# MODELING DENDRITIC SHAPES – USING PATH PLANNING

A Thesis Submitted to the

College of Graduate Studies and Research

In Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By Ling Xu

# PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

# ABSTRACT

Dendritic shapes are commonplace in the natural world such as trees, lichens, coral and lightning. Models of dendritic shapes are widely needed in many areas. Because of their branching fractal and erratic structures modeling dendritic shapes is a tricky task. Existing methods for modeling dendritic shapes are slow and complicated.

In this thesis we present a procedural algorithm of using path planning to model dendritic shapes. We generate a dendrite by finding the least-cost paths from multiple endpoints to a common generator and use the dendrite to build the geometric model. With the control handles of endpoint placement, fractal shape, edge weights distribution and path width, we create different shapes of dendrites that simulate different kinds of dendritic shapes very well. Compared with some existing methods, our algorithm is fast and simple.

# ACKNOWLEDGEMENTS

The author would like to express the most sincere thanks to Professor David Mould. He helped me to take the first step and guided me with much kindness and patience. Without him, there is not this thesis.

I would like to thank my parents and husband for their support and encouragement.

And thanks to the thesis committee for their great comments and suggestions.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

As one of the major parts of computer graphics, modeling plays an important role in many areas such as 3D movie industry and scientific research. Modeling is the technique to "deal with the mathematical specification of shape and appearance properties in a way that can be stored on the computer" [58]. In this thesis we will introduce our algorithm for modeling dendritic shapes, which are commonly seen but tricky for modeling task.

Our discussion will focus on the method of creating geometric models. Geometric models describe objects using collections of geometric components such as spheres, cubes, cones and polygons. We often visualize the geometric models by the synthetic images. There are some popular methods for creating geometric models described as follows.

A common method to generate models is to use interactive modeling software such as 3D Studio Max, Maya, Blender, LightWave and so on. Artists or engineers manipulate the model shape by hand according to their desire. The main advantage of this method is that it can create a very complicated model such as a human face. But the drawback is that the process is time-consuming and needs a lot of human labor.

Scanning is another way to obtain 3D models. A 3D-scanner samples points from the surface of a physical object. The output includes the information of geometry, color and texture data. The advantages of the method are fast 3D data acquisition and precise description of the real object. Compared with the method of using interactive modeling software, scanning does not need much manual manipulation especially for complicated models, and the input is just a real object, so it can be applied to a wide range of objects. But most 3D scanning devices are too specialized and complex for ordinary users to operate themselves [6] and there are strict criteria for getting satisfying scanned data. These require the operator to be skilled and familiar with the expectations of the result. The premise of the scanning method is the existence of the real object. Sometimes it is an obstacle for creating models for objects not suitable for scanning (such as large sized natural objects) or not available in daily life.

Another method is procedural modeling. Procedural techniques refer to "code segments or algorithms that specify some characteristic of a computer-generated model or effect" [57]. Procedural modeling focuses on abstracting the complex details of the object into a function or algorithm and creating the model from these rules. Users can control the modeling by adjusting parameters in the function or algorithm to generate large amount of variations so can be released from the arduous manual manipulation of details. Procedural

modeling is often used when models are too complicated or tedious to create. Modeling dendritic shapes is just one of these cases.

## 1.1 Motivation and Problem Statement

Dendritic shapes are commonplace in the natural world. The term 'dendrite' comes from the Greek word to mean a tree. In this thesis we use it to denote the tree shaped structure that has many branches emerging from a root. There are many examples of objects exhibiting dendritic shapes including lichens, coral, lightning, trees, rivers and crystals. Models of dendritic forms often appear in art works, movies, games and scientific research areas. Although dendritic models are widely needed, modeling the dendritic shapes is a tricky task. The difficulty lies in the branching fractal structures and the erratic winding travels of individual branches with different widths as shown in some examples of dendritic objects in Figure 1.1.

**Figure 1.1:** Objects exhibiting dendritic shapes. From left to right: rivers, elm tree and corals

Some computer graphics practitioners have already made many efforts to implement the task and obtained good results. In general, common methods for modeling dendritic shapes can be roughly classified as following types.

1. *Physically Based Method*. Physically based methods tend to simulate the physical process to grow the dendrite, such as the visual simulation of ice formation [25, 23] and animation of lightning [24]. Because the models are obtained by the simulation of their natural formation process they look realistic. But this method needs the related physics knowledge; it is not easy for users who just want to create visually esthetic models.

   As a special case of physically based method, diffusion-limited aggregation (DLA) is often used to create dendritic structures. DLA is the process whereby particles undergo a random walk due to Brownian motion cluster together to form aggregation of such particles. The resulting clusters generated by DLA are similar to some natural dendritic shapes such as lichens, crystals, neurons, and

lightning [5]. Although DLA can generate very realistic dendrites, because most of time is spent on the random walk of particles before they aggregate, the whole process is very slow.

2. $L-system$. L-system is a parallel rewriting grammar beginning from an initial string and repeatedly uses a replacement rule to create strings which can be interpreted as a variety of botanical forms, particularly branching structures. L-system has been used to model plants and achieve realistic-looking results [46]. The main drawback to L-system lies in the difficulty of devising the system of replacement rules, and the connection between the rules and resulting shapes is not straightforward.

3. $Image-based\ Method$. Image-based methods use input images of the object to create models. Because the input images are photographs taken in the real world, the resulting model looks very natural and realistic. The drawback is that users need to prepare the input images and the modeling process always needs much human intervention.

According to the above analysis, the problem is to have an algorithm to model dendritic shapes in a simple, fast, automatic and straightforward way. Next, we will briefly introduce such an algorithm as our solution.

## 1.2 Our Solution

To satisfy the above requirements for modeling dendritic shapes, we present a procedural algorithm using path planning. Path planning is the problem of finding the least-cost path between two nodes in a weighted graph. The basic idea is that to model a dendritic object we first need to simulate the dendritic shape. Using path planning method we can find the least-cost paths from multiple endpoints to a common source point (generator), and the collection of the paths forms the dendrite. Since we use a single generator for all paths and the least-cost path from the generator to any node in the graph is unique, the least-cost paths from the generator to the endpoints will never intersect.

Our algorithm includes three steps described as follows.

In the first step, we create a weighted graph and the edges of the lattice are given weights from some distribution. Then we choose a node in the graph as the generator, which is the base node (we call it generator) with the path cost zero and path costs of all the other nodes in the graph will be evaluated by their path costs to the generator.

In the second step we apply Dijkstra's algorithm to the above graph. All the nodes in the graph will be updated with their least path costs to the generator.

Finally, we choose some endpoints in the graph. According to their path costs, we can trace their least-cost paths back from the endpoints to the generator. Because paths from multiple endpoints to the same generator may share some common routes when they approach toward the generator, the paths will appear

to emerge from the generator and then branch when overlapping paths deviate. The collection of the paths forms the dendrite shape (see Figure 1.2).

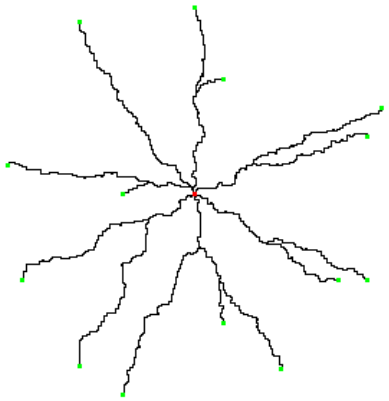


**Figure 1.2:** The dendritic shape formed by a collection of least-cost paths from multiple endpoints to a common source point.

For creating dendritic objects in different shapes, we provide control handles including endpoint placements, iteration count of fractal, distribution of edge weights and path width to obtain dendrites with special features. With these control handles we can easily control the global and local shapes of the dendrite and imitate important features of some natural dendritic objects. We describe our control mechanisms as follows.

*Endpoint Placement* : Endpoint placement is the way of choosing which node in the graph to be the endpoint. When the generators position is fixed, the positions of the endpoints decide the direction where we want the paths to go so control the global shape of the resulting dendrite.

*Iteration Count* : Fractal is an important feature of many dendritic shapes. To imitate the fractal shape we grow the dendrite with iteratively added smaller scaled paths attaching to the former structure to achieve the self-similar property. By setting the former structure with less cost, the later added paths will prefer to take the edges in the former structure and appear to attach to the former structure. The process can be applied to the current obtained dendrite on and on. We can control the iteration count of the above process to control the complexity of the fractal shape.

*Path Width* : To imitate the dendritic shapes with paths in different widths, we use the path cost of each node after treating the whole dendrite as a new set of generators. Each path in the dendrite will be assigned a width factor. The path costs and the width factor will be converted to intensity value of each node. The bigger sized aggregation of nodes with high intensity forms the wide path and smaller size of that forms the slimmer path.

*Distribution of Edge Weights* : Since edge weights decide the detailed shape of the paths, we use different distributions of edge weights to control the path shape. If we set the edge weights according to spatial locations, the paths will tend to take edges in cheaper areas and avoid edges in expensive areas. The resulting dendrite will demonstrate spatial bias which is similar to some features of natural dendritic objects.

Since the dendrite obtained by the above process is limited with the resolution of a fixed lattice, in order

to get high-resolution dendrite with more details we propose a refinement algorithm. By applying iterations of the refinement algorithm we can improve the resolution of the dendrite step by step.

The final stage of the work is to convert the dendrite to geometric models. Different converting methods will be applied according to specific requirements. Since the process to obtain the dendrite is the core of the whole work we will focus on describing this part.

The main contribution of our work is to provide a simple, fast way for generating dendritic forms. Because path planning has been well studied in computer science, many standard algorithms exist and should be familiar to computer graphics practitioners; in consequence, our algorithm is easy to implement. As stated previously, compared with many existing methods for generating dendrites our algorithm is very fast and most dendrites shown in the thesis can be achieved within seconds. Furthermore, an intuitive control mechanism is another advantage of our algorithm. By using different control handles we can produce different shapes of dendrites which are qualitatively similar to different natural objects possessing the complex features.

This thesis is organized as follows:

In chapter 2 we will discuss some previous work in the related areas of path planning, and some methods for generating dendrites including physically based processes especially DLA, L-systems and image based tree modeling. We will also introduce the background knowledge for path planning. Based on these introductions we can understand the advantages of our algorithm which will be described in chapter 3 and chapter 4.

In chapter 3, we first introduce our basic algorithm of creating a simple dendrite by finding the least-cost paths between multiple endpoints and a common generator. We then generate more complicated and realistic dendrites by endpoint placement and simulation of fractal shape. We place the endpoints manually or randomly according to some criteria to control the global and local shape of the dendrite. To imitate the fractal shape, we reduce the edge weights of the obtained dendrite and add new small-scale paths. Preferring to take cheaper edges, the newly added paths will attach to the former structure. Applying the process iteratively we can obtain the fractal shape.

In order to imitate more complicated dendritic objects, we need more improvements and controls to achieve dendrites with special features. In chapter 4 we will introduce our method of creating complicated dendrites by the control of edge weights and path width. We set edge weights according to different distributions to control the regularity of the path. We set the edge weights according to spatial locations to obtain dendrites that have spatial biases. We control the path width to imitate the dendritic shapes with different thickness of branches. After that, we will introduce how to break the limitation of fixed resolution to obtain highly detailed dendrite by our path refinement method. In the end, the process to convert these dendrites to specific geometric models will be described.

Chapter 5 will discuss our algorithm and give some results. For demonstrating the versatility of our algorithm, we give the result of different shaped elm tree models created within the same framework. We

also imitate the behavior of space competition of trees and lichens which was once implemented by open L-systems [35] and DLA method [13]. We give the esthetic dendritic letters similar to the lichen-writing described by Desbenoit et al [13]. We also show our model of lightning compared with the model created by Kim and Lin's physically-based method [24]. After that we will make the comparison between our method and other related methods including DLA, L-systems and image-based algorithms by pointing out the advantages and drawbacks.

In the last chapter we will conclude our algorithm and restate the contributions of our work. Finally, we will discuss possible future work.

# CHAPTER 2

# PREVIOUS WORK

Generating realistic images of geometric models is a major goal in computer graphics [28]. A common class of complex objects arises from dendritic shapes, such as trees, river system, lichen and coral. Before we start introducing our method of modeling dendritic shapes in chapter 3, we review some related work in this chapter. We begin from the survey of important properties of dendrites such as fractal. After that, we will introduce some related work for modeling dendritic shapes. Finally, we will give the background knowledge of our solution.

## 2.1 Fractal

As Mandelbrot stated in "The Fractal Geometry of Nature" [33]: "Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line." Fractal is an important property existing in many objects. A fractal is "a rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole" [33]. The concept of fractal is closely linked to properties of self-similarity and scale invariance [18].

Self-similarity is the property that an object is formed by parts that are similar to the whole. Many natural objects such as coastlines demonstrate self-similarity property, since the parts of them show similar statistical features at many scales [32]. Scale invariance refers to the invariance of property under the change of scales, that is, the smaller parts of the object are always similar to the whole at any scales.

Many dendritic objects have fractal shape. A simple example is a snowflake which shows self-similarity and scale invariance at the same time ( see Figure 2.1).

The complex and widespread features of fractals attract the interest of many researchers. Fractals have often been used for simulating some complex structures, such as soil structures [44], mountains [47] and river network [55].

Among the various approaches for modeling fractal dendritic shapes, we are more interested in two algorithms: L-systems and Diffusion-limited aggregation (DLA), not only because both of these two algorithms can be applied to describe a rich set of phenomena but also because the resulting shape is similar to what we expect to create, that is the dendritic fractal structures with the erratic winding travels of individual branches.

**Figure 2.1:** A snowflake with fractal property

## 2.2 L-systems

Lindenmayer systems or L-systems for short are a theoretical framework widely used for studying the growth of branching processes. They were introduced for modeling the development of simple multicellular organisms [29, 30]. Most recent applications of L-systems are focused on modeling growth of plants. Beginning from an initial string, L-systems repeatedly use a replacement grammar to create strings which can be interpreted as a variety of botanical forms, particularly branching structures.

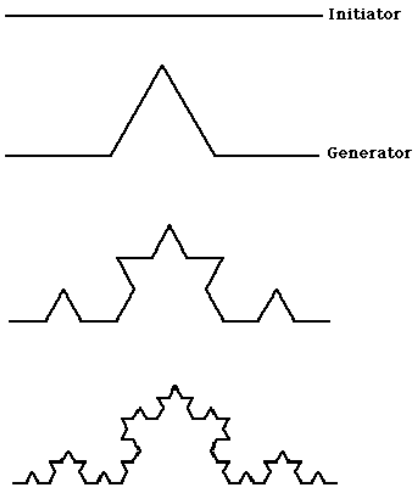



**Figure 2.2:** Construction of the Koch curve

The central idea of L-systems is that of rewriting. 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 [50]. A simple fractal example of a graphic object is the Koch curve [66], which is constructed by using an iterative process consisting of an initiator(initial state) and a generator (iterative operation) as shown in Figure 2.2.

Borrowing from the word 'generator' here for its function of generating successive structures, we call the nodes in our algorithm whose path costs are zero and can act as base nodes for generating newly added paths 'generators'.

The simplest class of L-systems is D0L-systems(Deterministic 0-context L-system). We cite the example in 'The Geometric Beauty of plants' [50] as follows.

> 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 $abaab$ which in turn yields $abaababa$, then $abaababaabaab$, and so on (Figure 2.3).

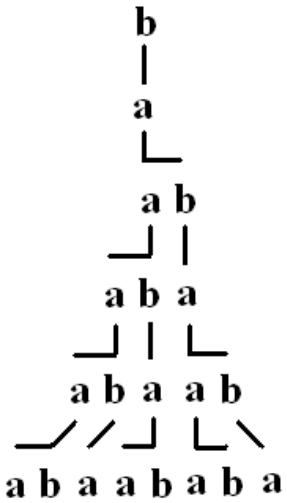



**Figure 2.3:** Example of a derivation in a DOL-system

Strings in the D0L-system can be interpreted in the following way [46, 45].

| | |
|---|---|
| $F$ | Move forward a step of length $d$ |
| $+$ | Turn left by angle $\delta$ |
| $-$ | Turn right by angle $\delta$ |
| [ | Push the current state onto a pushdown stack |
| ] | Pop the current state onto a pushdown stack |

By this means, L-systems can create the branching structure shown in Figure 2.4 by applying the rewriting process for $n$ iterations.

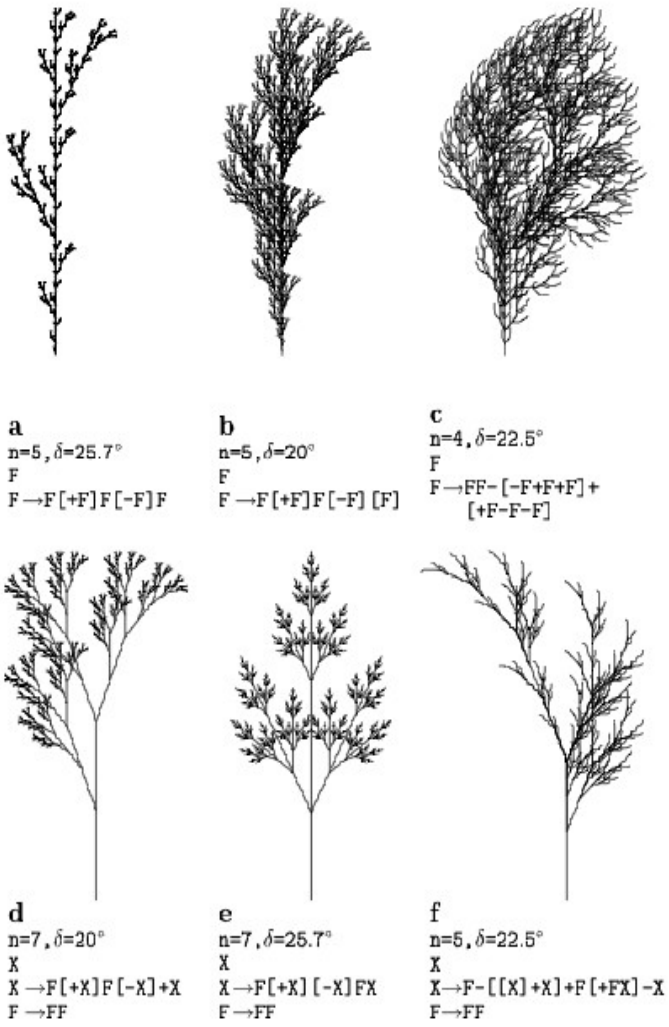



**Figure 2.4:** Plant-like structures generated by L-systems. The figure is taken from 'The Geometric Beauty of Plants' [50].

Because the basic L-systems are deterministic and context-free, the structures generated by the same framework are identical and the growth process is not affected by surroundings. As improvements, the stochastic L-systems [14, 48] and context-sensitive L-systems [27, 19] are introduced. The stochastic L-systems brought into the idea of the production probabilities. A same string can be replaced using different productions according to the production probabilities. This results in variations of structures. Context-sensitive L-systems allow the rewriting to select the production depending on the context of the interested string, so the information about the environment can be taken into account.

As an extension of early context-sensitive L-systems [29], environmentally-sensitive L-systems [49] introduced query symbols returning current position or orientation of the turtle in the underlying coordination system. These parameters are used to influence development at the queried location.

Later, open L-systems [35] was devised to implement bilateral interaction. They imported a communication symbol in the grammar. Parameters associated with an occurrence of the communication symbol can be set by the environment and transferred to the plant model, or set by the plant model and transferred to the environment.

Although the later extended L-systems brought in some elements to affect the rewriting rules, the rewriting rules are only control handles to decide the resulting structure. For all the L-systems introduced above, the connection between the rules and resulting structure is not straightforward so it is difficult to devise and control the system.

## 2.3 DLA

Diffusion-limited aggregation (DLA) is one of the most popular algorithms for generating dendrites. It was proposed in 1981 by Witten and Sander to explain the aggregation of smoke particles [63] and since then the model has been studied a lot by many researchers [34, 65, 2].

The basic DLA algorithm works as follows. Given a discrete 2D lattice, a seed particle in the center of the lattice is set to be the ”origin”. A second particle situated in an arbitrary direction far away from the origin is released. This particle undergoes Brownian motion until it visits a site adjacent to the seed particle. Then it stops walking and sticks to the seed particle. Another particle is released now and repeats the random walk until it joins the cluster (the former aggregation of particles), and so forth. The above process is repeated until the particle cluster has reached a desired number of particles. Figure 2.5 shows the process and resulting dendrite.

When the first released particle stuck to the seed, it blocked certain paths that later released particles can approach the seed. On the other hand, it enhances the possibility of sticking at either end of the aggregation and reduces the possibility of sticking along the edge of the aggregation. During the aggregation process more particles join to the cluster and more tips appear. The probability of sticking to the tips is bigger than that of sticking to the edges. Meanwhile, it is hard for a particle to avoid sticking to any part of the cluster

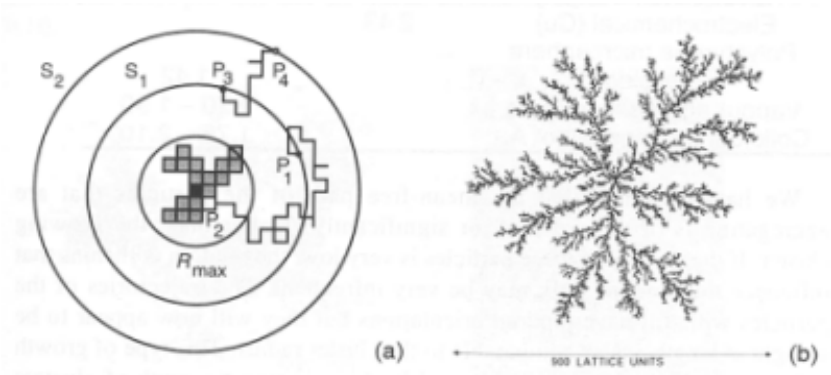



**Figure 2.5:** Left: Illustration of the growth of a diffusion-limited aggregation happening in a square lattice. Trajectories are shown for a particle wandering outside of S1, and a particle that took random steps and finally stuck to the cluster. Right: The result dendrite of such a growth for 50000 particles on a hexagonal lattice. The figure is taken from "Fractals in Chemistry" [21]

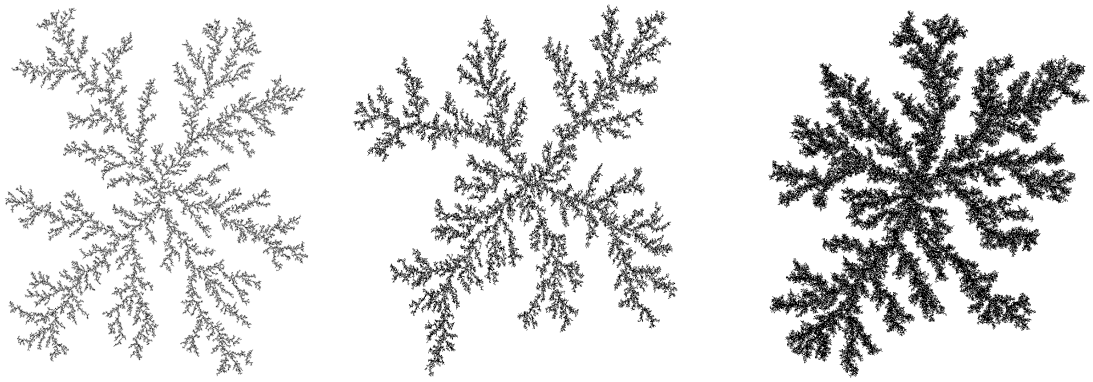


**Figure 2.6:** From left to right: The dendrites obtained with sticking probabilities of 0.2, 0.05, 0.01. The figure is taken from "DLA - Diffusion Limited Aggregation" [4]

along its route of diffusing into the center of the cluster. This results in the dendritic shape of the cluster.

As extensions of the initial DLA algorithm, some modified models were devised. One type of extension is the introduction of "sticking probability". That means when the particle reaches the cluster it is not always stuck but keeps walking until it is finally stuck somewhere. A low sticking probability gives the particle longer random walking process and it has more possibility to stick to the part that is hard to reach before. So the dendrite will look thicker. The dendrites obtained with different sticking probabilities are shown in Figure 2.6.

To take advantage of parallel computational resources, parallel models of DLA [67, 37] release multiple particles instead of the single one in the original DLA model. Another modification direction is to change the shape of seeds. Using a line instead of a single particle as the origin seed can create forest-like clusters which is known as diffusion-limited deposition (DLD) [62].

The dendrites generated by DLA can be observed in many systems such as electrodeposition [69, 36, 68], mineral deposits [64], dielectric breakdown [3] and neurite outgrowth [16]. DLA has also been used in computer graphics to model some common natural dendritic shapes, such as lichens [13] and ice crystals [25, 23]. The results are of high visual quality but the modeling process is very time-consuming. The random walk process of the particles since they are released until they are stuck to the cluster is very slow. We can decide the shape of the seeds and the direction of releasing the particles to control the general shape of the resulting dendrite. But we lack of efficient way to control the local details. Although the use of "sticking probability" can roughly adjust the local thickness of the dendrite we still cannot decide the specific details.

## 2.4 Some Related Algorithms

Besides L-systems and DLA, there are some other algorithms for modeling dendritic shapes.

Trees are a classic category of dendritic objects. Modeling complex trees has been studied by computer graphic practitioners for decades. Besides the popular L-system models image-based methods are used recently to create models for trees. Image-based modeling refers to the use of images to drive the reconstruction of 3D models [42]. Since many complex shaped objects such as trees or people cannot easily be described by the polygonal representations which are commonly used in computer graphics [40, 10, 72], image-based method can make use of photographs taken from the real world to build 3D models. The approaches for modeling trees range from using a single image [20, 41] to multiple images [54, 59, 52, 51, 61, 38]. Methods of using a single image always need good priors. For example, Han et al. [20] use a Bayesian approach to model 3D shapes and scenes from a single image with the assumption of surface and boundary smoothness, 3D angle symmetry etc. Methods of using multiple images always use a sequence of images taken from well-controlled camera positions to find point constraints for 3D reconstruction. Among these methods we are more interested in the methods of Reche-Martinez et al [52], Tan et al [61] and Neubert et al [38] because of their great tree models and ways of dealing with branches. We will discuss these three

methods in detail in chapter 5.

Another dendritic shape which has been paid much attention is lightning. Because the special physical characteristics of lightning, some researchers took the physically-based method to create models [17, 24]. In Glassner's method [17], the lightning geometry is generated by using statistics gathered from real lightning. In Kim and Lin's algorithm [24], they simulate lightning based on the dielectric breakdown model for electrical discharge. Rather than providing the generating mechanism of lightning, some modeling works focus on creating the visual simulation of lightning. One example is Reed and Wyvill's algorithm [53]. They build the lightning model based on the observation that most lightning branches deviate from parent branches by 16 degrees on average. The lightning model are generated by rotating line segments with angles normally distributed around 16 degrees and branching is controlled by a probability function.

## 2.5 Path Planning

To model the dendritic shapes our solution is using path planning algorithm to obtain the dendrite and then convert the dendrite to the specific geometric model. The key part is the path planning algorithm.

Path planning problems arise in many different applications and written in many publications especially on artificial intelligence and computational geometry [70, 43]. Path planning is in general a problem concerned with finding paths connecting different locations in an environment, such as a network, a graph, or a geometric space [7]. The desired paths often need to meet specific criteria such as various distance metrics [26, 1], cost functions [7], obstacles [22, 39] or coverage of an area [9, 12]. A well-known problem of path planning is computing shortest paths in graphs (i.e., finding a path in a graph connecting two vertices with the minimum total length) [15, 7]. In our algorithm we use the cumulative distance metric in a graph as the criterion. That is, we are given a weighted graph $G =( N, E )$ where $N$ is the group of nodes and $E$ is the group of edges. The cost of path $p = <n_0, n_1, ..., n_k>$ is the sum of the weights of its constituent edges. Our goal is to find the path with the least cost.

Breadth-first algorithm is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms [11]. The input of the algorithm consists of a weighted graph $G$ and a source node $s$. Originated from $s$ it explores all the neighboring nodes of $s$. Then for each of the neighboring nodes, it keeps on exploring their unexplored neighboring nodes. Repeat the process until it finds the goal node. Breadth-first algorithm is so named because it expands the frontier of unexplored nodes uniformly across the breadth of the frontier[11].

Using similar idea to breadth-first algorithm, Dijkstra's algorithm maintains a tree $T$ rooted at the source node $s$ that spans all nodes reachable from $s$. At each step, the edges incident to every node of $T$ are examined, and one edge is added to $T$ that (a) reaches a node $x$ outside of $T$: $x \in G$, and (b) such that the path cost to $x$ from $s$ is the shortest among all nodes satisfying (a). The point of (b) is to ensure that $x$ is the next node to be reached. By storing information at each node indicating from which direction first to be

reached, it is possible to trace backwards and find the complete shortest path to any node [43]. Dijkstra's algorithm is summarized and shown in Figure 2.7.

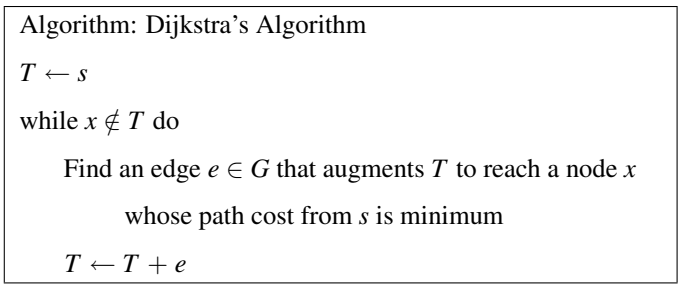



**Figure 2.7:** Dijkstra's shortest path algorithm. The figure is taken from "Computational Geometry in C" [43].

In our modeling work, we take the method described by Xu and Mould [71]. Firstly we choose a source node. By applying Dijkstra's algorithm, for each node we store the path cost value to the source node (we call it 'generator'). According to these path cost values, we can trace backwards from the endpoints to the generator to find the shortest paths. The collection of the shortest paths forms the dendrite. To create the fractal dendrite, we treat all the nodes in the dendrite as generators. With Dijkstra's algorithm we can find the paths between new endpoints and the generators. The paths attaching to the former structure form the dendrite. The basic algorithm will be described in the next chapter.

Based on the similar idea, Long [31] uses a non-scalar distance metric instead of the traditional cumulative distance metric in the path planning to model dendritic structures for artistic effects. The non-scalar distance metric intends to minimize the maximum edge costs in the path. Because the paths generated using cumulative distance metric tends to take short cuts over high cost areas in order to avoid long, winding routes and the short cuts can ruin the fine details and stylized winding structures, using the non-scalar distance metric is suitable for this application.

In our modeling work we use the traditional cumulative distance metric which defines the cost of a path equal to the sum of edge weights along the path. Because our modeling objects are natural dendritic objects or phenomena such as coral, trees and lightning, they do not have dramatic long and winding structures as in the stylized art patterns. We can simulate these dendritic structures using the cumulative distance metric very well.

# CHAPTER 4

# DENDRITIC MODELING

In chapter 3, we described how to obtain a simple dendrite using a path planning algorithm and how to generate a fractal dendrite with simple control handles of endpoint placement and fractal shape. Based on the basic fractal dendrite, now we start to create specific dendritic models. We choose three common dendritic shapes in nature as our modeling objects: staghorn coral, lightning, and elm tree. We choose these three shapes because all of them are commonly seen and familiar to readers; each of them shows the common dendritic structure with respective different features. Although the details of individual objects may be different, the modeling mechanism is similar and users only need to adjust some parameters to meet different requirements. In this chapter, firstly we will give our methods for building geometric models. Next we will describe the detailed modeling techniques. For the problem of the resolution limitation that appeared in practice, we will give the solution of path refinement at the end.

Before creating specific dendritic models, we must know some important features of our modeling objects. In most occasions, our modeling objects are always natural dendritic shapes possessing following features.

1. Many featured dendritic objects have specific characteristics, such as erratic branch shape and growth tropisms.

   As introduced in chapter 3, the position of generator decides where the root of the dendrite is; the positions of endpoints decide the relative directions and distances to the generator. So positions of endpoints and generator can decide the directions and rough lengths of the paths. But for the detailed path shape, we need more methods to control the path shape according to our requirement. In section 4.2, we explore the control of edge weights, and combining the former controls we can obtain different featured dendrites with erratic branch shape and growth tropisms.

2. Most dendritic shapes have different widths of branches.

   There are many natural objects demonstrating different widths of branches in different parts within the overall dendritic shapes, such as lightning, trees and river systems. Simulating the different widths of branches is an important part of our modeling work. Section 4.3 explains how we control path width for 2D and 3D dendritic models. We give the models of lightning and elm tree as the examples of two different methods in 2D and 3D cases respectively.

3. Different dendritic objects have different branching details.

   Most natural dendritic objects possess different branching details. For example, elm trees usually have more complicated details such as small twigs and winding stems than a staghorn coral, whose structure is more regular and less variable. We should be able to generate dendrites with different details.

   The straightforward method to control the dendrite detail is to change the resolution. We introduce how to obtain a higher resolution dendrite by the path refinement algorithm in section 4.4. We create an initial coarse dendrite first and build a hooked new sublattice around the dendrite, then apply the path planning algorithm within this new refined graph. The new dendrite obtained in the refined graph has the higher resolution and the refinement can be applied for times according to users desire.

## 4.1 Modeling Methods

In order to use the dendrites to create models for specific dendritic objects, we apply the following two methods for 2D and 3D cases respectively.

1. For 3D models, we build the model by composing geometric primitives.

   Because the dendrite we obtained are composed of nodes and edges, if we convert the nodes and edges to the specific geometric primitives such as circles and lines in 2D or spheres and cylinders in 3D, by assembling these geometric primitives we can create the model. The advantage to do so is that the process is simple and we can easily control the size of these geometric primitives.

   Figure 4.1 shows our staghorn coral model. The model was created by manually placing endpoints in a 3D graph; the points were not chosen to exactly duplicate the input model, but to give a visually similar appearance, i.e., the synthetic coral could plausibly have come from the same underlying growth process. The dendrite is converted to the geometric model by building a sphere for each node. Despite the small amount of information provided by the modeler (only the endpoints of the branches were specified), the synthetic coral model resembles the real coral quite well. The synthetic image was rendered using Pixie (pixie.sourceforge.net), with the high-frequency structure (thorns) on the surface of the branches obtained from a Renderman displacement shader.

2. For 2D models, we make use of path costs of the nodes in the graph to obtain intensity values.

   As described in the basic algorithm (see chapter 3), when we treat the existing dendrite as a new set of generators, after applying Dijkstra's algorithm each node in the graph will have an updated path cost $d$ towards the nearest generator. Here we take scalar value $V = \exp(-\alpha \times d^2)$ for every node in the graph where $\alpha$ is a constant value around 1/10000. For each node (pixel), the value of $V$ can be converted to the brightness.

   We can also use the method of composing geometric primitives to create the 2D model. Figure 4.2

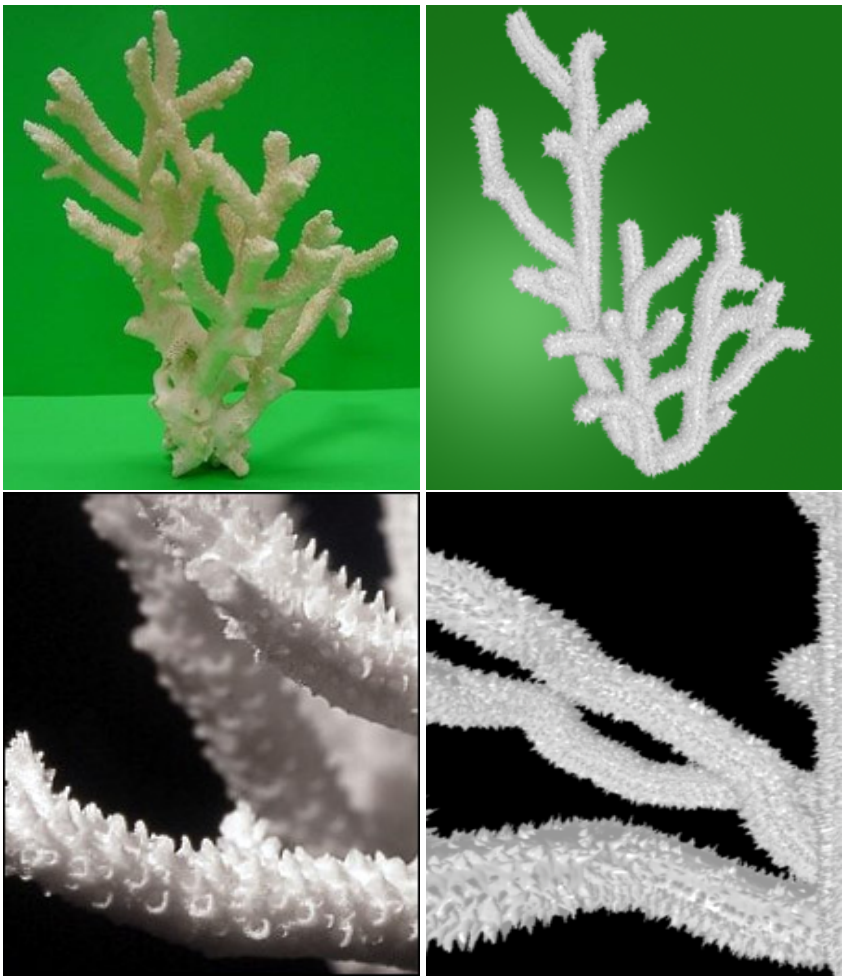


**Figure 4.1:** Left: real coral. Right: coral generated using path planing

shows the lightning models created by the above two methods. For the left lightning model, we choose a single node as the generator at the top of the image and placed a small number of endpoints at the bottom of the image to obtain the dendrite by path planning. We build a circle for each node in the dendrite and side branches of the lightning are obtained by circles with smaller radius. The right lightning model is created by using path costs of the nodes in the graph. Compared with the left model, the right model looks more realistic and natural, but the process is more complicated.

There are also other possible methods for converting dendrite to geometry. One way to convert a 3D dendrite is to use path costs to the dendrite as a scalar field. And the field can be converted to geometry using an existing isosurface extraction algorithm such as marching cubes. But the surface obtained can be no better than the resolution of the lattice and further mesh smooth steps are needed. A side-by-side comparison between the raw model and smoothed model is shown in Figure 4.3. Another possible method is to build a circle for each node and choose some points on the circle. For each adjacent pair of points, we connect them together to build meshes. The advantage is that we can control the thickness of the branch by setting circles with different radius values. The drawback is that the mesh building process is not easy. Finding the corresponding pair of points in the adjacent circles may be difficult.

**Figure 4.2:** Left:lightning model created using geometric primitives; right:lightning model created using path costs in the graph.

In this thesis, we build geometric models for three kinds of dendritic objects: staghorn coral, elm tree and lightning. As stated at the beginning, since the basic fractal dendrite cannot meet the requirements of simulating complicated structures of these natural objects, as the further improvement we apply two techniques: edge weight control and path width control. With different edge weight distributions we can produce dendrites in featured spatial tropism. By controlling path widths, we can simulate many natural objects with different widths of branches. Using two modeling methods introduced in the early part of this section, we build our models to illustrate these two techniques in section 4.2 and 4.3.

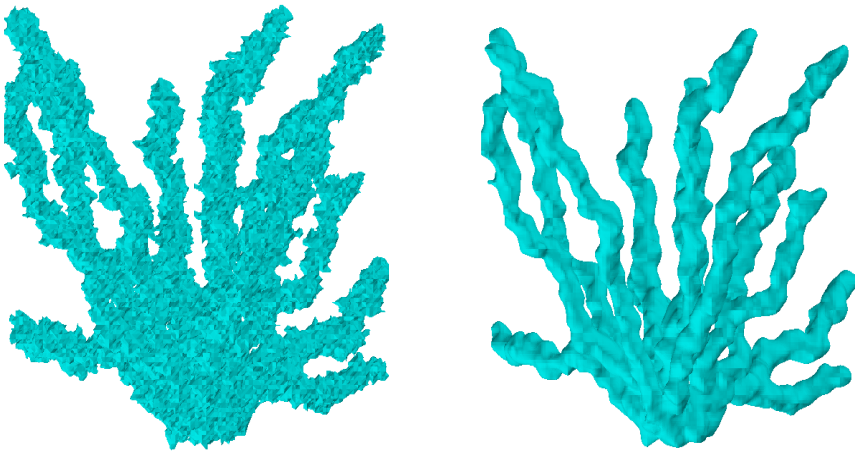


**Figure 4.3:** Raw model(left) and smoothed model(right).

## 4.2 Edge Weight Control

In our real lives we often observe some dendritic objects illustrating different styles. For example, some branches of lightning in the sky appear more winding and erratic while some branches appear straighter and less variable. Another example is that an elm tree shows a different posture in the wind than in the static status though in both situations it possesses the same dendritic branches. To imitate these specific features of the dendrites, we use the control of edge weight.

The simple dendrite we obtained initially exists in the graph with edge weights $W = 1 + \alpha R$. $R$ is a random value between $(0,1)$, $\alpha = 100$, as introduced in Section 3.1. We are already able to use endpoint placement and generator shape to control the global shape of the whole dendrite. When locations of generator (starting point) and endpoints are decided, edge weights of a graph are crucial for the shape of every individual path. By controlling the edge weights, we can control local shapes as well to generate different shapes of dendrites. In some other applications, the edge weights can take other local information into account, such as gradient and intensity values [31].

Figure 4.4 shows the dendrites obtained by setting edge weight $W = 1 + v^n$, where $v$ is a random value between $(0,100)$. While value of $n$ is increasing, the disparity between the cheapest and most expensive edges becomes greater, and therefore, the path planner has more incentive to seek cheap edges. For example, for two paths between the same endpoint and starting point, say $p1$ consisting of one edge with cost $e$ and $p2$ consisting of four edges with cost $a$, $b$, $c$ and $d$ respectively. For $n > 1$, when $a+b+c+d = e$, $a^n + b^n + c^n + d^n < e^n$, that is, the path consisting of more and individually cheap edges is cheaper than the path consisting of fewer but expensive edges, though the sum of edge weights for these two paths are equal. And greater value of $n$ will result in longer and more erratic paths than lower $n$.

We can see the difference from Figure 4.4: since the edge weights are constant values when $n = 0$,

the obtained paths are Manhattan paths; when $n = 1$, the edge weights follow the uniform distribution, the obtained dendrite is the simple dendrite discussed in chapter 3 and the paths are no longer straight; for the dendrite with $n = 3$ and $n = 4$, the paths become less direct and more tortuous with greater value of $n$. The value of $n$ can be chosen according to specific modeling desire. For modeling some objects with straight branches such as staghorn coral as shown in Figure 4.1, we choose $n = 1$ to generate a more regular dendrite; for modeling more irregular dendritic objects such as lightning, we choose greater value of $n$, for example, 3 or 4 is enough.

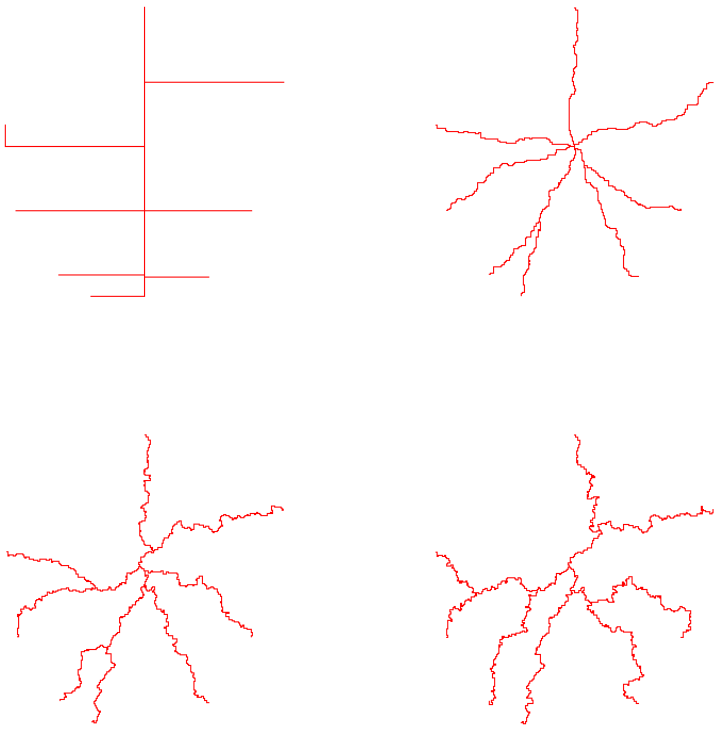


**Figure 4.4:** From top to bottom, left to right: dendrites with edge weight $W = 1 + v^n$, $n$ = 0, 1, 4, 8.

Another way to control the edge weights is biasing edge weights according to spatial locations. That is, the weight of an edge is decided by spatial positions of its two endpoints. So some areas of the graph become cheaper and some areas become more expensive for path planning. Because paths prefer to take the cheaper edges, paths from fixed endpoints to the generator will have a tropism towards the cheap regions and departing from the expensive regions.

We illustrate dendrites with different tropisms in the 2D graph in Figure 4.5. All the dendrites are obtained by placing the root at the bottom of the graph, choosing random endpoints and path planning. The differences come from different distributions of edge weights. For the dendrites $a$ and $b$, we set edges

cheaper when they are closer to the left boundary of the graph. Consequently, the paths prefer to take the cheap edges and demonstrate the tropism towards the left. Although the generating process for $a$ and $b$ is same, we place the root at the bottom left for dendrite $a$, the resulting structure looks similar to the posture of trees blown by wind from left; we place the root at the bottom right for $b$ and create a dendrite similar to some kind of bush growing up from the right corner. In a contrast, for the dendrite $c$ and $d$, since the edges become cheaper when closer to the right boundary of the graph, the paths demonstrate the tropism towards the right. The difference between $c$ and $d$ is also the position of root. For the dendrite $e$, we set the edge weights cheaper when closer to the center in horizontal direction. Conversely, we set the edge weights in $f$ more expensive when closer to the center in the horizontal direction. Compared with $e$, dendrite $f$ shows a quite contrary tropism of bending outward to avoid the expensive area. Another contrast pair is $g$ and $h$. For the dendrite $g$, the edges are cheaper the closer to the top they are. For the dendrite $h$, the edges are cheaper when closer to the bottom. As the result, the former dendrite looks like a tree and the latter looks more like a shrub.

Based on the similar idea, we created different 3D models by varying the edge weights. The difference is, we calculate the edge weights in a 3D space where the horizontal radius is used instead of horizontal distance used in 2D. Figure 4.6 shows those 3D dendritic models which look similar to some natural dendritic objects with erratic branches and growth tropisms. Figure 4.7 shows our 3D models simulating trees in the wind. In our model, the branches bending towards certain directions are similar to the state that trees are blown by wind. We generate the 3D dendrite by placing the endpoints at one side of the root and setting edge weights decreasing towards the root side.

## 4.3 Path Width

Most natural dendritic objects such as lightning, trees and river system have different widths of their branches. For example, the river system often has wider main flows and slimmer side flows. For the lightning and trees, besides the different width between main stem and side branches their branches always have the tapering property. To imitate these features, we need more improvements on the dendrite with an even width obtained so far. We wish to implement different widths for different parts along a single path and different widths for different paths in a dendrite.

For a path in 2D graph, we take the following algorithm to implement the task. For simplicity, we describe the process for a single path. To setting different widths for a dendrite, we can apply the process for each single path independently.

First, we set the endpoint of the intended path with a width factor $T_s$. Then after tracing the path back from the endpoint to the generator, the nodes along the path will have a corresponding width factor $T_s$. If we treat all the nodes in the path as a new set of generators (in the same way described in section 3.4.1), after Dijkstra's algorithm each node $N_i$ in the graph will have an updated path cost $c(N_i)$. $V(N_i) = exp(-$

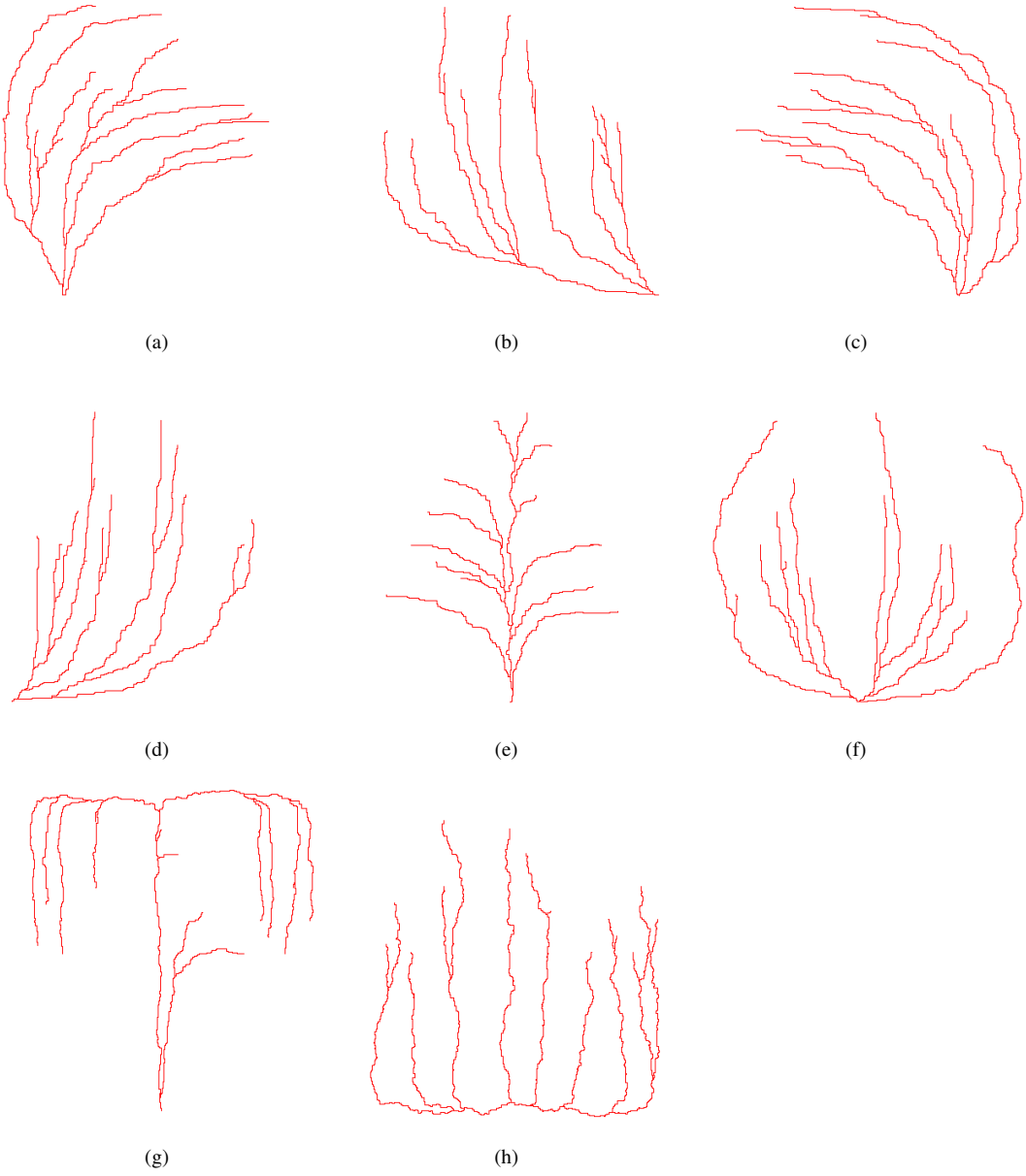
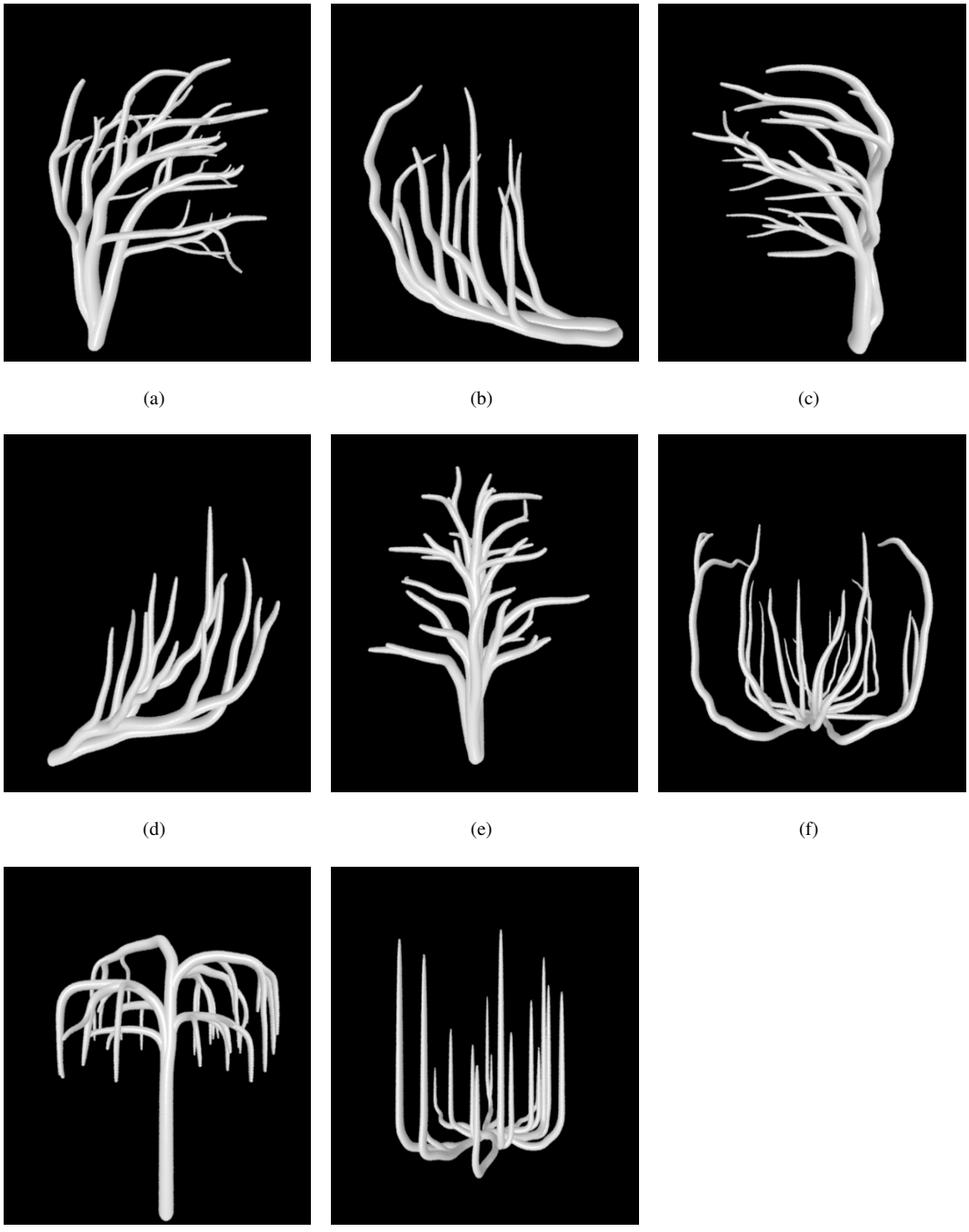



**Figure 4.5:** Different dendrites obtained by varying the edge weights.



(a) (b) (c)

(d) (e) (f)

(g) (h)

**Figure 4.6:** 3D dendrites obtained by varying edge weights.

**Figure 4.7:** Upper: trees bending in the wind; Lower: our tree models

$\alpha \times(\mathrm{c}(\mathrm{N}_i) \times \mathrm{T}_s)^2)$ is a scalar value for each node $N_i$ in the graph. The value of $V(N_i)$ is between 0 and 1. We can use $V(N_i)$ to get color intensity values. To increase the brightness of the path, we treat all the pixel intensities above 0.7 as 1. By adjusting the value of $T_s$, we can adjust the value $V(N_i)$ and finally control the width of individual branches. Here we found the value of $\alpha$ around 1/10000 to work well. The pseudocode is shown in Figure 4.8. Figure 4.9 shows the same path with different widths by setting to different width factor $T_s$ of 0.6 and 0.2 respectively. If we decrease the width factor $T_s$ gradually from the root to the tip of the path, we can obtain a tapering path as shown in Figure 4.10.

Input: weighted graph $G$, consisting of nodes $N$ and edges $E$; $N_s$, one endpoint of the path; the generator $Z$; and $T_s$, width factor for the path.
Output: intensity value $I(N_i)$ of each node $N_i$.
1. Apply Dijkstra's algorithm.
2. Find the least-cost path $P$ from $N_s$ to $Z$ by greedy hill climbing.
3. For each node $N_i \in N$:
if $N_i \in P$, append $N_i$ to $Z$ and set $c(N_i)$ to 0;
otherwise, set $c(N_i)$ to Max.
4. Apply Dijkstra's algorithm. Each node $N_i$ has an updated cost $c(N_i)$ and a width factor $T_s$, then calculate $V(N_i) = exp(-\alpha \times(\mathrm{c}(\mathrm{N}_i) \times \mathrm{T}_s)^2)$ .
5. For each node $N_i \in N$:
if $V(N_i) > 0.7$, intensity value $I(N_i) = 1$;
otherwise $I(N_i) = V(N_i)$.

**Figure 4.8:** Pseudocode for getting a path with a certain width.

Different widths and tapering of the path are key components for modeling lightning branches. Using the method stated above we can create models for lightning. Figure 4.11 shows the process to create a complicated twisted structure of lightning. We use two separate 2D graphs with different edge weights to do the path planning. The dendrites in both graphs are generated by placing a same generator at the top part of the graph and some random endpoints below the generator. The purpose to do so is to get a dendrite with branches scattering below the generator, which simulates the real lightning branches spreading downwards from the source. The dendrite in the first graph contains the main branch and some additional side branches; the dendrite in the second graph contains only side branches. By adding the two dendrites together we can get a compound structure with crossing twisted branches, which possess more details and difficult to obtain in a single graph. We put the lightning structure onto a background image of stormy sky to create an image as shown in Figure 4.12. The synthetic lightning image is convincing and difficult to be identified along

with the photographs of real lightning.

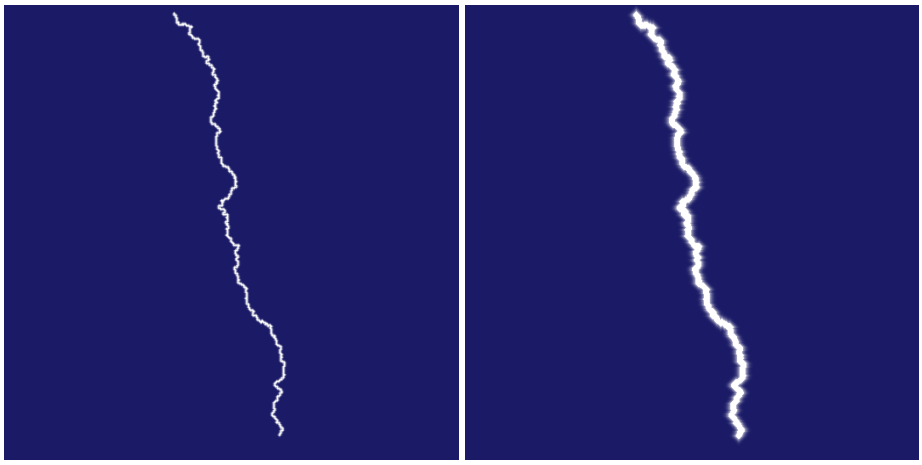


**Figure 4.9:** Paths with different widths.

**Figure 4.10:** Width tapering of a path.

As stated before, in 3D cases, we build the model by composing geometric primitives. The path in 3D space can be converted to a geometric model by building a sphere for each node. We can control the radius of the spheres to create different widths of branches. Greater radius gives thicker branches and smaller radius gives slimmer branches. Tapering branches can be generated by gradually decreasing the sphere radius along the path from the root to the tip.

Figure 4.13 shows the elm tree model created by this approach. The similarity between our model and the real tree lies in the erratic spreading branches with tapering twigs growing from a common root. The different 3D dendrites shown in Figure 4.6 in the last section are also implemented using this approach.

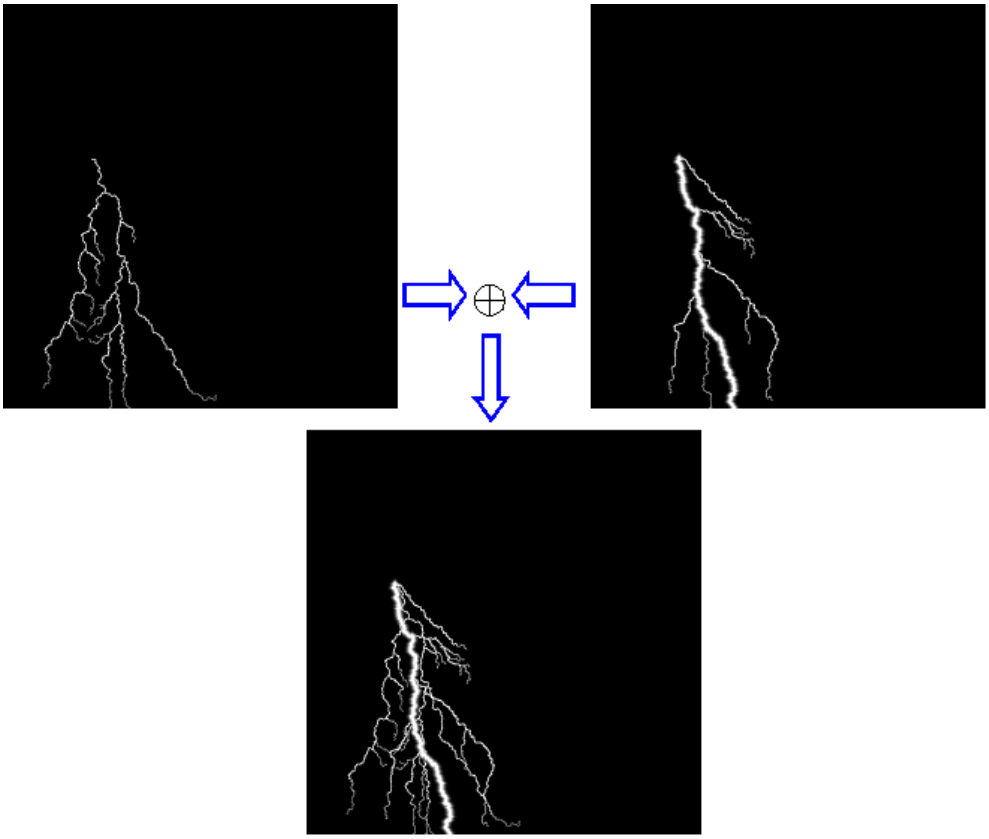


**Figure 4.11:** Process of building a lightning model.


**Figure 4.12:** 1:our lightning; 2 and 3: real lightning

**Figure 4.13:** Left: real elm tree; right: our elm tree model.

## 4.4 Path Refinement

Our algorithm as described so far creates dendrites which are limited by the fixed resolution of the lattice in 2D or 3D where the dendrite exists. We often need to create dendrites with different resolution requirements. For some dendritic objects possessing many details we wish to imitate it with a dendrite in higher resolution. To increase the resolution of the dendrite, one method is to increase the size of lattice. In theory, we can choose different sizes of the lattice to create different detailed dendrites. But in practice the computer memory limitation problem arises along with the size increasing. Here we give our solution of path refinement method. By applying the path refinement algorithm to the current low-resolution dendrite iteratively, we can improve the resolution step by step. The details are as follows.

Firstly, we apply our path planning algorithm to obtain the raw path which has the same resolution as the lattice. For each node in the dendrite, we build a sublattice with resolution $n \times n$ ($n > 1$) and $n$ is an odd number for the purpose to locate the node in the sublattice center.

Secondly, we hook each pair of adjacent sublattices together according to their relative positions. For two adjacent sublattices $L_i$ with center $N_i$ and $L_{i+1}$ with center $N_{i+1}$, we set the edge weights of each sublattice $W_{sub} = eW_{int} + (1-e)W_{ran}$. $W_{int}$ is the value obtained by linear interpolation of the raw edge weight $W$. $W_{ran}$ is a random value between 1 and $W$. To keep the basic shape of the raw path, we choose $0.5 < e < 1$, by which we can avoid the overwhelming random elements in the edge weights. The hooked $L_i$ and $L_{i+1}$ form an $2n \times n$(if $N_{i+1}$ locates on the right or left side of $N_i$) or $n \times (2n)$(if $N_{i+1}$ locates on the upper or lower side of $N_i$) lattice. By this means, we can hook all the sublattices together to form a connected new graph.

In the new connected graph, we take a path planning process to get the new paths. When the size of the old lattice is fixed, the resolution of the new paths is decided by the size of the sublattice. Figure 4.14 illustrates the process. Pseudocode describing the refinement process for a single path is shown in Figure 4.15.

One advantage of the path refinement is that the size of the sublattice is relatively small and they are

temporary, so the burden of memory usage is reduced. Furthermore, this refinement process can be repeated in the region of interest if desired. Users can easily control the resolution of refined version they want to see by setting different sublattice sizes. Figure 4.16 shows the original coarse path and different refined versions obtained by $7\times7$, $9\times9$ and $15\times15$ sublattices. A side-by-side comparison between a coarse dendrite and a refined dendrite is shown in Figure 4.17. The refined dendrite is visually smoother and shows more details.

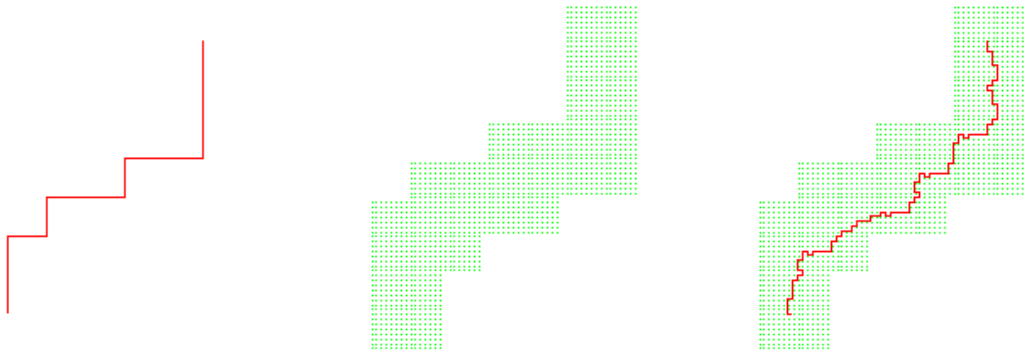


**Figure 4.14:** Left: a coarse path; the refined lattice will be generated around it. Middle: refined lattice. Right: a new path computed inside the refined lattice.

| |
|---|
| Input: a coarse path D consisting of m nodes, and n to decide the size of the sublattice. |
| Output: a refined path D'. |
| 1. For each node in D, say $N_i$, create a regular lattice $L_i$ of size $n \times n$. Assign positions to nodes in $L_i$ relative to $N_i$. |
| 2. For i=0 to m-2, stitch each pair of adjacent lattices together by adding edges between adjacent nodes in $L_i$ and $L_{i+1}$. Call the resulting graph $G_0$. |
| 3. Perform a path planning task within $G_0$ and return the result. |

**Figure 4.15:** Pseudocode for refining a path.

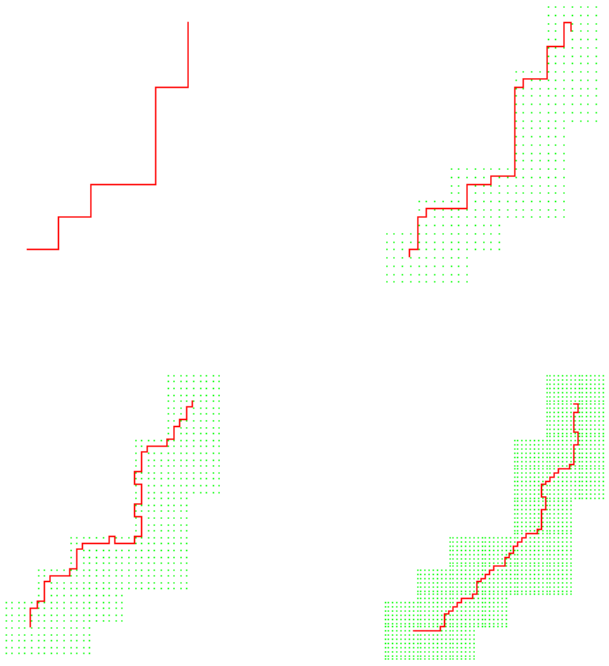


**Figure 4.16:** From left to right: single paths with different resolutions by setting different sublattice sizes of $7\times7$, $9\times9$ and $15\times15$.

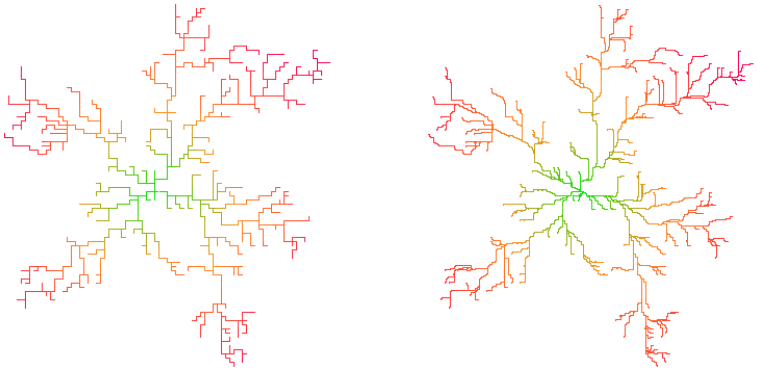


**Figure 4.17:** Left: a dendrite generated on a coarse graph. Right: a refined version of the coarse dendrie.

# CHAPTER 3

# BASIC ALGORITHM

## 3.1 Overview

The goal of our algorithm is to build geometric models for dendritic shapes. The key part is to obtain a dendrite composed of paths used for building models. Since path planning is a well-known problem of dealing with paths, we are inspired to use path planning to find the paths. Based on the observation that key element of our modeling objects is the branching structure with a common root, the dendrite we are pursuing must meet the following three basic criteria: erratic individual branches, a common root of all the branches, and bifurcating twigs from some main branches. In our algorithm, we do path planning to find the least-cost paths between multiple endpoints and a common starting point. The collection of these paths forms the dendrite. We can make the resulting dendrite satisfy the above requirements in the following way. The erratic individual branches can be achieved by setting irregular path costs (edge weights). Therefore the Euclidean distance is no longer the shortest path cost between two points. The least cost path will take a wandering irregular route instead of a straight line. If we choose a single node as the common starting point, all the least cost paths will lead the different endpoints to the same starting point and the paths will share a common root (the starting point). At the same time, paths to nearby endpoints will often share the early portion of their routes, which appear to be a single paths emerging from the source and bifurcating to form twigs.

Based on this general idea, we apply the path planning algorithm including three steps. Firstly, we build a weighted graph which is a regular lattice composed of nodes and edges that connect the nodes. Weights on the edges are chosen randomly. The random edge weights guarantee the irregular path costs and irregular paths later. Users need to choose a node in the graph as the starting point (generator). A single starting point guarantees the common root. Secondly, we apply a Dijkstra's computation of path costs from the generator to all nodes in the lattice. Each node is updated with a path cost value, which is the sum of edge weights along the least cost path connecting the node to the generator. Finally, we select a set of endpoints and trace the paths back from the endpoints to the generator. Proper located endpoints guarantee the bifurcating branches. The paths obtained by above three steps form a simple dendrite and can meet the basic requirements mentioned above (see Figure 3.1). From the figure, we can see the dendrite has branches emerging from a source point and scattering around toward different directions. Each individual branch has

an irregular and tortuous shape. Some branches overlapping in the early portion form the dendritic twigs.

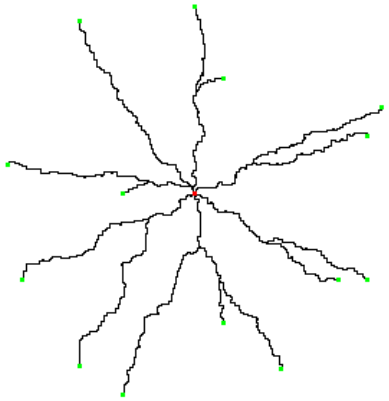


**Figure 3.1:** A simple dendrite obtained by path planning.

For the simple dendrite in Figure 3.1, we choose a node located in the center of the graph (lattice) as the starting point and some random nodes around the generator as the endpoints. Here the word 'simple' means once the endpoints and starting point are chosen, we only need to apply one iteration of path planning. The obtained dendrite does not possess the fractal feature. To create more complicated fractal dendrites, we need to treat different groups of nodes as generators and apply the path planning algorithm for more iterations. But just for the simplicity of the illustration, we only introduce the path planning algorithm for creating a simple dendrite in section 3.2. Since such kind of simple dendrites cannot be used for modeling natural dendritic shapes with more complicated structures, we make further improvements for the algorithm including endpoint placement and fractal shape, which are crucial for creating useful dendrites throughout all the modeling work in this thesis. Section 3.3 will describe this part in detail.

In this chapter, we place more emphasis on the basic algorithm of generating a dendrite rather than creating geometric models for specific sophisticated objects. Based on the introduction of this chapter, in chapter 4 we will discuss how to create models for natural dendritic objects with different features.

## 3.2 Path Planning Algorithm

Path planning is the problem of finding the least cost path between two nodes in a weighted graph. Here we use the algorithm to find the least cost paths between multiple endpoints and a common starting point. The group of the paths forms a dendrite. Now we explain our path planning algorithm for creating a simple dendrite in 2D. In general, the path planning process includes the following three steps.

The first step is to create a weighted graph and choose a node as the generator. The weighted graph where the path planning processes take place is a regular lattice, 4-connected in 2D or 26-connected in 3D space. We set weights of lattice edges $W = 1 + \alpha R$. $R$ is a random value between $(0, 1)$; $\alpha$ is a parameter determining the top amount of fluctuation permitted in the weights. Because small $\alpha$ will result in the nearly

constant edge weights, the resulting paths are close to Manhattan paths. On the other hand, larger $\alpha$ will result in more erratic paths because the path finder tends to find the cheapest path within a wider range of random path costs. In practice, we found the value of $\alpha$ between 10 and 100 work well. Figure 3.3 shows such a weighted graph. Then we choose a node in the graph as the generator. We call it 'generator' because the path cost value of the generator is 0, and path cost values of all the other nodes will be evaluated according to their path costs to the generator. In the future, 'generators' will be used to refer a group of nodes at path cost zero. Generators are often used as the base nodes for computing the path costs of other nodes in the graph, which can be seen in the following chapters. Since we try to find paths between different endpoints to a sole generator, the generator appears to be the root of all the paths.

Input: weighted graph $G$, consisting of nodes $N$ and edges $E$; set of generators $Z$, for each $Z_i \in Z$, $Z_i \in N$. Edge $E_{ij}$ links node $N_i$ and node $N_j$; we denote the cost of this edge by $E^c_{ij}$.
Output: set of cost values: for each node $N_i$, the cost of the shortest path to any node in $Z$, written $c(N_i)$.

1. For each node $N_i$, set $c(N_i)$ to $MAX$.
2. Create a heap $H$, initially empty.
3. For each node $Z_i$, set $c(Z_i)$ to 0. Add $Z_i$ to $H$.
4. Repeat the following until $H$ is empty:
4A. Take the minimum cost node from $H$, say $N_m$.
4B. For each neighbour of $N_m$, say $N_n$, execute step 4C.
4C. If $c(N_n) > c(N_m) + E^c_{nm}$, set $c(N_n) := c(N_m) + E^c_{nm}$, and add $N_n$ to $H$.

**Figure 3.2:** Pseudocode for Dijkstra's algorithm.

In the second step, we apply Dijkstra's algorithm to the weighted graph. The expected path cost value of a node is the cost of the shortest path, that is, the sum of edge weights along the shortest path. In Dijkstra's algorithm, starting from the generator it explores all the neighboring nodes and sets their path cost values to the costs of paths from current edge direction to the generator. Then for each of the neighboring node, it explores their unexplored neighboring nodes and sets their path cost values in the same way. Subsequently all the neighboring nodes of the existing explored nodes are visited. For each node, if we find there exists a smaller path cost than the current path cost, which means there exists a shorter path between the generator and this node, we replace the current path cost with the less cost value. This guarantees that when Dijkstra's algorithm finishes, the final path cost value of each node is the cost of the shortest path to the generator.

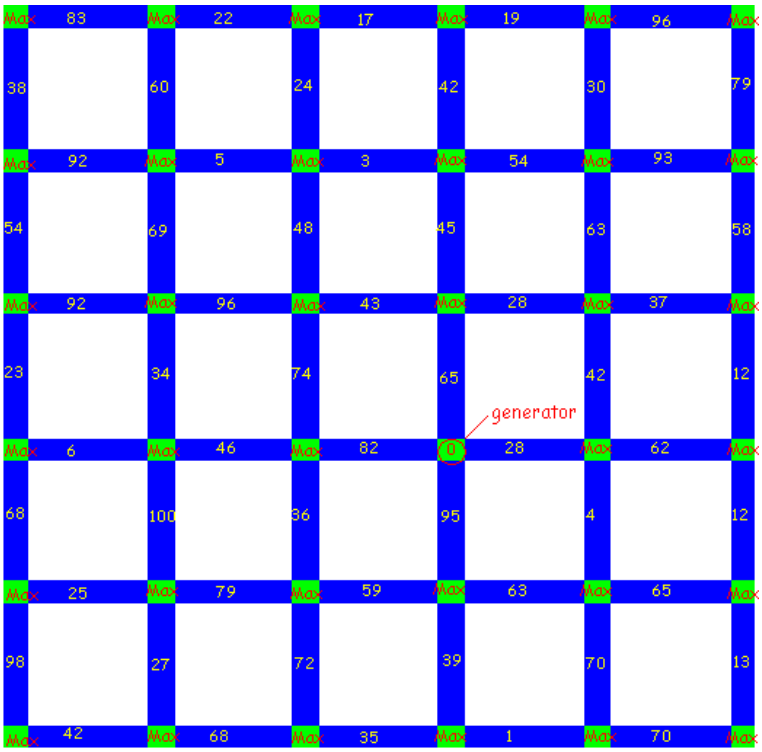



**Figure 3.3:** Initial lattice with random edge weights

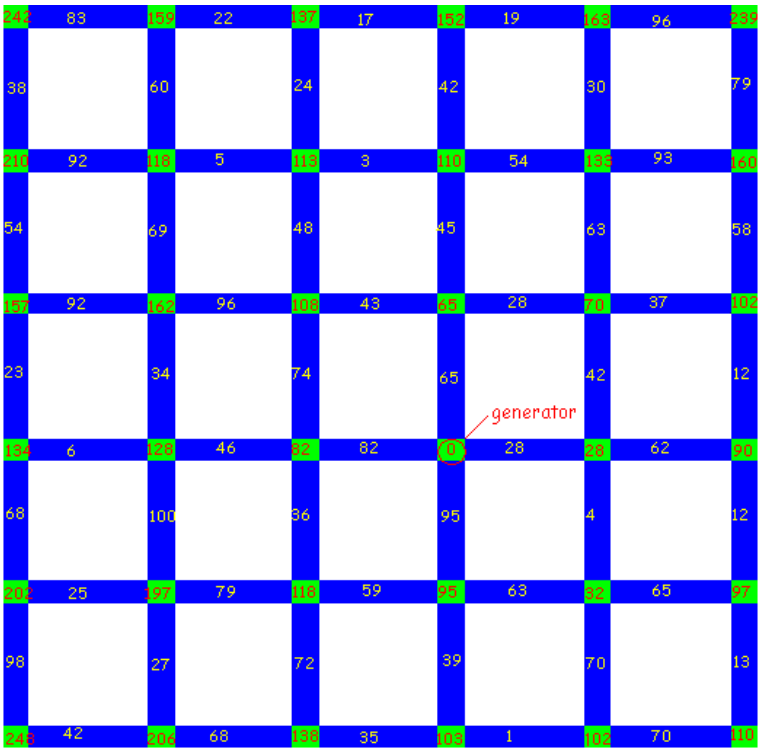



**Figure 3.4:** Lattice after applying Dijkstra's algorithm

The pseudocode of the process is shown in Figure 3.2. Readers may notice that the pseudocode here looks different with the pseudocode we provided in chapter 2 (see Figure 2.7). In fact, instead of keeping a tree which consists of the shortest paths(edges) to each node in the graph, we store the information of the shortest paths(the least path cost) at each node. The ideas of these two psedocodes are same. The graph after applying Dijkstra's algorithm is shown in Figure 3.4.

In the last step, we select a set of endpoints in the lattice and trace the paths back from these endpoints. Since the path cost of each node we calculated in the second step is the cost of shortest path to the generator, to find a shortest path from an endpoint node to the generator, we check all edges connecting to the current node to find the edge that by decreasing its weight along the edge we can arrive at the other ending node, and this ending node is closer to the generator for the path cost of the edge weight than the current node. Then we take the found node as the current node and check its connecting edges in the same way. By tracing edges node by node from the endpoint and approaching closer and closer to the generator, we can find the least cost path composed of these nodes and edges.

If we choose multiple endpoints close in position, when the paths approach the generator they have a high chance of meeting. The result is that they may meet and share some common edges. Due to the fact that we use a same generator for all paths and there is a unique path from the generator to any nodes in the graph, paths will never cross each other but may share the same remaining route to the generator. For the same reason, paths can only share the rest part to the generator and never share any other middle parts. So paths will appear to emerge from the source and then branch when overlapping paths deviate. The union of the paths obtained form the dendrite that we want. The pseudocode is shown in Figure 3.5. The process of endpoint placement and path finding are shown in Figure 3.6 and Figure 3.7 respectively.

Input: weighted graph $G$, consisting of nodes $N$ and edges $E$; generator $Z$; for each node $N_i$ in $N$, the cost of the shortest path to $Z$, written $c(N_i)$; and $N_s$, one endpoint of the path.
Output: a list of nodes $P$, where $P_i \in N$, describing the shortest path from $N_s$ to $Z$.
1. Set $P$ to null; set $N_i$ to $N_s$; append $N_i$ to $P$.
2. Repeat the following until $c(N_i) = 0$.
2A. Find the neighbour of $N_i$, $N_n$, with the lowest $c(N_n)$ such that $c(N_n) = c(N_i) - E^c_{ni}$.
2B. Set $N_i$ to $N_n$.
2C. Append $N_i$ to $P$.

**Figure 3.5:** Pseudocode for pathfinding in a labeled graph.

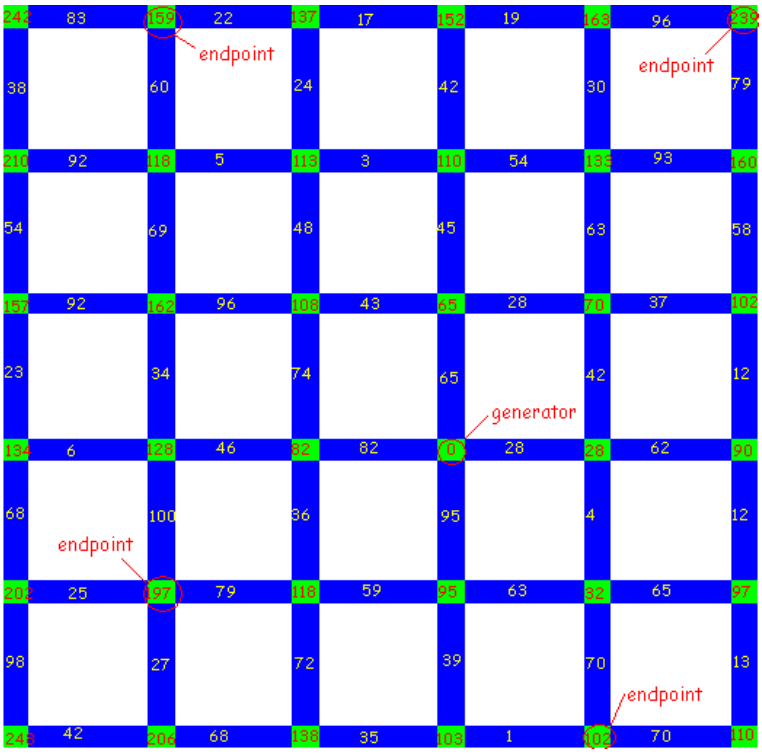



**Figure 3.6:** Endpoints placement.

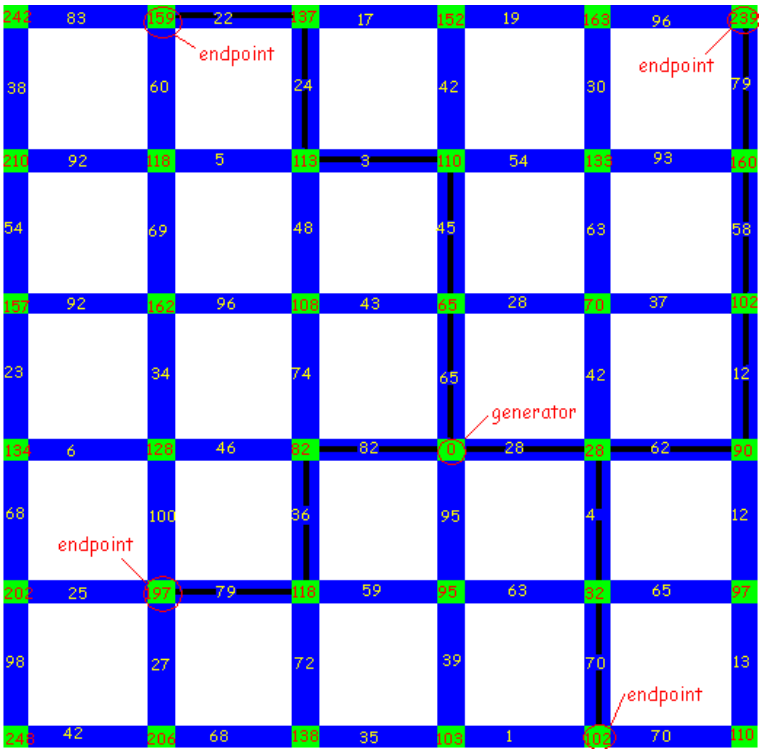



**Figure 3.7:** Tracing paths back from endpoints to the generator.

## 3.3 Simple Controls

Since we already obtained the simple dendrite, readers are eager to use it for some applications. But we find although the simple dendrite we obtained so far demonstrates some obvious characteristics we are interested in, such as branching structures, path converging to the generator and erratic winding travels of individual branches, it still cannot represent any specific dendritic object we often see in the real world. Without further controls the bunch of paths means nothing. We now add some controls of endpoint placement and fractal shape to the basic algorithm. By the control of the endpoint placement we can control the global structure of the dendrite. By controlling the fractal shape, we can control more details of our dendrite. These two control handles play important roles throughout all our modeling work.

### 3.3.1 Endpoint placement

When the generators position is fixed, the positions of the endpoints decide the direction where we want the paths to go. Endpoint placement refers to the method of choosing nodes in the graph as the endpoints. It is very useful when we have a clear thought about what the expected dendrite should be. In this section, we introduce two endpoint placement methods: random placement and manual placement.

Random placement is very simple. We choose endpoints at random positions within the lattice or within some areas in the lattice. The resulting dendrites have many random and variable features, but in most occasions they always do not meet our specific modeling requirements. Even if our modeling objects seemingly have many random spread branches, these branches always manifest specific features. In fact, this method shows its merit when combined with other supplementary criteria for choosing endpoints.

One supplementary criterion along with random placement is the limitation of endpoints within areas of interest. Figure 3.8 shows three simple cases of placing endpoints within geometric areas. The dendrites from left to right are obtained by choosing random endpoints within a square, a circle and a triangle.

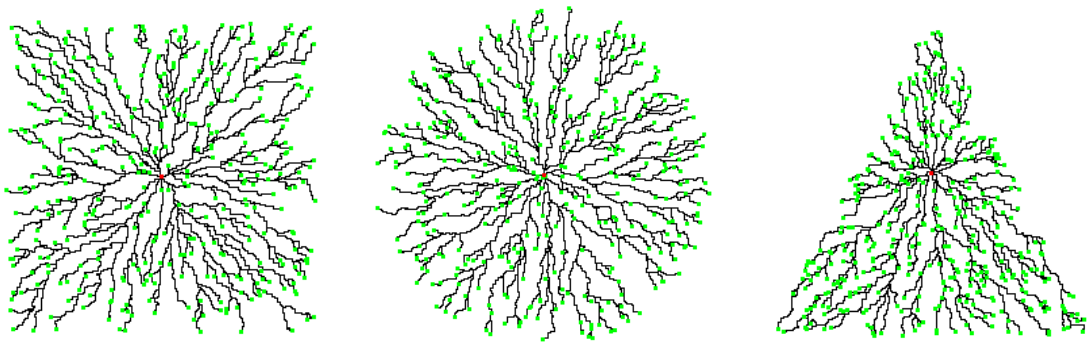


**Figure 3.8:** Dendrites obtained by placing endpoints within different geometric shapes.

One of the most useful criteria for choosing endpoints is the path cost of each node in the graph. In our modeling work, placing endpoints according to their path costs towards the generator shows great advantage

in many applications such as modeling lightning, imitating DLA and modeling elm trees, which will be discussed in chapter 4. Because of the randomness of graph edge weights, nodes having the same spatial distance towards the generator generally have different path costs. Figure 3.9 shows isocontours of spatial distances comparing with isocontours of path costs in the same graph. We can see that isocontours of path costs show much variation and endpoints chosen according to path costs will create more natural looking dendrite.

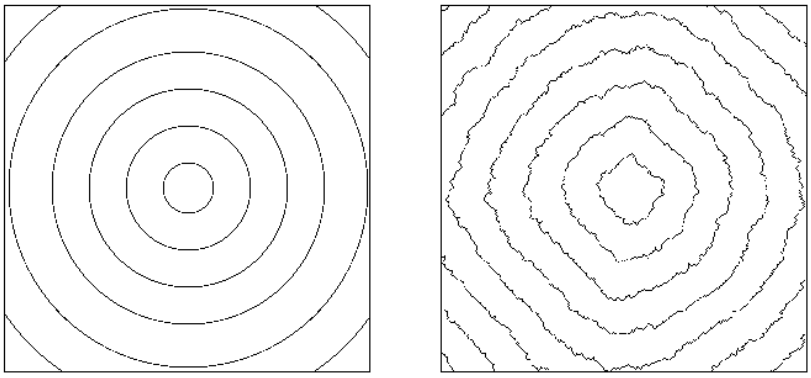


**Figure 3.9:** Left:isocontours of spatial distance; right:isocontours of path costs.

We always use manual placement to create models for objects with simple structure and few branches. We select endpoints in the lattice according to the following principles.

1. We place the endpoints at the positions where we wish the paths to go.
2. The candidate endpoint position is not unique, and all the points within a reasonable area according to the relative position towards the generator can be considered.
3. In order to create a relatively similar model, if the endpoint creates a path quite different from the object branch we discard this one and choose another endpoint near the former one because of principle two.

In general, we use random endpoint placement when we want to add a large amount of paths or we wish the general shape of the paths display more variable features; we use manual endpoint placement when we care more about each individual path and want to control the basic structure of the dendrite. For different modeling tasks, we always choose different endpoint placement methods or combine them together according to specific features of our modeling objects.

### 3.3.2 Fractal Shape

Besides endpoint placement, fractal shape is another important control handle of our algorithm by which we can create more complicated dendrite for our later modeling uses.

**Figure 3.10:** Some fractal dendritic objects in the real world

A fractal is "a rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole" [33]. There are many dendritic objects in nature possessing fractal features, such as ferns, lightning, trees, and snowflakes (see Figure 3.10). To model such kinds of fractal dendritic objects, the simple dendrite created by our former path planning method is not enough. We wish to control the subsequent paths to repeatedly attach to the former dendrite structure. According to this requirement, we take the methods of new generators and cheap former paths to achieve the goal.

### New Generators

The basic idea of the new generators method is, since we know paths are always traced back from the endpoints towards the generator, that is, paths always attach to the generator, we can use the dendrite as a new set of generators. Then the newly added paths will attach to the former dendrite.

Firstly, when we obtain a dendrite we treat all the nodes in the current dendrite as generators. The path costs of all these nodes are set to zero. Next, we apply Dijkstra's algorithm and all the nodes in the graph will be updated with a new path cost. When additional endpoints are placed near the former dendrite, we do the path planning again to find the paths from the additional endpoints to the generators. These paths attaching to the generators (the former dendrite) form the fractal shape.

The process of adding new endpoints and getting the corresponding paths to the former structure is iterative according to our requirement of fractal depth. At each iteration, we increase the number of endpoints by multiplying a branching factor $\beta$. At the same time, the path cost for choosing new endpoints is reduced which can be achieved by dividing an attenuation factor $\alpha$. Because edge weights $W$ are random values, there are no or few nodes in the graph strictly having the path cost $d$. Setting a range parameter $r$ and choosing endpoints by the path cost $d-r<c<d+r$ allows us to have sufficient candidates. The process can be applied $n$ times if our intended fractal depth is $n$ and path cost $d$ is no less than some small value $\varepsilon$, say 200. Notice a larger value of $\alpha$ means the path cost decays faster, so endpoints are closer to the former structure and result a sparser dendrite. Larger value of $\beta$ means more endpoints will be chosen in the next iteration and dendrite will have more branches and appears denser. Figure 3.11 shows a visualization of this process. The upper left structure is the obtained dendrite. The upper right image shows the path cost isocontours of the dendrite if we treat the dendrite as a new set of generators. The lower left image shows the endpoints chosen from the isocontours. The lower right image shows the resulting dendrite with newly added paths. The pseudocode is given in Figure 3.12. Dendrites obtained by different values of $\alpha$ and $\beta$ are shown in Figure 3.13.

### Cheap Former Paths

Following the idea of the new generators method, we find that the key element to make newly added paths attach to the former dendrite is the cheaper cost of the former dendrite. In fact, if the cost of the former

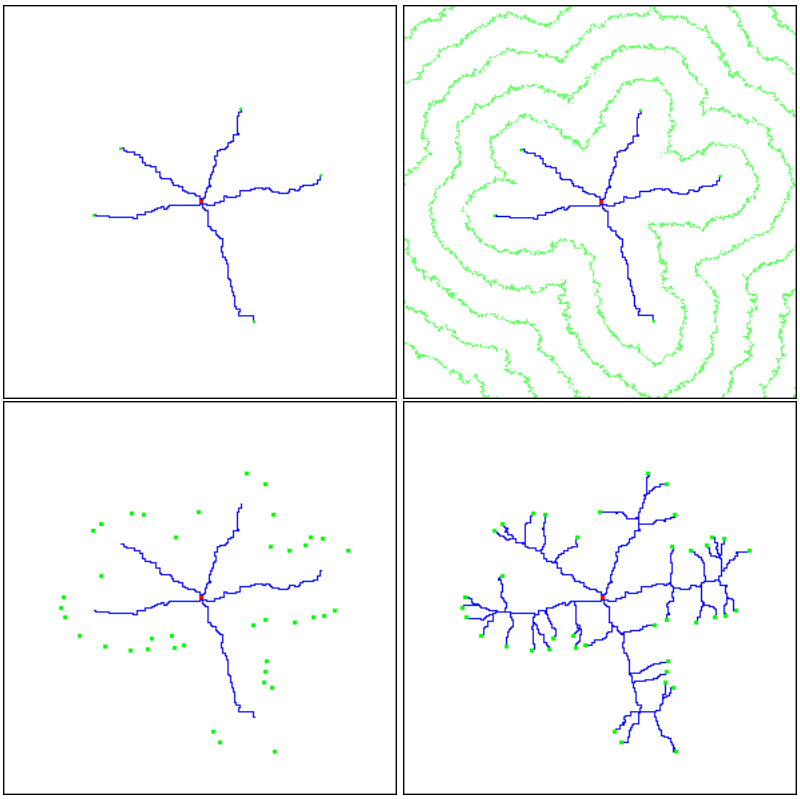


**Figure 3.11:** Fractal dendrite construction process

Input: weighted graph $G$, containing nodes $N$ and edges $E$; an initial generator $Z$; parameters $\alpha$ and $\beta$; initial number of endpoints $m$; initial path cost $d$; cost limit $\varepsilon$ and a range parameter $r$.

Output: a list of nodes $P$ on the fractal dendrite.

1. Set $P$ to null; append $Z$ to $P$.

2. Repeat the following while $d > \varepsilon$.

2A. Find $m$ endpoints at path cost $d - r < c(m) < d + r$.

2B. Find a path from each endpoint to $Z$. Append each path to $P$.

2C. Set $m$ to $m * \beta$.

2D. Set $d$ to $d/\alpha$.

2E. Set $Z$ to $P$.

**Figure 3.12:** Pseudocode for creating fractal dendrites.

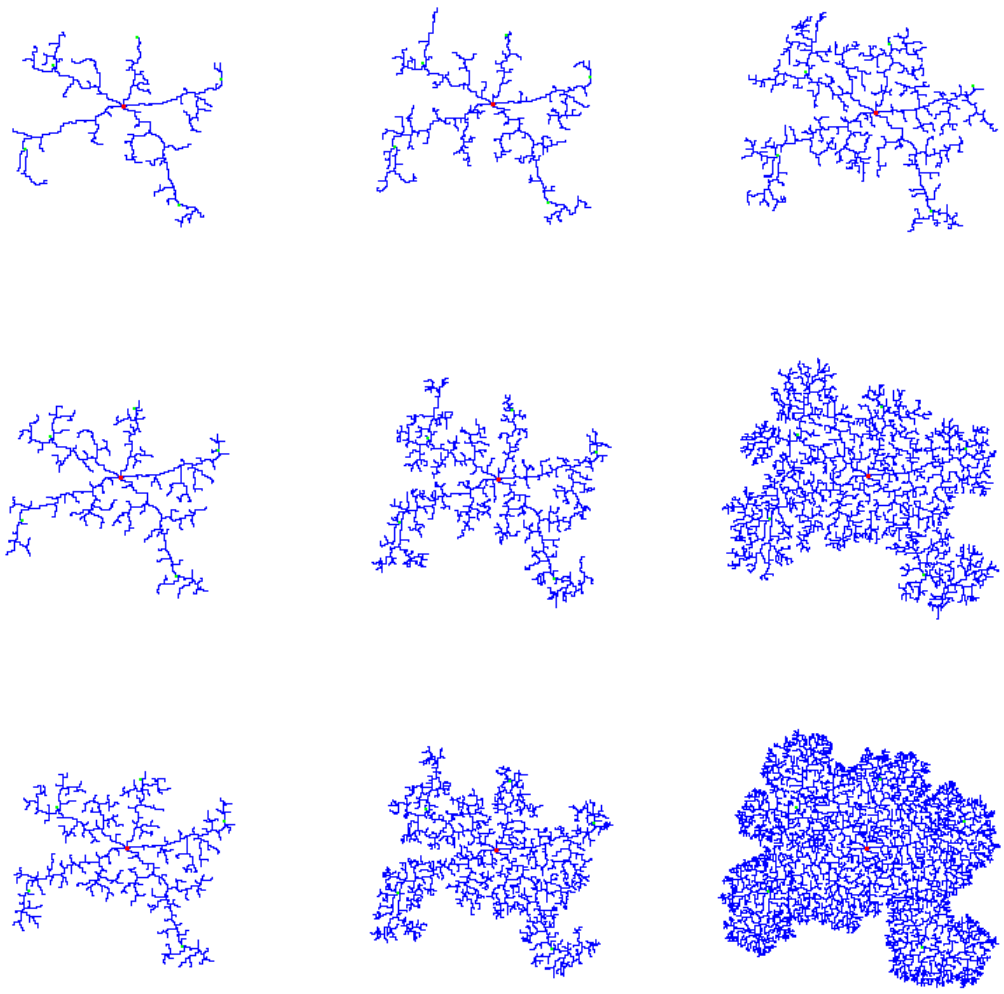


**Figure 3.13:** A few fractal dendrites, with different parameters governing the branching factor and path cost limit at each iteration. Left to right: $\alpha = 2, 1.5, 1.2$; top to bottom, $\beta = 2, 3, 4$.

paths(dendrite) becomes cheaper, since the paths from newly added endpoints prefer to take the cheaper edges, these paths will tend to take edges in the former dendrite. The cheaper the edges in the former dendrite are, the more eagerly the newly added paths take more edges in the former dendrite, and therefore, the longer routes new and former paths overlap.

In the method of cheap former paths, after we obtain a dendrite, we first decrease the weights of edges in the dendrite by multiplying a decreasing factor $D$, that is, the new edge weight $W_{new} = W \times D$. $D$ is always smaller than one and edges become cheaper with smaller value of $D$. Then we apply Dijkstra's algorithm to update the path cost of each node in the graph. Since the edges forming the dendrite are already the least-cost paths from endpoints to the generator, after Dijkstra's algorithm paths will still take these edges to reach the generator. But when additional endpoints are placed in the vicinity of the existing dendrite, because paths will tend to take cheaper edges, the added paths will tend to take the edges in the existing dendrite. When $D$ becomes smaller, which means the edge weights in the existing dendrite decrease faster, the added paths will have more tendency to take more edges in the existing dendrite. Different tendencies lead to different attaching preferences of the added paths to the existing structure.

From the above description, we can find that treating the former dendrite as a new set of generators in fact is setting the cost of the former dendrite zero. So the new generators method is a special case of cheap former paths method when the decreasing factor $D = 0$, and the cheap former paths method is the generalization of the new generators method with more situations. In most occasions we prefer to use the method of cheap former paths, because this method not only creates the fractal shape but also affect the tendency of new path attaching to former dendrite. This control is very useful when we want to control our models to be more outspread or less.

Figure 3.14 shows the later added paths attaching to the former path with different preferences by setting decreasing factor $D$ different values. We can see that when $D$ decreases from 1 to 0.3, the added paths attach to the former structure more eagerly and the paths look more outspread. We can create a fractal structure by iteratively decreasing the weights of edges in the former dendrite, and therefore the later paths will share some common routes of the former dendrite and form a fractal dendrite. Figure 3.15 gives a visualization of this process. At each iteration more endpoints are added in the vicinity of former paths and successively more paths attach to the current structure. The dendrite obtained after several iterations demonstrate a fractal shape which can be seen from many natural objects such as snowflake and lichen.

Although the fractal dendrite we can create so far is not enough for our modeling purpose since most natural dendritic objects possess more complicated structures such as growth tropism and branch width, the simple dendrite produced in this chapter gives us a basic structure. According to the basic dendritic structure, in the next chapter we will explore more different methods to enrich the shape of current simple dendrites, and we will create the specific models for dendritic objects with complicated structures.

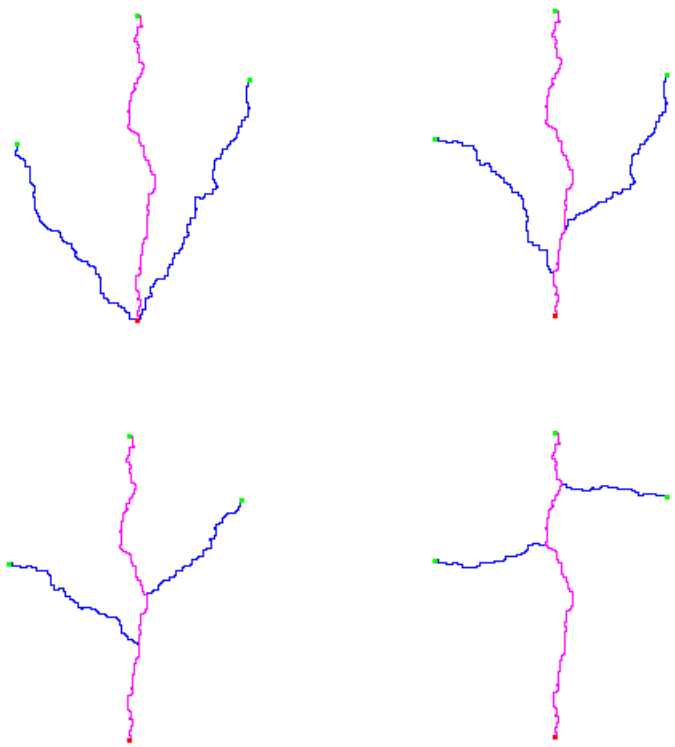


**Figure 3.14:** From top to bottom, from left to right: paths obtained with D=1, 0.8, 0.6, 0.3

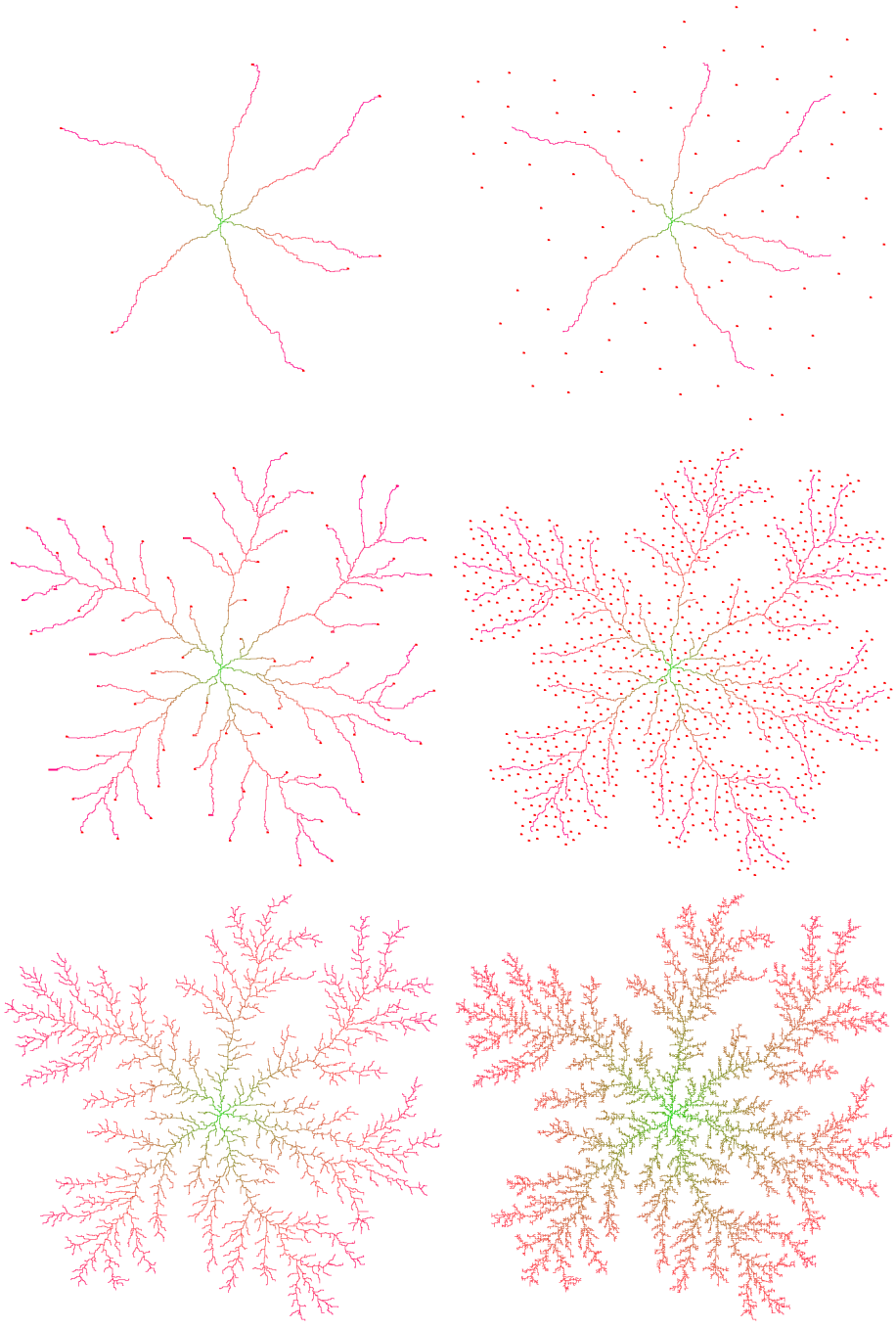


**Figure 3.15:** A fractal dendrite (four iterations). Initially, we have only a few branches, but successively more endpoints are placed at successively smaller path cost from the structure.

# CHAPTER 5

# RESULTS AND DISCUSSION

In previous chapters we introduced our basic algorithm of generating a simple dendrite and by adding kinds of controls we created variable dendrites with specific features. We have also shown the models of some dendritic shapes such as staghorn coral, elm tree and lightning. In this chapter we will give more results to demonstrate the various types of models that our algorithm can generate. From these examples we can get a deeper impression on the performance of our algorithm. After that, we will make comparisons between our algorithm and existing related algorithms including DLA, L-systems and image-based algorithms.

## 5.1 Results

In this section, we focus on demonstrating the versatility of our algorithm by giving further results in different applications. Firstly, we will give more elm tree models generated by the same process. As the extension of modeling elm trees, we will show our imitation of the behavior of space competition between dendritic objects including elm trees and lichens. Next we will illustrate the application of our algorithm for creating artistic esthetic dendritic letters. After that, we will give a lightning model compared with Kim and Lin's lightning model. Finally, we will give timing results.

Our algorithm has some random elements such as random edge weights and random endpoint placement. Many examples given previously have already shown that different endpoints will lead to distinct paths pointing to different directions with different lengths, and different sets of edge weights will produce different shapes of dendrites (e.g. dendrites in Figure 4.5). With these random elements we can easily create various models within the same framework. In chapter 4 we showed one elm tree model created by our path planning algorithm (see Figure 4.13). Now we show four more elm trees that created by the same process in Figure 5.1. Each of these four elm trees looks different from each other. The variations come from different edge weights in the 3D lattice and different endpoint placement. But all these four models share some common characteristics such as random spread branches, tapering property of each branch and slim side twigs attaching to strong main branches. From this example we can see that the modeling variation is a great advantage of our algorithm. It is very easy to create as many models as you want if different random elements are provided.

As an extension of modeling elm trees, we are interested in the behavior of space competition between

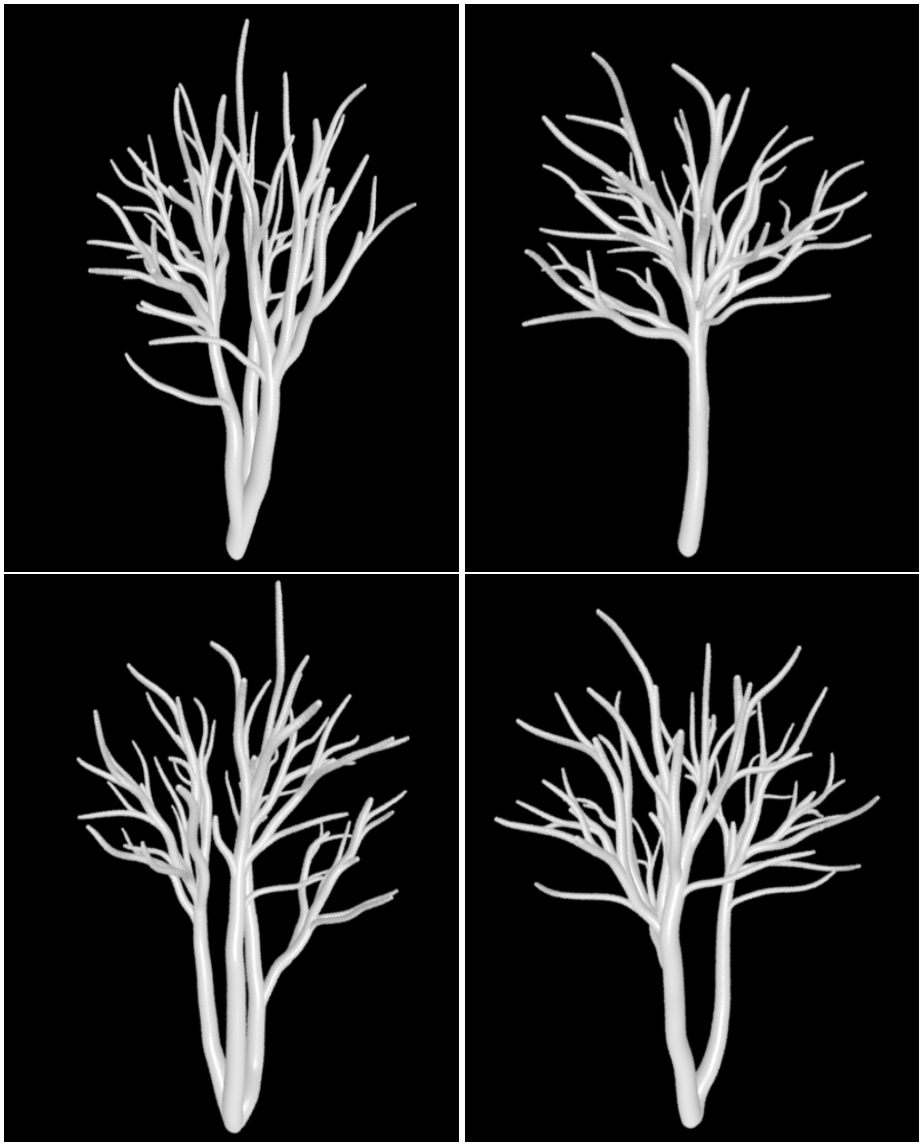


**Figure 5.1:** Elm tree models created by path planning algorithm.

trees. Competition for space is a common phenomenon that occurs through interactions among immediate neighbors and each individual often suppresses the growth of its neighbors to obtain a disproportionate share of the contested space [60] [56]. It is one of the phenomena that can be well simulated by open L-systems [35]. As introduced in chapter 2, open L-systems use the communication symbol for bilateral communication between the plant model and the environment. With the occurrence of the communication symbol, the environment can react to the information from the plant, and plants can send and receive information from the environment. Figure 5.2 shows the competition for space between two tiers simulated with open L-system. The branching in each tier is controlled by collisions between its apices and its own or the neighbor's leaf clusters.

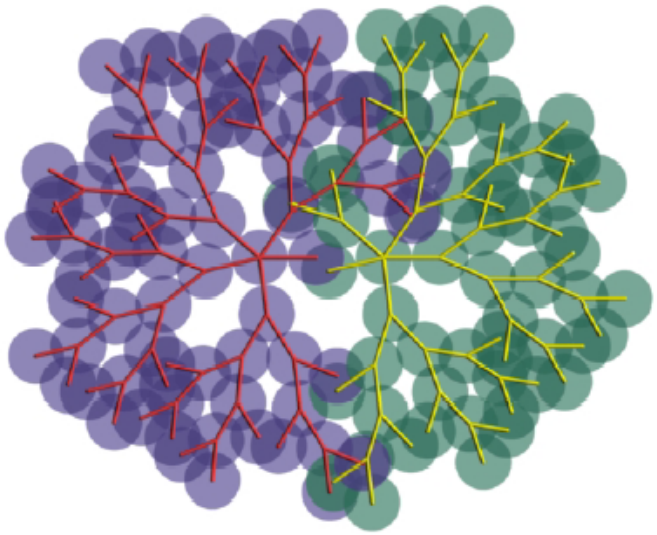


**Figure 5.2:** Competition for space between two tiers of branches simulated by Radomir Mech and Przemyslaw Prusinkiewicz with open L-systems [35].

Now we introduce our algorithm to imitate the space competition between two elm trees. We place two disjoint generators within a graph as roots of the two trees and scatter endpoints nearby to get paths as tree branches. Since paths from each node tend to reach to their nearest generator, all the paths form two dendrites exhibiting avoidance of touching between each other. Although the dendrites do not actually communicate with each other during the path planning process, the avoidance comes from the process of setting each node with the least path costs to their nearest generator by Dijkstra's algorithm. Since each endpoint has its own nearest generator, paths from all the endpoints to the two generators form two separate dendrites. The same method can be used to imitate the space competition between two lichens. Figure 5.3 shows the results.

Competition for space between lichens was also simulated by Desbenoit et al [13], who use the open DLA algorithm to simulate the propagation of lichens over the substrate. Their method first places a seed

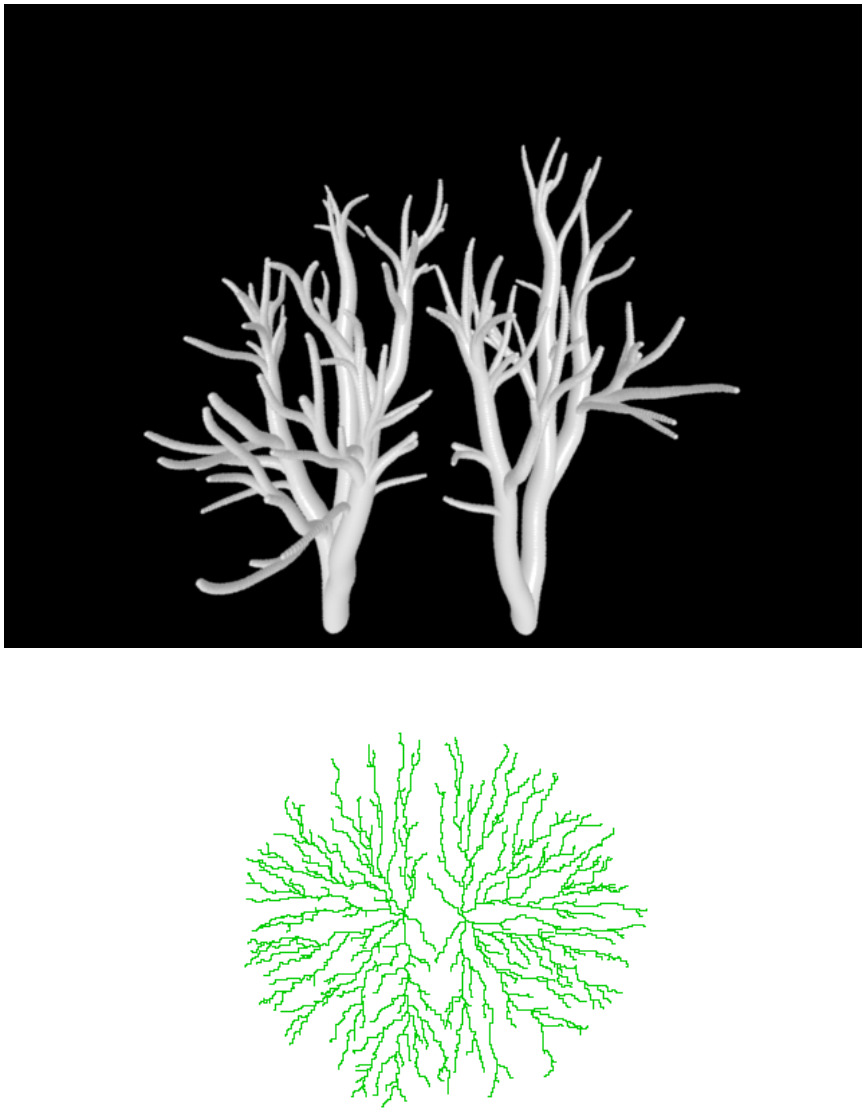


**Figure 5.3:** Imitated competition behavior between two elm trees and two lichens.

particle then creates a cluster of particles by randomly moving new particles and aggregating them when they meet.

Now we use our method to do the simulation of a lichen cluster formed by several kinds of lichens competing for space. Just to obtain a relatively similar visual result of the lichen cluster, we ignore the individual lichen texture and specific geometric patterns. We first choose the endpoints and a generator then apply the path planning algorithm to obtain the dendrite. Next we set all the nodes in the dendrite as a new set of generators and apply Dijkstra's algorithm. After that all the nodes in the graph have a path cost to their corresponding generators. We use the path cost to choose nodes as the lichen particles. Nodes close to the dendrite form the cluster of one kind of lichen while nodes further act as another kind of lichen particles. Because all the path costs form isocontours of the dendrite (see Figure 3.9 and Figure 3.11), we can imitate the shape that one kind of lichen embracing another kind of lichen by placing enough nodes between two adjacent isocontours. The lichen clusters created by Desbenoit's method and our method are shown in Figure 5.4. We must admit that our current imitated lichen cluster has some drawbacks, such as the outer layer of lichen looks coherent while natural lichen cluster often has some irregular openings and isolated lichen particles. But our algorithm shows the possibility of doing the simulation work in a simpler and faster way.

It is also easy to use our algorithm to create the similar result of lichen-writing described by Desbenoit et al. They created the dendritic letters by distributing seeds for DLA in letter shaped areas painted by the designer. Though they did not give the timing figures, we know the open DLA algorithm is very slow. Here we introduce our method. Figure 5.5 demonstrates the application of our algorithm for creating a dendritic letter. Given a stylized letter shape, we create a graph for the letter and set nodes out of the letter shape illegal for path planning. Then we choose a node as the generator and a random set of nodes within the letter-shaped region as endpoints. By finding paths using our path planning algorithm, the resulting paths fill the letter shape in the graph and cause the letter to become visible, so we can obtain a dendritic letter. Based on the method of edge weight distribution described in chapter 4, in order to promote the basic skeleton of the letter, we set edges close to the skeleton of the letter with cheaper weights. The obtained paths show the biasing towards the skeleton of the letter. Figure 5.6 shows different styles of dendritic hello. To generate this figure, we built a separate graph and applied the above process for each letter. From the figures we can see our 2D result is comparable to the dendritic letters of Desbenoit et al (see Figure 5.7). It took less than one second to create a dendritic letter composed of 600 paths within a $300\times$ 300 lattice.

We already described the process of modeling lightning using our path planning algorithm in chapter 3. Here we show another lightning model created by the same process. Figure 5.8 shows our lightning models compared with Kim and Lins lightning model [24]. In our algorithm, we use the path costs of nodes in the graph and width factors of paths to obtain lightning paths with different widths and tapering parts. The resulting erratic branching model simulates the natural properties of lightning branches very well. In Kim and Lin's physically based method, they use the dielectric breakdown model for electrical discharge

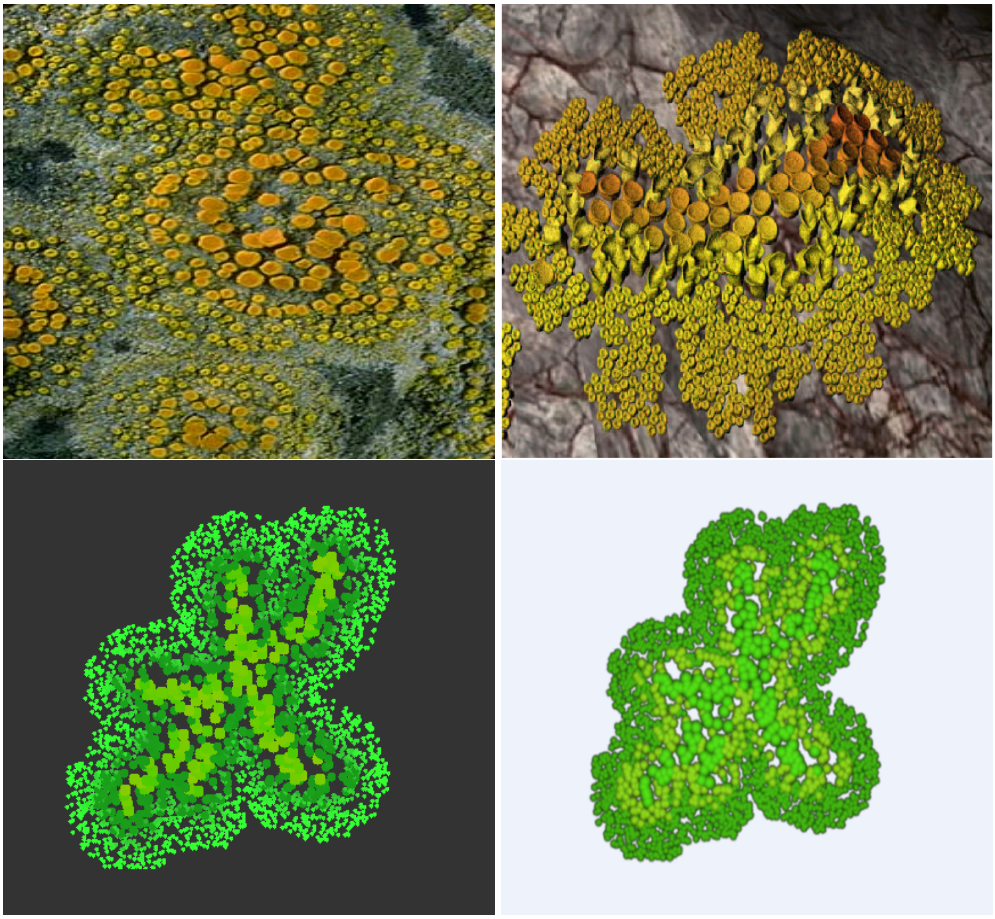


**Figure 5.4:** Top left: real lichen; top right: the lichen model created by Desbenoit et al; bottom: our lichen model in 2D and 3D

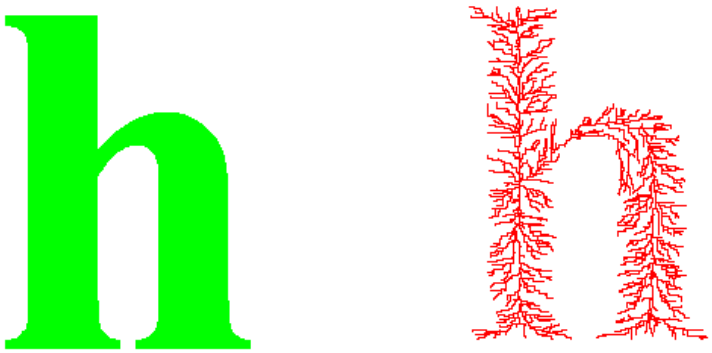


**Figure 5.5:** Left: a letter shape; right: dendritic letter according to the given shape.

**Figure 5.6:** different styles of hello written with dendrites.

**Figure 5.7:** Lichen letters created by Desbenoit et al.

to simulate lightning and get a pretty nice result. Although the modeling mechanisms are quite different, the two algorithms achieved similar visual result performance. But our algorithm is much simpler for non-physicist practitioners. It is easy to create a realistic lightning model without worrying about complicated physical processes. From the timing aspect, it takes less than 5 seconds for creating the lightning model shown in Figure 5.8 by our method, while it takes several minutes for Kim and Lin's method. Judged from above points our algorithm shows more flexibility and has a wider application prospect.

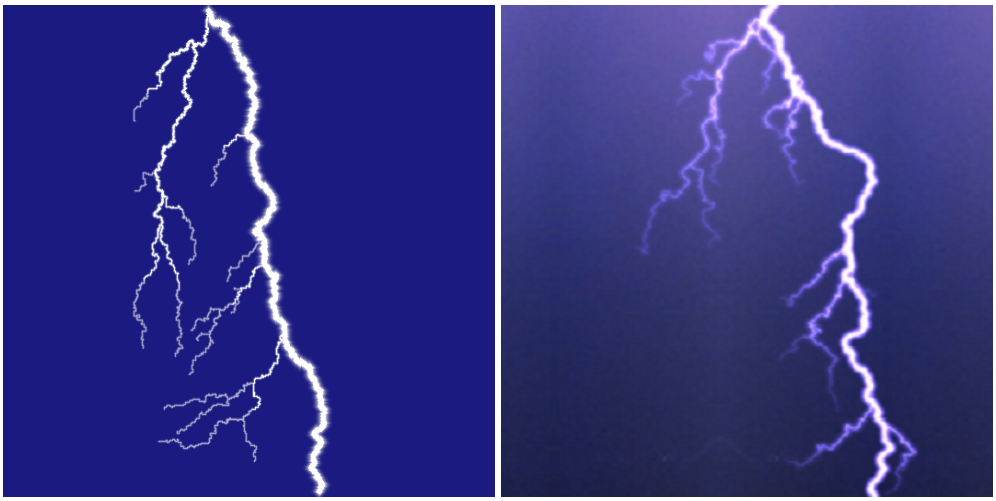


**Figure 5.8:** Left: our lightning model by path planning method; Right: Kim and Lins lightning model by physically based method

Table 5.1 show modeling time results from our method. The timing results are given for a 1.8GHz P4 with 512 MB RAM. Here the modeling time refers to the time needed to create the lattice and produce the dendritic shape. For 3D tree models shown in the thesis, besides the modeling time, the average rendering time(the time needed to build spheres for each node in the paths) is less than 4 seconds.

**Table 5.1:** Modeling time results

| Model | lattice | endpoints | time |
|---|---|---|---|
| Simple 2D dendrite | $600^2$ | 15 | 0.94s |
| Fractal 2D dendrite | $512^2$ | 8930 | 7.55s |
| Lightning | $600^2$ | 24 | 4.16s |
| Coral (with refinement) | $50^3$ | 24 | 3.06s |
| 3D trees (no refinement) | $80^3$ | 17 | 3.65s |

## 5.2 Comparison with existing algorithms

We have presented our algorithm of modeling dendritic shapes using path planning. We used the method to create models for three common dendritic objects: staghorn coral, lightning and elm tree. We also demonstrated the versatility of our algorithm for creating similar results in some applications. Next we will discuss the characteristics of our algorithm by comparing our algorithm with some popular algorithms for modeling dendritic shapes.

As mentioned in chapter 2, there exist some algorithms for generating dendrites such as L-systems, DLA and some image-based methods. Each of them has been used for creating dendritic models and obtained good results. Here we will make some comparisons between these algorithms and our algorithm. We are not trying to find an omnipotent algorithm suitable for modeling all dendritic shapes and we will not review all the details of these algorithms (which can be seen in chapter 2). We just wish to show readers the advantages and drawbacks of our algorithm when compared with these algorithms and suitability of them for creating some specific dendritic models.

### 5.2.1 Our algorithm and DLA

DLA is a common algorithmic process for generating dendritic shapes. The standard DLA process starts from a seed particle. When a random particle is released from a distant boundary and reach the seed after a random walk due to Brownian motion, they will stick into a cluster. The cluster forms a dendritic shape in a fractal pattern as shown in Figure 5.9.

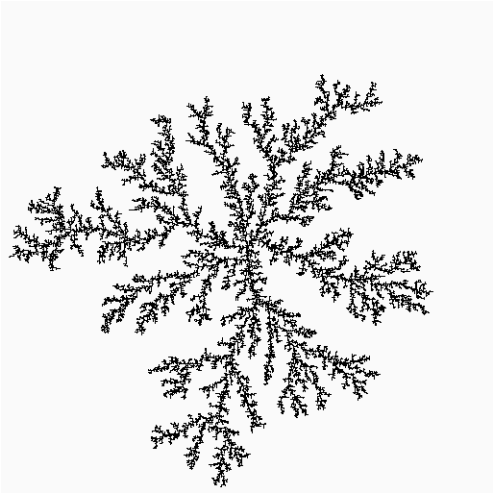


**Figure 5.9:** Dendrite form generated by diffusion-limited aggregation.

The main control handle of DLA is the global control. That is, we can only control the general shape of the resulting dendrite. Because the cluster can aggregate more particles from the direction that particles are released, the cluster will tend to grow toward that direction. We can control the overall dendritic shape by

controlling positions of releasing particles but we cannot tell what the detail paths look like. Although the basic DLA algorithm does not have the local control, the improvement of adding 'stickiness' to the particles partly provides the control of local details.

We use our path planning algorithm to imitate the dendritic cluster of DLA and obtain a good result as shown in Figure 5.10. We choose a generator in the graph center and manually place some endpoints in a outer contour far away. By this means we can control the global shape of the dendrite. Then we successively place more random endpoints at successively smaller path cost from the structure, so that after some iterations we can obtain the fractal dendrite. We can control the local details by controlling the iteration counts, the tendency of the side paths attaching to the main structure, endpoint number and criterion of path costs for choosing endpoints. The iteration counts can control the complexity of the fractal dendrite; the attaching tendency of the side paths to the main structure affects the spread degree of local paths (see section 3.3.2 for details); endpoint number decides how many side paths attaching to the structure; the criterion of path costs decides the general length of side paths. We can see the two models show similar visual characteristics such as dendritic shape, fractal structure and branches spreading outward from the center of the cluster.

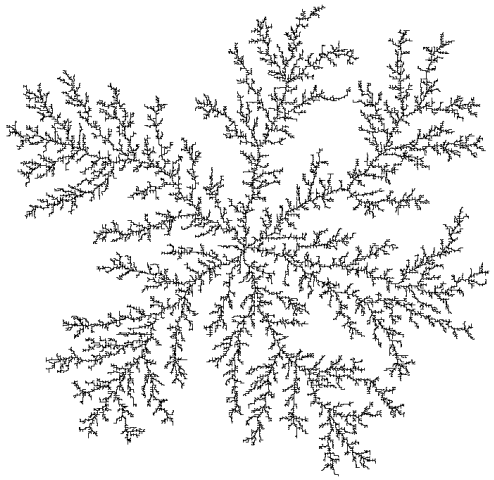


**Figure 5.10:** Imitation of DLA with a path planned fractal (4 iterations).

The efficiency is a big problem for DLA. The random walk process from the particles are released until they are stuck to the cluster is time consuming. For creating a dendrite shown in Figure 5.9 it needs about 8 minutes.

The process to create an imitated dendrite of DLA using our path planning algorithm is much faster. The most time consuming part is Dijkstra's step. Because all the path costs of nodes need to be checked and updated according to generators, more generators mean more checking times. Despite the costly Dijkstra's steps our algorithm still show the dominant advantage over DLA in timing. To create a similar dendrite as shown in Figure 5.10, our algorithm only need less than eight seconds which is about 100 times less computer time than DLA.

Of course our simulation is not perfect. Although to the unaided human eyes these two models are visually similar, if investigated in detail, the dendrite created by DLA obviously look more similar to some natural dendritic objects especially the dendrites obtained by aggregation of particles such as ice crystals, neurons and lichens. Because the DLA process is very close to their natural forming process. One drawback of our imitated dendrite is that the fractal is only over a small range of scales that needs users to explicitly program in. Although increasing the fractal iterations will lead to more computer time, by measuring between the extremely similar results and huge timing difference our algorithm is still a good choice for creating a DLA dendrite.

### 5.2.2 Our algorithm and L-system

L-system is a parallel rewriting grammar often used to model the growth processes of plant development. Beginning from an initial string, L-system repeatedly uses a replacement grammar to create strings which can be interpreted as a variety of botanical forms, particularly branching structures.

The most obvious character of models created by L-system is regularity. The work is based on the premise that the growth of the structure follows the fixed production pattern. The resulting structure is obtained by the recursive replacements of the production string. We can see from Figure 5.11 that the left branching structure has a very regular fractal shape. Although the right branching structure does not look so obviously regular as the left one, its branches still bear same features such as branching angle and twig length. Even if from the great result of tree model created by open L-system (see Figure 5.12) and different thickness of branches are imported, we can easily find most branches are same. But in the real world, not all the trees look so regular, and even trees in the same species still have many variances in shapes.


**Figure 5.11:** Branching structures created by L-system

**Figure 5.12:** A tree model created by open L-system

Another drawback to L-system lies in the difficulty of devising the system of replacement rules. Even if users have an image of the intended structure in mind, it is still hard to build the connection between the resulting shape and the replacement rules. On the other hand, users can hardly imagine what the specific resulting structure looks like if only provided with an abstract string.

Compared with L-system, the modeling process of our algorithm is more intuitive. Users can manually place endpoints to decide the exact paths. The resulting connection between endpoints and paths is very straightforward. Users can also use the control of fractal shape to control the fractal complexity of the structure. At this point, our algorithm can achieve the same effect which is obtained by recursive times of replacement rules in L-system. Variation of the results is one of the most important features of our algorithm. We can place endpoints randomly to add side branches and change the edge weights to control the path to be more straight or erratic. We can control the tropism of the dendrite easily by setting the edge weights according to spatial locations. From these aspects, our algorithm shows more flexibility than L-system. Our Figure 5.13 shows tree models created by L-system and our path planning algorithm separately. Compared with the tree model created by L-system, our tree has many random side branches and each branch has different spread status. These variations of our model simulate the natural properties of real trees better and look more realistic than the L-system tree model. And more important as illustrated in the last section, we can use the same mechanism to create models in different shapes while possessing some common features at the same time as most L-system one string can only be used to create one structure. Although stochastic L-systems bring into the concept of production probability and a same string can be replaced using different productions according to the production probabilities, it is still easier to set different edge weights and choose different endpoints than setting different production rules. So we can say that our algorithm has a more powerful productivity than L-system.

**Figure 5.13:** Left: tree model created by L-system. The figure is taken from "L-System Plant Geometry Generator" [8]; Right: elm tree model created by path planning algorithm.

### 5.2.3 Our algorithm and image-based algorithms

Unlike L-system focusing on the plant development process, some computer graphics researchers pay more attention to simulating the appearance of trees. Image-based modeling method is a popular method widely used to create 3D models based on images of a scene provided by users instead of geometric primitives. Just for comparison we are more interested in the image-based modeling work of trees.

In the algorithm of Reche et al [52], the tree is considered as a volume with opacity and color values. The values of each cell in the volume are computed by a set of textures obtained from the original image. Importing the texture information into the algorithm helps to create a appealing realistic model. But because the lighting information is also incorporated into the textures the approach cannot simulate the tree under different lighting conditions. A tree model is shown in Figure 5.14.

In the recent work of this area, Tan et al [61]and Neubert et al [38] have created great tree models.

Tan et al first captured the appearance of the tree from a number of different overlapping view and drew a 3D point cloud. Then the 3D point cloud was used to reconstruct the visible branches. Users need to click on 3D points in the primary branch to select branch clusters. A sub-graph was built for each branch cluster. Then they found the shortest paths from the root point to all other points and extracted the skeleton as the branches. Their bare tree model is shown in Figure 5.15.

In the algorithm of Neubert et al they used the information of input photographs to build an approximate voxel-based tree volume with voxels containing density values. The density values are used to locate a set of particles. Then they perform a 3D flow simulation to trace the particles downwards to the tree basis to form twigs and branches. A tree skeleton of an oak tree created by their method is shown in Figure 5.16

**Figure 5.14:** A tree model created by Reche-Martinez et al.

**Figure 5.15:** A bare tree model created by Tan et al

**Figure 5.16:** An oak tree model created by Neubert et al

The premise of these image-based methods is that users need to take a set of photographs for the real object. Images for the same object can only be used to create one model. This obviously limits the productivity. For our algorithm these problems do not exist. We do not need any preparation work and our modeling is not fixed with specific objects existing in the real world. Our algorithm is very prolific since the same mechanism can be used to create a variety of models. From the aspect of user interaction, the methods all need users intervention more or less. In our algorithm users can decide to place few endpoints for the main branches since it is a very small amount of work or let the algorithm to do it automatically. Compared from the result aspect, image-based method shows more advantages. Since the method makes use of information obtained from the real world the resulting models look very realistic. So it is much more suitable for creating models to precisely simulate existing objects while our algorithm is more flexible and efficient to create multiple models with variations and demonstrating common features.

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

Dendritic shapes are common in nature and models of dendritic shapes are widely needed in many areas. Because of their branching fractal and erratic structures modeling dendritic shapes is a tricky task. Existing methods are slow and complicated.

In this thesis we present a procedural algorithm of using path planning to model dendritic shapes. We generate a dendrite by finding the least-cost paths from multiple endpoints to a common generator and use the dendrite to build the geometric model. With the control handles of endpoint placement, fractal shape, edge weights distribution and path width, we create different shapes of dendrites that simulate different kinds of dendritic shapes very well. The main contribution of the thesis is the introduction of path planning method for modeling task. Compared with some existing methods including physically based method, L-system and image-based method, our algorithm is fast and simple.

Here we give an overview for the whole thesis. At the beginning of the thesis we surveyed some previous work in the related areas of path planning and existing methods for generating dendrites. Based on the review we pointed out the drawbacks of these methods and presented our solution. In chapter 3 and chapter 4, we introduced our algorithm in details and showed our models for three kinds of commonly seen dendritic forms: staghorn, elm tree and lightning. After that, we gave further results demonstrating the versatility of our algorithm and made comparisons with several existing methods. With the results and comparisons, we can evaluate our algorithm as an effective and simple method. Compared with the real dendritic shapes, although we cannot duplicate the original object, our models grasped the basic characteristics and simulated the erratic branching structure very well. Compared with the existing methods, our method can obtain similar or better results in a simpler process with much less computer time. Furthermore, the intuitive control mechanism guarantees the productivity of various models with efficiency.

## 6.2 Future Work

One possibility of future work would be building models in a 3D space with higher resolution. Although we give the path refinement method as the solution of lattice resolution problem in chapter 4, building a model

with a high resolution for each part is not efficient. For example, for a tree, we may pay more attention on the tiny twigs demonstrating more details than the straight main branches with less detail. Since building the whole lattice with high resolution is costly, we believe building a lattice with flexible resolution is a promising way. We could build the part we are more interested in a higher resolution, while build the part with less details in a low resolution.

Since the models we created so far are built in a regular 4-connected in 2D or 6-connected in 3D lattice. As one possible improvement, non-regular lattice could be used instead of current regular lattice. The advantage is that our algorithm could be applied on the surface of an existing 2D or 3D object. Another possible improvement is that we can use the 8-connected lattice in 2D or 26-connected lattice in 3D. By doing so, we could generate a smoother dendrite.

Another direction is to build models for more dendritic shapes. We would explore to modeling different kinds of trees, rivers, cactus and antlers. One possible approach for modeling such dendritic forms with different features could be exploring more control handles from our current algorithm, such as endpoint placement and edge weights distribution. For the endpoint placement, we would find a more flexible and easy to control method instead of current spatial distribution. For the edge weights distribution, we could permit a user to draw a sketch in the lattice. The edge weights could be set with different distributions in different sketched areas.

We would also explore the application of our algorithm for Non-Photorealistic Rendering (NPR). The dendrites can be used for stylized aesthetic designs [31]. And animation of dendrites is also a promising direction to explore.

# REFERENCES


[1] Nancy M. Amato, O. B Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. *International Conference on Robotics and Automation*, 16:442–447, 1998.

[2] Felipe Barra, Benny Davidovitch, Anders Levermann, and Itamar Procaccia. Laplacian growth and diffusion limited aggregation: Different universality classes. *Physical Review Letters*, 87, 2001.

[3] Paul Bourke. Constrained diffusion limited aggregation in 3 dimensions. *Computer and Graphics*, 30:646–649, 2006.

[4] Paul Bourke. DLA - diffusion limited aggregation. *http://local.wasp.uwa.edu.au/ pbourke/fractals/dla/*, accessed December,2007.

[5] A. Bunde and S. Havlin. *Fractals and Disordered Systems*. Springer-Verlag, Berlin, 1996.

[6] Tom Capizzi. *3D Modeling And Texture Mapping*. Premier Press, 1982.

[7] Danny Z. Chen and Notre Dame. Developing algorithms and software for geometric path planning problems. *Computing Surveys*, 1996.

[8] Hung-Wen Chen. L-System plant geometry generator. *http://www.nbb.cornell.edu /neurobio /land /OldStudentProjects /cs490-94to95 /hwchen/*, 1995.

[9] Howie Choset. Coverage for robotics - a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31:113–126, 2001.

[10] Dana Cobzas and Martin Jagersand. A survey of image-based modeling and rendering. *IEEE Virtual Reality 2003 Tutorial*, 2003.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[12] R. Neumann de Carvalho, H.A. Vidal, P. Vieira, and M.I. Ribeiro. Complete coverage path planning and guidance for cleaning robots. *Industrial Electronics, 1997. ISIE '97., Proceedings of the IEEE International Symposium*, 2:677–682, 1997.

[13] Brett Desbenoit, Eric Galin, and Samir Akkouche. Simulating and modeling lichen growth. *Computer Graphics Forum*, pages 341–350, 2004.

[14] Peter Eichhorst and Walter J. Savitch. Growth functions of stochastic lindenmayer systems. *Information and Control*, 45:217 – 228, 1980.

[15] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[16] G.Albinet and P.Pelce. Computer simulation of neurite outgrowth. *Europhysics Letters*, 33:569–574, 1996.

[17] Andrew S. Glassner. The digital ceraunoscope: Synthetic thunder and lightning. *Computer Graphics and Applications*, 20:89–93, 2000.

[18] Jean-Francois Gouyet. *Physics and Fractal Structures*. Springer-Verlag, 1996.

[19] G.T.Herman and G.Rozenberg. *Developmental systems and languages*. North-Holland, Amsterdam, 1975.

[20] Feng Han and Song-Chun Zhu. Bayesian reconstruction of 3d shapes and scenes from a single image. *HLK '03: Proceedings of the First IEEE International Workshop on Higher-Level Knowledge in 3D Modeling and Motion Analysis*, page 12, 2003.

[21] Andrew Harrison. *Fractals in Chemistry*. Oxford University Press, 1995.

[22] Marcelo Kallmann. Path planning in triangulations. Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games, pages 49–54. International Joint Conference on Artificial Intelligence (IJCAI), 2005.

[23] Theodore Kim and Ming C.Lin. Visual simulation of ice crystal growth. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2003.

[24] Theodore Kim and Ming C.Lin. Physically based animation and rendering of lightning. In *Pacific Conference on Computer Graphics and Applications*, pages 267 – 275, 2004.

[25] Theodore Kim, Michael Henson, and Ming C.Lin. A hybrid algorithm for modeling ice formation. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2004.

[26] James J. Kuffner. Effective sampling and distance metrics for 3d rigid body path planning. *Robotics and Automation, Proceedings. ICRA '04. 2004 IEEE International Conference*, 4:3993–3998, 2004.

[27] Zhuming Lam and Scott A.King. Animation of tree development. *Image and Vision Computing New Zealand*, 2003.

[28] Ming C. Lin and Dinesh Manocha. Applied computational geormetry, towards geometric engineering, fcrc'96 workshop, wacg'96, selected papers. volume 1148 of *Lecture Notes in Computer Science*. Springer, 1996.

[29] Aristid Lindenmayer. Mathematical models for cellular interaction in development, parts i and ii. *Journal of Theoretical Biology*, 18:180–315, 1968.

[30] Aristid Lindenmayer. Developmental systems without cellular interaction, their languages and gramars. *Journal of Theoretical Biology*, 30:455–484, 1971.

[31] Jeremy Long. Modeling dendritic structures for artistic effects. *Master thesis*, 2007.

[32] Benoit B. Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *Science, New Series, Vol. 156, No. 3775*, 1967.

[33] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., USA, 1982.

[34] P. Meakin. Diffusion-controlled cluster formation in two, three and four dimensions. *Physical Review A*, 27:604–607, 1983.

[35] Radomir Mech and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, 1996.

[36] Hiroshi Mizuseki and Yoshiyuki Kawazoe. Simulation of electrochemical deposition process by a multiparticle diffusive aggregation model. *Journal of Applied Physics*, 87:4611–4616, 2000.

[37] K. Moriarty, J. Machta, and R. Greenlaw. Parallel algorithm and dynamic exponent for diffusion-limited aggregation. *Phys. Rev. E*, 55:6211–6218, 1997.

[38] Boris Neubert, Thomas Franken, and Oliver Deussen. Approximate image-based tree-modeling using particle flows. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2007)*, 26(3), 2007.

[39] Christoph Niederberger, Dejan Radovic, and Markus Gross. Generic path planning for real-time applications. In *CGI '04: Proceedings of the Computer Graphics International (CGI'04)*, pages 299–306. IEEE Computer Society, 2004.

[40] Byong Mok Oh, Max Chen, Julie Dorsey, and Fredo Durand. Image-based modeling and photo editing. *SIGGRAPH 2001 Proceedings*, 2001.

[41] Byong Mok Oh, Max Chen, Julie Dorsey, and Fredo Durand. Image-based modeling and photo editing. *SIGGRAPH 2001 Proceedings*, 2001.

[42] Manuel M. Oliveira. Image-based modeling and rendering techniques: A survey. *RITA*, 9:37–66, 2002.

[43] Joseph O'Rourke. *Computational Geometry in C (Second Edition)*. Cambridge University Press, 1998.

[44] Edith Perrier, Christian Mullon, and Michel Rieu. Computer construction of fractal soil structures: Simulation of their hydraulic and shrinkage properties. *WATER RESOURCES RESEARCH, VOL.31, NO.12, PAGES 2927-2943*, 1995.

[45] P. Prusinkiewicz and J. Hanan. Lindenmayer systems, fractals and plants. *Lecture Notes on Biomathematics*, 1989.

[46] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphical Interface 86*, pages 247 – 253, 1986.

[47] Przemyslaw Prusinkiewicz and Mark Hammel. A fractal model of mountains with rivers. In *Proceeding of Graphics Interface 93*, pages 174 – 180. Springer, 1993.

[48] Przemyslaw Prusinkiewicz, Mark Hammel, Radomir Mech, and Jim Hanan. The artificial life of plants. *Artificial life for graphics, animation, and virtual reality,SIGGRAPH 95 Course Notes*, 7:1:1 – 38, 1995.

[49] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. *Computer Graphics*, 28:351–358, 1994.

[50] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[51] Long Quan, Ping Tan, Gang Zeng, Lu Yuan, Jingdong Wang, and Sing Bing Kang. Image-based plant modeling. *ACM Trans. Graph.*, 25:599–604, 2006.

[52] Alex Reche, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, 23:720–727, 2004.

[53] Todd Reed and Brian Wyvill. Visual simulation of lightning. *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 359–364, 1994.

[54] Tatsumi Sakaguchi. Botanical tree structure modeling based on real image set. *ACM SIGGRAPH 98 Conference abstracts and applications*, page 272, 1998.

[55] V. B. Sapozhnikov and V. I. Nikora. Simple computer model of a fractal river network with fractal individual watercourses. *Journal of Physics A Mathematical General*, 26, 1993.

[56] Susanne Schwinning and Jacob Weiner. Mechanisms determining the degree of size asymmetry in competition among plants. *Oecologia*, pages 447–455, 1998.

[57] David S.Ebert, F.Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing Modeling,A Procedural Approach(Third Edition)*. Elsevier Science,USA, 2005.

[58] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen R.Marschner, Erik Reinhard, Kelvin Sung, William B.Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics*. A K Peters,Wellesley,Massachusetts,USA, 2005.

[59] Ilya Shlyakhter, Max Rozenoer, Julie Dorsey, and Seth J. Teller. Reconstructing 3d tree models from instrumented photographs. *IEEE Computer Graphics and Applications*, 21:53–61, 2001.

[60] Montgomery Slatkin and D. John Anderson. A model of competition for space. In *Ecology, Vol. 65*, pages 1840–1845, 1984.

[61] Ping Tan, Gang Zeng, Jingdong Wang, Sing Bing Kang, and Long Quan. Image-based tree modeling. In *International Conference on Computer Graphics and Interactive Techniques archive, ACM SIGGRAPH 2007*, 2007.

[62] M. Tasinkevych, J. M. Tavares, and F. de los Santos. Diffusion-limited deposition with dipolar interactions: Fractal dimension and multifractal structure. *Journal of Chemical Physics*, 124, 2006.

[63] T.A.Witten and L.M.Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Physical Review Letters*, 47:1400–1403, 1981.

[64] R. D. Theriault and A. D. Fowler. Harrisitic textures in the centre hill complex, munro township, ontario: product of diffusion limited growth. *Mineralogy and Petrology*, 54:35–44, 1995.

[65] T. Vicsek, F. Family, and P. Meakin. Multifractal geometry of diffusion-limited aggregates. *EUROPHYSICS LETTERS*, 12(3):217–222, 1990.

[66] H. von Koch. Une methode geometrique elementaire pour l'etude de certaines questions de la theorie des courbes planes. *Acta mathematica, 30:145-174*, 1905.

[67] Richard F. Voss. Multiparticle diffusive fractal aggregation. *Journal of Statistical Physics*, 36:861–872, 1984.

[68] Richard F. Voss and Micha Tomkiewicz. Computer simulation of dendritic electrodeposition. *J.Electrocheml.Soc.*, 132:371–375, 1985.

[69] Huang Weiguang and D. Brynn Hibbert. Computer modeling of electrochemical growth with convection and migration in a rectangular cell. *Physical Review E*, 53:727–730, 1996.

[70] Patrick Winston. *Artificial Intelligence*. Addison-Wesley, USA, 1992.

[71] Ling Xu and David Mould. Modeling dendritic shapes - using path planning. In *GRAPP (GM/R)*, pages 29–36, 2007.

[72] Z. Zhang. Image-based modeling of objects and human faces. *Proceedings of SPIE*, 4309:1–15, 2001.