L-System分形图形的绘制

1753557 崔延珅

1753424 宋亮

项目简介

本项目是基于OpenGL实现，用Python语言编写，使用L-System生成算法绘制分形图形的程序.

L-System是由美国生物学家Aristid LinderMayer于1968年提出的一种分形生成算法，起初LinderMayer提出L-System是为了构建一个描述植物生长的数学模型，研究植物的进化和造型，后来Smith、Anono和Kunii率先将L-System引入到计算机图形学中，为计算机模拟植物的真实感图形提供了强有力的工具.

本项目将实现L-System的绘制程序，并定义一些常见的分形图形，以及模拟一个森林的生成.

绘制程序的搭建

基本窗口的建立与绘制

本项目中利用OpenGL和pygame实现窗口的建立和绘制，在窗口建立时，绘制一些基本内容，如背景图，坐标系，并通过pygame进行对按键的响应。

(1)Camera：本项目通过一个Camera类来模拟视点，记录视点的位置，并提供按键响应和视点更新的方法。

按键响应：可以通过按键进行视角的缩放，旋转和还原

```
# 处理按键
    def event(self, e):
        if e.type == pygame.KEYDOWN:
            if e.unicode == 'a':
                self.angleZ += 1
            if e.unicode == 'd':
                self.angleZ -= 1
            if e.unicode == 'w':
                self.angleX += 1
            if e.unicode == 's':
                self.angleX -= 1
            if e.unicode == 'q':
                self.angleY += 1
            if e.unicode == 'e':
                self.angleY -= 1
            if e.key == pygame.K_RETURN:
                self.angleX = 0
                self.angleZ = 0
```

```
                self.distance = 100
        if e.type == pygame.MOUSEBUTTONDOWN:
            if e.button == 5:
                self.distance *= 1.1
            if e.button == 4:
                self.distance *= 0.9
        pass
```

视点更新：根据按键的输入，相关参数会发生变化，视点会每秒更新一次

```
    def look(self, lookat):  # 每隔一段时间更新视点
        # 更新时间
        if time.time() - self.lookat[3] > Conf.GRAPHX.CAMERA_UPDATE_PERIOD or lookat
is (0, 0, 0):
            self.lookat = (lookat[0], lookat[1], lookat[2], time.time())
        gluLookAt(self.lookat[0] + self.distance, self.lookat[1] + self.distance,
self.lookat[2] + self.distance,
                  self.lookat[0], self.lookat[1], self.lookat[2],
                  0, 1, 0)
        glRotated(self.angleY, 0, 1, 0)
        glRotated(self.angleX, 1, 0, 0)
        glRotated(self.angleZ, 0, 0, 1)
```

(2)Graph：本项目利用一个Graph类来进行对OpenGL和pygame的初始化，并提供绘制基本图形等方法，同时会调用Camera的方法，提供更新视点，按键响应的方法。

初始化：

```
        pygame.init()
        self.screen = pygame.display.set_mode((self.width, self.height), OPENGL |
DOUBLEBUF)

        glMatrixMode(GL_PROJECTION)  # 设置投影矩阵
        glLoadIdentity()
         # 透视投影，参数为角度，宽高比，远近裁剪平面距离
        gluPerspective(65.0, self.width / float(self.height), 0.01, 1000.0)
        glMatrixMode(GL_MODELVIEW)  # 设置模型矩阵
        glEnable(GL_DEPTH_TEST)  # 开启深度测试，实现遮挡关系
        self.camera = Camera((0.0, 0, 0))
```

指定着色器：通过glUseProgram()指定着色器，并获取统一变量

```
# 指定着色器：glAttachShader()->glCompileShader()->glLinkProgram()->glUseProgram()
    def setShader(self, shader, unif):
        glUseProgram(shader)
        self.shader = shader
        self.uniform_values = unif
        for name in unif:
            self.uniform_locations[name] = glGetUniformLocation(self.shader, name)
```

绘制基本内容：

```
# 绘制背景，底部，更新视点并展示
    def update(self, lookat=(0, 0, 0)):
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        self.camera.look(lookat)

        for name in self.uniform_values:
            glUniform1f(self.uniform_locations[name], self.uniform_values[name]())

        self.draw_base()

        glClearColor(1, 1, 1, 1)
        pygame.display.flip()
```

Turtle绘制

LoGo - Style Turtle表示法：用于解释L系统重写后的字符串。设v是一个字符串，Turtle有一个初始状态，包括位置与朝向，Turtle会根据字符串的每个字符逐步地做出动作，如前进，后退，旋转，并进行绘制，通过Turtle对字符串的解释所画出的的图称为v的turtle表示。

本项目通过一个Turtle类提供各种解释动作的方法，并且会记录自身的状态，提供用于保存状态的栈，以及存储openGL要绘制的所有点的列表。

初始化：

```
# 初始化当前位置，朝向，颜色
        if pos is None:
            self.pos = Vector(Conf.TURTLE.INIT_POS)
        else:
            self.pos = Vector(pos)
        self.heading = Vector(Conf.TURTLE.INIT_HEADING)
        self.color = Vector(
            colorsys.rgb_to_hls(Conf.TURTLE.INIT_COLOR[0], Conf.TURTLE.INIT_COLOR[1],
Conf.TURTLE.INIT_COLOR[2]))

        self.stateStack = []  # 保存状态的栈，用于绘制分支
        self.stochasticFactor = 0  # 随机因子

        # 保存要绘制的所有点
        self.vertexBuffer = []
        self.vertexBufferLength = 0
        self.vertexBufferChanged = False
        self.vertexPositions = None
        self.draw_type = GL_TRIANGLES

        self.point()
```

记录绘制点：该方法会将Turtle的当前位置记录到要绘制的点列表中，每当位置变化时，此方法都会被调用，最终OpenGL会将此列表作为输入，根据绘制类型绘制出最终图案

```
    # 将当前点加入到要绘制的点集中，breakline代表分支的结束
    def point(self, breakline=0):
        c = colorsys.hls_to_rgb(self.color.x, self.color.y, self.color.z)
        self.vertexBuffer.append([float(self.pos.x), float(self.pos.y),
float(self.pos.z),
                                  c[0], c[1], c[2], breakline])
        self.vertexBufferChanged = True
        self.vertexBufferLength += 1
```

状态操作：将状态入栈和出栈的操作，因为出栈时会回到过去的位置和朝向，所以可以用于分支的绘制

函数	描述
push(self)	将当前状态入栈
pop(self)	取出栈顶的状态，并设为当前状态

基本动作：这些方法通过修改Turtle的状态提供了基本的前进，后退，旋转等操作

函数	描述
forward(self, step)	向着朝向移动step
backward(self, step)	向朝向的反方向移动step
rotX(self, angle)	绕x轴逆时针旋转angle
irotX(self, angle)	绕x轴顺时针旋转angle
rotY(self, angle)	绕y轴逆时针旋转angle
irotY(self, angle)	绕y轴顺时针旋转angle
rotZ(self, angle)	绕z轴逆时针旋转angle
irotZ(self, angle)	绕z轴顺时针旋转angle
setColor(self, color)	将颜色设为color

绘制：这个方法会将绘制点列表中的点全部绘制

```
# 创建vbo，并将点全部绘制
    def draw(self):
        if self.vertexBufferChanged or self.vertexPositions is None:
            vertices = np.array(self.vertexBuffer,
                                dtype=np.float32)
            self.vertexPositions = vbo.VBO(vertices)
            print("Total number of vertex: ", self.vertexBufferLength)
            self.vertexBufferChanged = False

        self.vertexPositions.bind()
        glEnableClientState(GL_VERTEX_ARRAY)
        glEnableClientState(GL_COLOR_ARRAY)

        glVertexPointer(3, GL_FLOAT, 28, self.vertexPositions)  # 三个位置坐标
        glColorPointer(3, GL_FLOAT, 28, self.vertexPositions + 12)  # 三个颜色值

        glDrawArrays(self.draw_type, 0, self.vertexBufferLength)
```

```
        self.vertexPositions.unbind()
        glDisableClientState(GL_VERTEX_ARRAY)
        glDisableClientState(GL_COLOR_ARRAY)
```

L-System

L-System简介

L-System是一种分形的生成算法，由美国生物学家Aristid LinderMayer于1968年提出的，L-System本质是一个重写系统，类似于我们学到编译原理语法推导的过程，起初LinderMayer提出L-System是为了构建一个描述植物生长的数学模型，研究植物的进化和造型，后来Smith、Anono和Kunii率先将L-System引入到计算机图形学中，显示了L-System在计算机模拟植物方面的潜力，为计算机模拟植物的真实感图形提供了强有力的工具.

L-System生成算法

L-System实际上是字符串重写系统，如果我们把字符串解释成图形，那么只要能生成字符串，也就等于生成了图形，L-System的工作原理很简单，仅仅是对几个简单的字符进行操作，L-System可以由以下3部分描述：

- **字符表：**字符串的集合，可以给每个字符赋予不同的图形操作意义；
- **起始元：**作为起始字符串
- **产生式：**是一种替换规则，从某个字符到字符串的变换

例如Koch曲线可以通过如下方式生成：定义字符表{F，-，+}，定义起始符{F}，定义产生式{F => F – F + + F – F}，对于字符表中的3中字符我们定义对应的操作：

F：前进一步（步长是定值）

-：逆时针旋转一定的角度

+：顺时针旋转一定的角度

迭代开始时：F

迭代1次后：F-F++F-F

迭代2次后：F-F++F-F-F-F++F-F++F-F-F-F++F-F

以此类推，通过不断的迭代获得的字符串，转换成图形之后，我们就可以得到Koch曲线的图像.

L-System的实现

定义一个LSystem基类，根据上述的L-System生成算法，L-System的生成过程需要指定以下变量：图形名、字符对应的画图操作、画图操作参数（例如步长和旋转角度等）、迭代次数、开始符、产生式等，所以L-System基础属性如下：

```
class LSystem(object):
    """abstract class"""
```

```
    def __init__(self, turtle):
        self.LSName = "Undefined"                   # 图形名称
        self.LSRules = {}                           # 产生式
        self.LSVars = {}                            # 与turtle的方法对应
        self.LSParams = {}                          # 变换参数
        self.LSAngle = math.pi / 2                  # 默认旋转角度
        self.LSSegment = 1.0                        # 默认移动距离
        self.LSSteps = 5                            # 默认迭代次数
        self.LSStartingString = "F"                 # 默认开始符
        self.LSCode = ""                            # 当前字符串
        self.LSCodeLen = 0                          # 当前字符串长度
```

为了实现作图、着色和控制迭代，需要加入其他属性，这里不一一列出了.

生成图像的过程分为两步，第一步计算迭代产生的最终字符串，通过递归调用generate_param_rec(step)函数实现，每一次调用替换当前字符串中所有可以替换的字符，根据预先设置的迭代次数替换.

```
    def generate_param_rec(self, itern):
        self.LSCodeLen = len(self.LSCode)
        if itern == 0:
            return

        rplc_list = []
        la = [0]

        while la[0] < self.LSCodeLen:
            rplc_list.append(self.apply_rule(la))

        newcode = ''.join(rplc_list)

        self.LSCode = newcode

        self.generate_param_rec(itern - 1)
```

其中apply_rule()函数用于使用相应的产生式进行替换，并将参数写入替换后的字符串，参数的表示用'()'，所以字符表中不能出现'('和')'两个字符.

```
    def apply_rule(self, la):
        char = self.LSCode[la[0]]
        rule = self.rules_contain(char)  # get the corresponding rule
        if rule is not False:
            arg = self.find_arg(la)
            if not arg:
                return rule
            param = self.find_param(rule)  # find the param name of the rule
            parametized_rule = self.LSRules[rule].replace(param, arg)
            return parametized_rule

        else:
            la[0] += 1
            return char
```

得到了最终产生的字符串之后，调用excute()顺序读取字符串，然后对于每一个字符，在self.LSVars中找到对应的turtle操作来绘图，excute()的实现过程：

```
    def execute(self, pos):
        dbg = pos[0]
```

```
        char = self.LSCode[pos[0]]
        if pos[0] + 1 < self.LSCodeLen and self.LSCode[pos[0] + 1] == '(':
            arg = self.find_arg(pos)
            if arg is None:
                return

            for param in self.LSParams:
                if param in arg:
                    arg = arg.replace(param, str(float(self.LSParams[param])))
            res = eval(arg)  # calculate param

            if char in self.LSVars:
                self.LSVars[char](res)  # call corresponding function
        else:
            if char in self.LSVars:
                self.LSVars[char](self.LSParams[char])  # call corresponding function
            pos[0] += 1
```

至此L-System基类的框架就完成了

以下我们生成一些简单的分形图形来测试一下L-System

基本分形图形的生成

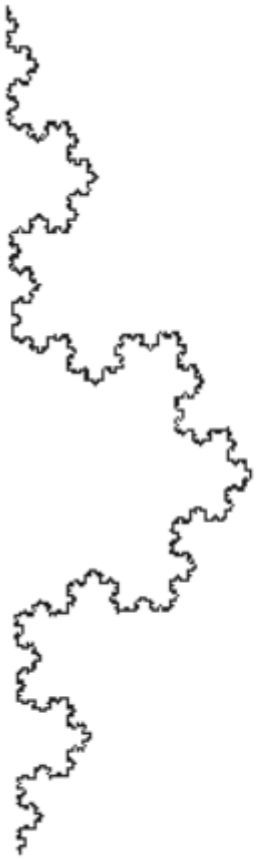
2D Koch曲线

在之前我们提到了2D的Koch曲线如何生成，下面来尝试一下，定义字符表{F，-，+}，定义起始符{F}，定义产生式{F => F – F + + F – F}，步长为0.01，旋转角度定义为$\frac{\pi}{2}$，定义代码如下：

```
class KochCurve(LSystem):
    def defineParams(self):
        self.LSName = "Koch curve"
        self.LSAngle = math.pi / 3
        self.LSSegment = 0.01
        self.LSSteps = 6
        self.LSStartingString = "F"

    def createVars(self):
        self.LSVars = {
            'F':    self.turtle.forward,
            '+':    self.turtle.rotZ,
            '-':    self.turtle.irotZ
        }
        self.LSParams = {
            'F':    self.LSSegment,
            '+':    self.LSAngle,
            '-':    self.LSAngle
        }
    def createRules(self):
        self.LSRules = {
            'F':    "F-F++F-F"
        }
```

迭代次数设为6时，可以观察到生成如下图形



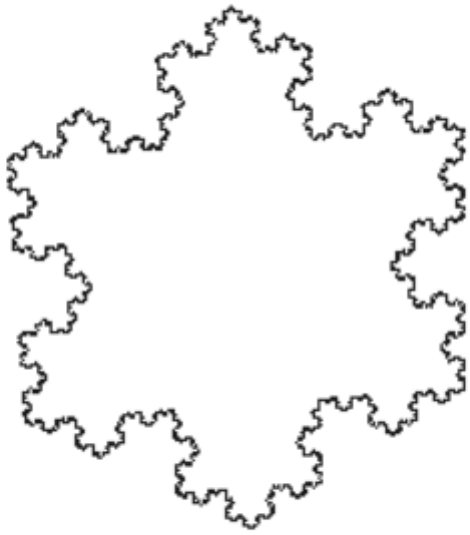
上图其实很像雪花的轮廓，如果将3个上图所示的Koch曲线拼在一起，那么就可以形成一个雪花图案.

Koch雪花

不改变Koch曲线的产生式，要生成Koch雪花图案，则需要改变起始符，原先的起始符为'F'，随着迭代次数的增加，它会朝一个固定的方向延伸下去，所以要使它闭合，我们需要从一个三角形开始迭代，那么这个三角形可以定义为'F++F++F'，所以将上述的KochCurve中的self.LSStartingString修改一下:

```
self.LSStartingString = "F++F++F"
```

就可以产生Koch雪花图案:



Levy曲线

Levy曲线分形,它是将一条线段不停地分形成两条长度相等且相互垂直的线段而生成的.

1827年英国植物学家布朗(R.Brown,1773-1858)用显微镜发现微细颗粒在液体中作无规行走，此现象被称为布朗运动。后来科学家对布朗运动进行了多方面的研究，维纳(N.Wiener，1894-1964)等人在此基础上创立随机过程理论。进入80年代，人们以分形的眼光看待布朗运动，并与“列维飞行”(Levy flight)相联系，找到了确定论与随机论的内在联系。

Levy曲线的定义非常简单，只需要用到两个方向旋转和一个前进操作就可以完成，完整的定义如下：

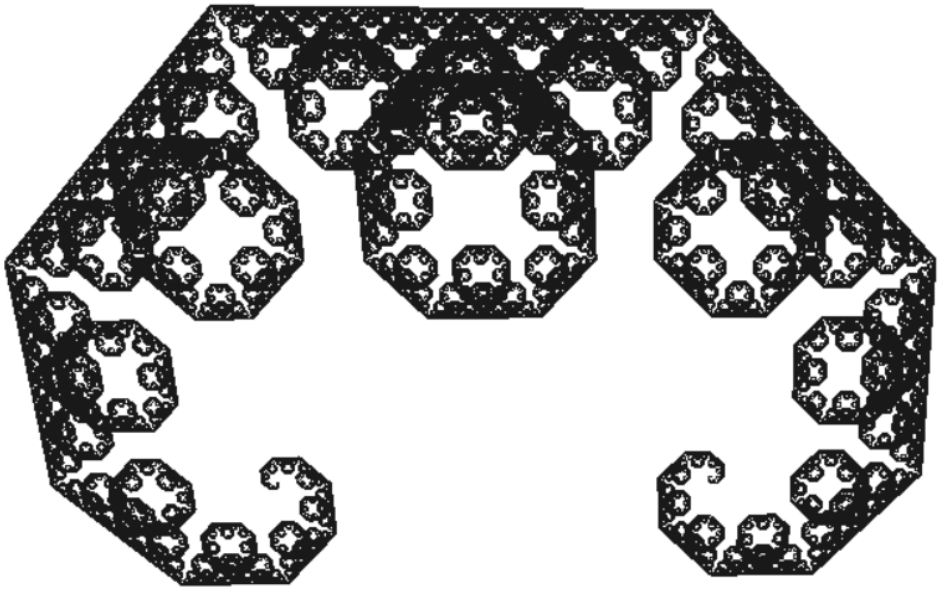
```
class LevyC(LSystem):
    def defineParams(self):
        self.LSName = "Levy"
        self.LSAngle = math.pi / 4
        self.LSSegment = 0.01
        self.LSSteps = 18
        self.LSStartingString = "--F"

    def createVars(self):
        self.LSVars = {
            'F':    self.turtle.forward,
            '+':    self.turtle.rotZ,
            '-':    self.turtle.irotZ
        }
        self.LSParams = {
            'F':    self.LSSegment,
            '+':    self.LSAngle,
            '-':    self.LSAngle
        }

    def createRules(self):
        self.LSRules = {
            'F':    "+F--F+",
```

```
        }
```

运行绘制一个Levy曲线：



Hilbert曲线

Hilbert曲线，只要恰当选择函数，画出一条连续的参数曲线，当参数t在[0，1]区间取值时，曲线将遍历单位正方形中所有的点，得到一条充满空间的曲线。 希尔伯特曲线是一条连续而又不可导的曲线。

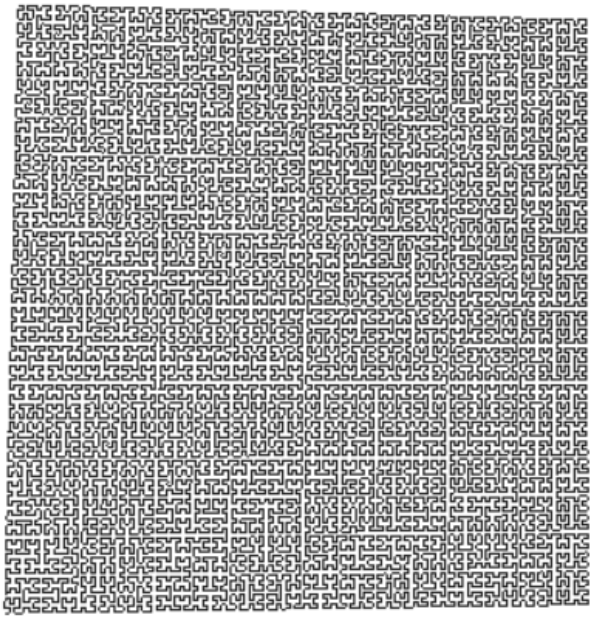
Hilbert曲线可以采用如下的产生式表示：

- A => -BF+AFA+FB-
- B => +AF-BFB-FA+

其中'+'和'-'表示顺时针、逆时针旋转，F表示前进，起始符设为'A'，其他参数定义如下：

```
    def defineParams(self):
        self.LSName = "Hilbert Curve"
        self.LSAngle = math.pi / 2
        self.LSSegment = 0.01
        self.LSSteps = 7
        self.LSStartingString = "A"
```

运行程序绘制Hilbert曲线：



如果继续增大迭代次数，会看到一个全黑色的平面，说明该正方形已经逐渐被Hilbert曲线填满了.

2D树木

生成2D树木的过程会出现分叉的情况，这时整个图形不能通过一条线连成，所以要用到入栈和出栈的操作，入栈时将当前状态存入一个栈中，出栈时会到过去的位置和朝向，从而产生分支.

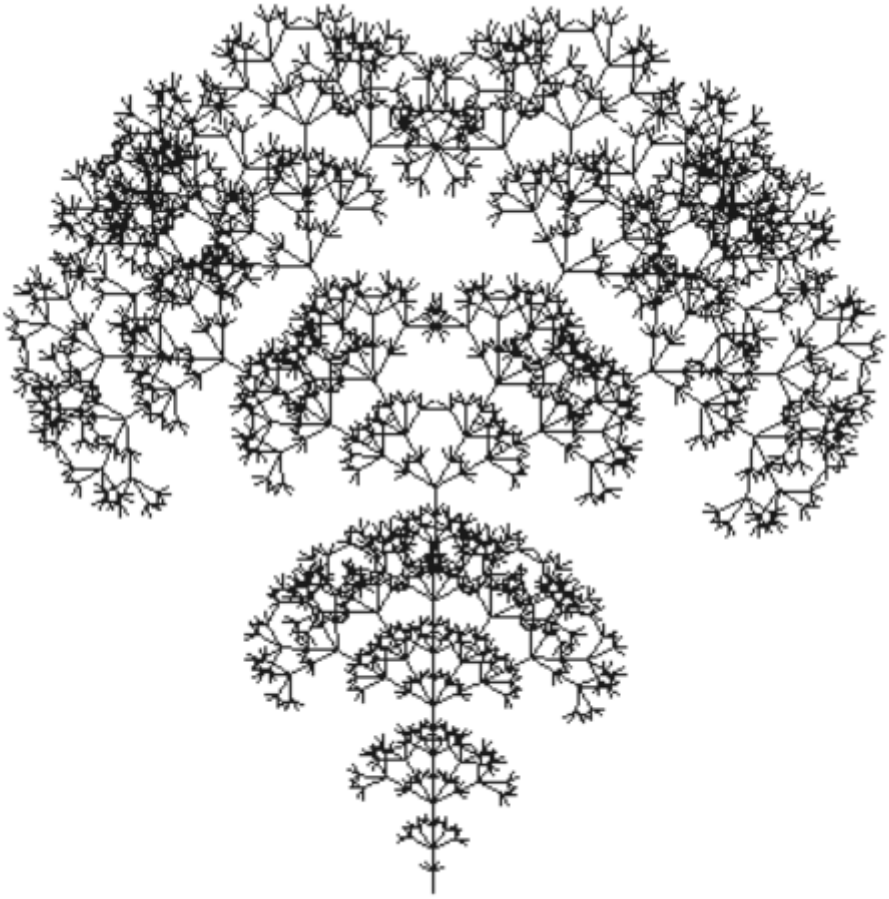
用'['和']'两个字符分别指代入栈和出栈操作，那么我们可以定义一个2D树木的产生式：

```
    def createRules(self):
        self.LSRules = {
            'F':    "FF[--F][-F][+F][++F]"
        }
```

每次迭代可以产生4个分叉，这个分叉数可以根据自己的喜好修改，角度定义为$\frac{\pi}{6}$，起始符定义为'F'.

```
    def defineParams(self):
        self.LSName = "Tree2D"
        self.LSAngle = math.pi / 6
        self.LSSegment = 0.01
        self.LSSteps = 5
        self.LSStartingString = "F"
```

运行之后产生图像：



3D树木

之前2D树木的生成过程用进栈出栈实现了分支，要生成3D树木，不能只在一个轴进行旋转操作，就需要在不同的轴旋转产生新的分支，定义两个X轴的旋转操作'<'和'>'.

另外，要产生更加真实的树木，需要在颜色上做出改变，定义两个设置颜色的操作'I'和'Y'，分别对应的是树干和叶子，每次迭代产生4个分支，并将分支的颜色设置为绿色，就可以得出如下的产生式：

```
    def createRules(self):
        self.LSRules = {
            'F':    "IFF[Y-FF][Y+FF][Y<FF][Y>FF]"
        }
```

此时我们的树木太过于对称，所以要加入一些随机元素，所以将self.LSStochastic设置为True然后将随机因子self.LSStochRange设为1，这里的随机数会影响到绘图时的步长和角度，我们首先尝试将迭代次数设为1，效果如下：

效果不错，增大迭代次数到4，可以看到生成的树木效果如下：

使用同样的分支变色思路来定义一个稍加偏斜的树：

```
    def createRules(self):
        self.LSRules = {
            'F':    "IFF-[Y-F+F+F]+[Y+F-F-F]<+[Y<++F>--F>--F]>-[Y>--F<++F<++F]"
        }
```

运行之后效果如图：

森林的生成

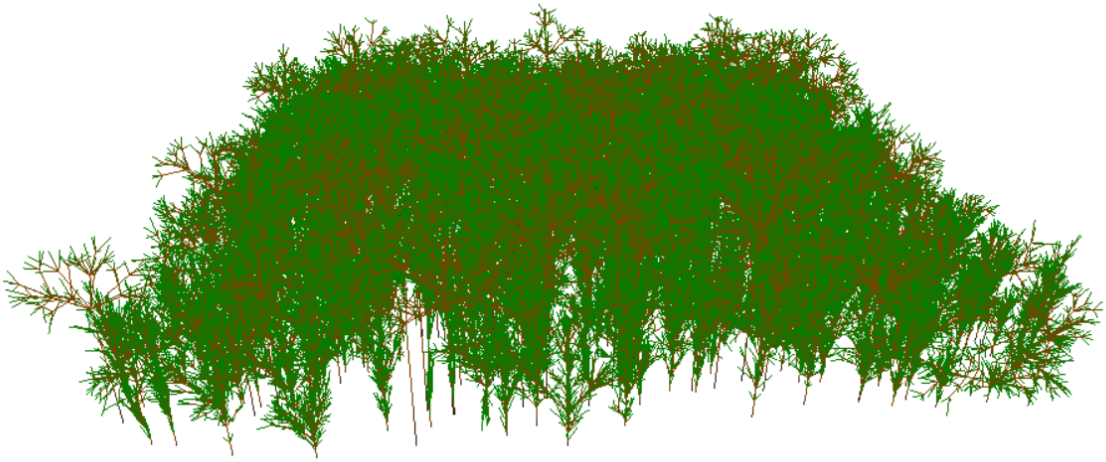
使用前一节类似的方法可以定义更多不同形态的树木，例如增加更多不同角度的分支，甚至通过颜色设置做出由深到浅的变化效果，或者产生树干高低不同的树木（通过在起始符中定义不变的forward操作），通过对同一个树木的变换参数做一些修正也可以产生形态不同的树木.

为了避免生成的森林过于单调，设置一些不同形态的树木：

```
USED = [
    lambda t: Tree1.Tree1(t),
    lambda t: Tree1.Tree1(t, -math.pi / 32),
    lambda t: Tree1.Tree1(t, -math.pi / 64),
    lambda t: Tree3.Tree3(t),
    lambda t: Tree3.Tree3(t, 0.2, 0),
    lambda t: Tree3.Tree3(t, -0.2, 0),
    lambda t: Tree3.Tree3(t, 0, -math.pi / 16),
    lambda t: Tree3.Tree3(t, 0.1, -math.pi / 16),
    lambda t: Tree3.Tree3(t, -0.1, -math.pi / 16),
    lambda t: Tree3.Tree3(t, 0, math.pi / 8),
    lambda t: Tree4.Tree4(t, 1, 0),
    lambda t: Tree4.Tree4(t, 5, 0),
    lambda t: Tree4.Tree4(t, 3, 0),
]
```

将边长设置为30，也就是说生成30*30=900个分形树，然后把他们随机摆放在一个正方形区域里：

```
    def init_grid(self, length, rand_factor):
        for x in range(length):
            for y in range(length):
                randx = random.random() * rand_factor
                posx = -2.5*length + x * length * 5 / (length - 1) + randx
                randy = random.random() * rand_factor
                posy = -2.5*length + y * length * 5 / (length - 1) + randy
                print(posx, ' and  ', posy)
                t = Turtle((posx, 0, posy))
                self.turtles.append(t)
                self.fractals.append(USED[random.randint(0, len(USED) - 1)](t))
```



可改进的方向

- 一个字符对应多个产生式，通过随机选取产生式来构造字符串
- 通过贝塞尔曲线构造曲线的绘图效果

参考文献

[OpenGL] L系统 分形树的实现（L-System植物建模:

https://blog.csdn.net/Mahabharata_/article/details/66967837