CHAPTER-3
CHAPTER-3

STUDY OF GRAMMAR TREE USING L-SYSTEM

Generation is a needful solution to render fractal objects like trees that have a complex geometry characterized by a huge quantity of details. This method is based on the L-system in which branching geometry is used to generate a plant which includes length of trunk, branch angle transfer. Experiments show fractal plant generation in two dimensional planes.

3.1 Rewriting System

The central concept of L-systems is that of rewriting. In general, rewriting is a technique for defining complex objects by successively replacing parts of a simple initial object using a set of rewriting rules or productions.

The most extensively studied and the best understood rewriting systems operate on character strings. The first formal definition of such a system was given at the beginning of this century by Thue [28], but a wide interest in string rewriting was spawned in the late 1950s by Chomsky’s work on formal grammars [66]. He applied the concept of rewriting to describe the syntactic features of natural languages. A few years later Backus and Naur introduced a rewriting-based notation in order to provide a formal definition of the programming language. The equivalence of the Backus-Naur Form (BNF) and the context-free class of Chomsky grammars was soon recognized [105], and a period of fascination with syntax, grammars and their application to computer science began.
In 1968 a biologist, Aristid Lindenmayer, introduced a new type of string-rewriting mechanism, subsequently termed L-systems [5]. The essential difference between Chomsky grammars and L-systems lies in the method of applying productions. In Chomsky grammars productions are applied sequentially, whereas in L-systems they are applied in parallel and simultaneously replace all letters in a given word. This difference reflects the biological motivation of L-systems. Productions are intended to capture cell divisions in multicellular organisms, where many divisions may occur at the same time. Parallel production application has an essential impact on the formal properties of rewriting systems.

3.2 DOL-System

The simplest class of L-systems, those which are deterministic and context-free, called DOL-systems. Consider strings (words) built of two letters a and b, which may occur many times in a string. Each letter is associated with a rewriting rule. The rule $a \rightarrow ab$ means that the letter $a$ is to be replaced by the string $ab$, and the rule $b \rightarrow a$ means that the letter $b$ is to be replaced by $a$. The rewriting process starts from a distinguished string called the axiom. Assume that it consists of a single letter $b$. In the first derivation step (the first step of rewriting) the axiom $b$ is replaced by $a$ using production $b \rightarrow a$. In the second step $a$ is replaced by $ab$ using production $a \rightarrow ab$. The word $ab$ consists of two letters, both of which are simultaneously replaced in the next derivation step. Thus, $a$ is replaced by $ab$, $b$ is replaced by $a$, and the string $aba$ results.

In a similar way, the string $aba$ yields $abab$ which in turn yields $abaababa$, then $abaababaabaab$, and so on.
Formal definitions describing DOL-systems and their operation are given below.

**Definition 1** - Let $V$ denote an alphabet, $V^*$ the set of all words over $V$, and $V^+$ the set of all nonempty words over $V$. OL-system is an ordered triplet $G = (V, \omega, P)$, where $V$ is the alphabet of the system, $\omega \in V^+$ is a nonempty word called the axiom and $P \subseteq V \times V^*$ is a finite set of production. A production $(a, X) \in P$ is usually written as $a \rightarrow X$. The letter $a$ and the word $X$ are called the predecessor and the successor of the production, respectively. An element of $V$ is described in the form $a \rightarrow b$. Furthermore, it is assumed that for $\forall a \in V$, there exists at least one word $b \in V^+$. If no production is explicitly specified for a given predecessor $a \in V$, the identity production $a \rightarrow a$ is assumed to belong to the set of productions $P$.

**Definition 2** - An OL-system is deterministic if and only if for each $a \in V$ there is exactly one $X \in V^*$ such that $a \rightarrow X$.
Definition 3- Let $\mu = a_1a_2...a_m$ be an arbitrary word over $V$. A production $p : a \rightarrow X$ matches a letter $a_i, 1 \leq i \leq m$, if $a = a_i$. The matching production $p$ can be applied to the letter $a_i$, producing the word $X$. If a letter $a_i$ produces a word $X$ as a result of a production application, it is noted $a_i \mapsto X$. The word $\nu = X_1...X_m \in V^*$ is directly derived from $\mu$, noted $\mu \Rightarrow \nu$, if and only if $a_i \mapsto X$ for all $i = 1...m$. A word $\nu$ is generated by $G$ in a derivation of length $n$ if there exists a developmental sequence of words $\mu_0, \mu_1, ..., \mu_n$ such that $\mu_0 = \omega$, $\mu_n = \nu$ and $\mu_i \Rightarrow \mu_{i+1} \Rightarrow ... \Rightarrow \mu_n$.

3.3 Turtle Interpretation of Strings

In order to model higher plants, a more sophisticated graphical interpretation of L-systems is needed. The first results in this direction were published in 1974 by Frijters and Lindenmayer [25], and Hogeweg and Hesper [86]. In both cases, L-systems were used primarily to determine the branching topology of the modeled plants. The geometric aspects, such as the lengths of line segments and the angle values, were added in a post-processing phase. The results of Hogeweg and Hesper were subsequently extended by Smith [10, 11], who demonstrated the potential of L-systems for realistic image synthesis.

The basic idea of turtle interpretation is given below. A state of the turtle is defined as a triplet $(x, y, \alpha)$, where the Cartesian coordinates $(x, y)$ represent the turtle’s position, and the angle $\alpha$, called the heading, is interpreted as the direction in which the turtle is facing. Given the step size $d$ and the angle increment $\delta$, the turtle can respond to commands represented by the following symbols:
Move forward a step of length $d$. The state of the turtle changes to $(x', y', \alpha)$, where $x' = x + d \cos \alpha$ and $y' = y + d \sin \alpha$. A line segment between points $(x, y)$ and $(x', y')$ is drawn.

Move forward a step of length $d$ without drawing a line.

Turn left by angle $\delta$. The next state of the turtle is $(x, y, \alpha + \delta)$. The positive orientation of angles is counterclockwise.

Turn right by angle $\delta$. The next state of the turtle is $(x, y, \alpha - \delta)$.

**Figure 3.3.1 Symbols that cause the turtle to move and draw**

- $F(s)$, $G(s)$ Move forward a step of length $s$ and draw a line segment from the original to the new position of the turtle.
- $f(s)$, $g(s)$ Move forward a step of length $s$ without drawing a line.
- $@O(r)$ Draw a sphere of radius $r$ at the current position.

**Symbols that control turtle orientation in space**

- $+ (\theta)$ Turn left by angle $\theta$ around the $\mathbf{U}$ axis.

- $- (\theta)$ Turn right by angle $\theta$ around the $\mathbf{U}$ axis.

- $\& (\theta)$ Pitch down by angle $\theta$ around the $\mathbf{L}$ axis.

- $\wedge (\theta)$ Pitch up by angle $\theta$ around the $\mathbf{L}$ axis.
\( l(\theta) \)
Roll left by angle \( \theta \) around the \( \vec{H} \) axis.

\( r(\theta) \)
Roll right by angle \( \theta \) around the \( \vec{H} \) axis.

\( | \)
Turn \( 180^\circ \) around the \( \vec{U} \) axis. This is equivalent to \( + (180^\circ) \) or \( - (180^\circ) \).

**Symbols for modeling structures with branches**

[  
Push the current state of the turtle (position, orientation and drawing attributes) onto a pushdown stack.

]  
Pop a state from the stack and make it the current state of the turtle. No line is drawn, although in general the position and orientation of the turtle are changed.

**Symbols for creating and incorporating surfaces**

\{  
Start saving the subsequent positions of the turtle as the vertices of a polygon to be filled.

\}  
Fill the saved polygon.

\~ X(s)  
Draw the surface identified by symbol \( X \), scaled by \( s \), at the turtle's current location and orientation. Such a surface is usually defined as a bicubic patch [37, 83].

**Symbols that change the drawing attributes**

\# (w)  
Set line width to \( w \), or increase the value of the current line width by the default width increment if no parameter is given.

! (w)  
Set line width to \( w \), or decrease the value of the current line width by the default width decrement if no parameter is given.

; (n)  
Set the index of the color map to \( n \), or increase the value of the current
index by the default colour increment if no parameter is given.

\( n \)  
Set the index of the color map to \( n \), or decrease the value of the current index by the default colour decrement if no parameter is given.

### 3.4 Axial Trees

A rooted tree has edges that are labeled and directed. The edge sequences form paths from a distinguished node, called the root or base, to the terminal nodes. An axial tree is a special type of rooted tree (Figure 3.4.1). At each of its nodes, at most one outgoing straight segment is distinguished. All remaining edges are called lateral or side segments. A sequence of segments is called an axis if:

- The first segment in the sequence originates at the root of the tree or as a lateral segment at some node,
- Each subsequent segment is a straight segment, and
- The last segment is not followed by any straight segment in the tree.

Together with all its descendants, an axis constitutes a branch. A branch is itself an axial (sub) tree. Axes and branches are ordered. The axis originating at the root of the entire plant has order zero. An axis originating as a lateral segment of an \( n \)-order parent axis has order \( n+1 \). The order of a branch is equal to the order of its lowest-order or main axis.
In order to model development of branching structures, a rewriting mechanism can be used that operates directly on axial trees. A rewriting rule, or tree production, replaces a predecessor edge by a successor axial tree in such a way that the starting node of the predecessor is identified with the successor’s base and the ending node is identified with the successor’s top.

A tree OL-system $G$ is specified by three components: a set of edge labels $V$, an initial tree $\omega$ with labels from $V$, and a set of tree productions $P$. Given the $L$-system $G$, an axial tree $T_2$ is directly derived from a tree $T_1$, noted $T_1 \Rightarrow T_2$, if $T_2$ is obtained from
by simultaneously replacing each edge in $T_i$ by its successor according to the production set $P$. A tree $T$ is generated by $G$ in a derivation of length $n$ if there exists a sequence of trees $T_0, T_1, \ldots, T_n$ such that $T_0 = \omega, T_n = T$ and $T_0 \Rightarrow T_1 \Rightarrow \ldots \Rightarrow T_n$.

### 3.6 Branching Structure

Lindenmayer [7] introduced the notion of bracketed strings to describe the branching structure of plants. Left and right brackets, "[" and "]", are added to the alphabet of an L-system. In a correctly formed bracketed string, left and right brackets must occur in matching pairs in the same way as parentheses are used in an arithmetic expression. By considering the brackets as delimiters of branches, the bracketed strings can be interpreted as branching structures. Specifically, a left bracket indicates the node of the mother branch to which the daughter branch is to be attached, while the matching right bracket terminates branch specification. The brackets may be nested, indicating higher-order branches. If more than one branch occurs at a site, the bracketed substrings representing each one are listed consecutively in an arbitrary order.

The plant kingdom is dominated by branching structures; thus a mathematical description of tree-like shapes and methods for generating them are needed for modeling purposes. An axial tree [8, 76] complements the graph-theoretic notion of a rooted tree [29] with the botanically motivated notion of branch axis.
Now we can give the code for the branching in Java language:

```java
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class branching extends Applet implements Runnable{
    Thread th;
    Image buf_i;
    Graphics buf_g;

    boolean goFlag=true,growFlag=true,lineFlag=true,plineFlag=true,fpolyFlag=true;
    int w,h,xp0,yp0,stack,step;
    int MAX=100;
    double x[]=new double[MAX];
    double y[]=new double[MAX];
    double z[]=new double[MAX];
    int t[]=new int[MAX];
    int p[]=new int[MAX];
    String str,str_next="";

    int phi=25,alpha=32,beta=25,scale=35,stepmax=15,view=0;

    //GUI
    Button startstop,reset,b_line,b_polyline,b_fillpoly;
    public void init(){
        //get screen size
        w=getSize().width-20;
```


h=getSize().height-100;

//create buffer layer
buf_i=createImage(w,h);
buf_g=buf_i.getGraphics();
init_values();

//GUI
setLayout(new FlowLayout(1,20,h+30));
add(b_line=new Button("hide"));
add(b_polyline=new Button("hide"));
add(b_fillpoly=new Button("hide"));
add(startstop=new Button("stop"));
add(reset=new Button("reset"));
b_line.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        linectrl();
    }
});
b_polyline.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        polylinectrl();
    }
});
b_fillpoly.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){
    fillpolyctrl();
}

startstop.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        startstop_p();
    }
});

reset.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        reset_p();
    }
});

public void start(){
    th=new Thread(this);
    th.start();
}

public void stop(){
    goFlag=false;
}
public void run()
{
    while(goFlag)
    {
        repaint();
        try{Thread.sleep(750);}catch(InterruptedException e){}
    }
}

public void update(Graphics g)
{
    this.paint(g);
}

public void paint(Graphics g)
{
    //initialize
    if(growFlag)step++;
    else view+=5;
    if(goFlag) draw();
    //System.out.println(str);
    g.drawImage(buf_i,10,10,this);
    if(step>=stepmax)growFlag=false;
    //start message
}
g.drawString("frame", w/2-125, h+75);
g.drawString("polygon", w/2-70, h+75);
g.drawString("filled", w/2-5, h+70);
g.drawString("polygon", w/2-10, h+80);
g.drawString("start", w/2+55, h+70);
g.drawString("/stop", w/2+55, h+80);
g.drawString("initialize", w/2+105, h+75);
}

public void draw(){
    init_b();
    init_point();
    str_next="";
    read();
    str=str_next;
}

public void init_point(){
    x[0]=0; y[0]=0; z[0]=0; t[0]=0; p[0]=45;
    xp0=w/2; yp0=0;
}

public void init_values(){

public void init_b()
{
    buf_g.setColor(Color.white);
    buf_g.fillRect(0,0,w,h);
}

public void read()
{
    int len,now=0,param,endp;
    String str_buf;
    char type;
    //get length of string pn
    len=str.length();
    //decode string one by one
    while(now=Math.random()){
        //p1
        if(prob>Math.random()){
            str_next="[+("+(-alpha)+")F(1)A("+(k+1)+")-"+(beta)+")F(1)A("+(k+1)+")";
        }
        //p2
        else{
            //
        }
    }
}
str_next+="B(1)-("+(-beta)+")F(1)A("+(k+1)+")";
}

else{
  //p1
  if(prob>Math.random()){
    str_next+="[+("+alpha+")F(1)A("+(k+1)+")]-("+beta+")F(1)A("+(k+1)+")";
  }
  //p2
  else{
    str_next+="B(1)-("+beta+")F(1)A("+(k+1)+")";
  }
} /*
if(0.5>Math.random()){
  if(0.5>Math.random()){
    //p1
    if(prob>Math.random()){
      str_next+="/("+phi+")[+("+(-alpha)+")F(1)A("+(k+1)+")]-("+(-beta)+") F(1) A ("+ (k+1) +")";
    }
    //p2
    else{
      str_next+="/("+phi+")B(1)-("+(-beta)+")F(1)A("+(k+1)+")";
    }
  }
}*/
```java
else{
    // p1
    if (prob > Math.random()) {
        str_next += ";(+phi+)[+(+alpha+)F(1)A(+k+1+)]-(+beta+)F(1)A(+k+1+);"
    }
    // p2
    else {
        str_next += ";(+phi+)B(1)-(+beta+)F(1)A(+k+1+);"
    }
}
else{
    if (0.5 > Math.random()) {
        // p1
        if (prob > Math.random()) {
            str_next += ";(-phi+)[+(-alpha+)F(1)A(+k+1+)]-(-beta+) F(1) A(+k+1+);"
        }
        // p2
        else {
            str_next += ";(-phi+)[+(-alpha+)F(1)A(+k+1+)]-(-beta+) F(1) A(+k+1+);"
        }
    }
    else {
        // p1
        if (prob > Math.random()) {
            str_next += ";(+phi+)B(1)-(+beta+)F(1)A(+k+1+);"
        }
        // p2
        else {
            str_next += ";(+phi+)B(1)-(+beta+)F(1)A(+k+1+);"
        }
    }
}
```
str_next+="/("+(phi)+")B(1)-("+(beta)+")F(1)A("+(k+1)+")";

    }
}

else{
    //p1
    if(prob>Math.random()){
        str_next+="/("+(phi)+")[+("+alpha")F(1)A("+(k+1)+")]-("+beta")F(1)A("+(k+1)+")";
    }
    //p2
    else{
        str_next+="/("+(phi)+")B(1)-("+beta")F(1)A("+(k+1)+")";
    }
}
else{
    str_next+="A("+k")";
}

public void bud(int s){
    //start draw circle
    xp0=(int)((double)w/2+y[0]);

    57
yp0=(int)((double)h-z[0]);

//buf_g.setColor(Color.green);
//buf_g.fillOval(xp0,yp0,s,s);
//end draw cirle

if(growFlag)str_next+="B("+(s+1)+")";
elsestr_next+="B("+s+")";
}

public void startstop_p(){
if(goFlag){
    goFlag=false;
    startstop.setLabel("start");
}
else{
    goFlag=true;
    start();
    startstop.setLabel("stop");
}
}

public void reset_p(){
    init_b();
    init_point();
    init_values();
}
growFlag=true;

goFlag=false;

startstop.setLabel("start");

startstop_p();

}

public void linecrl(){
  if(lineFlag){
    lineFlag=false;
    b_line.setLabel("show");
  }
  else{
    lineFlag=true;
    b_line.setLabel("hide");
  }
}

public void polylinctrl(){
  if(plineFlag){
    plineFlag=false;
    b_polyline.setLabel("show");
  }
  else{
    plineFlag=true;
    b_polyline.setLabel("hide");
  }
}
plineFlag=true;
  
b_polyline.setLabel("hide");
}
}

public void fillpolyctrl()
{
  if(fpolyFlag)
  {
    fpolyFlag=false;
    b_fillpoly.setLabel("show");
  }
  else
  {
    fpolyFlag=true;
    b_fillpoly.setLabel("hide");
  }
}

Figure-3.6.1 Branching tree (4, 0.5, [12,-12], [0, 0], [0, 1])
3.7 Conclusion:

L-system model integrate local process, taking place at the level of individual modules, into developmental patterns and structures of entire plants. We suggest a grammar for generating a tree which consist a short trunk and subtrees.