y_a)(x_c - x_b) - (y_c - y_b)(x_b - x_a)$$

Where $s > 0$ means clockwise, $s = 0$ means colinear and $s < 0$ means anticlockwise.

Our function therefore can look like this:

```
def orientation(a, b, c):
    s = (b.y-a.y)*(c.x-b.x) - (c.y-b.y)*(b.x-a.x)
    if s > 0:
        return 0
    elif s < 0:
        return 1
    else:
        return 2
```

Next we can add a function, `intersects`, to our `LineSegment` class. It will accept another line segment, `other`, and will return `True` if the line segments intersect, and `False` otherwise.

It will use the following theorem: Generally speaking, the two line segments intersect if and only if the triplets `self.a, self.b, other.a` and `self.a, self.b, other.b` have different orientations, *and* the triplets `other.a, other.b, self.a` and `other.a, other.b, self.b` have different orientations. In the special case where all four of those triplets are colinear (which is equivalent to saying that the first two are colinear) then we find the range of x values for both line segments and see whether they intersect. We do the same for the y values and if both of them intersect, then the line segments intersect.

Our `LineSegment.intersects` function therefore looks like this:

```
    def intersects(self, other):
        o1 = orientation(self.a, self.b, other.a)
        o2 = orientation(self.a, self.b, other.b)

        if o1 == o2 == 2:
            minX = min(self.a.x, self.b.x)
            maxX = max(self.a.x, self.b.x)

            xIntersects = (minX <= other.a.x <= maxX
                           or minX <= other.b.x <= maxX)
            if xIntersects:
                minY = min(self.a.y, self.b.y)
                maxY = max(self.a.y, self.b.y)

                yIntersects = (minY <= other.a.y <= maxY
                               or minY <= other.b.y <= maxY)
                return yIntersects
            else:
                return False
        elif o1 != o2:
            o3 = orientation(other.a, other.b, self.a)
            o4 = orientation(other.a, other.b, self.b)
            return o3 != o4
        else:
            return False
```

Don't worry if this function looks a little daunting. It is just applying the above theorem step by step, making sure to avoid unnecessary steps where possible.

So why did we solve this line-segment-intersection problem? We needed this function to answer the more pressing question: is a given point inside a given polygon. The way we answer this is to split the polygon into line segments, and draw another line starting from the point we are testing and extending to the right a very long way. We can then ask: How many of the line segments does this new line intersect? If the answer is odd, then the point is inside the polygon. If the answer is even, then the point is outside the polygon.

In this spirit, let's create a class, `Polygon`, whose constructor will accept a list of point vectors which will be the vertices of the polygon. We will assign this list to an attribute named `self.points`. We will also create a list called `self.lineSegs` which will at first contain only the line segment between the first and the last point. We will then iterate over each point in the points list except for the last one, and for each one we create a `LineSegment` object between it and the next point, adding this to the `self.lineSegs` list.

```
class Polygon:
    def __init__(self, points):
        self.points = points
        self.lineSegs = [LineSegment(points[0], points[-1])]

        for i in range(len(points)-1):
            self.lineSegs.append(LineSegment(points[i],
                                             points[i+1]))
```

Before we mentioned taking our point we want to test and creating a line segment starting at that point and extending to the right a long distance. As long as this distance is greater than the greatest width of the polygon, then we'll be okay. We therefore need to add a function to the `Polygon` class called `bounds` which will return two vectors describing the smallest rectangle aligned with the axes which contains polygon– the first being the position vector of the top-left corner of such a rectangle, and the second vector encoding the width and height. If you're wondering why we need to work out all of this when we only need the width, it's because we will need the rest of the information later and so we might as well implement the function now.

This function will be fairly simple. It will iterate over every point in the polygon and keep track of the maximum and minimum `x` and `y` values it encounters.

```
    def bounds(self):
        minX = None
        maxX = None

        minY = None
        maxY = None

        for point in self.points:
            if minX is None or point.x < minX:
                minX = point.x
            if maxX is None or point.x > maxX:
                maxX = point.x

            if minY is None or point.y < minY:
                minY = point.y
            if maxY is None or point.y > maxY:
                maxY = point.y

```

```
        topLeft = Vector(minX, minY)
        size = Vector(maxX-minX,
                      maxY-minY)

        return topLeft, size
```

Now we can go back to the constructor and set these vectors as attributes

```
    def __init__(self, points):
        self.points = points
        self.lineSegs = [LineSegment(points[0], points[-1])]

        for i in range(len(points)-1):
            self.lineSegs.append(LineSegment(points[i],
                                             points[i+1]))

        self.topLeft, self.size = self.bounds()
```

Next we can finally add our function `Polygon.contains` which accepts a point vector as its parameter and will return `True` if the point is inside the polygon and `False` if not.

Inside this function we create a line segment between this point and some other point which is a distance of `self.size.x` to the right of it. We will call this line segment `testSeg`.

```
    def contains(self, point):
        extendedPoint = Vector(point.x+self.size.x,
                               point.y)

        testSeg = LineSegment(point, extendedPoint)
```

Now we need to count the number of elements of the list `self.lineSegs` with which `testSeg` intersects. We will have a counter variable, `intersections`, which starts at 0, and then we will iterate over each line segment `i` in the list. If `testSeg.intersects(i)` returns `True`, then increment `intersections`.

```
        intersections = 0
        for i in self.lineSegs:
            if testSeg.intersects(i):
                intersections += 1
```

In fact we don't even need to count the intersections, we just care about whether the number is even or odd. As such we can instead replace the counter with a boolean flag called `inside` which will start off as `False` and instead of incrementing it, we just toggle it.

```
        inside = False
        for i in self.lineSegs:
            if testSeg.intersects(i):
                inside = not inside
```

In essence, we want `inside` to be `True` if and only if either it is already `True` and `i` doesnt intersect `testSeg` *or* it was already `False` and the lines do indeed intersect. As such, for each line segment we just want to xor the `inside` flag with the result of the intersection test using Python's xor operator, "^".

```
        inside = False
        for i in self.lineSegs:
            inside ^= testSeg.intersects(i)
```

After this loop we can return `inside` and we are done with the the `geometry` module. Its code looks like this:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.roundTuple = (int(self.x), int(self.y))

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def __sub__(self, other):
        return Vector(self.x - other.x,
                      self.y - other.y)

    def __truediv__(self, k):
```

```
        return Vector(self.x/k,
                      self.y/k)

    def __idiv__(self, k):
        self.x /= k
        self.y /= k
        self.roundTuple = (int(self.x), int(self.y))
        return self

    def copy(self):
        return Vector(self.x, self.y)

    def sqr_dist(self, other):
        return (self.x-other.x)**2 + (self.y-other.y)**2

def orientation(a, b, c):
    s = (b.y-a.y)*(c.x-b.x) - (c.y-b.y)*(b.x-a.x)
    if s > 0:
        return 0
    elif s < 0:
        return 1
    else:
        return 2

class LineSegment:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def intersects(self, other):
        o1 = orientation(self.a, self.b, other.a)
        o2 = orientation(self.a, self.b, other.b)

        if o1 == o2 == 2:
            minX = min(self.a.x, self.b.x)
            maxX = max(self.a.x, self.b.x)

            xIntersects = (minX <= other.a.x <= maxX
                           or minX <= other.b.x <= maxX)
            if xIntersects:
                minY = min(self.a.y, self.b.y)
                maxY = max(self.a.y, self.b.y)

                yIntersects = (minY <= other.a.y <= maxY
                               or minY <= other.b.y <= maxY)
                return yIntersects
            else:
                return False
        elif o1 != o2:
            o3 = orientation(other.a, other.b, self.a)
            o4 = orientation(other.a, other.b, self.b)
```

```
                return o3 != o4
            else:
                return False

class Polygon:
    def __init__(self, points):
        self.points = points
        self.lineSegs = [LineSegment(points[0], points[-1])]

        for i in range(len(points)-1):
            self.lineSegs.append(LineSegment(points[i],
                                             points[i+1]))

        self.topLeft, self.size = self.bounds()

    def bounds(self):
        minX = None
        maxX = None

        minY = None
        maxY = None

        for point in self.points:
            if minX is None or point.x < minX:
                minX = point.x
            if maxX is None or point.x > maxX:
                maxX = point.x

            if minY is None or point.y < minY:
                minY = point.y
            if maxY is None or point.y > maxY:
                maxY = point.y

        topLeft = Vector(minX, minY)
        size = Vector(maxX-minX,
                      maxY-minY)

        return topLeft, size

    def contains(self, point):
        extendedPoint = Vector(point.x+self.size.x,
                               point.y)

        testSeg = LineSegment(point, extendedPoint)

        inside = False
        for i in self.lineSegs:
            inside ^= testSeg.intersects(i)

        return inside
```

We can now go back to the main space colonisation script. We left off with the code looking like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *
from geometry import Vector

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(topLeft, size):
        x = topLeft.x + uniform(size.x)
        y = topLeft.y + uniform(size.y)

        pos = Vector(x, y)
        center = topLeft+(size/2)

        if pos.sqr_dist(center) < (size.x/2) ** 2:
            return Leaf(pos)
        else:
```

```
            return Leaf.random(topLeft, size)

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

        self.forces = []

    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        topLeft = Vector(270, 100)
        size = Vector(100, 100)
        self.leaves = list(Leaf.random(topLeft, size) for i in range(250))

        self.extending = True

    def grow(self):
```

```python
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

tree = Tree(Vector(width/2, height),
            Vector(0, -1))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()

    tree.draw(screen)

    pygame.display.update()
```

We now need to modify the imports to include the `Polygon` class.

```
from geometry import Vector, Polygon
```

We will also modify the `Tree` constructor to accept a `Polygon` instance which it will pass to the `Leaf.random` function (we will change this function in a second such that it actually wants this parameter).

```
class Tree:
    def __init__(self, base, sprout, polygon, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.extending = True
```

We will go to the `Leaf.random` function to make it accept this parameter. The `x` and `y` coordinates that it generates will be inside the polygon's bounds.

```
    def random(polygon):
        x = polygon.topLeft.x + uniform(polygon.size.x)
        y = polygon.topLeft.y + uniform(polygon.size.y)

        pos = Vector(x, y)
```

If the polygon contains the point `pos`, then return a leaf at that position. Otherwise, try again by recursively calling the function.

```
    def random(polygon):
        x = polygon.topLeft.x + uniform(polygon.size.x)
        y = polygon.topLeft.y + uniform(polygon.size.y)

        pos = Vector(x, y)

        if polygon.contains(pos):
            return Leaf(pos)
```

```
        else:
            return Leaf.random(polygon)
```

Next we go to where we create the `tree` object right before the game loop. We need to define a polygon. We will use the points $(270, 100)$, $(320, 200)$, $(370, 100)$ which define a triangle. We will then pass this polygon object to the `Tree` constructor.

```
polygon = Polygon([Vector(270, 100),
                   Vector(320, 200),
                   Vector(370, 100)])

tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon)
```

Our script now looks like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *
from geometry import Vector, Polygon

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest
```

```

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(polygon):
        x = polygon.topLeft.x + uniform(polygon.size.x)
        y = polygon.topLeft.y + uniform(polygon.size.y)

        pos = Vector(x, y)

        if polygon.contains(pos):
            return Leaf(pos)
        else:
            return Leaf.random(polygon)

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

        self.forces = []

    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
```

```
                              self.tip.roundTuple,
                              1)

class Tree:
    def __init__(self, base, sprout, polygon, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

polygon = Polygon([Vector(270, 100),
```

```
                     Vector(320, 200),
                     Vector(370, 100)])

tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()

    tree.draw(screen)

    pygame.display.update()
```

When we run it, the result looks like this:

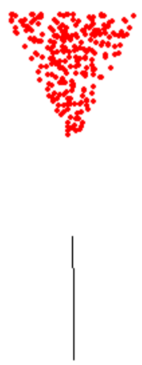


Figure 2.41: Growing Triangular Tree

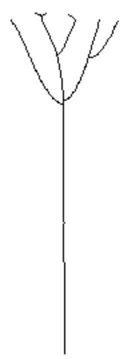


Figure 2.42: Grown Triangular Tree

We can now be more experimental with the polygon we use. I like to use

```
polygon = Polygon([Vector(320.000000, 68.750000),
                   Vector(322.950850, 57.287111),
                   Vector(340.307481, 35.005082),
                   Vector(372.950850, 25.641105),
                   Vector(406.023870, 41.429232),
                   Vector(420.000000, 75.000000),
                   Vector(406.023870, 111.869918),
                   Vector(372.950850, 144.932621),
                   Vector(340.307481, 174.195768),
                   Vector(322.950850, 197.139164),
                   Vector(320.000000, 206.250000),
                   Vector(317.049150, 197.139164),
                   Vector(299.692519, 174.195768),
                   Vector(267.049150, 144.932621),
                   Vector(233.976130, 111.869918),
                   Vector(220.000000, 75.000000),
                   Vector(233.976130, 41.429232),
                   Vector(267.049150, 25.641105),
                   Vector(299.692519, 35.005082),
                   Vector(317.049150, 57.287111)])
```

Because it looks heart-shaped, and I also like to use `minDist=5` for this polygon so that the branches have to get really close to the leaves

```
tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon,
            minDist=5)
```

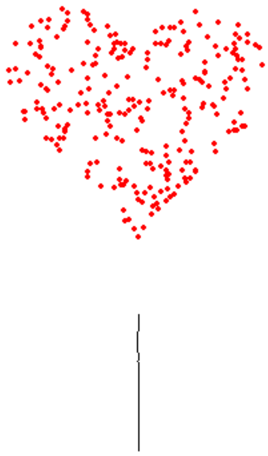


Figure 2.43: Growing Heart-Shaped Tree

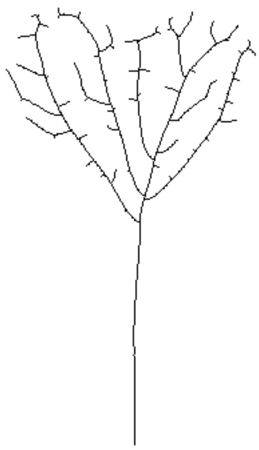


Figure 2.44: Grown Heart-Shaped Tree

At this point you might be wondering where those polygon coordinates came from. We're going to take a little detour to show how you can generate a polygon from any parametric curve.

2.3.4 Generating a Heart-Shaped Polygon

A parametric curve is one for which both the x and y coordinates of the curve are functions of a shared parameter (usually called t and thought of as *time*). We will create a new script which I will call *parametric.py*. Inside it we will create a function called `heart` whose parameters will be a list, `center`, containing the coordinates for the centre of the heart, a float, `width`, which will be the width of the heart, and a keyword argument, `precision` which will be used to determine the number of vertices we want to sample from the parametric curve, and hence how similar the polygon we generate will be to the parametric curve itself. By default let's make this 20.

```python
def heart(center, width, precision=20):
```

Inside this function we will create a list called `points` which will initially be empty and we will use it to store the coordinates of the points we sample from the curve. We also need to store the minimum and maximum x coordinates we come across. This is so we can calculate the width of the polygon, and so we can scale each point to match the `width` parameter we passed in. By default we will make both of these values `None`.

```python
def heart(center, width, precision=20):
    points = []
    minX = maxX = None
```

Next we can add our loop which will iterate over the curve. We want a number of samples equal to the value of `precision` so the loop will have `i` iterate from `0` (inclusive) to `precision-1` (inclusive). Inside this loop we can calculate the appropriate value of the curve's t parameter. For this specific curve we want t to vary from 0 to 2π (note that if you are using a different parametric curve you will need to check the appropriate range for t). We can therefore calculate evenly spaced values of t using `t = (i/precision) * 2 * math.pi`, so don't forget to add `import math` to the script.

```python
import math

def heart(center, width, precision=20):
    points = []
    minX = maxX = None
    for i in range(precision):
        t = (i/precision) * 2 * math.pi
```

Now we need to look at our curve itself. There are many heart-shaped curves that exist[2], but in particular we will be using:

$$
\begin{aligned}
x(t) &= 16\sin^3(t) \\
y(t) &= 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)
\end{aligned}
$$

$$0 \leq t < 2\pi$$

We already have our `t` value, so we can now calculate `x` and `y`.

```python
    for i in range(precision):
        t = (i/precision) * 2 * math.pi
```

[2] http://mathworld.wolfram.com/HeartCurve.html

```
        x = 16 * math.sin(t)**3
        y = (- 13 * math.cos(t)
             + 5 * math.cos(2*t)
             + 2 * math.cos(3*t)
             + math.cos(4*t))
```

It is important to note here that our `y` value is the negative of that of the curve itself. This is again because in regular mathematical graphs y increases upwards but in graphical programs such as PyGame, y increases downwards.

We then need to update our `minX` and `maxX` variables if appropriate, and then append our point to the list.

```
    for i in range(precision):
        t = (i/precision) * 2 * math.pi
        x = 16 * math.sin(t)**3
        y = (- 13 * math.cos(t)
             + 5 * math.cos(2*t)
             + 2 * math.cos(3*t)
             + math.cos(4*t))

        if minX is None or x < minX:
            minX = x
        if maxX is None or x > maxX:
            maxX = x

        points.append([x, y])
```

Now we need to handle scaling the points such that the width of the polygon is equal to the `width` parameter. We will also shift these points such that the centre of the polygon is at the `center` parameter. Note that when I say centre I do not mean geometric centre, I just mean the origin of the parametric curve. We will store these transformed points in a new list called `transformed` which will initially be empty.

Since the width of the polygon is currently `maxX - minX` and we want it to be `width`, we can multiply the coordinates of each of the points by a variable, `scale`, which will be equal to `width/(maxX - minX)`. We can then add on the coordinates of the centre.

```
import math

def heart(center, width, precision=20):
    points = []
    minX = maxX = None
    for i in range(precision):
```

```
        t = (i/precision) * 2 * math.pi
        x = 16 * math.sin(t)**3
        y = (- 13 * math.cos(t)
             + 5 * math.cos(2*t)
             + 2 * math.cos(3*t)
             + math.cos(4*t))

        if minX is None or x < minX:
            minX = x
        if maxX is None or x > maxX:
            maxX = x

        points.append([x, y])

    transformed = []
    scale = width/(maxX-minX)

    for x, y in points:
        transformed.append([x*scale + center[0],
                            y*scale + center[1]])

    return transformed
```

That's the code which generates the points on the heart curve. We can test it by writing using:

```
print(heart([320, 100], 200))
```

This will generate a heart of width 200 pixels centred on the point $(320, 100)$. After running this, you should see the result:

```
[[320.0, 68.75], [322.95084971874735, 57.28711099373523],
↪  [340.30748101455663, 35.005081636722366], [372.95084971874735,
↪  25.641104508486606], [406.02387002944835, 41.429231917974995], [420.0,
↪  75.0], [406.0238700294484, 111.86991836327762], [372.95084971874735,
↪  144.93262091339233], [340.30748101455663, 174.19576808202498],
↪  [322.95084971874735, 197.13916358438584], [320.0, 206.25],
↪  [317.0491502812526, 197.13916358438584], [299.69251898544337,
↪  174.195768082025], [267.04915028125265, 144.93262091339236],
↪  [233.97612997055165, 111.86991836327765], [220.0, 75.00000000000003],
↪  [233.97612997055163, 41.42923191797501], [267.0491502812526,
↪  25.641104508486606], [299.6925189854433, 35.00508163672234],
↪  [317.04915028125265, 57.28711099373521]]
```

As you can see, the output is a list containing pairs of x and y coordinates of vertices of the polygon. If you look closely you might recognise these values as the coordinates we used earlier to generate the heart shaped tree. The only problem is that it is very time-consuming to transcribe these values into your code by hand. Therefore we will instead print this out using

```
header = "polygon = Polygon(["
string = header
for x, y in heart([320, 100], 200):
    string += "Vector(%f, %f),\n"%(x,y) + " "*len(header)
string = string[:-len(header)-2]
string += "])"
```

Don't worry if you don't quite understand what this code is doing. There's nothing interesting going on except for some helpful string manipulation. The important part is that now when we run the code, we will get the following output:

```
polygon = Polygon([Vector(320.000000, 68.750000),
                   Vector(322.950850, 57.287111),
                   Vector(340.307481, 35.005082),
                   Vector(372.950850, 25.641105),
                   Vector(406.023870, 41.429232),
                   Vector(420.000000, 75.000000),
                   Vector(406.023870, 111.869918),
                   Vector(372.950850, 144.932621),
                   Vector(340.307481, 174.195768),
                   Vector(322.950850, 197.139164),
                   Vector(320.000000, 206.250000),
                   Vector(317.049150, 197.139164),
                   Vector(299.692519, 174.195768),
                   Vector(267.049150, 144.932621),
                   Vector(233.976130, 111.869918),
                   Vector(220.000000, 75.000000),
                   Vector(233.976130, 41.429232),
                   Vector(267.049150, 25.641105),
                   Vector(299.692519, 35.005082),
                   Vector(317.049150, 57.287111)])
```

We could now copy and paste this directly into our space colonisation code. Our final polygon generation code now looks like this:

```
import math

def heart(center, width, precision=20):
    points = []
```

```python
    minX = maxX = None
    for i in range(precision):
        t = (i/precision) * 2 * math.pi
        x = 16 * math.sin(t)**3
        y = (- 13 * math.cos(t)
             + 5 * math.cos(2*t)
             + 2 * math.cos(3*t)
             + math.cos(4*t))

        if minX is None or x < minX:
            minX = x
        if maxX is None or x > maxX:
            maxX = x

        points.append([x, y])

    transformed = []
    scale = width/(maxX-minX)

    for x, y in points:
        transformed.append([x*scale + center[0],
                            y*scale + center[1]])

    return transformed

header = "polygon = Polygon(["
string = header
for x, y in heart([320, 100], 200):
    string += "Vector(%f, %f),\n"%(x,y) + " "*len(header)
string = string[:-len(header)-2]
string += "])"

print(string)
```

I recommend you try this out using other parametric curves, and be creative with the type of shapes you create.

2.3.5 Prettifying the Trees

We will now go back to our space colonisation code. We left off with the code looking like this:

```python
import pygame
import sys
import math
from numpy.random import uniform
```

```
from pygame.locals import *
from geometry import Vector, Polygon

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest

    def draw(self, screen):
        pygame.draw.circle(screen,
                           (255, 0, 0),
                           self.pos.roundTuple,
                           2,
                           0)

    def random(polygon):
        x = polygon.topLeft.x + uniform(polygon.size.x)
        y = polygon.topLeft.y + uniform(polygon.size.y)

        pos = Vector(x, y)

        if polygon.contains(pos):
            return Leaf(pos)
        else:
            return Leaf.random(polygon)

class Branch:
    def __init__(self, parent, tip, direction):
        self.parent = parent
        self.tip = tip
        self.direction = direction

        self.forces = []

```

```
    def grow(self):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def extend(self):
        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy())

    def draw(self, screen):
        if self.parent is not None:
            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             1)

class Tree:
    def __init__(self, base, sprout, polygon, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend())

        else:
            for i in range(len(self.leaves)-1, -1, -1):
```

```
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow())

    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)

        for leaf in self.leaves:
            leaf.draw(screen)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

polygon = Polygon([Vector(320.000000, 68.750000),
                   Vector(322.950850, 57.287111),
                   Vector(340.307481, 35.005082),
                   Vector(372.950850, 25.641105),
                   Vector(406.023870, 41.429232),
                   Vector(420.000000, 75.000000),
                   Vector(406.023870, 111.869918),
                   Vector(372.950850, 144.932621),
                   Vector(340.307481, 174.195768),
                   Vector(322.950850, 197.139164),
                   Vector(320.000000, 206.250000),
                   Vector(317.049150, 197.139164),
                   Vector(299.692519, 174.195768),
                   Vector(267.049150, 144.932621),
                   Vector(233.976130, 111.869918),
                   Vector(220.000000, 75.000000),
                   Vector(233.976130, 41.429232),
                   Vector(267.049150, 25.641105),
                   Vector(299.692519, 35.005082),
                   Vector(317.049150, 57.287111)])

tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon,
            minDist=5)

while True:
    for event in pygame.event.get():
```

```
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        screen.fill((255, 255, 255))
        tree.grow()

        tree.draw(screen)

        pygame.display.update()
```

Now that we've got the algorithm itself working, it's time to make the tree itself more visually appealing.

The first change we'll make is to entirely delete the `Leaf.draw` function. The red dots were helpful to visualise how the algorithm actually works, but they will not be part of the final product. As well as deleting this function, we also have to go to `Tree.draw` and delete the loop which draws the leaves. The function now looks like this.

```
    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen)
```

The next change we'll make is to give the branches some thickness. I would like the branches near the bottom of the tree to be thicker and the branches near the top to be thinner. We therefore need to give the branches some notion of how far through the tree they are. To achieve this we will make two main changes.

- The `Branch` class will have an attribute called `depth` which keeps track of how far through the tree it is. The root will have a `depth` of 0 and each of its children will have a `depth` of 1 etc.

- The `Tree` class will have an attribute called `maxDepth` which will keep track of the greatest value of `depth` of any of its branches.

To achieve this, we will first go to the `Tree` constructor and add the `maxDepth` parameter which by default will be 0.

```
    def __init__(self, base, sprout, polygon, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

```

```
        self.maxDepth = 0

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.extending = True
```

Now we can go to the Branch constructor and make it accept a keyword argument depth which will be by default equal to 0. It will then assign this as a parameter.

```
    def __init__(self, parent, tip, direction, depth=0):
        self.parent = parent
        self.tip = tip
        self.direction = direction
        self.depth = depth

        self.forces = []
```

Now we can go down to the Branch.grow function and make it accept an additional parameter, tree, which will be the Tree object from which it is growing. We want to check whether the tree's maxDepth attribute is less than or equal to the branch's depth attribute. If so, set tree.maxDepth equal to self.depth + 1. Then we pass self.depth + 1 to the constructor of the child Branch object we create.

```
    def grow(self, tree):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        if tree.maxDepth <= self.depth:
            tree.maxDepth = self.depth + 1

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy(),
                      self.depth + 1)
```

We can now make the same changes to the `Branch.extend` function.

```
    def extend(self, tree):
        if tree.maxDepth <= self.depth:
            tree.maxDepth = self.depth + 1

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy(),
                      self.depth + 1)
```

Now we go down to the `Tree.grow` function and find where we call `currentBranch.extend` and `branch.grow` and just make sure to pass `self` to those function calls.

```
                self.branches.append(currentBranch.extend(self))
```

```
                    self.branches.append(branch.grow(self))
```

This function now looks like this:

```
    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
            else:
                self.branches.append(currentBranch.extend(self))

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
```

```
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow(self))
```

We can now go up to the `Branch.grow` function and modify it to accept an additional parameter, `maxDepth`. We can now work out a value `p` which is equal to `self.depth/maxDepth`. This will be equal to 0 at the root and 1 at the deepest branch. We can therefore change the thickness of the line from 1 to `int(20-18*p)`. This means that the thickness will be 20 at the root and 2 at the deepest branch. Of course, these values are entirely up to your own personal taste. These are just the ones I think look nice.

```
    def draw(self, screen, maxDepth):
        if self.parent is not None:
            p = self.depth/maxDepth

            pygame.draw.line(screen,
                             (0, 0, 0),
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             int(20-18*p))
```

Next we'll go down to `Tree.draw` and locate the call to `branch.draw`, ensuring to also pass it `self.maxDepth` as a parameter.

```
    def draw(self, screen):
        for branch in self.branches:
            branch.draw(screen, self.maxDepth)
```

If we now run the code this is what we will see.

Figure 2.45: Growing Heart-Shaped Tree with Depth Gradient

Figure 2.46: Growing Heart-Shaped Tree with Depth Gradient

Figure 2.47: Growing Heart-Shaped Tree with Depth Gradient

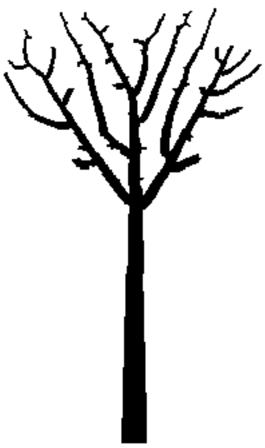


Figure 2.48: Grown Heart-Shaped Tree with Depth Gradient

This is certainly an improvement, but the tree still looks incredibly barren and lifeless. We can add some bright colourful leaves to remedy this. For this project, we will achieve this by drawing coloured circles at all of the branches. To ensure that the trunk of the tree remains relatively leaf-free, we will make the radius of the circle drawn increase as we go from the base of the tree to the tips.

We will create a function `Branch.draw_circle`. While the name `Branch.draw_leaf` might seem more appropriate, I will not use this because it might get a little confusing, as this has nothing to do with the `Leaf` objects we used to generate the tree.

`Branch.draw_circle` will accept as parameters a PyGame surface on which to draw and the `maxDepth` of the tree. It will then calculate the same value of `p` as we calculated in `Branch.draw`. It will then call `pygame.draw.circle`. For the moment the colour will just be bright green (RGB 0, 255, 0) but we will make this more interesting later. The centre of the circle will be `self.tip.roundTuple` and we will set the radius equal to `int(2+25*(p**2))`. This is similar to the branch thickness calculation. At the root, the radius will be 2 and at the deepest branch, the radius will be 27. However, the `p**2` term means that the radius will grow faster at the deeper branches, as opposed to the branch thickness which grows at a constant rate.

```
    def draw_circle(self, screen, maxDepth):
        p = self.depth/maxDepth

        pygame.draw.circle(screen,
                           (0, 255, 0),
                           self.tip.roundTuple,
                           int(2+25*(p**2)),
                           0)
```

We can now go back to `Tree.draw` and add a loop to draw all of the branches' circles before drawing the branches themselves. This is so the branches appear in front of the circles.

```
    def draw(self, screen):
        for branch in self.branches:
            branch.draw_circle(screen, self.maxDepth)

        for branch in self.branches:
            branch.draw(screen, self.maxDepth)
```

Figure 2.49: Growing Heart-Shaped Tree with Circles

Figure 2.50: Growing Heart-Shaped Tree with Circles

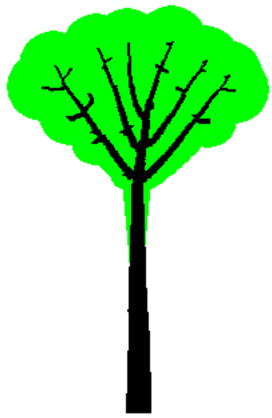


Figure 2.51: Growing Heart-Shaped Tree with Circles

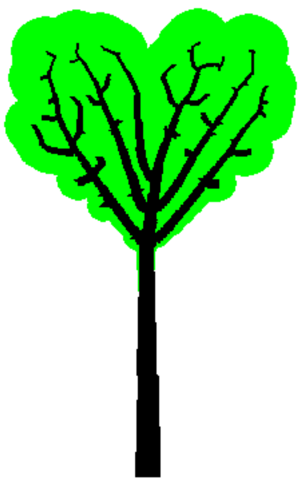


Figure 2.52: Grown Heart-Shaped Tree with Circles

We're getting somewhere now. The next thing to change would be the colour scheme. Instead of hard-coding in colours for the branches and circles, we will make these colours properties of a class called `Palette`. We will then pass an instance of this class to our `Tree` object which will in turn pass it to the the `Branch.draw` and `Branch.draw_circle` functions.

At the top of our script we will create the `Palette` class. It will accept two parameters: a tuple containing HSV colour values for the branches, and one containing HSV colour values for the circles.

```
class Palette:
    def __init__(self, branch, circle):
        self.branch = branch
        self.circle = circle
```

We can now go down to the `Tree` constructor and make it accept a `Palette` instance as a parameter, which it will then assign as an attribute.

```
    def __init__(self, base, sprout, polygon, palette, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.maxDepth = 0

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.palette = palette
        self.extending = True
```

Now we can go to the `Branch.draw_circle` function and make it accept `palette` as a parameter. It will then create a `pygame.Color` object which is initially black, and then we will set the `hsva` attribute of it to `palette.circle`. Since we won't be storing alpha values in the palette, we can use `color.hsva = (*palette.circle, 100)` to just use a hard-coded value of 100 for alpha. We then pass this `color` object to `pygame.draw.circle`.

```
    def draw_circle(self, screen, maxDepth, palette):
        p = self.depth/maxDepth

        color = pygame.Color(0, 0, 0)
        color.hsva = (*palette.circle, 100)

        pygame.draw.circle(screen,
                           color,
```

```
                                self.tip.roundTuple,
                                int(2+25*(p**2)),
                                0)
```

We then make similar changes to `Branch.draw`, but using `palette.branch` rather than `palette.circle`.

```
    def draw(self, screen, maxDepth, palette):
        if self.parent is not None:
            p = self.depth/maxDepth

            color = pygame.Color(0, 0, 0)
            color.hsva = (*palette.branch, 100)

            pygame.draw.line(screen,
                                 color,
                                 self.parent.tip.roundTuple,
                                 self.tip.roundTuple,
                                 int(20-18*p))
```

Now we can go to `Tree.draw` and pass the palette object to these functions.

```
    def draw(self, screen):
        for branch in self.branches:
            branch.draw_circle(screen, self.maxDepth, self.palette)

        for branch in self.branches:
            branch.draw(screen, self.maxDepth, self.palette)
```

Now underneath where we create the polygon, we can also create an instance of `Palette`. I will use dark purple (HSV 265, 68, 31) for the branches, and light pink (HSV 318, 25, 100) for the circles, as I feel that these fit the heart theme very nicely. We will then pass this palette object to the `Tree` constructor.

```
palette = Palette((265, 68, 31),
                  (318, 25, 100))

tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon,
            palette,
            minDist=5)
```

If we were to run the code now, we would see these new colours.

Figure 2.53: Pink Growing Heart-Shaped Tree

Figure 2.54: Pink Growing Heart-Shaped Tree

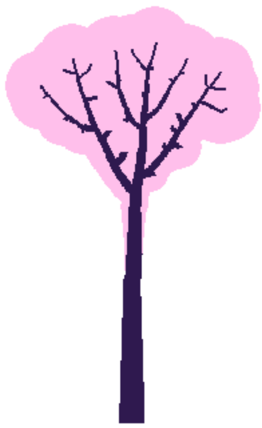


Figure 2.55: Pink Growing Heart-Shaped Tree

Figure 2.56: Pink Grown Heart-Shaped Tree

This image still looks a little bit flat to me, so we will make another change. Instead of having one branch colour and one circle colour, we will have two of each, and will linearly interpolate between them using `p` as the parameter. This is why we used the HSV colour space; this makes the linear interpolation nicer.

We will add to our script our familiar linear interpolation functions from page 72.

```
def lerp(a, b, t):
    return a + t * (b - a)

def lerp_color(a, b, t):
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))
```

We will also modify the `Palette` class to accept these additional colours.

```
class Palette:
    def __init__(self, branch0, branch1, circle0, circle1):
        self.branch0 = branch0
        self.branch1 = branch1
        self.circle0 = circle0
        self.circle1 = circle1
```

Down in `Branch.draw_circle` we will create a variable called `hsv` which will be the result of linearly interpolating between `palette.circle0` and `palette.circle1` using `p` as the parameter. We will then use this to assign `color.hsva`.

```
    def draw_circle(self, screen, maxDepth, palette):
        p = self.depth/maxDepth

        hsv = lerp_color(palette.circle0,
                         palette.circle1,
                         p)

        color = pygame.Color(0, 0, 0)
        color.hsva = (*hsv, 100)

        pygame.draw.circle(screen,
                           color,
                           self.tip.roundTuple,
                           int(2+25*(p**2)),
                           0)
```

We will make the same change to `Branch.draw` except using `palette.branch0` and `palette.branch1` instead of `palette.circle0` and `palette.circle1` respectively.

```
    def draw(self, screen, maxDepth, palette):
        if self.parent is not None:
            p = self.depth/maxDepth

            hsv = lerp_color(palette.branch0,
                             palette.branch1,
                             p)

            color = pygame.Color(0, 0, 0)
            color.hsva = (*hsv, 100)

            pygame.draw.line(screen,
                             color,
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             int(20-18*p))
```

Now we can go down to where we created our palette object and specify two more colours. I will use a lighter purple (HSV 265, 83, 58) as the additional branch colour and a darker pink (HSV 326, 100, 100) as the additional leaf colour.

```
palette = Palette((265, 68, 31),
                  (265, 83, 58),
```

```
                      (318, 25, 100),
                      (326, 100, 100))
```

The code now looks like this:

```
import pygame
import sys
import math
from numpy.random import uniform
from pygame.locals import *
from geometry import Vector, Polygon

def lerp(a, b, t):
    return a + t * (b - a)

def lerp_color(a, b, t):
    return tuple(lerp(a[i], b[i], t) for i in range(len(a)))

class Palette:
    def __init__(self, branch0, branch1, circle0, circle1):
        self.branch0 = branch0
        self.branch1 = branch1
        self.circle0 = circle0
        self.circle1 = circle1

class Leaf:
    def __init__(self, pos):
        self.pos = pos

    def influence(self, branches, sqrMinDist, sqrMaxDist):
        closest = None
        closestSqrDist = -1
        for branch in branches:
            sqrDist = self.pos.sqr_dist(branch.tip)
            if sqrDist < sqrMinDist:
                return False
            elif sqrDist <= sqrMaxDist:
                if closest is None or sqrDist < closestSqrDist:
                    closest = branch
                    closestSqrDist = sqrDist

        if closest is not None:
            force = (self.pos - closest.tip)/closestSqrDist**0.5
            closest.forces.append(force)

        return closest

    def random(polygon):
```

```
        x = polygon.topLeft.x + uniform(polygon.size.x)
        y = polygon.topLeft.y + uniform(polygon.size.y)

        pos = Vector(x, y)

        if polygon.contains(pos):
            return Leaf(pos)
        else:
            return Leaf.random(polygon)

class Branch:
    def __init__(self, parent, tip, direction, depth=0):
        self.parent = parent
        self.tip = tip
        self.direction = direction
        self.depth = depth

        self.forces = []

    def grow(self, tree):
        for force in self.forces:
            self.direction += force

        magnitude = (self.direction.x**2 + self.direction.y**2)**0.5

        self.direction /= magnitude

        self.forces = []

        if tree.maxDepth <= self.depth:
            tree.maxDepth = self.depth + 1

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy(),
                      self.depth + 1)

    def extend(self, tree):
        if tree.maxDepth <= self.depth:
            tree.maxDepth = self.depth + 1

        return Branch(self,
                      self.tip + self.direction,
                      self.direction.copy(),
                      self.depth + 1)

    def draw_circle(self, screen, maxDepth, palette):
        p = self.depth/maxDepth

        hsv = lerp_color(palette.circle0,
                         palette.circle1,
```

```
                             p)

            color = pygame.Color(0, 0, 0)
            color.hsva = (*hsv, 100)

            pygame.draw.circle(screen,
                               color,
                               self.tip.roundTuple,
                               int(2+25*(p**2)),
                               0)

    def draw(self, screen, maxDepth, palette):
        if self.parent is not None:
            p = self.depth/maxDepth

            hsv = lerp_color(palette.branch0,
                             palette.branch1,
                             p)

            color = pygame.Color(0, 0, 0)
            color.hsva = (*hsv, 100)

            pygame.draw.line(screen,
                             color,
                             self.parent.tip.roundTuple,
                             self.tip.roundTuple,
                             int(20-18*p))

class Tree:
    def __init__(self, base, sprout, polygon, palette, minDist=10, maxDist=100):
        self.sqrMinDist = minDist ** 2
        self.sqrMaxDist = maxDist ** 2

        self.maxDepth = 0

        self.branches = [Branch(None, base, sprout)]

        self.leaves = list(Leaf.random(polygon) for i in range(250))

        self.palette = palette
        self.extending = True

    def grow(self):
        if self.extending:
            currentBranch = self.branches[-1]
            for leaf in self.leaves:
                sqrDist = leaf.pos.sqr_dist(currentBranch.tip)
                if sqrDist < self.sqrMaxDist:
                    self.extending = False
                    break
```

```
                else:
                    self.branches.append(currentBranch.extend(self))

        else:
            for i in range(len(self.leaves)-1, -1, -1):
                returned = self.leaves[i].influence(self.branches,
                                                    self.sqrMinDist,
                                                    self.sqrMaxDist)

                if returned == False:
                    del self.leaves[i]

            for branch in self.branches[::-1]:
                if len(branch.forces):
                    self.branches.append(branch.grow(self))

    def draw(self, screen):
        for branch in self.branches:
            branch.draw_circle(screen, self.maxDepth, self.palette)

        for branch in self.branches:
            branch.draw(screen, self.maxDepth, self.palette)

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

polygon = Polygon([Vector(320.000000, 68.750000),
                   Vector(322.950850, 57.287111),
                   Vector(340.307481, 35.005082),
                   Vector(372.950850, 25.641105),
                   Vector(406.023870, 41.429232),
                   Vector(420.000000, 75.000000),
                   Vector(406.023870, 111.869918),
                   Vector(372.950850, 144.932621),
                   Vector(340.307481, 174.195768),
                   Vector(322.950850, 197.139164),
                   Vector(320.000000, 206.250000),
                   Vector(317.049150, 197.139164),
                   Vector(299.692519, 174.195768),
                   Vector(267.049150, 144.932621),
                   Vector(233.976130, 111.869918),
                   Vector(220.000000, 75.000000),
                   Vector(233.976130, 41.429232),
                   Vector(267.049150, 25.641105),
                   Vector(299.692519, 35.005082),
                   Vector(317.049150, 57.287111)])

palette = Palette((265, 68, 31),
                  (265, 83, 58),
```

```
                     (318, 25, 100),
                     (326, 100, 100))

tree = Tree(Vector(width/2, height),
            Vector(0, -1),
            polygon,
            palette,
            minDist=5)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((255, 255, 255))
    tree.grow()

    tree.draw(screen)

    pygame.display.update()
```

If we run it, we see the following result.

Figure 2.57: Pink Growing Heart-Shaped Tree With Linear Interpolation

Figure 2.58: Pink Growing Heart-Shaped Tree With Linear Interpolation

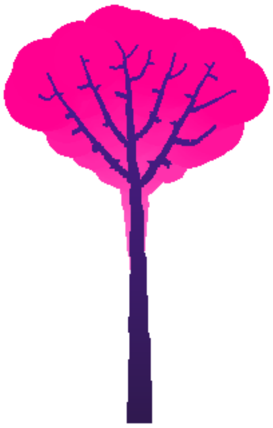


Figure 2.59: Pink Growing Heart-Shaped Tree With Linear Interpolation

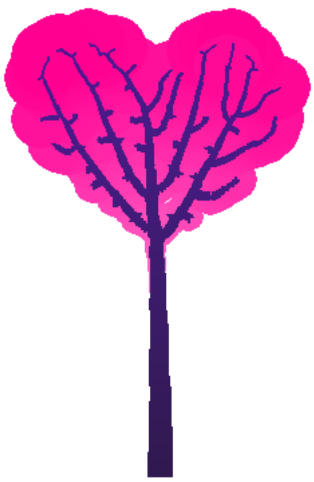


Figure 2.60: Pink Grown Heart-Shaped Tree With Linear Interpolation

I will now show a few more examples of trees you can generate with this code, just by changing the colours and polygon.

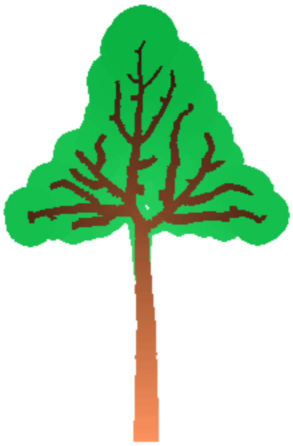


Figure 2.61: Triangular Brown-Green Tree

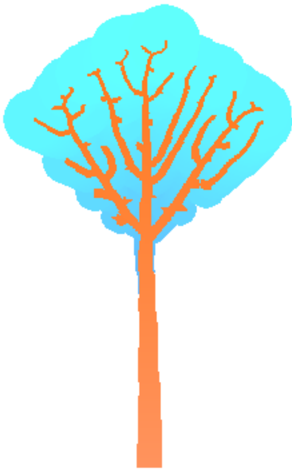


Figure 2.62: Diamond-Shaped Orange-Blue Tree

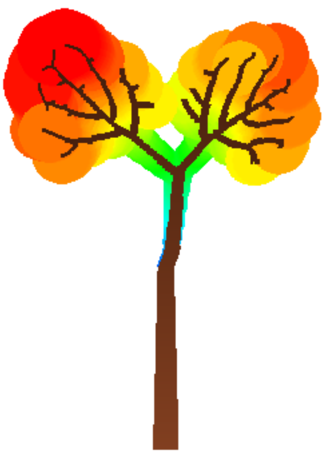


Figure 2.63: Dual-Headed Rainbow Tree

As you can see, space colonisation gives you a great deal of freedom to make the tree look however you like. Here are some ideas for where you might want to take this program.

- Render the growth process into a video or GIF.
- Render the tree growth into a spritesheet.
- Animate the tree blowing in the wind.
- Make the colours change over time, perhaps with Perlin noise.

3 Rendering Your Art

We've covered a lot of algorithms in the previous chapter. However, actually generating the art is only half of the story. Without a doubt it's the more interesting half, but nevertheless we still need to worry about how to make our program produce a usable output such an an image or video.

In this chapter, we will render much simpler projects (block coloured rectangles on a black background) but this is only to demonstrate the rendering process. These exact same processes can be used to render any more complicated project such as the beautiful trees we created in the previous chapter.

3.1 Rendering to an Image

Luckily, PyGame makes it very easy to render our screen to an image file. It has a built-in function called `pygame.image.save` which takes in two parameters: a Surface object to save and a string representing the path to the file where the image should be saved. To demonstrate how to use this we will consider a simple script.

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((0, 0, 0))
    pygame.draw.rect(screen,            # Draw a rectangle to the screen
                     (0, 255, 0),       # Green
                     (10, 10, 100, 50), # At position (10,10) size 100x50
```

```
                        0)                # Filled

    pygame.display.update()
```

If we run this script we should see something like this:

Figure 3.1: Green Rectangle on Black Background

Now let's say we wanted to save this to an image named, for example, *greenrect.png*. We could do this by inserting this line:

```
    pygame.image.save(screen, "greenrect.png")
```

We could insert this line into the game loop, such that the script looked like this:

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
```

```
            sys.exit()

    screen.fill((0, 0, 0))
    pygame.draw.rect(screen,            # Draw a rectangle to the screen
                     (0, 255, 0),       # Green
                     (10, 10, 100, 50), # At position (10, 10) size 100x50
                     0)                 # Filled

    pygame.image.save(screen, "greenrect.png")

    pygame.display.update()
```

In this case, we would find that once we run the program we would indeed find the file *greenrect.png* appear in the same directory as the script, and it would contain the image displayed on the screen.

However, this is not the most robust or efficient technique. Since the function call is inside the game loop, it means that this image is being saved and re-saved once per frame. Instead we can put the call into an event handler. Inside the event loop, underneath where we check for the `QUIT` event, we can also check for another event such as a space bar press. We do this by checking whether `event.type` is equal to the `KEYDOWN` variable imported from `pygame.locals` and then further checking whether `event.key` is equal to the `K_SPACE` variable. If so, we will save the screen and print a nice little debugging message to the console. Our script now looks like this:

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        elif event.type == KEYDOWN:
            if event.key == K_SPACE:
                pygame.image.save(screen, "greenrect.png")
                print("Screen saved ^-^")

    screen.fill((0, 0, 0))
    pygame.draw.rect(screen,            # Draw a rectangle to the screen
```

```
                     (0, 255, 0),        # Green
                     (10, 10, 100, 50), # At position (10, 10) size 100x50
                     0)                  # Filled

    pygame.display.update()
```

If we run the program and hit the space bar while it's running, we will see the file *greenrect.png* appear and we will see the message appear in the console.

```
Screen saved ^-^
```

Now the image will only be rendered whenever the space bar is pressed. This might be helpful particularly if the screen is changing over time e.g. with the space colonisation project, and you want to have control over when the "screenshot" is taken.

Note also that the call to `pygame.image.save` will overwrite the image at the given filepath if it already exists. You therefore might also want to use a counter variable so that the filename changes each time the screen is saved. You might implement such a system like this:

```
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

counter = 0

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        elif event.type == KEYDOWN:
            if event.key == K_SPACE:
                filename = "greenrect%i.png"%counter
                pygame.image.save(screen, filename)
                print("Screen saved to '%s' ^-^"%filename)
                counter += 1

    screen.fill((0, 0, 0))
```

```
    pygame.draw.rect(screen,            # Draw a rectangle to the screen
                     (0, 255, 0),       # Green
                     (10, 10, 100, 50), # At position (10, 10) size 100x50
                     0)                 # Filled

    pygame.display.update()
```

If we run the program now, we would see a new console message each time we press space

```
Screen saved to 'greenrect0.png' ^-^
Screen saved to 'greenrect1.png' ^-^
Screen saved to 'greenrect2.png' ^-^
Screen saved to 'greenrect3.png' ^-^
Screen saved to 'greenrect4.png' ^-^
Screen saved to 'greenrect5.png' ^-^
Screen saved to 'greenrect6.png' ^-^
```

Of course we would also see the corresponding image files appear.

One further thing which you might want to consider is changing what triggers the screen to be saved. We have written code such that the screen gets saved on a space bar press, but it may instead be useful to have it save every few frames or seconds. Implementing this will be left as an exercise for the reader.

3.2 Rendering to a Video

Unfortunately, rendering the PyGame window to a video file is slightly more complicated. It may instead be beneficial to render each frame as an image, and then use an external tool such as FFmpeg[1] to stitch the frames together into a video file.

However if you do indeed decide to render the video directly from Python, we will require an incredibly powerful library called OpenCV[2]. It can be installed via PyPI under the name `opencv-python`.

We will start with a very similar script to the previous section, except now the x coordinate of the rectangle will increase over time.

[1] https://www.ffmpeg.org/
[2] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

```python
import pygame
import sys
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

x = 0

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    screen.fill((0, 0, 0))
    pygame.draw.rect(screen,
                     (0, 255, 0),
                     (x, 10, 100, 50),
                     0)
    x += 1

    pygame.display.update()
```

If we run the script we would see the rectangle moving slowly across the screen.

Figure 3.2: Moving rectangle

Figure 3.3: Moving rectangle

Figure 3.4: Moving rectangle

The first step in rendering to a video is to import OpenCV using the line `import cv2`.

Next, before the game loop, we create a `VideoWriter` object. This is a class included in the OpenCV library. Its constructor requires the following parameters:

1. A string representing the path to the output video file

2. A video codec to use. Specifically, a four-character-code (FOURCC) identifier for a video codec[3].

[3] `https://www.fourcc.org/fourcc.php`

3. A number of frames per second for the output video.

4. A tuple containing the width and height of the video respectively.

Luckily these are all fairly simple apart from the FOURCC. We need to use OpenCV's `VideoWriter_fourcc` function whose parameters are four strings each containing a single character. For example, for the *MJPG* codec (which might be appropriate if we wanted to render a ".avi" video), we would use:

```
cv2.VideoWriter_fourcc("M", "J", "P", "G")
```

Or, more succinctly,

```
cv2.VideoWriter_fourcc(*"MJPG")
```

We can therefore create a VideoWriter object like this:

```
writer = cv2.VideoWriter("greenrect.mp4",
                         cv2.VideoWriter_fourcc(*"FMP4"),
                         24,
                         (width, height))
```

Note that since we are rendering to a ".mp4" file, we will use the FMP4 codec. The codec you will need may differ depending on your output format.

Also note that the size of the video window needs to match the size of the PyGame window.

Finally, note that we somewhat arbitrarily chose 24 frames per second. This may differ significantly from the actual framerate of the PyGame program as it runs. The result of this is that the animation you see in the PyGame window may differ in speed from the video output. This is something of which you need to be mindful when you render to a video.

Next, inside the game loop we want to take the current frame and convert it into a form which is usable by OpenCV. OpenCV wants the image to be a NumPy array, and luckily, PyGame has a built-in function to convert from a Surface object to a NumPy array. We can do this using the `pygame.surfarray.array3d` function, passing it the screen object as its parameter.

```
    frame = pygame.surfarray.array3d(screen)
```

The `frame` object is now a $640 \times 360 \times 3$ NumPy array containing the RGB pixel data of the screen.

To write the frame to the video, we simply call

```
    writer.write(frame)
```

Finally we need to release the VideoWriter resources once we are done rendering. We do this inside the `QUIT` conditional block.

```
        if event.type == QUIT:
            writer.release()
            pygame.quit()
            sys.exit()
```

Our code now looks like this:

```
import pygame
import sys
import cv2
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

x = 0

writer = cv2.VideoWriter("greenrect.mp4",
                         cv2.VideoWriter_fourcc(*"FMP4"),
                         24,
                         (width, height))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            writer.release()
            pygame.quit()
```

```
                sys.exit()

        screen.fill((0, 0, 0))
        pygame.draw.rect(screen,
                         (0, 255, 0),
                         (x, 10, 100, 50),
                         0)
        x += 1

        frame = pygame.surfarray.array3d(screen)
        writer.write(frame)

        pygame.display.update()
```

When we run this program and then quit it after a while, we will indeed see the file *green-rect.mp4* appear in the same directory as the script. However, it will be totally unplayable and corrupted. Since we didn't see any error messages while the program was running, this could be a particularly difficult problem to debug.

I highlighted this particular issue for a reason: it is an incredibly easy trap to fall into. The problem was that PyGame images (and indeed the frame that we have generated) are specified in the format width×height, whereas OpenCV images are specified as height×width.

Though this may seem unusual, the reason for this is so that if you iterate over an OpenCV pixel array in order (e.g. increasing height followed by increasing width), you access each pixel in reading order.

Luckily, the fix is quite simple. We need to take the pixel array we have generated, and swap the first two axes. NumPy has a function, `swapaxes`, which will handle this for us, so we need to `import numpy as np` at the top of our script and then change our code to

```
        frame = np.swapaxes(pygame.surfarray.array3d(screen), 0, 1)
```

If we now run the program, our video will render perfectly.

However, we are still not done. If we changed the rectangle from green (RGB 0, 255, 0) to red (RGB 255, 0, 0) and then run the program, the output video will show the rectangle being blue. This is also an important and very common error to highlight. How is it that green is rendered correctly, but red is not?

The answer is that OpenCV, infuriatingly, stores its colours in BGR format rather than RGB. That's right. Let that sink in. The blue and red channels are swapped. You might reasonably be wondering why on earth OpenCV stores its colours this way when almost every other graphical program uses RGB. The answer is that BGR used to be quite popular with camera manufacturers[4].

OpenCV (while incredibly powerful) and specifically the Python binding for it is a mess of style violations and unintuitive naming conventions, but for me the BGR colour space is the most infuriating subversion of the programmatical norm.

Let's be clear: I hate this. We're stuck with it. Let's move on.

Fortunately, OpenCV does provide us with the functionality to convert between the colour spaces using the function `cv2.cvtColor`. We need to pass in our frame, as well as a variable which indicates which converstion we want to do. We will use `cv2.COLOR_RGB2BGR`.

```
    frame = np.swapaxes(pygame.surfarray.array3d(screen), 0, 1)
    frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
    writer.write(frame)
```

Our finished code now looks like this:

```
import pygame
import sys
import cv2
import numpy as np
from pygame.locals import *

pygame.init()

width, height = (640, 360)
screen = pygame.display.set_mode((width, height), 0)

x = 0

writer = cv2.VideoWriter("greenrect.mp4",
                         cv2.VideoWriter_fourcc(*"FMP4"),
                         24,
                         (width, height))

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            writer.release()
            pygame.quit()
            sys.exit()
```

[4] https://www.learnopencv.com/why-does-opencv-use-bgr-color-format/

```python

        screen.fill((0, 0, 0))
        pygame.draw.rect(screen,
                         (255, 0, 0),
                         (x, 10, 100, 50),
                         0)
        x += 1

        frame = np.swapaxes(pygame.surfarray.array3d(screen), 0, 1)
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
        writer.write(frame)

        pygame.display.update()
```

Perhaps it would be fitting to change the name of the output file to *redrect.mp4* but this is ultimately unimportant. Upon running this script and then quitting it after a while, we will see the output file *greenrect.mp4* appear and it will contain the animation we've created.

One thing to note is that your choice of codec will affect how your finished product looks in terms of compression and artifacts. I encourage you to experiment with different codecs and framerates etc. in order to make your video look exactly how you want.

4 Afterword

As I hope you may have noticed, this book is not simply about how to make pictures of trees. Instead I wrote this as more of a general guide for beginner and intermediate-level programmers to help put programming paradigms into the context of real projects.

I wrote this book to teach about concepts such as why OOP is important and why it's relevant to pick a good colour space. It's easy to learn the theory of these ideas in the abstract but it's no substitute for actually going and picking a project and making it, exploring these notions for yourself.

Sure, the algorithms themselves are botanical. Nevertheless, the path we took to implement them – the endless refactoring and rethinking and rearranging – is universal in programming and will no doubt translate seamlessly to other areas of the field.

Furthermore I chose to write the code in Python because of its intuitive readability and its abundance of external libraries. However, except for the final chapter on rendering, none of the overarching techniques or algorithms described in this book are Python-specific. Procedural art is also often created in Processing[1] and the JavaScript port thereof, p5.js[2], is often used to make web-based interactive art.

I suppose that what I'm saying here is that there is no resource more valuable to somebody learning programming than guided experimentation. I encourage you to implement the code discussed herein in a different language, or with a different flair and a creative twist. If you're willing to put in the effort, programming is an art form which can be so much fun to master.

[1] `https://processing.org/`
[2] `https://p5js.org/`