Procedural generating of plants models using L-system

Jungming Suh
University of Southern California
suhjungm@usc.edu

Han Zhang
University of Southern California
zhan704@usc.edu

Zian Wang
University of Southern California
zianwang@usc.edu

ABSTRACT

In this project we made a demonstration of generating 3D natural-like plants models using L-system grammar with customized featured graphic interpreter.

Our works can be summarized as follows. Firstly, We extended the growing principles to three-dimension and built a mesh model generator that is capable of generating plants models meshes on demand. Secondly we build our own OpenGL renderer for rendering the generated models. Finally, we designed an interactive GUI to view and manipulate the plant models in real time.

1 INTRODUCTION

Plants models are everywhere in a game scene, such as *MineCraft*[1] (Figure 1), and *Super Mario Bro*'s wild natural-like environments. However in such an environment a huge number of plants need to be generated to satisfy the requirements that they are slightly different from each other. Here the main challenge is that the task of making a realistic and creditable environment normally takes a team of professional designers several months, which is not affordable in larger game design. Fortunately, a principle of *Procedural Content Generation* (also referred as *PCG*)[2] was introduced to game design in 70's. Designers can thus utilize predefined algorithms to automatically generate large scale of models or environment efficiently.

L-system[3] is one of those principles for procedural generation. L-system (or Lindenmayer system) was initially brought by biologist Lindermayer when he attempted to modeling various growth patterns of bacteria[4][5]. The recursive nature of L-system is fit for modeling plants because plants have branch structures which can be properly described by a fractal-like grammar system. Thus using L-system aided plants models generator is widely accepted by the computer graphic industrial[6].

We designed our own generating algorithm and show it in our interactive GUI using OpenGL. To describe our work accurately, this paper is organized as follows. First we make series of definitions in Section 2. In Section 3, we present our 3D models generating algorithms. After that detailed software features are shown in Section 4, where reader can get information about technical details such as OpenGL and GUI. Finally, in Section 5 we show the future works needed to be done when we apply this to potential commercial use.

2 DEFINITIONS AND NOTATIONS

In this section we make several fundamental definitions. The core spirit of L-system is recursively rewriting a string following certain rules. These rules are referred as "Grammars" in the rest of this paper. We give a formal definition of grammar to avoid potential ambiguities.

Figure 1: MineCraft[1]'s wild scenes are generated using *PCG*. One of the advantages is that the scenes are capable of extending to a wider space efficiently. The game is thus worth playing again by changing levels or quests and hence offering new stories and impressions in each new session while the game mechanism remains the same.

2.1 Grammar

Definition 2.1. A *Grammar* is *a set of production rules* for rewriting strings[6].

Here we denote a *Grammar* with G. Each element $g_v \in G$ is a replacing rule that replace a single symbol v with another symbol u or another string A consisting several symbols during a string rewriting process. A simple example works as follows:

Example 2.2. $G = \{A \rightarrow AB, B \rightarrow b\}$
In a single rewrite process: $AB \xrightarrow{G} ABb$

In the example above, when rewriting starts, we replace A with string AB following the first rule of G. At the same time, the second symbol in the original string AB, namely B is rewritten as b. So the new string after such transformation is ABb.

So far we have introduced the concept of Grammar. More specifically, Grammar can be further divided into two groups based on whether the rules are determined for each symbol. For those deterministic Grammars, each replaceable symbol maps to a unique rule. However this is not the case for another group of Grammars, whose rules are non-deterministic. A symbol may have more than one rewriting rules, when programs incur such symbols, they choose to execute one of the rules based on certain conditions. Although the ambiguity of the latter Grammars set difficulties for us to predict or reproduce the results, they provide us with more flexibility to create various sentences

2.2 L-system

Generally speaking, L-systems are a class of grammars whose defining feature is parallel rewriting. By recursively rewriting the string,

we create the self-similarity of a growing complicated system between its local and global structures. We make the following indispensable definitions before the definition of L-system:

- **Variables**: denoted as $\mathcal{V}$, is a set of symbols used in a string that are being replace in every recursion level.
- **Constants**: denoted as $\mathcal{S}$, is a set of symbols used in a string that are always retained with no corresponding replacing rules.
- **Axiom**: denoted as ω, is the initial string from the beginning of the growth.
- **Rules**: denoted as **P**, is a set of *Grammars* we defined above such that for every symbol $v \in \mathcal{S}$, there is a rule $g_v \in \mathbf{P}$ to replace v with a certain string A, where for each symbol $a \in A$, $a \in \mathcal{S} \parallel a \in \mathcal{V}$ holds true.

Now we can come to the definition of L-systems:

Definition 2.3. L-systems are four-element tuples consisting $\{\mathcal{V}, \mathcal{S}, \omega, \mathbf{P}\}$. They start from an initial string ω. In each iteration, every variable symbol s is rewritten based on rules **P**. It terminates when certain conditions are satisfied.

We list a simple example to illustrate our definitions.

Table 1: Fibonacci example of L-system

variables	constants	axiom	rules
A, B	*none*	A	$A \rightarrow B, B \rightarrow AB$

Table 2: Fibonacci String in six iterations

n=0	A
n=1	B
n=2	AB
n=3	BAB
n=4	ABBAB
n=5	BABABBAB
n=6	ABBABBABABBAB

2.3 Graphic Interpreter

The story so far is that we have already discussed L-system. However, the essence of L-system is a string rewriting system. A set of smart interpretation rules should be created for the purpose of generating models on demand. Intuitively, the idea is that the program read the generated string from left to right. Let the symbols represent the direction of the next stroke. The program reads the string while holding a "pen", draw a stroke to the next point following the direction the current symbol represented. This method works successfully in *Sierpiński Arrowhead Curve* example[7]. Unfortunately, the use of this method is only limited to linear drawing system, where models are drawn in one stroke without rolling back. In a more realistic case, for example a typical branched tree, we need a more powerful way that allow us to come back to the stem when we finished building a branch.

Algorithm 1: The Rewrite Algorithm

```
Input  : sentence
Output : newsentence
initialize an empty sentence: newsentence;
for symbol in sentence do
    switch symbol do
        case A or [ or ] or L do
            Append to newsentence with ;
        end
        case T do
            if the current branch is long enough and with a
              probability of p1 then
                Append to newsentence with [T][T]...[T];
            end
            else if the current tree is high enough and with a
              probability of p2 then
                Append to newsentence with ALT;
            end
            else
                Append to newsentence with AT;
            end
        end
    end
end
```

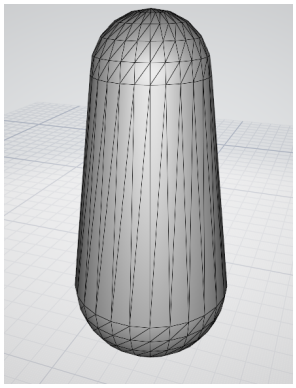
2.3.1 Bracketed Sequential Reading. We let the program maintain a stack data structure when it is reading the string. A node recording the current location and direction will be pushed into the stack when it incurs a left bracket symbol "[". After a short period at the time it incurs a right bracket symbol "]", it pops up the stack and restore to the location and direction recorded in the node. A more specific example will be shown in next section where we present our algorithm for 3D modeling.

3 3-D MODELS GENERATOR

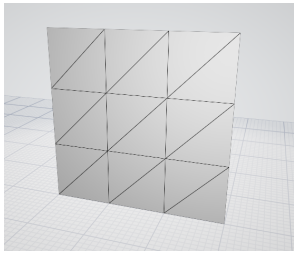
In this section we will go to details of our 3-D generator system. The algorithms are inspired from ideas of J.Knulzen's thesis[8] and Viruchpintu's thesis[9]. Specifically, in our project we focus on generating tree models. This system comprises of two algorithms. The first one is a sentences generator which accepts parameters from users and make sentences with a non-deterministic L-system's rules. The second part is what we have discussed as a graphic interpreter. The function of the interpreter is to translate the sentences that comprise symbols to a 3-D model mesh constructed by combining several kinds of meta models. We will discuss the sentences generator part first.

3.1 Sentences Generator

First of all we introduce the meaning of our symbols. For a growing tree one of the most important components can be the the growing branches. So here we use **A** to denote a piece of branch. To simulate the growing process, an **A** has a specific physical length, we let the branches grow by appending several **A**s to the current branch. Similarly, an **L** represents a "Leaf" in our system. A "Leaf" has its



(a) Branch Meta Model



(b) Leaf Meta Model

Figure 2: Meta Models

own physical properties such as length and width as well, but leaves are not extended to bigger ones. Please note that both **A** and **L** are **Constants** in grammar. On the contrary, **T** is the only **Variable** that can be replaced in the rewriting process. **T** in essence is analogy to growing points of plants where new cells are bifurcated from old cells actively. A single **T** is rewritten by prepending the original **T** with some **A**s or **L** or even more **T**s depending on the condition of current branch. Besides, as mentioned in the previous section, we have symbols **[** and **]** to facilitate our rewriting algorithm.

A tricky mechanism here is created to control bifurcating rules. Three parameters are passed in, among which two are the thresholds– *branchheight* and *leafheight*. When a branch is growing, we continuously add **A**s to the branch so the physical length keeps growing. At the time it hits the threshold *branchheight*, the **T**'s replacing rule significantly changes by allowing it to bifurcate with a certain probability specified ahead of time. The third parameter—*branchnumber* determines the number of new **T**s that the old **T** should be replaced with. Similarly, when another threshold *leafheight* is hit, we allow it to have a leaf on the branch also with a probability. The detailed algorithm can be find in Algorithm 1.

3.2 Meta models

Even if with the help of L-system as blueprints, modeling a bifurcating plant is still difficult. To further simplify this problem, we introduce the idea that we use meta models(Figure 2) to represent branches and leaves. However, such meta models are not manually designed models. They are procedural generated on demand by specifying height, width etc. In the next step we just naively assemble these models using our graphic interpreter by putting them together. In practice, the joint parts look smooth even if we don't do further processing. The possible reasons may rely on the fact that branches are tall and thin, the intersection parts are too minor to be seen with eyes. Besides, proper texture mapping with tree barks can be cheating thus partially covers up the defects.

3.3 Building Tree models

In order to build a 3D model by an L-system sentence, we need to interpret the sentence as branches and leaves with parameters.

Algorithm 2: 3D Model Generation Algorithm

```
Input : sentence, currentBottom, currentTop,
        currentTopRadius, currentBottomRadius,
        currentLength
Output: vertexVector, indexVector
initialize empty vectors: vertexVector, indexVector;
for symbol in sentence do
    switch symbol do
        case A do
            M=computeTransMatrix(currentBottom,
             currentTop);
            vertexBuffer,indexBuffer
             =genBranchMetaModel(currentTopRadius,
             currentBottomRadius, currentLength);
            vertexVector <--pushback-- M · vertexBuffer;
            indexVector <--pushback-- indexBuffer;
            update currentBottom, currentTop,
             currentTopRadius, currentBottomRadius,
             currentLength by M, Length ratio, Radius Ratio;
        end
        case L do
            M=computeTransMatrix(currentBottom,
             currentTop);
            vertexBuffer,indexBuffer
             =genLeafMetaModel(currentLength);
            vertexVector <--pushback-- M · vertexBuffer;
            indexVector <--pushback-- indexBuffer;
        end
        case [ do
            randomly generate Vertical angle and
             Horizontal angle;
            M=computeFurcateTransMatrix(currentBottom,
             currentTop, Vertical angle, Horizontal angle);
            compute Shrinkage ratio by Vertical angle;
            update currentBottom, currentTop,
             currentTopRadius, currentBottomRadius,
             currentLength by M, Shrinkage ratio;
            3DModelGeneration(sentence.substring(symbol),
             currentBottom, currentTop, currentTopRadius,
             currentBottomRadius, currentLength);
        end
        case ] do
            restore currentBottom, currentTop,
             currentTopRadius, currentBottomRadius,
             currentLength;
        end
    end
end
```

Specifically, we translate **A** to branches, **L** to leaves, and **[** to parameters which control how to generate child branches from a parent

Algorithm 3: Compute Transformation Matrix

input : currentBottom, currentTop
output : X_{wm}

orientation = currentTop- currentBottom;
θ_1 = arccos(orientation.y / length(orientation));
lengthXZ = length((orientation.x, orientation.z));
θ_2 = lengthXZ == 0 ? 0 : arccos(orientation.z / lengthXZ);
if orientation.x < 0 **then**
  θ_2 = 2π - θ_2;
end
X_{wm} = T(currentBottom) · $R_y(\theta_2)$ · $R_x(\theta_1)$;

Algorithm 4: Compute Furcate Transformation Matrix

input : currentBottom, currentTop, verticalAngle, horizontalAngle
output : X_{wm}

generate θ_1 and θ_2 as Algorithm 3 does;
orientation = currentTop- currentBottom;
normalVector = (orientation.y, -orientation.x, 0);
X_{wm} = T(currentBottom) · $R_{\text{orientation}}$(horizontalAngle) · $R_{\text{normalVector}}$(verticalAngle) · $R_y(\theta_2)$ · $R_x(\theta_1)$;

branch[9]. In addition, we use **[** and **]** to save and restore these parameters. Before we build the modeling system, two additional factors should be considered in order to make the tree look more natural.

The first thing are the shapes of our meta models. A common observation is that the branches are thinner and shorter as the tree gets taller. Therefore, we use a truncated cone with two hemispheres to model a branch (Figure 2a). The rates at which the branches become thinner and shorter is determined by **length ratio** and **radius ratio** respectively. In order to render different kinds of leaves, we build a leaf model as a rectangle panel (Figure 2b) and render it into the shape of a leaf by mapping its texture to images with transparent channels.

The second fact is that we observe that trunks of trees are more likely to be thicker than side branches. In order to implement such mechanism, we let those side branches whose have larger angles with the their branches become thinner. The ratio of this angle to the reciprocal of thickness of child branch is called the **shrinkage rate**.

Besides the above parameters, We list some other parameters that control the tree's growth.

- **Length ratio**: is the ratio of the length of a parent branch to it of its child branches.
- **Radius ratio**: is the ratio of the top surface radius to the base radius of a truncated cone.
- **Vertical angle**: is the angle between the child branch and the parent branch.
- **Horizontal angle**: is the angle between the child branches projected on the parent branch's normal plane.
- **Shrinkage rate**: is the ratio of the thickness of the child branch to it of the parent branch.

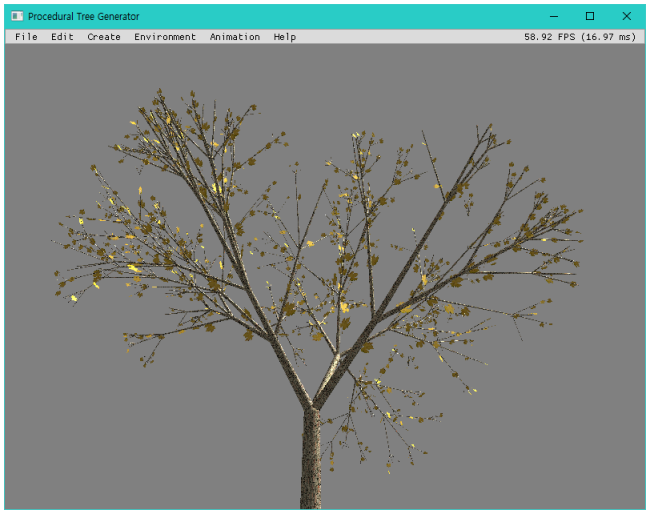


Figure 3: View port

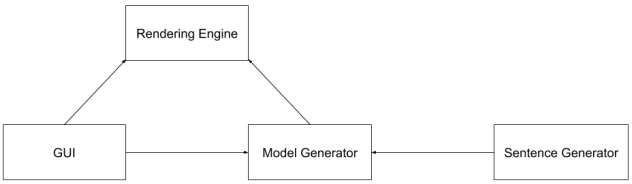


Figure 4: Model Diagram

The algorithm for generating 3D models is shown in Algorithm 2. When we encounter an **A** or an **L**, we need to generate a meta model and transfer its vertices coordinates from its model space to world space. The generation of the meta model require *currentTopRadius*, *currentBottomRadius*, and *currentLength*. Two more parameters—*currentBottom* and *currentTop* are needed in order to save the position and orientation of current branch, which create transformation matrices to move meta models to correct positions and orientations. The process of calculating transformation matrices is shown in the algorithm 3. Because the branches get thinner, we update *currentTopRadius*, *currentBottomRadius*, and *currentLength* after generating a branch.

When we encounter a [, which means a parent branch will furcate to generate several child branches, we need to save the position and the orientation of the growing branch and use randomly generated Vertical angle and Horizontal angle which change the orientation of child branches. Because the difference between *currentTop* and *currentBottom* can be regarded as the orientation of branches. We will use a transformation matrix to move *currentTop* and achieve the goal of changing orientation. The process of calculating the matrices is shown in the algorithm 4. In the algorithm 4, $T(vector_A)$ denotes translating by vector $vector_A$, $R_{vector_B}(angle_C)$ denotes rotating about the axis of vector $vector_B$ by angle $angle_C$. In addition, we will restore them until we encounter a].

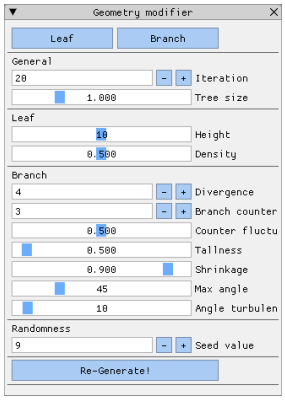


Figure 5: Geometry modifier

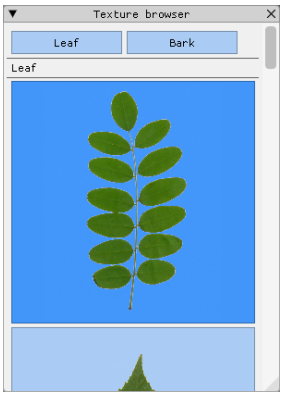


Figure 6: Texture browser

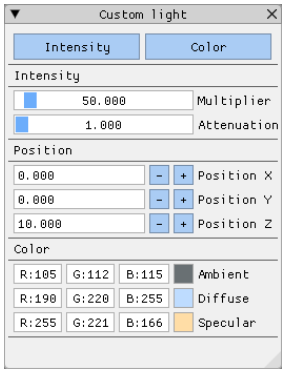


Figure 7: Custom light modifier

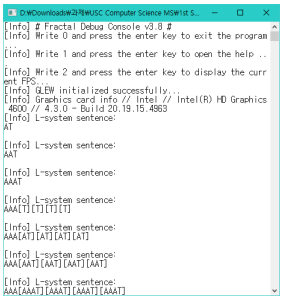


Figure 8: Debug console

4 SOFTWARE IMPLEMENTS

Our system integrates OpenGL, *ImGui*, 3D model generator, and L-system, which can generate and display trees in nature (Figure 3). Our system works as shown in Figure 4. Specifically, **Sentence Generator** constructs sentences representing trees based on L-system. **Model Generator** constructs tree models according to the sentences generated by **Sentence Generator** and parameters received from users. **Rendering Engine** renders the models generated by **Sentence Generator** and renders them differently based on various user inputs. GUI accepts user inputs and passes parameters to **Rendering Engine** and **Model Generator**.

OpenGL enables our **Rendering Engine** to efficiently implement coordinate transformations, lighting and shading, loading image texture, anti-aliasing, and transparency.

Our GUI is based on *ImGui*, which accepts mouse and keyboard inputs and generates various trees. It has GUI functions such as I/O features, bark and leaf texture changing, user-customized lighting and tree generating by user-defined parameters.

4.1 Features

Our procedural generating system contains a variety of textures of bark and leaves that can be replaced to simulate trees more realistically. Texture browser (Figure 6) can help us to quickly replace the texture. We have built-in lights and custom lights to more clearly show the trees in different lighting conditions (Figure 7). Moreover, in order to generate various trees, we have a geometry modifier (Figure 5) that allows the user to modify multiple parameters such as the number of iterations of growth, max vertical angle and the number of child branches when a parent branch furcates, etc.

We summarize the parameters that users can set by themselves as follows.

- **Iteration**: how long the tree grows.
- **Tree size**: the thickness of the tree.
- **Leaf height**: how tall trees begin to grow leaves.
- **Leaf density**: the density of leaves.

(a) Iterate 10 times (b) Iterate 20 times (c) Iterate 30 times

Figure 9: Tree with different iteration times

(a) 2 child branches (b) 3 child branches (c) 4 child branches

Figure 10: Tree with different child branch numbers

(a) Texture 1 (b) Texture 2

Figure 11: Tree with different textures

- **Branch divergence**: how many child branches when a parent branch furcates.
- **Branch counter**: the length of the branches.
- **Branch counter fluctuation**: the probability of branching.
- **Branch tallness**: the height of the tree.
- **Branch shrinkage**: the extent to which the child branches become thinner when a parent branch furcates.
- **Branch max angle**: the upper limit of the vertical angle between the child branch and the parent branch.
- **Branch angle turbulence**: the degree of randomization of the horizontal angle of the child branches.
- **Random seed**: the number used to initialize a pseudorandom number generator.

The specific effects of changing part of the parameters would be shown in next section. The user can get the L-system sentences generated by the sentence generator through debug console (Figure 8).

4.2 Results

In this section, we show some trees with different parameters or textures. For example, the number of iterations controls the time of tree growth, Figure 9 shows trees with different iterations. We can also control the number of child branches when the parent branch are furcating. Figure 10 shows the result of trees with different child branches. In addition, we can change the bark and leaf texture or trees (Figure 11).

5 FUTURE WORKS

Several features can be extended in our project to improve the degree of completion for the potential commercial use. Here we list some of the features we could added due to the limited time for this project.

- **Shadow mapping**: Shadow mapping[10] can be considered for a more realistic scene like a forest.
- **Bump mapping**: Currently we could see specular light on the tree's body because the tree's body is nearly round and smooth where normals are perpendicular to the geometric shape. To make it looks more natural where specular lights are invisible and the barks are rough and wrinkled, bump mapping[11] can be applied here by slightly turbulent each normal on the surface.
- **Animation**: This feature is to give users an intuitive impression of how the models evolve during each iteration. Since we rewrite the sentence in each iteration, we can simply trace the evolution history by interpreting each sentence in the history to a individual model. These models serve as "snapshots" of the tree's growth and form each frame in the animation.
- **Migration**: The L-system tree has a very tiny file size, regardless of the complexity of the model. If it can be applied to commercial game engines, it will contribute greatly to reducing file size of open world games.

REFERENCES

[1] Mojang. Minecraft. Video Game, 2009.

[2] Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1:4, 2017.

[3] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[4] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18:280–99, 04 1968.

[5] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of theoretical biology*, 18:300–15, 04 1968.

[6] Julian Togelius, Noor Shaker, and Joris Dormans. *Grammars and L-systems with applications to vegetation and levels*, pages 73–98. Springer International Publishing, Cham, 2016.

[7] B.B. Mandelbrot, W.H. Freeman, and Company. *The Fractal Geometry of Nature*. Einaudi paperbacks. 1997, 1983.

[8] J. Knutzen and Institutionen för data-och informationsteknik. *Generating Climbing Plants Using L-systems*. Chalmers University of Technology, 2009.

[9] Rawin Viruchpintu and Noppadon Khiripet. Real-time 3d plant structure modeling by l-system with actual measurement parameters. *National Electronics and Computer Technology Center, Bangkok*, 2005.

[10] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[11] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 286–292, New York, NY, USA, 1978. ACM.