Path Planning for Natural Phenomena

Thesis submitted at Bournemouth University
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Animation and Visual Effects

by

Erika Camilleri
Faculty of Media and Communication
Bournemouth University

22 August 2016

To two exemplary people who proved that the best path planning heuristic for success is persistance.

Perit. Alfred Briffa

&

David Briffa

Declaration

All sentences or passages quoted in this project dissertation from other people's work have been specifically acknowledged by clear cross referencing to author, work and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Acknowledgements

There are a few people I would like to acknowledge as without them this thesis would not have possible. I would like to thank Jonathan Macey, Hammadi Nait-Charif and Ian Stephenson for their invaluable guidance and input throughout this project. I am indebted to David Briffa for his encouragement over the past year and his assistance with the creation of grayscale images used in this project. Lastly, I would like to thank my parents, in-laws and friends for their continued support.

Abstract

The ability to create sufficiently detailed natural phenomena is important for building virtual environments, unfortunately this is a laborious task. Although various procedural techniques are available, new techniques that can ease the burden on digital artists are highly sought after. The use of path planning techniques that originate from artificial intelligence is a viable avenue for the generation of models resembling natural phenomena characterized by their dendritic nature such as plants, coral, trees and lightning. The technique involves finding the least-cost paths through a weighted graph from a single source to a number of destination points. This work presents a new method of informing the behaviour of the path planner through images in order to build models that emulate certain familiar objects. The effectiveness of the method is shown in three examples, two for trees and one for coral.

Keywords: path-planning, artificial intelligence, image based weighting, dendrites, natural phenomena, procedural modeling

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

It is quite common to see objects that may be familiar or fantastical in computer games and computer animated films. Creating a sufficient number of detailed models for a virtual environment is a very laborious task. Although various procedural techniques are available, new techniques are highly sought after with the intention of easing the burden on digital artists. The use of artificial intelligence techniques in computer graphics is quite a recent research area. Computer graphics practitioners are realizing the rich possibilities of implementing such techniques in order to solve difficult problems [13].

In this work, we investigate the use of path planning which is a well-studied area in the field of artificial intelligence. The research presented here is heavily based on the work of Xu and Mould ([22]) who introduced path planning as a means of procedurally modeling natural phenomena. The technique involves finding the least-cost paths through a randomly weighted graph from a source to a number of destination points. Here, we have attempted to implement some extensions and improvements.

Our main contribution is a new and controlled way of automatically assigning weights to a graph through the use of images. Every edge in the graph is assigned a cost that corresponds to a pixel in the image. A minor contribution is a method of providing more control in the selection of the goal destinations used by the path planner to generate branching structures from a single source. Our results show that these improvements considerably aid in informing the behaviour of the path planning algorithm and the general shape of the model. The technique was applied to model trees and coral.

The remainder of the document is organized into five chapters. Chapter 2 provides an overview of the relevant works with respect to natural phenomena modeling and an overview of path planning in the context of artificial intelligence. The application of path planning for this problem is described in detail in Chapter 3 after which, Chapter 4 presents implementation specifics. In Chapter 5, we show and our results in the form of reference images and renderings of the generated models and discuss the advantages and disadvantages of this method. Finally, we conclude with a few recommendations for future work.

Chapter 2

Background

This chapter gives some background on the relevant areas related to this project. First we present the established methods for procedurally modelling natural phenomena such as L-Systems and DLA. This is followed by a detailed look at the relevant literature for the method that we have focused on in this project which is path finding. Finally, we provide a brief overview of how artificial intelligence techniques have developed in order to tackle the problem of path finding.

2.1 Established Methods for Modelling Natural Phenomena

We start by presenting some common methods for procedurally modeling of natural phenomena with a dendritic nature. We say procedurally because as mentioned in [7], this provides for a flexible and parametric approach to building structures instead of storing vast numbers of low level primitives. Dendrites, as mentioned in [21], are characterized by individual branching and erratic winding travels as can be found in natural objects such as trees, coral and lightning. They cite two common approaches to obtaining such structures; L-Systems and Diffusion Limited Aggregation (DLA). While our solution does not make direct use of these, it is worth mentioning their underlying logic as part of a snapshot through the literature.

Figure 2.1: Figures of dendritic structures in nature

In 1968, Aristid Lindenmayer introduced L-Systems as a formalism for simulating the development of multicellular organisms. Further work by Lindenmayer and Prusinkiewicz in [15] presents replacement grammar L-Systems, where a combination of grammar and rules are used to describe how 'tokens' are subsequently changed to sequences of tokens. As part of a modeling system, these can be used to describe geometric shapes such as dendrites by mapping each token in the string to an action or movement such as forward, backward, left or right. L-systems resemble fractals in their self-similarity aspect although not being as well mathematically defined [7].

Deussen et al [5] built a system for automatically generating realistic natural scenes and ecosystems, making use of such L-Systems for the modeling. The framework provided a very powerful means for generating trees and plants, but lacked in artistic control. Talton et al present an approach that deals with this problem, providing some measure of artistic control [19]. There has also been further work on augmenting L-Systems with extensions to create stochastic L-Systems, open L-Systems [12] and environmentally-sensitive L-systems[14]. The main advantage provided by open L-Systems is that the grammar provides for two-way communication between the system and the environment, thereby allowing the system to consider information about its surroundings.

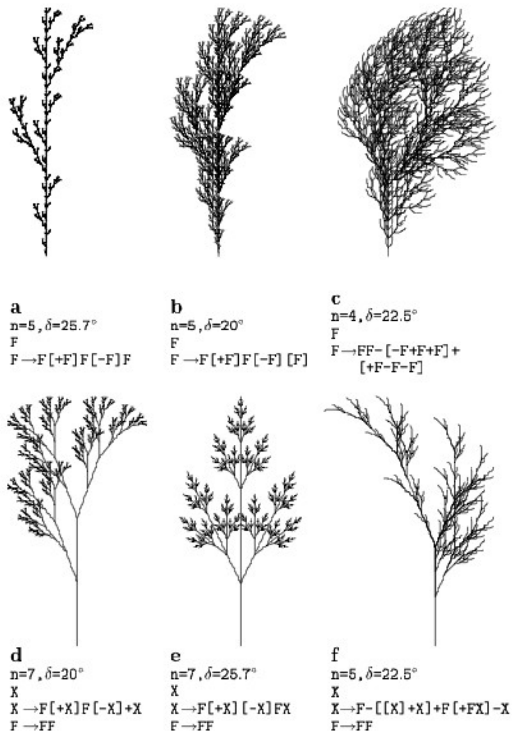


Figure 2.2: L-System examples [15]

L-systems have also been taken further to generate virtual creatures that evolve over time. Hornby and Pollack in [9] present a method where such systems are used as the encoding structure for an evolutionary algorithm, resulting in creatures with a more natural look. The interest to artificial intelligence researchers here is that the creature is built progressively through L-System development and guided by a fitness function that evaluates how well the creature moves and reconfigures the L-System accordingly.

The next modeling method, Diffusion-limited aggregation (DLA) refers to the gradual process whereby matter clusters to form dust, soot and other phenomena of a dendritic nature. The algorithm for DLA was first pioneered by Witten and Sandler in [20] who have extensively studied the growth, gelation and structure factor of such phenomena. The model is governed by very simple rules akin to those seen in a number of natural objects. It starts off with a seed particle at the origin of a lattice. As part of an iterative process, other particles are allowed to undergo a random walk due to Brownian motion from a distance until they finally reach a point that is adjacent to another one which is occupied [20]. This method yields large attractive structures similar to Figure 2.3.

The DLA algorithm has been used in graphics because of the rich set of phenomena that can be modeled using this method such as frost as presented by Kim and Lin in [10] or lightning presented by Reed and Wyvill in [16]. As with other physically based simulations, the major drawback of this approach is the computational time required to get highly intricate models. Having said that, despite the wide applicability in chemistry and botany, there still seems to be limited use for DLA in graphics, prompting graphics practitioners to seek alternative procedural methods to represent these complex models [21].

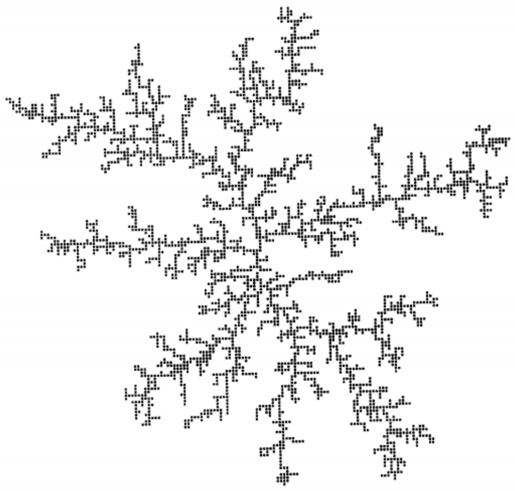


Figure 2.3: DLA structure [20]

Another avenue of exploration into the modeling of dendritic structures was through the use of artifical intelligence (AI) techniques in the form of path planning. Path planning attempts to model such structures by applying AI algorithms such as Djikstra over weighted lattices. This technique has been shown to yield good results, with trees and lightning as examples shown in [22]. Xu and Mould present a number of works ([21, 22, 23]) relating to this area, which serve as the main references for our work. In this regard, further detail on path planning is presented in the next section.

2.2 Path Planning for Modeling Natural Phenomena

The use of path planning techniques to model dendritic natural phenomena was first presented by Xu and Mould in [21]. Their method generates a regular lattice with randomly weighted edges. Consequently, it is possible to generate paths from a single starting point to a number of destinations using an algorithm such as Dijkstra [22]. The resulting dendrites (Figure 2.4) are then converted to geometry (Figure 2.5) for which the authors have provided a number of approaches. The simplest way is to directly convert the lines by placing spheres at each pixel. However, in a previous work they also propose converting the nodes remaining in the acyclic graph into an iso-surface using the Marching Cubes algorithm [21].

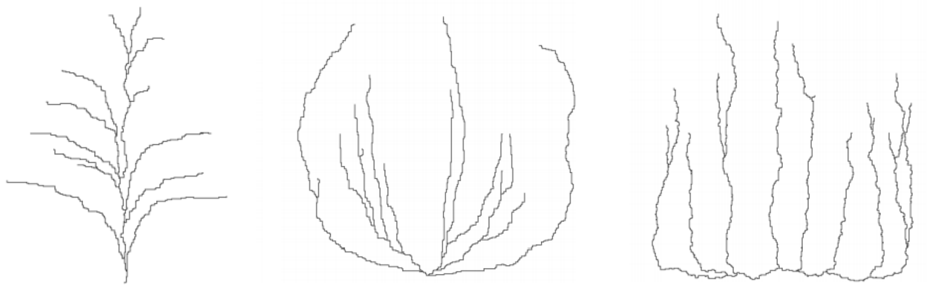


Figure 2.4: Dendrites produced by Xu and Mould [22]

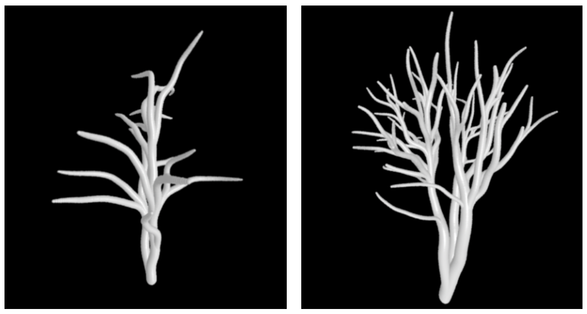


Figure 2.5: Models created by Xu and Mould using path planning [22]

By employing an iterative approach, the authors succeeded in producing models highly resembling coral or other types of marine life. Moreover, they boast very attractive computational times in contrast to the costly DLA simulation required to get similar results. Their algorithm is shown in Algorithm 2.1.

Algorithm 2.1 Path Planning [22]

```
1. Create a regular square lattice of nodes: 4-connected in 2D, 6-connected in 3D
2. Choose edge weights for the graph
3. Choose a node to be the root of the structure
4. Apply Dijkstra's algorithm to find path costs to all nodes from the root
5. Choose path endpoints
6. Use a greedy algorithm to backtrack from the endpoints to the root, giving the paths
7. Render the paths
```

They also investigate a more sophisticated way of distributing the weights in the graph in order to be able to control the shape of the dendrite.

They recommend that every edge should be assigned a weight value of e which would be

$$e = R^{\alpha}$$

where R is a uniformly drawn random positive value and α is a parameter controlling the amount of path variation. The authors proved that the larger theα, the greater the disparity between the cheapest and most expensive edges. Therefore, there is greater incentive for the path planning algorithm to seek paths that are made up of a lot of cheap edges rather than considerably few edges which may be more expensive. This improvement contributed towards better aesthetics, however it was critiqued that the structures that were generated from a single graph were still overly simple.

In their most recent publication ([23]) the model has been extended so that it becomes hierarchical. This means that the paths are generated in a recursive fashion mimicking fractal behaviour. This was implemented by introducing the concept of a 'lifespan' which is synonymous to the number of levels in the structure.

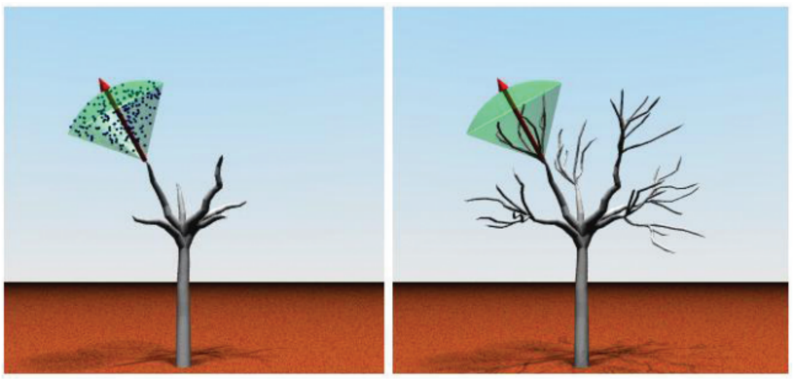


Figure 2.6: Dendrite built iteratively [23]

Furthermore, Xu and Mould have investigated the task of the lattice building, which according to them, is the step that influences the general shape of the model. They have come up with a generic way of building irregular lattices that can be shaped either by a composite of primitive geometries or by processing a sketched curve to determine the dendrite outline boundary. This requires additional computational effort although it was a small trade-off considering that the results they were able to get look much more intricate and detailed.

To conclude, path finding techniques traditionally used in artificial intelligence are plausibly a good and efficient approach to create almost life like dendritic models without having much prior knowledge about the morphology. Xu and Mould have shown that this technique is more versatile than other traditional methods. Moreover, there are so many components in this method that is remains rich ground for further exploration and extension.

2.3 Artificial Intelligence Techniques for Path Planning

Path planning is a problem much studied in the context of artificial intelligence, with many applications such as robotics, navigation systems and computer games. As of currently, it is still the most used method to determine how a predefined destination can be reached from a particular configuration [22]. In general, the algorithm is designed to minimize a particular factor which may or may not be distance [3]. In fact, this could take the form of danger or perhaps fuel consumption. This type of planning is relatively simple for us humans [3]; we do not realize that in reality there are many facets of the problem and representing them formally for a robot is particularly challenging. Hence, path planning has been a very important topic of research in the realm of artificial intelligence. Several algorithms have been developed to tackle this problem and many of them are optimal and so can be done in real time [1].

As Algfoor et al explain, path finding consists of two major aspects: the construction of the navigation environment and the path finding algorithm. The most convenient method of representing the navigation environment is by defining a graph, which may also be termed as a grid or lattice

$$G = (V, E)$$

where V is the set of vertices that maps to a coordinate space and E is the set of edges connecting the vertices that are in the line-of-sight of one another. Normally a grid is described as either being regular or irregular with regards to its tessellation (Figure 2.7). Since both kinds each have their unique set of desirable properties, the decision of whether to use one or the other hugely depends on the problem at hand and the behavior being sought which brings about the mention of the path finding algorithm itself [1].

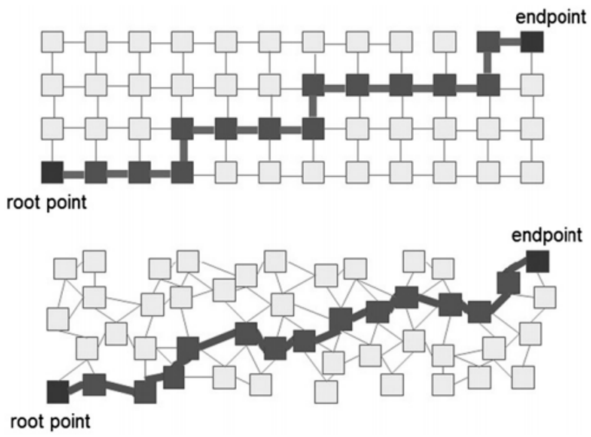


Figure 2.7: Regular and irregular tessellations [23]

All path finding processes aim to return the most optimal path in an efficient manner and a variety of techniques are available [3]. The most popular group of path planning algorithms are based on Dijkstra's algorithm ([6]), which was developed by Edsger Dijkstra in 1959 and is the path planning algorithm of choice for this project. Taking into consideration a grid as defined previously, the sole objective of the algorithm is to return an ordered list of vertices which constitute the least-cost path. This works under the premise that every traversal possible between two vertices incurs a cost of a positive value which directly depends on the problem at hand. This requires the iterative maintenance of the list of vertices which need to be evaluated, each time selecting the one with the lowest cost and calculating the score for its neighbours [3]. The evaluation process is repeated until either the goal is reached or no more vertices remain. The latter case is an indication that a path with a finite cost could not be found. If the goal is reached however, then the path can be easily traced by backtracking from the goal and repeatedly jumping to the most inexpensive point. In this document we have provided the pseudo code for this algorithm in Chapter 3.

In just under ten years since Dijkstra was introduced, the A* algorithm emerged and it is considered to be the founder algorithm for modern path planning [3]. Both algorithms are in actual fact similar with the exception of the additional heuristic element whereby the effort required to reach the goal from a vertex is taken into account alongside the cost. According to Bell, under most circumstances A* performs better than its father algorithm. More importantly, the heuristic element is completely configurable so that various types of behaviour can be supported making the algorithm suitable for a wider range of tasks [3]. Even though in the context of this project the heuristic element is set as a constant, the assignment of cost is realized to be a very important part of the project which is investigated in order to have more control over the behavior of the path planning algorithm.

Chapter 3

Path Planning for Natural Phenomena

The objective of this project is to make use of simple path planning techniques to create geometric models that are characterized by the dendritic appearance. These structures are akin to various types of natural phenomena that exist for example plants, trees, coral and lightning. For the purpose of this project, trees were used as the main subject matter for emulation and discussion. As part of this investigation into path planning for dendrites, we attempt to make certain improvements over Xu and Mould's method, particularly in the ability for the user to influence the resulting dendrite through image based weighting and improved endpoint selection.

In this chapter we present the different parts of the method and concepts as applied to our pipeline. While the pipeline itself is described in Chapter 4, at this stage it helps the reader to know that it is made up of two parts; the path generation module built as an application based on the approach first introduced by Xu and Mould in [21]; and a geometry generation part that converts the exported paths into a detailed model. Most of the content in this chapter focuses on the former, showing the graph generation, weighting and path generation processes. In particular, this project investigates the distribution of the cost in order to inform the behaviour of the path planning algorithm and in turn the general shape of the model. Alongside the original formula by Xu and Mould, we also present an alternative way of assigning weights for 2D graphs through user created images. This is in contrast to their approach of using user sketches to determine the outline of the resulting dendrite. In our method for weight distribution, every vertex in the lattice corresponds to a pixel in the input image and the value of the colour pixel is assigned as the cost. In the cases that we present, the path planning algorithm is directly encouraged to seek paths that correspond to the darker regions within the image. Furthermore, we explore the use of irregular lattices as well as a simpe method to improve endpoint selection such that the user may shape a dendrite by selecting only endpoints from desired regions.

The last sub-section touches on the aspect of modeling after a dendrite has been generated as a set of paths, including how the paths can be refined individually with some post-processing using an acceptable amount of computation and resources. In [22], the authors proposed an iterative refinement approach by constructing a new higher resolution lattice in the region of interest. Our take on this problem is to use procedural noise in order to enhance the erratic aesthetic of the model, thereby saving on computation and memory.

3.1 Graph Generation

In the graph generation step, our application allows a user to choose between a number of tesselation option for lattices. Figure 3.1 shows two of the lattice types provided; square and hexagon. The 2D square lattice is generated using a basic algorithm which uses an outer and an inner loop to set values for the x and y position for each node based on a input number of steps. To explain, a step value of 2 would result in a four by four lattice, and so on. The order in which nodes are generated can be seen in the node indices superimposed on the lattices shown in Figure 3.1, built with a step value of 2. The hexagon lattice was implemented to address one of the drawbacks of the square lattice in that such lattices produce no slanting edges, which may be undesirable for cases such as modeling of lightning. Hexagon lattices produce more available edges from each node, resulting in more slanting lines available.

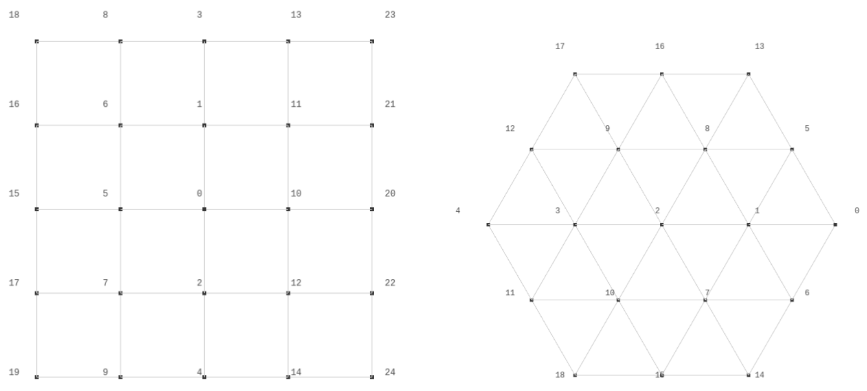


Figure 3.1: 2D lattice types - square (left) and hexagon (right)

One of the limitations mentioned by Xu and Mould in [22] is that for regular lattices, the resulting path may look more uniform than desired due to the right angle path changes. Their own improvement to the regular lattice was to use point clouds instead of a procedural lattice generation method. We have attempted a different approach whereby a regular lattice built using the previously mentioned method would be skewed such that each uniform point would be moved by a random margin less than the edge length in the x and y axes. This greatly reduces the chance of having too many right angle or straight long path sections, at very little cost as the skewing can be done in one pass after the initial lattice generation. This can be seen in the side by side comparison in Figure 3.2.

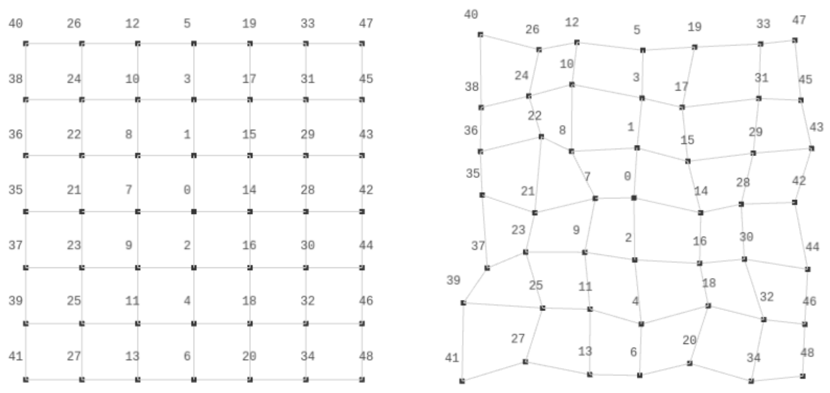


Figure 3.2: Regular (left) vs irregular (right) lattice

A natural extension of the simple 2D square lattice is the use of the z-axis to create a 3D cube shaped lattice. This allows paths to be generated in 3D space which produces more realistic models later in the pipeline. An example of such a 3D lattice is shown in Figure 3.3.

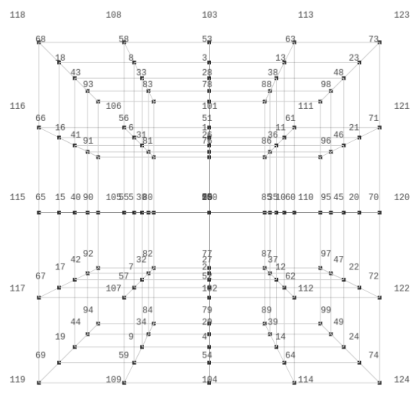


Figure 3.3: 3D Square lattice

3.2 Weight Distribution

A key step in the path planning process is the application of weights to the edges in the lattice. These are ultimately required to be able to apply an algorithm to find a path through the network, such as the Dijkstra algorithm (mentioned in the next section). These weights should be synonymous to the effort required to traverse that edge. Having said that, in the problem we are trying to solve, we would prefer that the path planner does not choose the most direct route to the end point, thereby resulting in a more erratic path resembling the dendritic structure we would like to emulate.

We have initially implemented the original costing function put forward by Xu and Mould in [22] as a benchmark. This uses the following formula to assign random weights to each edge. They recommend that every edge should be assigned a weight value of e which would be

$$e = R^{\alpha} \tag{3.1}$$

where R is a uniformly drawn random positive value and α is a parameter controlling the amount of path variation. One of the contributions of this project is an alternative weight distribution function, which makes use of an image to assign weights to the edges. To be clear, this differs from Xu and Mould's sketch approach, as that was used to inform the outline of the dendrite rather than the internal weights. The basic idea here is that this would enable a better measure of control over the resulting dendrite, moving away from randomness.

For this to be possible, the image provided is scaled to the aspect ratio of the lattice, such that each edge on the lattice corresponds to a pixel in the image. By mapping the colour value of the individual pixel to a grayscale value, a weight can be assigned to the respective edge. Figure 3.4 shows a simple example of this method and how the resulting weights are generated. One can notice that edges on the white areas of the image are assigned the maximum weight value of 1, indicating a high cost to the path planner. The other weights can be seen to vary based on the darkness of the image colour regions, thus making the darker regions of the image more desireable.

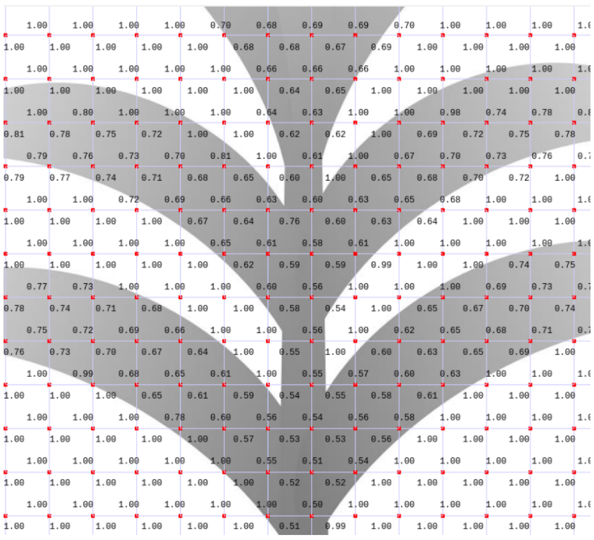


Figure 3.4: Weighted lattice with superimposed image

The intention of this method is not to provide an exact rendering of the desired dendrite, but rather to provide an easy and generic way of controlling the behaviour of the path planner without the need of adding complex heuristics to the path planner, or edit weights manually. Furthermore the user has the ability to clearly define regions through which the path planner will not be able to pass. This can be used to create dendrites that must adhere to some outline shape, similar to Xu and Mould's sketch approach in [23]. An example of such an image, and the resulting dendrite is shown in Figure 3.5.

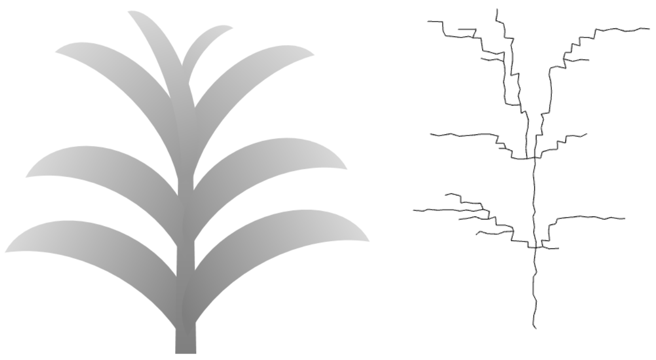


Figure 3.5: Sample image (left) and resulting dendrite (right)

3.3 Path Generation

Once the edges have weights, the next step is to choose a root and subsequently calculate the distances from all nodes in the lattice to the root. By this, we mean the sum of the shortest path of edge weights leading from the respective node to the root. This is done using Dijkstra's algorithm (Algorithm 3.1) and results in an ordered list of nodes with corresponding distances to the root.

Algorithm 3.1 Dijkstra's Algorithm [2]

```
1: function Dijkstra(Graph, source):
2:     for each vertex v in Graph: // Initialization
3:         dist[v] := infinity // initial distance from source to vertex v is set to infinite
4:         previous[v] := undefined // Previous node in optimal path from source
5:     dist[source] := 0 // Distance from source to source
6:     Q := the set of all nodes in Graph // all nodes in the graph are unoptimized - thus are in Q
7:     while Q is not empty: // main loop
8:         u := node in Q with smallest dist[ ]
9:         remove u from Q
10:        for each neighbor v of u: // where v has not yet been removed from Q.
11:            alt := dist[u] + dist_between(u, v)
12:            if alt < dist[v] // Relax (u,v)
13:                dist[v] := alt
14:                previous[v] := u
15:    return previous[ ]
```

The distances calculated are therefore very much dependent on the chosen root, as can be seen in Figure 3.6 showing, for the same set of weights, the resulting distances on the nodes for two different roots chosen. The path planning application allows the user to view these similarly to the node indices and edge weights so as to fully understand the network and pick the root as desired.

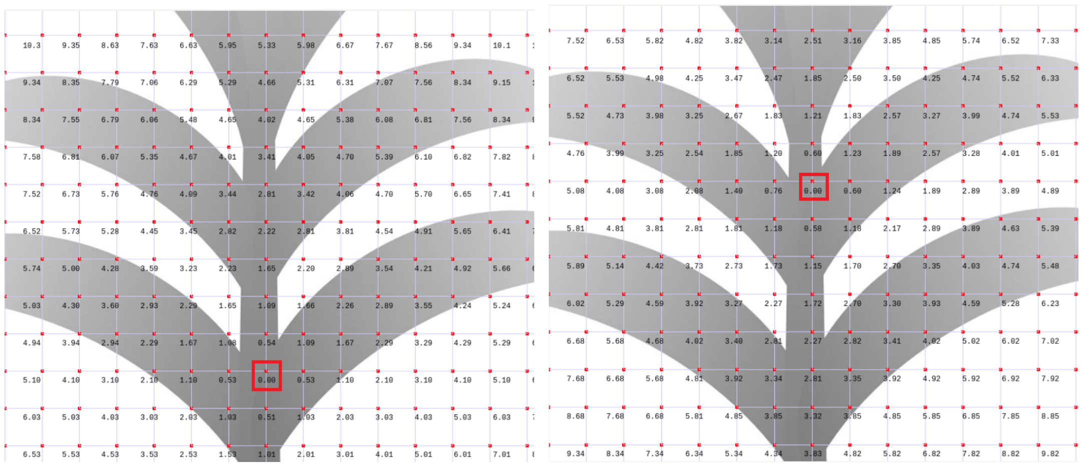


Figure 3.6: Comparison of node distances generated for different roots

The last step in path generation is choosing the desired endpoints and for each one, using a greedy algorithm to traverse from the endpoint to the root through the next cheapest node. In order to avoid cycles, a heuristic was added such that a node cannot be used more than once in a single path.

One of the challenges encountered at this stage was getting enough density in the lattice to result in dense erratic paths. As an example, using a step of 5, yields Figure 3.7, where it can be seen that the overall number of edges is low. One way to treat this would be to make the lattice denser by increasing the step, resulting in more edges from the endpoints to the root, however this would incur a high computational cost of retaining all the unused edges in memory. Xu and Mould, in [22], suggest an optimization for this in the refinement of the path through sub lattices only on nodes in the path. In this way, the extra computational cost and memory would only be expended on nodes of interest lying on the already established path. By creating a lattice on each, the straight line to the next node can be broken down into a path of its own, making it more erratic by a factor of the step used. We attempted to replicate this method, with limited success and was ultimately not taken forward for path refinement in our pipeline. Instead of this approach, we opted to perform post processing on the dendrite at the coversion to geomerty stage, described in the next section.

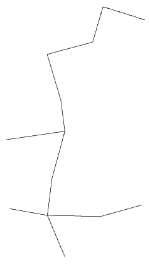


Figure 3.7: Low density path using a lattice of step value 5

3.4 Converting to Geometry

The conversion of the paths to geometry is the final step of this method. This was done with the aid of Houdini™ by SideFX® which is an industry standard modeling and visual effects tool. After importing, each path in the dendrite is smoothened into a NURBS curve. At this stage an artist may choose to instigate procedural noise to flow along the curve as a means of refining the skeleton of the model. Each path is then individually skinned by using low polygon circles as the cross-section profile and sweeping them along the curved path. The resulting effect is that each edge of the path is covered by a tube. To allow for various kinds of organic shapes one would need to control the variation of the radii in the tubes. As an example, tapered strokes in a model of a tree are highly sought after and can be achieved with little effort using an adaption of Xu and Mould's [22]formula:

$$r_i = r_s \times \left(1 - (p_i/p_k)^{\beta}\right) + r_f \times (p_i/p_k)^{\beta} \tag{3.2}$$

where r_i is the radius of the circle profile at a current point i in the path, $r_s$and r_f are the radius at the start and finish of the curve respectively, p_i is the index of the current point, p_k is the index of the last point and finally, β is the parameter that governs the amount of tapering allowed. The ability of the above equation to produce tapering strokes is illustrated in the next chapter.

Chapter 4

Implementation

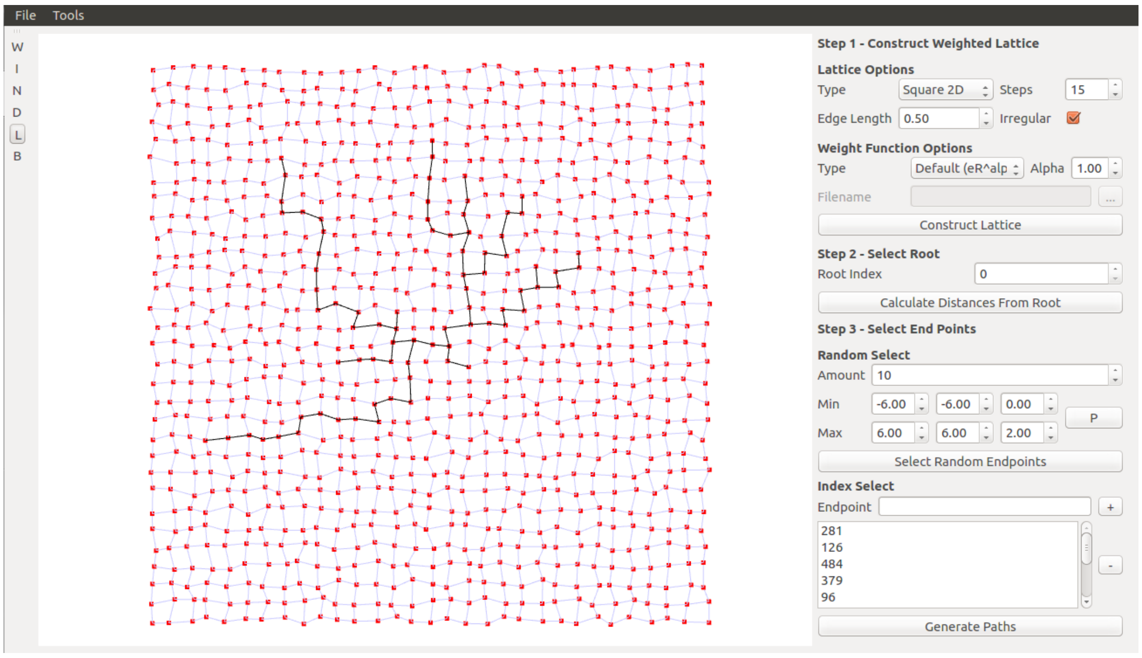
In this chapter, we present the implementation specific details relating to the path planning application and the Houdini$^{\text{TM}}$ scene. The project makes use of a pipeline made up of two distinct modules. The first module, explained in Section 4.1, is a path generation application built in C++ and OpenGL (using the NGL library [11]) and handles the process through which paths are generated over a lattice, based on the same general algorithm as that used by Xu and Mould in Algorithm 2.1. The second module, shown in Section 4.2, consists of using an industry third party tool; Houdini$^{\text{TM}}$, and handles the importing of the path data from XML via a Python script, followed by the conversion to geometry.

4.1 Path Planning Application

In this subsection we take a look at the path planning appliction and the features it provides. We also look at the architecture details in terms of classes and interaction between them.

4.1.1 User Interface

A user interface is provided (see Figure 4.1) to allow the user to progressively build dendrite paths by setting parameters over a number of steps. On the right of the viewport, the parameters for the three main steps in the process are shown; weighted lattice construction, root selection and endpoint selection. Each step has a number of action buttons, explained in Table 4.1. For each step, Table 4.2 gives details on each of the parameters affecting the aforementioned actions, and which should be read in the context of material presented in the previous two chapters. Finally, on the left of the viewport, a view toggle toolbar is provided for the user to show and hide different elements in the viewport; lattice [L] , node indices [N] , edge weights [W], node distances [D], image background [I], bounding box [B]. A number of views of the viewport with different toggle configurations is shown in Figure 4.2.



Step 1 - Construct Weighted Lattice

Lattice Options

Type: Square 2D   Steps: 15

Edge Length: 0.50   Irregular: ✓

Weight Function Options

Type: Default (eR^alp   Alpha: 1.00

Filename

Step 2 - Select Root

Root Index: 0

Step 3 - Select End Points

Random Select

Amount: 10

Min: -6.00   -6.00   0.00

Max: 6.00   6.00   2.00

Index Select

Endpoint

281
126
484
379
96

Figure 4.1: Path planning application user inteface

Step	Action button	Use
1	**Construct lattice**	Invokes the construction of the weighted lattice based on selected parameters. Any paths previously generated in the viewport are removed.
2	**Calculate distances from root**	Runs Dijkstra path planning algorithm based on selected root. Generated distances may be viewed by toggling 'D' in view toggle toolbar. May be run with already existing paths in place if building dendrite iteratively.
3	**Select random endpoints**	Selects a number of random endpoints based on the chosen bounding box and amount. These are added to endpoint list view. User may view bounding box by toggling 'B' in view toggle toolbar.
3	**Generate paths**	Runs through endpoint list and uses greedy algorithm to trace a path from each endpoint to root. Any path endpoint selected in listbar is shown in red.

Table 4.1: Action button details

Step	Parameter	Allowed values	Effect on dendrite
1	**Lattice Type**	Square2D, Hexagon2D, Cube	Determines the construction method of the lattice.
1	**Steps**	Integer	Determines the size of the lattice.
1	**Edge Length**	Real	World space distance between one node and another. Inconsequential to dendrite shape, but useful for managing viewport.
1	**Irregular**	Ticked/Unticked	If ticked, lattice is distored to produce irregular edges.
1	**Weight Function Type**	Default, Image Based	Determines how weights are assigned. For 'Default', alpha value is required to calculate weights randomly. For 'Image based', an image URL.
1	**Alpha**	Real value	When using 'Default' weighting, affects the disparity between cheapest and most expensive edges.
1	**Filename**	String	When using 'Image based' weighting, specifies the image file to be used.
2	**Root Index**	Integer (from set of Indices)	Determines which node is selected as the root for the Dijkstra function to calculate distances.
3	**Random Select Amount**	Integer	Number of random endpoints to select from within the bounding box.
3	**Min and Max x,y,z**	Real	Determines the bounding edges of the endpoint selection bounding box.
3	**Endpoint**	Real	Position of desired endpoint, specified manually.

Table 4.2: Parameter details

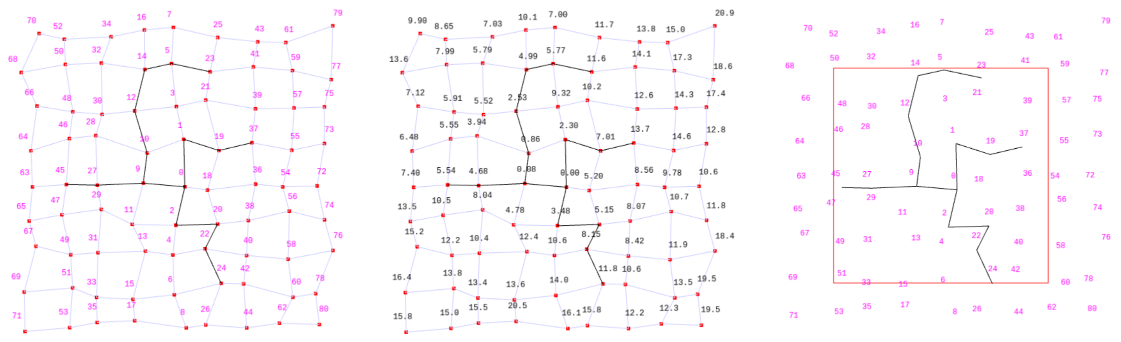
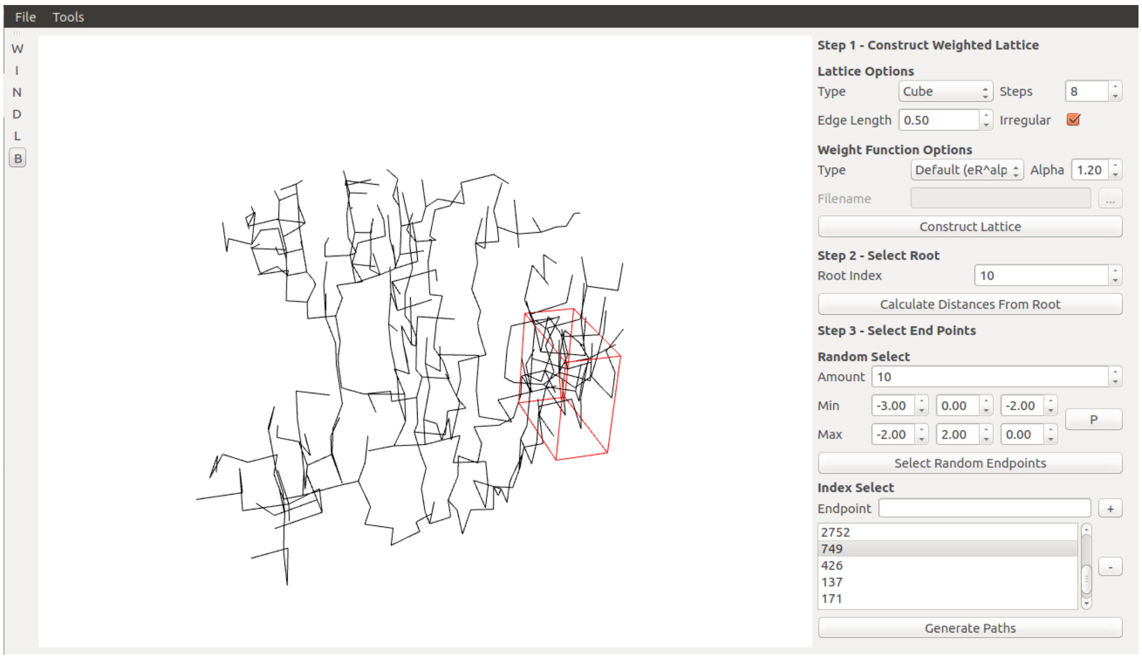


Figure 4.2: View toggles examples. Lattice and node indices (left). Lattice and distances (middle), Node indices and bounding box (right)

The general use of the application therefore involves primarily deciding the lattice structure based on the dendrite requirements, and constructing the lattice. Once this is done, the user may use the view toggle toolbar to select a root and invoke the calculation of the distances. The last step may be approached iteratively by moving the bounding box for the random endpoint generator and progressively generating endpoints in the desired region(s) (Figure 4.3). Each path is added to a list characterized by the endpoint index, where if selected, it is displayed in red to be identifyable. The user may then choose to use the remove button to remove any undesireable paths. The user may also choose to add endpoint indices to the list manually if this is required. Finally, from the top menu, the user may choose the 'Export' function to export all the paths in the current viewport to an XML file.



Step 1 - Construct Weighted Lattice

Lattice Options

Type: Cube — Steps: 8

Edge Length: 0.50 — Irregular: ✓

Weight Function Options

Type: Default (eR^alp — Alpha: 1.20

Filename:

Step 2 - Select Root

Root Index: 10

Step 3 - Select End Points

Random Select

Amount: 10

Min: -3.00 0.00 -2.00

Max: -2.00 2.00 0.00

Index Select

Endpoint:

2752
749
426
137
171

Figure 4.3: Restricting endpoint selection using the bounding box

4.1.2 Architecture

The architecture of this module consists of a number of high level classes to represent the real world problem of storing lattice information, performing functions over nodes and edges, and representing a dendrite. A high level UML diagram is shown in Figure 4.4, while the corresponding class summaries are summarized in the next three subsections. These should provide enough detail for anyone wishing to pursue development in this area to get a good idea of the backend structures.

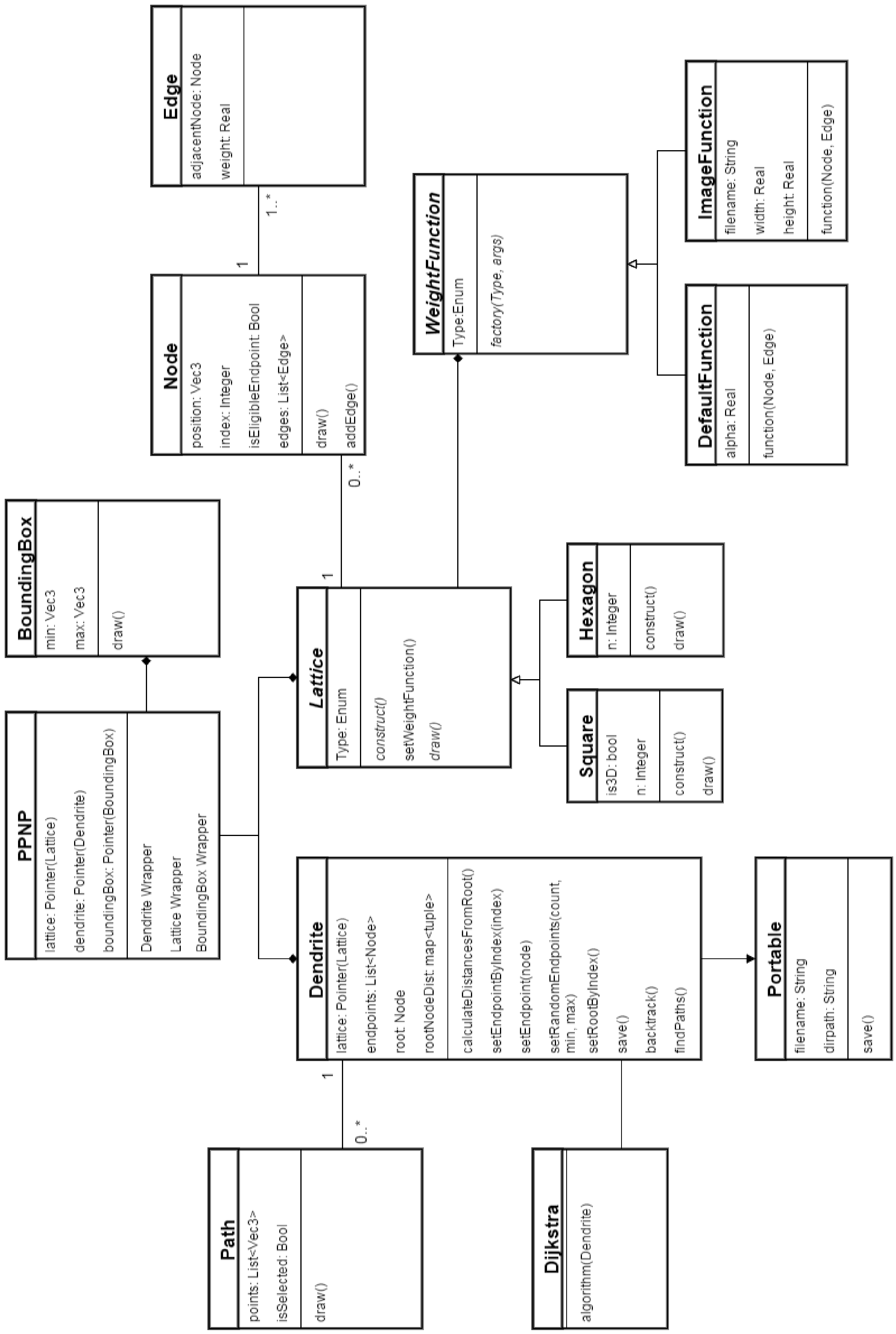


Figure 4.4: UML diagram

4.1.3 NGLScene and PPNP

As with other OpenGL applications using the NGL library, the **NGLScene** governs the high level flow of the application, where graphics resources are initialized, user interface signals are caught, and the screen updated through the *paintGL()* method. In terms of our application, it suffices to say that the main interaction point between the **NGLScene** and our architecture is through the **PPNP** class, which acts as a wrapper class to call all the necessary functions on the main objects initialized in the application; the dendrite, the lattice and the endpoint selection bounding box. The **BoundingBox** class is a very simple class that is used solely for user interaction purposes, allowing the **NGLScene** to maintain and draw the box from which endpoints can be chosen.

4.1.4 Lattice Class and Relative Objects

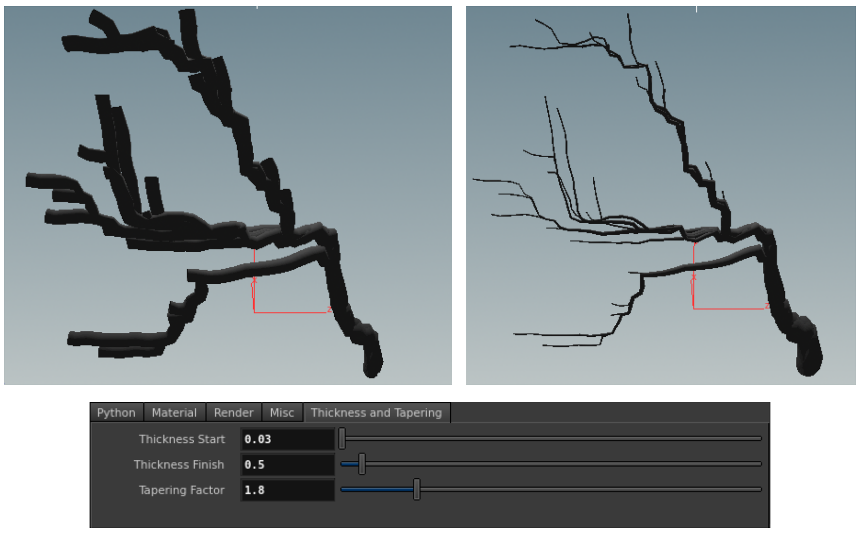
A number of classes in the architecture handle lattice related functions. An abstract class **Lattice** is inherited by two other classes; **Square** and **Hexagon**, which implement methods to construct and draw the lattice respectively. Any additional tessallation formats that need to be implemented can be created as classes that inherit from **Lattice**, making the system extendible. This class also makes use of the **Node** class to represent each node in the lattice. This in turn uses the **Edge** struct to represent an edge between two nodes. Each node maintains a list of edges which join to that node, while each edge maintains the node to which it is connected. This therefore means that between each two nodes, there are also two edges, making the edge structure directional. Finally the lattice makes use of another abstract class **WeightFunction**, which is inherited by the two implemented function types **DefaultFunction** and **ImageFunction**, each implementing an algorithm to apply a weight to an edge.

4.1.5 Dendrite Class and Relative Objects

On the other side of our UML diagram are a number of classes related to the dendrite itself. Primarily, the **Dendrite** class provides a lot of functionality related to working with the lattice to generate paths and maintain them. It maintains a pointer to the currently active lattice, as well as the root and list of endpoints provided by the user. By calling the **Dikjstra** class, it populates a map of tuples *rootNodeDist*, which contains the respective distances from each node to the selected root. Once the path generation option is invoked for a set of endpoints, the *backtrack()* method is used to generate a **Path** object for each endpoint, containing a list of points in world space. For display purposes, each path can be marked *isSelected*, so that the *draw()* function uses a different colour for the respective path. To be able to use these paths outside the application, a **Portable** class is used to save these paths in an XML by providing a filename and directory.

4.2 Modeling in Houdini$^{\text{TM}}$

The dendrite data generated in the previous steps is exported into an XML file (a sample can be found in Appendix B) which can be parsed by the software using an in-built operator that interprets a Python script (available in Appendix A). It should be noted that while Houdini$^{\text{TM}}$ was used as the modeling tool in this case, the XML provided is structured so that it may be used by other tools in a similar way. The script makes use of the Houdini Object Model API ([17]) to invoke the appropriate modules and packages them in a simple model that can be used by an artist. The raw model that is generated is shown in Figure 4.5 (top left). We provide a set of parameters for thickness and tapering, which can be seen in Figure 4.5 (bottom). This allows an artist to easily change the thickness of the paths and tapering factor based on Equation 3.2. Figure 4.5 (top right) shows an example of how varying these parameters enhances the model.



Thickness Start 0.03
Thickness Finish 0.5
Tapering Factor 1.8

Figure 4.5: Modeling progression in Houdini$^{\text{TM}}$. Default model (top left). Tapered model (top right). Thickness and tapering parameters in Houdini$^{\text{TM}}$ (bottom).

Chapter 5

Results and Discussion

In this chapter we present the results of experiments run as part of this project, their implications and a general discussion on the outcomes of these experiments. We first perform a number of comparisons related to the tessellation of different lattices using different input parameters and assess how this affects the dendrites produced. We then assess the performance of the image based weighting function and the resulting dendrites. The last part of this chapter then looks at results from the last part of the pipeline, and what types of models are possible when exporting dendrites from the path generation application into Houdini$^{\text{TM}}$ using the developed Python script.

5.1 Tessellation Parameters

Looking at the tessellation parameters of the lattices and resulting dendrites allows us to make certain assessments about the effect of said parameters; type of lattice structure (2D square, 2D hexagon and 3D cube), regularity - regular or irregular (determining the angle variation between nodes and edges), and step size - determining the size and density of the lattice.

For the purpose of this section, the input image in Figure 3.5 was used as the weighting function. Figure 5.1 shows a comparison between two dendrites with varying lattice types. In both cases, a random set of 25 endpoints was chosen, keeping the step size constant and a using a regular lattice in both cases. One can observe that the type has a marked effect on the resulting dendrite. While the square lattice provides a dendrite more akin to a tree structure corresponding to the input image, the hexagonal lattice produces a dendrite which looks less like the structure of a tree.

Figure 5.2 uses the same dendrite, but instead looks at the effect of regularity of the lattice. In this case both dendrites use a square lattice and same lattice size, but one can see a clear effect of the irregularity of the dendrite on the right. The irregularity almost guarantees the absence of straight lines across multiple nodes, and produces a dendrite with branches that resemble what one would expect to see in the natural world, such as in trees.

The last experiment in this section, shown in Figure 5.3, looks at the effects of varying the step size and therefore having a denser lattice to work with. In theory, a denser lattice should provide more detail, and this clearly comes through from left to right as a very simple dendrite gets more detailed as the step size is increased. There is however an upper bound to the step size depending on the dendrite required. While each dendrite resembles the source image more than its counterpart on the left, one can also see how the right-most dendrite produces multiple 'trunks' as multiple paths of fairly equal cost are used by different endpoints. Depending on the dendrite required, one may therefore decide to keep a lower step size to maintain more common branches.

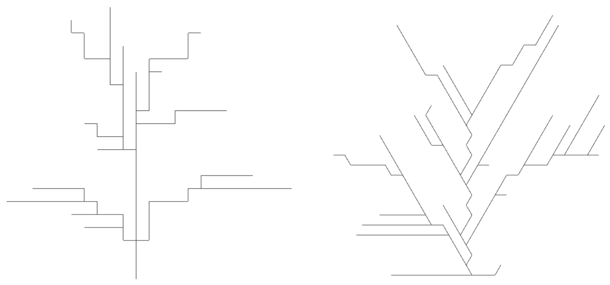


Figure 5.1: Comparison of dendrites with square lattice (left) and hexagon lattice (right)

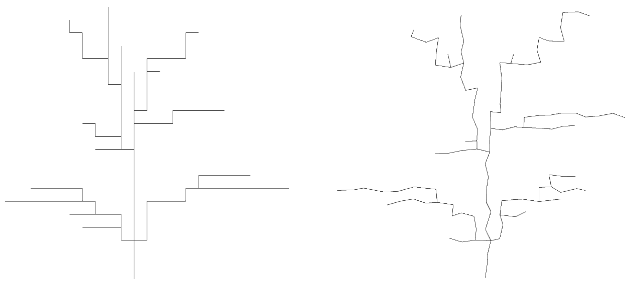


Figure 5.2: Comparison of dendrites with regular lattice (left) and irregular lattice (right)

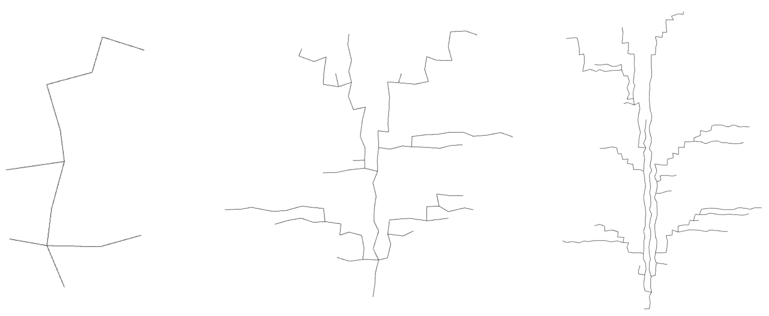


Figure 5.3: Comparison of dendrites with step 5 (left), step 15 (middle), step 35 (right) lattice sizes

5.2 Image based weighting

The next, perhaps more interesting assessment in this chapter is on the image based weighting function introduced in this project. As an initial evaluation, we attempted to reproduce some of the dendrites presented in [22] so as to show that the implemented image weighting function is able to produce a dendrite similar to an input image. In each of the cases shown in Figure 5.4, the middle image is an input gradient image corresponding to the dendrite by Xu and Mould its left. On the right we show the resulting dendrite using our image based weighting function. It can be noted that in each case, the resulting dendrite conforms very well to the input image provided, and therefore manages to produce dendrites similar to Xu and Mould's without manually changing any weights.

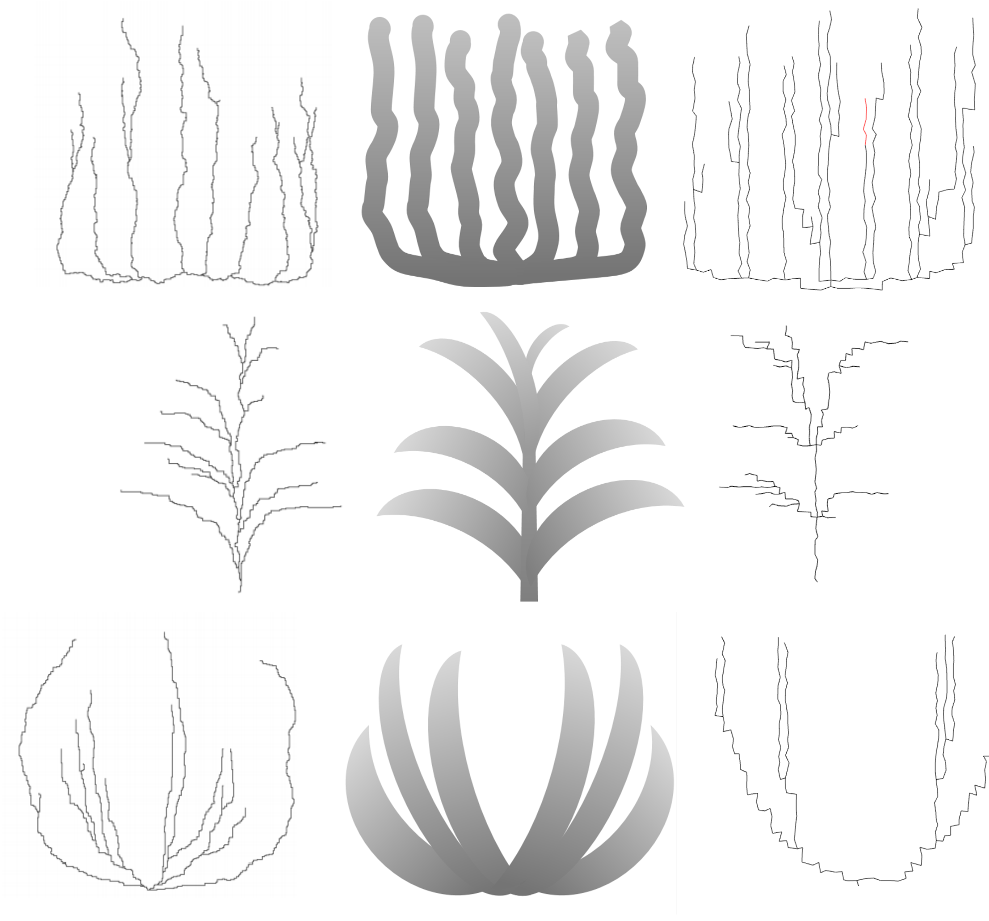


Figure 5.4: Dendrite produced by Xu and Mould [22] (left), input image for our image weighting algorithm (middle), resulting dendrite (right)

It is important to note that in the above cases, the input image needs to be created such that there is a color gradient in the desired direction towards the root. Aside from ensuring variance between weights, this is also critical to ensure that lower weights (lower costs) are closer to the root such that the greedy algorithm that traverses the node network always favors a direction towards the root. The importance of this can be shown in Figure 5.5, where two dendrites are produced using the same image shape but one with directional gradients and the other a solid black color. Even though the structure is the same, the resulting dendrites are different, as in the solid fill image, there is not enough weight variance and therefore the algorithm picks a route based on its internal bias of node checking.

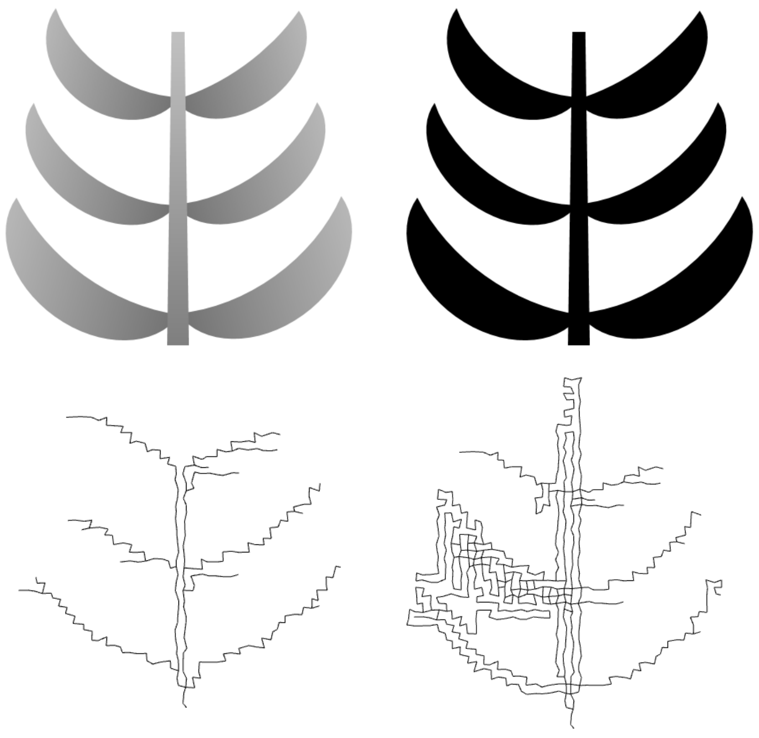


Figure 5.5: Comparison of gradient image (left) to solid image (right) for weighting and resulting dendrites (bottom)

5.3 Modeling Natural Phenomena

In this section, we use the results from the previous two subsections to showcase a number of models developed using our pipeline. In each case a real world reference image is used to give a general impression of the desired model output. The renders are presented in a very minimalistic scene so as to show what an artist's starting point may be.

Figure 5.6 shows a case study of a slanting tree structure, with the corresponding model rendered on the right. This was generated using image weighting, with the input image and dendrite shown in Figure 5.7. In this case, since image weighting was used, the dendrite was in 2D, however once in Houdini$^{\text{TM}}$, bending was used to give the structure a 3D feel. One can also notice that tapering was used in this case as each path gets thinner from root to branch. Some of the paths were also refined using procedural noise which resulted in more erratic variation, further enhancing the 3D feel.

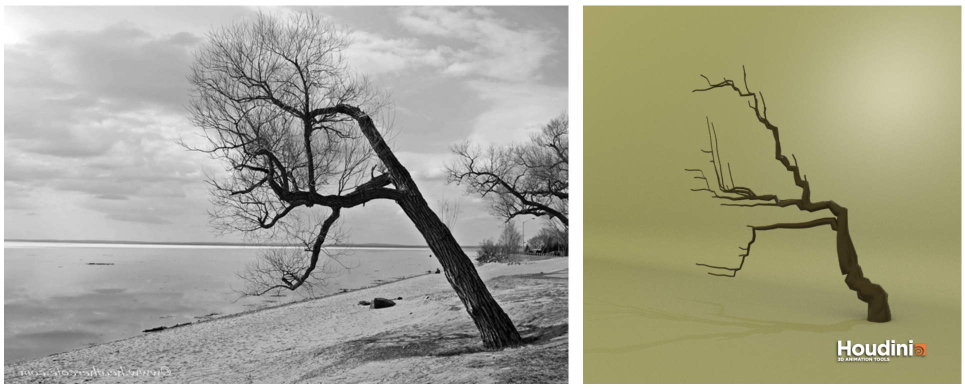


Figure 5.6: Modeling a natural tree [4] (left) in 2D with bending (right)

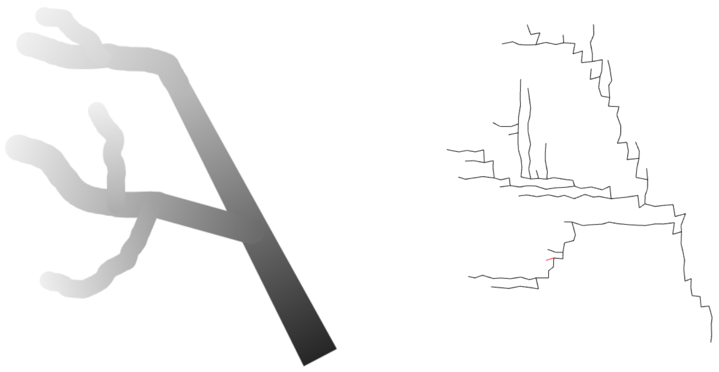


Figure 5.7: Dendrite evolution for Figure 5.5, input image weighting (left) and dendrite (right)

The second case study attempts to use a similar approach, but replicate 2D dendrites to produce a 3D model. In this case coral is used as an example, and therefore less tapering is used in this case to maintain radius even at the endpoints. Figures 5.8 and 5.9 show once again how a simple input image can produce a dendrite that through a modeling tool can be made to look very similar to the subject matter.

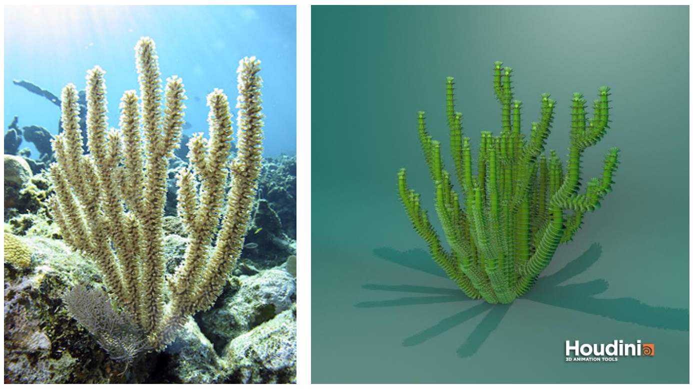


Figure 5.8: Modeling a natural coral [18] (left) in 2D using image weighting and replication in Houdini (right)

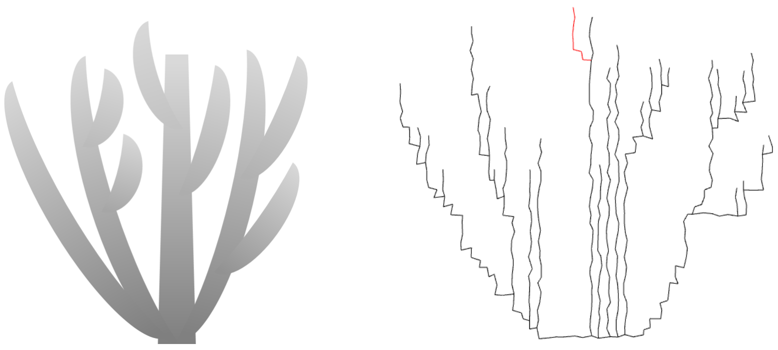


Figure 5.9: Input image and dendrite

The last case study, shown in Figure 5.10, attempts to treat two limitations of the previous case studies. Firstly, we generate a 3D dendrite using default alpha weighting. An iterative process is used to choose random endpoints within fixed bounded areas, so as to produce clusters of endpoints. A top view is shown in Figure 5.11 (left) to get an idea of the clusters. The same figure (middle) also shows how the bounding box can be used to make certain areas of the dendrite denser . The second limitation from the first case study is that while the branch struture matches the reference image very well, there is an absence of detail at the ends of the branches. In order to overcome this, we attempted to supplement the initial 3D model in this case study with smaller 2D dendrites scattered across some of 3D model tapered branches. For practicality, the scatter was only applied above a certain height, so as to match the clustered areas. Figure 5.12 shows a before and after comparison, where on the left we can see the model with just the 3D dendrite rendered, and on the right, we see it supplemented with 2D branches that were produced using image weighting. This shows that while image weighting may be limited in this implementation to 2D results, these can still be used in a hybrid environment when modeling a 3D structure.

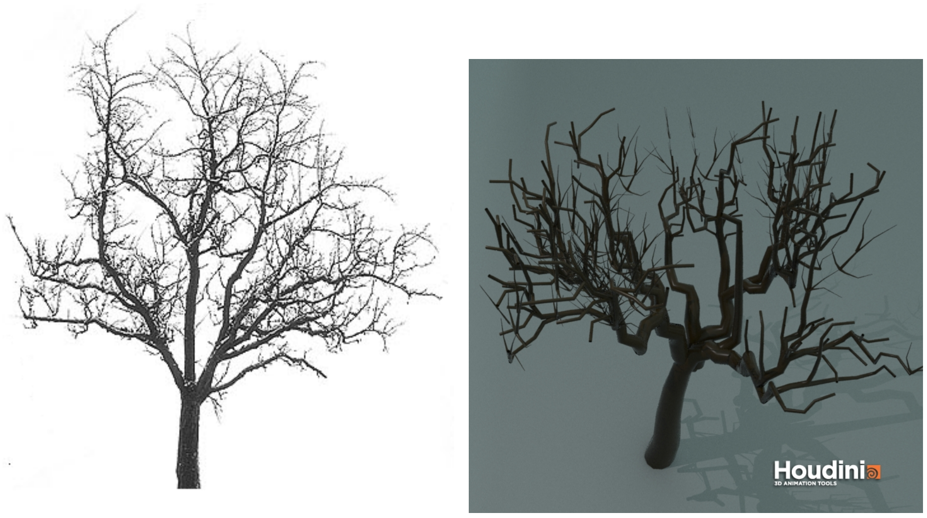


Figure 5.10: Modeling a tree [8] (left) in 3D, enhanced with 2D fractals (right)

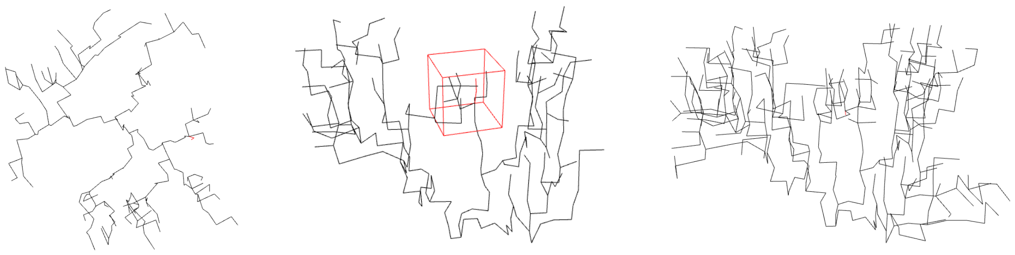


Figure 5.11: Dendrite evolution. 3D top view(left), endpoint selection via bounding box (middle), final dense dendrite side view (right)

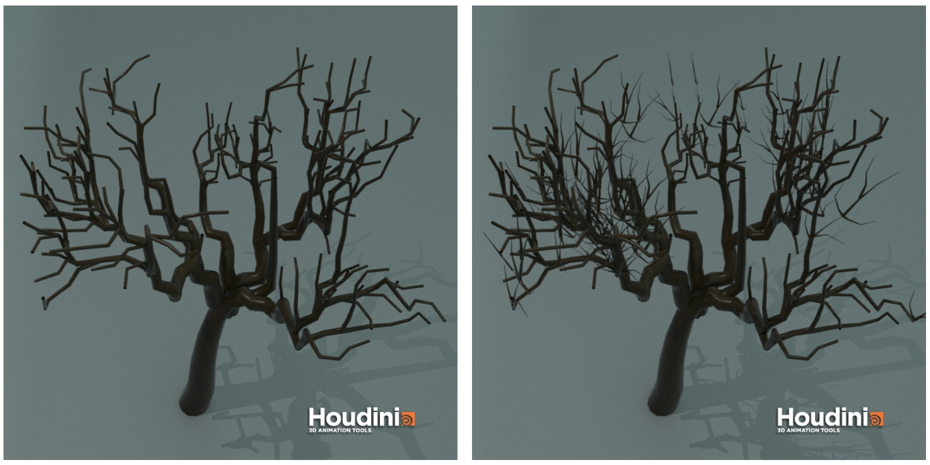


Figure 5.12: Model comparison; 3D dendrite only (left), 3D dendrite with 2D dendrite scatter (right)

Chapter 6

Conclusion and Future work

We have shown that path planning is a viable avenue for the generation of models resembling natural phenomena characterized by their dendritic nature such as plants, coral, trees and lightning. Since path planning is a well studied problem in the realm of artificial intelligence, there are many optimized algorithms available. This approach has the potential to be more computationally attractive than more established methods such as L systems and physically based techniques which can be very slow as the problem becomes complex. The second chapter provides the relevant background on these methods, and in particular makes reference to the path planning work first presented by Xu and Mould in [21] as a basis for our investigation into this area.

Our contribution in this project is an alternative method to distribute weights over the navgational lattice of the path planning algorithm. Our method takes a gray scale image as input whereby the pixel colours are used as weights. We have demonstrated its effectiveness in our experiments and it is clear that image based weighting is a simple way of informing the behaviour of the path planning algorithm and in turn influencing the general shape of the model. In the case studies we have presented the path planner was encouraged to seek inexpensive edges that correspond to the darker regions of the input image. For the experiments presented in the results section, we have picked out a number of real world case studies. Through this method we have succeeded in generating simple models that clearly resemble their real world counterpart with little information about the morphology of the natural phenomena itself except from what is visible in the source image. The versatility of our method is also demonstrated by the use of targeted endpoint selection, where random endpoints are selected within a bounded area, thereby controlling the overall structure of the model.

This approach was found to be a very straight forward and generic way of controlling the behaviour of the path planner without the need of adding complex heuristics. However, a major consequence of this method is that the detail elements that could be generates are limited to the resolution of the navigations graph, On the other hand, highly dense graphs could pose certain performance and memory problems. In fact, our implementation contains directional edges which doubles the amount of edges that need to be maintained in memory. Therefore, it would be worth looking into alternative structures which reduce the load and perform faster. Furthermore, the implementation of the edge selection may result in harsh changes in direction which may not occure in certain natural phenomena. This is due to the indiscriminate approach to edges angles. A possible improvement would be to introduce herusitics in the path planner to favour slightly more expensive edges with lower angle changes. Another way to address this would be to apply path refinement as suggested by Xu and Mould but only between nodes that have harsh angle changes.

As part of future work, a natural extension for our application is the ability of generating dendrites in an iterative fashion to get a hierarchicial effect and generate more complex models. Also, a lot of room for exploration in the selection of root and endpoints remains. Instead of manually selecting these points in a bounded area, it will be interesting to make used of more modern artificial intelligence techniques to infer the endpoints based on the outline shape of the image.

Bibliography

[1] Algfoor, Z. A., Sunar, M. S., and Kolivand, H. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *Int. J. Comput. Games Technol. 2015* (Jan. 2015), 7:7–7:7.

[2] Alliance, G. I. T. T. Dijkstra Algorithm: Short terms and Pseudocode. `http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html`. Online; Accessed 20 August 2016.

[3] Bell, S. An Overview of Optimal Graph Search Algorithms for Robot Path Planning in Dynamic or Uncertain Environments. `http://stanford.edu/~sebell/oc_projects/ieeepaper19Mar10_stevenbell.pdf`, 2010. Online; Accessed 15 August 2016.

[4] Cote, H. Black and White - Tree Photograph. `https://heathercote.com/2013/05/05/black-white/`, 2013. Online; Accessed 20 August 2016.

[5] Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., and Prusinkiewicz, P. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on computer graphics and interactive techniques* (1998), ACM, pp. 275–286.

[6] Dijkstra, E. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1*, 1 (1959), 269.

[7] Ebert, D. S. Advanced Modeling Techniques for Computer Graphics. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 153–156.

[8] Hogrefe, A. Winter Special 4 - Rendered trees image. `https://visualizingarchitecture.com/winter-special-4/`, 2014. Online; Accessed 20 August 2016.

[9] Hornby, G. S., and Pollack, J. B. Evolving l-systems to generate virtual creatures. *Computers & Graphics 25*, 6 (2001), 1041–1048.

[10] Kim, T., Henson, M., and Lin, M. C. A Physically Based Model of Ice. In *ACM SIGGRAPH 2004 Sketches* (2004), ACM, p. 13.

[11] Macey, J. NCCA Graphics Library. `https://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/index.html`. Online; Accessed 20 August 2016.

[12] Měch, R., and Prusinkiewicz, P. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 397–410.

[13] Plemenos, D., and Miaoulis, G. Intelligent Techniques for Computer Graphics. In *Artificial Intelligence Techniques for Computer Graphics* (2009), Springer, pp. 1–14.

[14] Prusinkiewicz, P., James, M., and Měch, R. Synthetic topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 351–358.

[15] Prusinkiewicz, P., and Lindenmayer, A. The algorithmic beauty of plants (the virtual laboratory).

[16] Reed, T., and Wyvill, B. Visual simulation of lightning. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 359–364.

[17] SideFX. Houdini Object Model. `http://www.sidefx.com/docs/houdini9.5/hom/`, 2016. Online; Accessed 20 August 2016.

[18] Society, M. C. Coral Reefs - MarineBio.org - Coral Photograph. `http://marinebio.org/oceans/coral-reefs//`. Online; Accessed 20 August 2016.

[19] Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., and Koltun, V. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG) 30*, 2 (2011), 11.

[20] Witten, T. A., and Sander, L. M. Diffusion-limited aggregation. *Physical Review B 27*, 9 (1983), 5686.

[21] Xu, L., and Mould, D. Modeling dendritic shapes using path planning. In *GRAPP* (2007), INSTICC - Institute for Systems and Technologies and Information, Control and Communication, pp. 29–36.

[22] Xu, L., and Mould, D. Constructive Path Planning for Natural Phenomena Modeling. In *Artificial Intelligence Techniques for Computer Graphics*. Springer, 2009, pp. 83–102.

[23] Xu, L., and Mould, D. Synthetic Tree Models from Iterated Discrete Graphs. In *Proceedings of Graphics Interface 2012* (2012), Canadian Information Processing Society, pp. 149–156.

Appendix A - Python import script for Houdini$^{\text{TM}}$

```
# import the stuff we need to access houdini and parse xml
import hou
import xml.etree.ElementTree as et
# let us parse the xml
filepath = hou.getenv("XML") + '/input.xml'
xmldata = et.parse(filepath)
skeleton = xmldata.getroot()
# let us init imp houdini variables to play around with the node network
node = hou.pwd()
root = hou.node('/obj')
# now we start
# first things first - create our geo object that will be the parent of everything
skeleton_node = root.createNode('geo')
skeleton_node.setName(skeleton.attrib['name'])
# delete the default nodes
for child in skeleton_node.children():
    child.destroy()
# edit the parameter interface of the skeleton node
# parmTemplate = hou.FloatParmTemplate("ths", "Thickness at Start", 1,
    default_value=([0]), min=0, max=10, min_is_strict=False)
# skeleton_node.addSpareParmTuple(parmTemplate, in_folder([Tapering]), create_missing_folders=True)
parm_group = node.parmTemplateGroup()
parm_folder = hou.FolderParmTemplate("folder", "Thickness and Tapering")
parm_folder.addParmTemplate(hou.FloatParmTemplate("ths", "Thickness Start", 1, default_value=([0.3])))
parm_folder.addParmTemplate(hou.FloatParmTemplate("thf", "Thickness Finish", 1, default_value=([0.3])))
parm_folder.addParmTemplate(hou.FloatParmTemplate("beta", "Tapering Factor", 1, default_value=([1])))
parm_group.append(parm_folder)
skeleton_node.setParmTemplateGroup(parm_group)
# for every path we must create a curve node
for path in skeleton:

    curve = skeleton_node.createNode('curve')
    curve.setName(path.attrib['name'])
    curve.parm('type').set(1)

    # empty list of vertices
    vertices = []
    for vertex in path:
        x = vertex.find('x')
        y = vertex.find('y')
        z = vertex.find('z')
        coord = x.text + ',' + y.text + ',' + z.text
        vertices.append(coord)

    coords = ' '.join(vertices)
    curve.parm('coords').set(coords)
sopcategory = hou.nodeTypeCategories()["Sop"]
curvetype = hou.nodeType(sopcategory, "curve")
i = 0
sweep_expr = str("ch('../ths')*(1-pow($"+"PT/$"+"NPT, ch('../beta')))
    + ch('../thf')*pow($"+"PT/$"+"NPT, ch('../beta'))")
circle = skeleton_node.createNode('circle')
circle.parm('type').set(1)
for child in skeleton_node.children():
    if child.type() == curvetype:
        sweep = skeleton_node.createNode('sweep')
        sweep.setInput(0, circle)
        sweep.setInput(1, child)
        sweep.parm('skin').set(1)
        sweep.parm('xformbyattribs').set(False)
        sweep.parm('scale').setExpression(sweep_expr)
    i = i + 1

# merge all nodes together
merge = skeleton_node.createNode('merge')
sweeptype = hou.nodeType(sopcategory, "sweep")
i = 0
for child in skeleton_node.children():
    if child.type() == sweeptype:
        merge.setInput(i, child)
    i = i + 1
merge.setDisplayFlag(1)
```

Appendix B - XML sample

```
<skeleton name='simple_example'>

     <path name='path_0'>

          <vertex>

               <x>2.88855</x>
               <y>2.85935</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>1.76534</x>
               <y>2.70711</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>1.80134</x>
               <y>1.84142</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>0.889024</x>
               <y>1.88056</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>0.566371</x>
               <y>0.817885</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>0.99892</x>
               <y>-0.479899</y>
               <z>0</z>

          </vertex>
          <vertex>

               <x>-0.467331</x>
               <y>-0.268224</y>
               <z>0</z>

          </vertex>

     </path>
     ...
```

```
      ...
      ...
      <path name='path_6'>

           <vertex>

                <x>-1.35514</x>
                <y>-2.402</y>
                <z>0</z>

           </vertex>
           <vertex>

                <x>-1.05441</x>
                <y>-1.21027</y>
                <z>0</z>

           </vertex>
           <vertex>

                <x>-1.46853</x>
                <y>-0.253649</y>
                <z>0</z>

           </vertex>
           <vertex>

                <x>-0.467331</x>
                <y>-0.268224</y>
                <z>0</z>

           </vertex>

      </path>

</skeleton>
```