Art-directing procedural vegetation in Houdini using a space colonization algorithm

Marta Feriani
s4900092

19th August 2017

MSc Computer Animation and Visual Effects

Acknowledgements

I would like to thank my supervisors Jon Macey and Phil Spicer for sharing their knowledge and offering guidance during the development of this project.

Another special thank comes to Yannis Ioannidis and Michail Agoulas for being extremely patient and supportive during hours of troubleshooting and brainstorming.

I would be eternally grateful to all my fellow students at MSc CAVE for the good advice, the honest feedback, the good food and the good time that made this year unforgettable.

Last but not least I would like to thank my family: my mother for always being supportive and pushing me every day to become the better version of myself and my brother who has always offered his time, knowledge and experience to help me succeed since we were children. I hope that one day I will be able to reciprocate what you did for me through all these years.

Contents

List of Figures

List of Algorithms

Abstract

Vegetation modelling plays a key role in shots composition. Achieving a believable and nature-like result has proven to be a tedious and time consuming task for artists. To overcome this issue we present a Houdini tool for trees modelling. Through specifying different parameters' values the tool will generate a variety of tree shapes. Tree structures are achieved using the space colonization algorithm by Runions et al (2007). Model parameters are directly mapped onto visual characteristics and offer an appropriate control over the silhouette and branching structure. This project shows that the toolset can produce a reasonable variety of trees.

1 Introduction

Vegetation plays a key role in the composition of a shot. The arrangement of trees and the choice of specific species can help conveying a particular mood and help defining the staging of a particular scene. The aim set for vegetation can vary greatly between different movies. For *The Jungle Book*, for example, MPC (2016) built a sophisticated pipeline to build visually rich and realistic environments relying on both photogrammetry and 3D software as SpeedTree and Maya. Even though the aim was high quality realistic look, the pipeline also integrates tools (Melson et al (2016), Cieri et al (2016)) that allow for art-directing such vegetation. The final look achieves both goals of realism and artistic staging of vegetation.

Sometimes, for animated movies, realism is not such as a primary target as the art-directability is. For the *Last Bastion,* animated short by Blizzard Animation (2017), the studio designed a complete toolset to handle vegetation scattering and simulations using plant models created in Maya.

In many animated movies from Disney, as well, vegetation plays a key role throughout the story. *Zootopia*'s mesmerizing landscapes were built with Disney in-house procedural vegetation modelling tool Bonsai (Keim, 2016). Their tool managed to produce variety not just across species but within the same tree or bush specie as well. All vegetation models are also able to interact with the main characters or with external forces such as wind.

Being the aim of this project the sole modelling of procedural vegetation, in particular trees and with a specific preference for art-directability over realism, the main inspiration was found in *Tangled* (Shek, 2010). For the production of this animated movie, Disney developed a whole new engine (Dendro Engine) that uses an artist sketched branches hierarchy and a rough shape for the crown to produce procedural tree modules. Unfortunately, no further details are given about how Dendro's inner algorithm calculates new branches. This project aims to reproduce a tool that offers similar features to Disney Dendro Engine.

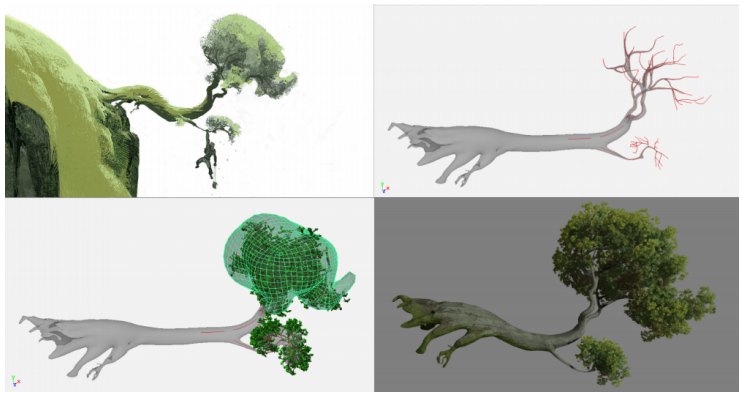

Figure 1: Dendro Engine

2 Related Works

Modelling natural shapes such as trees is difficult because of the richness of small details. The history of 3D tree modelling starts with a recursive algorithm proposed by Honda (1971). The recursive structure relies on a few geometrical attributes such as branching angles and length ratio between consecutive segments. Honda also studies the nature of tree branching addressing the monopodial branching pattern shown in figure 2 as a special case for the dichotomous branching for structures that are parallel to the gravity. Honda's recursive approach has been at the core of many later modelling algorithms.

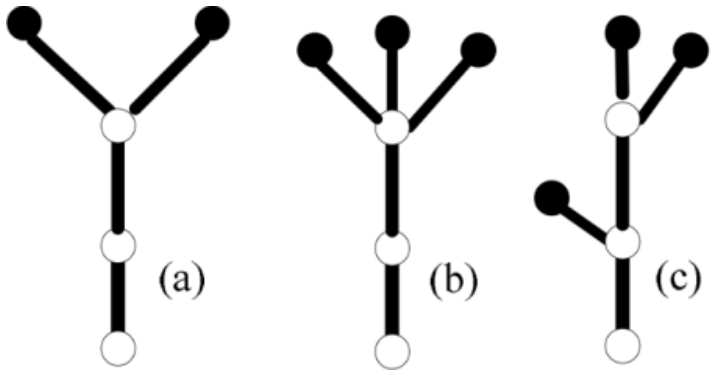

Figure 2: Branching pattern by Wang et al (2008)
(a) sympodial branching (binary tree), (b) sympodial branching (trinary tree), (c) monopodial branching

Lindenmayer (1996) introduced a string rewriting system know as L-System to describe cellular interactions. An L-System (section §7.1) is a formal grammar that consists of an alphabet, production rules and an axiom. The alphabet is a collection of symbols that can be used to produce strings. Production rules expand each symbol or string into some larger string. The axiom is used as starting point for the production. The string produced by iterating over the axiom can be later interpreted as geometric commands. Prusinkiewicz (1996) applies Lindenmayer rewriting system to plants structures adding few extensions such as context sensitivity and random variations. L-Systems produce good results but the self similarity and the production patterns remain clearly visible in the final model. To overcome the pattern visibility, literature (Prusinkiewicz, Lindenmayer 1996) has introduced stochastic L-System which involves the selection of a production rule from the set based on a probability value. Other improvements on L-Systems involve the definition of a container volume for the L-System to grow into (Ijiri et al, 2006). Given the volume and an initial segment, the system will automatically set the depth for the recursion.

Oppenheimer (1986) also relies on Honda recursive approach by developing natural patterns using fractals section §7.1. The fractal specification is based on parameters such as branching angles, branch-to-parent size-ratio and branch-

per-stem number. The fractal method by Oppenheimer was found to be affected by severe self similarity pattern visibility which gave the final shape a machine-made look. To overcome this problem Weber and Penn (1995) improved the recursive model by adding randomness and organized functions to affect the tree development.

The interpretation of a tree as a recursive structure is justified by the process of tree development but plays a smaller role for fully grown trees. Buds may have different fate such as growing to major limbs, being shed or remaining small twigs. In the architecture of a mature tree, therefore, the regularity of the recursive branching is mainly lost, overridden by subsequent development (Weber and Penn 1995). Furthermore, finding rules and fitting parameters for recursive structures is a non trivial task. A small variation on the initial conditions will propagate exponentially through the generations becoming more and more evident, to the point that trying to tweak the existing rules to produce small variations on the produced model will often lead to a drastically different whole new structure. Given the recursive nature of both L-System and fractals, even the use of random variables cannot completely overcome self-similarity issue and the resulting architectures give the machine-made impression (Rodkaev et al, 2003).

New techniques for 3D tree modelling find common ground in the Runions et al (2005) approach developed for leaves venation. Rodkaev et al (2003) inherit Runions approach and applies it to a new algorithm based on particles that produces realistic leaves venation on a 2D space or a branching structure in a 3D space. Rodkaev populates a shape or a volume with particles. Each particle will move towards a predefined goal position merging with neighbours particles when the distance is smaller than a certain threshold. The trail for the particles trajectories generates the tree graph.

3 Technical background

Similar to Rodkaev et al (2003), Runions et al (2007) suggest a space colonization algorithm as an extension of the 3D leaves venation by Runions et al (2005). Unlike Rodkaev's, Runions' branching structure is formed in a base-to-leaves order. Each iteration of the algorithm produces new elements that expand the tree structure formed in the previous steps. This approach results not only to be adaptive to obstacles and neighbour plants, but also provides controls over the growth process that results in a wide variety of tree structures.

3.1 Overview

The algorithm proceeds as follows. A 3D envelope for the tree crown is specified as input. Its volume is seeded with a set of attraction points. The attraction points influence the growth of the tree structure by signalling the availability of empty space within the tree crown envelope. A single point is specified as input and acts as the base of the tree. Having the attracting points and the first

tree point, the tree structure is generated as an iterative process. Each iteration produces new tree points in the direction of the neighbouring attraction points to extend the existing tree structure. The algorithm may stop after a user-defined number of iterations. If let running, the algorithm will either stop when there are no attraction points close enough to the tree to influence its growth (i.e. no tree points are within a threshold distance, called radius of influence, from the attraction points) or when all attraction points have been reached by tree points.

Further manipulation involves subdivision and smoothing of the internodes (branch that connects two consecutive nodes of the tree), skinning and scattering of organs such as leaves, flowers or fruits.

3.2 Space Colonization Algorithm

Competition for space plays a fundamental role in the colonization algorithm and determines the branching pattern for the tree structure. Before delving into the details of the algorithm some definitions and axioms are needed.

3.2.1 Definitions and axioms

1. Every point the algorithm uses is either a tree node or an attraction point. From now on we will refer to tree nodes with v and attraction point with s.

2. N is the set of all the attraction points.

$$\forall s,\ s \in N \tag{1}$$

3. T is the set for the tree nodes

$$\forall v,\ v \in T \tag{2}$$

4. If we refer to p as a generic point in the algorithm

$$\nexists p \mid p \in N \wedge p \in T \tag{3}$$

5. $S(v)$ is the set of attraction points s that influence the growth of the tree point v for the current iteration.

6. $d(a, b)$ is the Euler distance between point a and point b

$$d(a,b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2 + (a.z - b.z)^2} \tag{4}$$

7. di is the *radius of influence* which represents the maximum distance between s and v so that $s \in S(v)$

8. dk is the *kill distance*. If $\exists v \in T \mid d(s, v) < dk$, then s is removed.

9. D is the distance between consecutive nodes of the tree

3.2.2 Steps

1. Define a 3D shape for the tree crown. Initialize the parameters N, di, dk, D.

2. Populate the 3D envelope with attraction points using a certain random distribution.

3. Define at least one tree node to act as the base of the tree structure.

4. Each attraction point $s \in N$ may influence the tree node v that is closest to it. The influence occurs if $d(s, v) < di$. For each $s \in N$, check if there is any $v \in T$ that meets the condition. More than one attraction point may influence the same v tree point: all the attraction points that influence v are gathered in the $S(v)$ set.

5. For each tree node $v \in T$, if $S(v)$ is not empty, a new tree node v' is created. The new tree node lies at a distance D from v in the direction of the average of the normalized vectors towards the attraction points $s \in S(v)$

$$v' \in T \tag{5}$$

$$d(v, v') = D \tag{6}$$

$$\vec{n} = \sum_{s \in S(v)} \frac{s - v}{\| s - v \|} \tag{7}$$

$$\hat{n} = \frac{\vec{n}}{\| n \|} \tag{8}$$

$$v' = v + D\hat{n} \tag{9}$$

6. Perform a check to test if any attraction points $s \in N$ should be removed. This happens if the following condition is true for at least one $v \in T$:

$$d(s, v) < dk \tag{10}$$

7. repeat from steps 4 to 6 until the stopping conditions mentioned in section 3.1 are met.

4 Implementation

The network for the implementation of the growth algorithm relies on the layering of simple rules that eventually lead to the creation of the complexity that characterizes trees in nature. Houdini supports multiple scripting languages such as HScript, Python and VEX. Being Python widely used as scripting language and being a reality shared between many DCC software, it might seem the

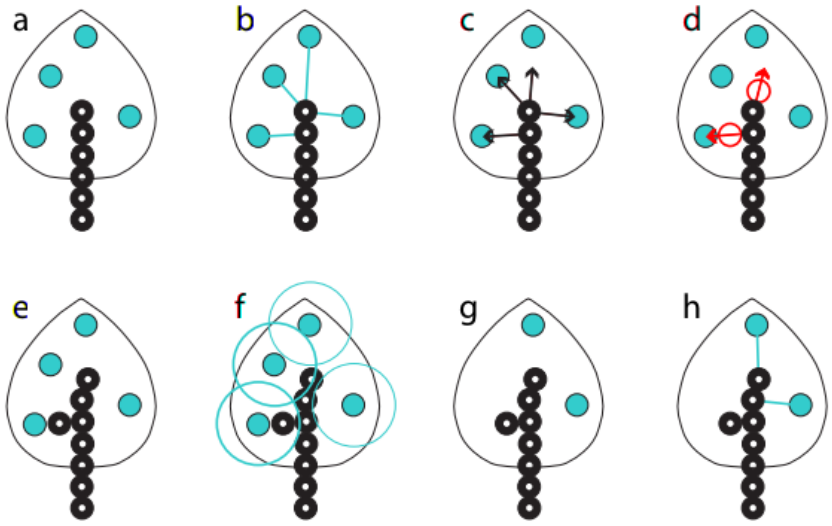

Figure 3: Space Colonization Algorithm by Runions et al (2007)

most sensible choice. On the other hand, VEX scripting language is the Houdini native language. VEX gives direct access to geometry data and provides a wide variety of built-in functions optimized for best performances on the Houdini geometry internal representation. In terms of performances, VEX has proven to be faster than Python because it implements automatic multithreading while Python does not have the same feature. Given the marked difference in performance, VEX was used throughout this project.

4.1 Wrangle Node

The *Wrangle* node (SideFX, 2017) is the most present node type in the network therefore it seemed only natural to give a brief explanation of how it works.

The *Attribute Wrangle* is a low level node that allows for the tweaking of the geometry attributes using VEX code. Its capabilities correspond to the *Attribute VOP SOP*. The main difference between the two of them is that *Wrangle* uses a textual editor whilst the *Attribute VOP SOP* uses a visual network.

The node can edit the input geometry by changing or adding attributes. It can also remove or create new geometry for example by adding points, linking points together to create polygons and so on. The snippet runs on details or on every point/primitive/vertex of the first input geometry. Using multiple sources to the node it is also possible to access data from different geometry at the same time. When having multiple sources, only the first input geometry will be passed down to the next node of the network whilst the other input geometries will be

lost.

When the snippet affects the attributes of the geometry, these attribute will not be available until the node is cooked. This often makes is difficult to populate geometry attributes such as arrays within the same *Wrangle* node. For this reason the implementation of particular parts of the colonization algorithm might not seem the most straightforward or obvious way to approach the problem, but have been deliberately designed to overcome this particular issue.

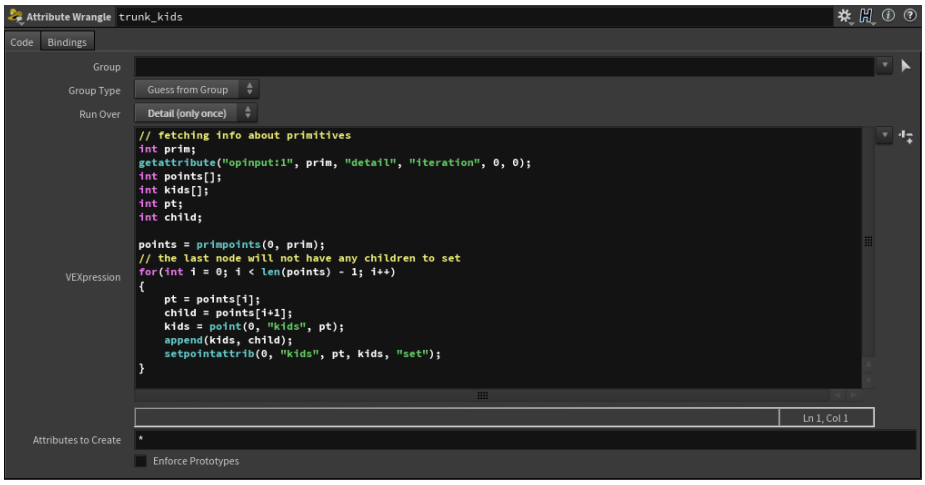

Figure 4: Wrangle node interface

4.2 Data structure

The colonization algorithm, as presented in section 3.2, mostly relies on points position. Each point has certain attributes to make it possible to create the tree structure. The attribute `@type` divides the point set in two main categories, respectively the tree nodes and the attraction points. Given the nature of the algorithm, the total number of attraction points will decrease each iteration whilst every new node that will be created will increase the count for the tree nodes. To give a better visualization for this partition, the colour attribute `@Cd` is initialized to $1, 0.5, 0$ for the tree node and to $0, 0.5, 1$ for the attraction points.

The aim of the algorithm is to create a believable branching structure to reproduce natural trees. Stems might either grow in a single direction or fork producing new stems. Older stems also grow thicker to balance the weight of the younger stems. This branching pattern can be represented by an acyclic graph data structure.

A graph is a finite collection of vertices and edges. Graphs are used to model pairwise relations between object. The vertices, or nodes, represent the objects while the edges are the relations. The different nature of the relations leads to different kinds of graphs. In the case of a directed graph, each edge

has an orientation, from one vertex to another vertex. A path is a sequence of edges. A node A is reachable by another node B if there exists a path that leads from A to B. If for each node in the graph there is no path that connects the node to itself, the graph is said to be *acyclic*. Being a graph a hierarchical structure, it is appropriate to address relations between nodes as parent/child relations. Each node is therefore aware either of its parent, its children or both. In computer science relations between nodes are implemented by storing in each node a pointer to its connected ones.

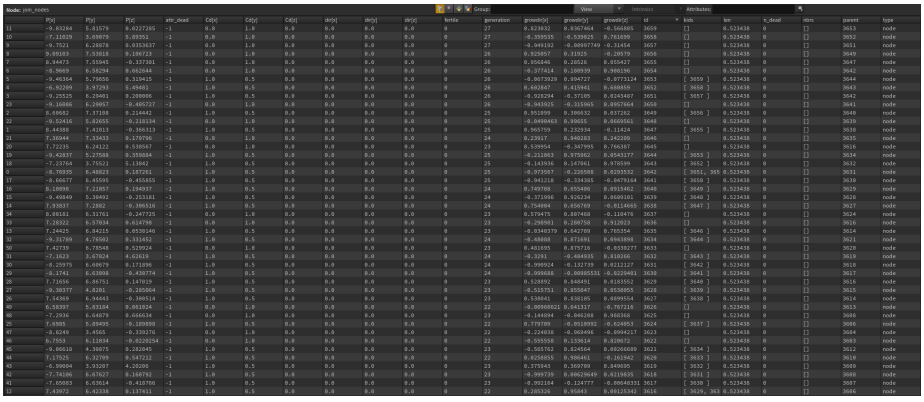

Figure 5: Data Structure (Geometry Spreadsheet)

The network developed for this project provides each tree node with two attributes, an integer and an array, to store references to the parent and the children respectively. Unfortunately, Houdini does not have pointers-like structures. The points of a geometry are stored in an array-like structure. The `@ptnum` attribute refers to the index of the point in that array. When referring to a point using the `@ptnum` attribute, one can gain direct access to other attributes, either built-in or custom, of that point. Unfortunately, being the `@ptnum` ultimately just an index in an array, its value is not consistent through the network: operations such as adding or removing points or merging together two geometries may shift or offset the `@ptnum` value. Given the unstable nature of `@ptnum,` building parent/child relation upon it did not seem to be a suitable choice. As a workaround, each tree node also stores a unique identifier `@id`. Smaller `@id` values belong to older tree nodes while greater values identify younger ones. When populating the `@parent` and `@kids` attributes, the reference to the other points will not be represented by `@ptnum` but by `@id` instead. Having to rely on a custom attribute such as `@id` automatically implies losing the direct access to the points granted by `@ptnum`. This greatly affects the performances of the whole algorithm because, in order to find a node with a specific `@id,` one has to cycle over each point of the geometry and check for a match. In order not to loose this convenient feature of direct access, every time the algorithm works on the tree nodes the *Sort* node is used to remap `@ptnum` based on the `@id` value. This solution allows for a reliable hierarchy structure

while maintaining speed of access.

Another important information about the tree nodes is the growth direction and the length of the internode (the distance between consecutive tree nodes). Storing this values makes it possible for a more detailed manipulation of branching angles at a later time. Each point therefore stores a `@growdir` attribute that holds the normalized growth vector that links the parent to the node point position.

As a final step, each node also has two flags respectively for death and fertility. The fertility flag signals if the node can produce a child in the current iteration, and can therefore assume "non fertile" and "fertile" values in two consecutive iteration. On the other hand, when the death flag is set to `true` it determines an irreversible condition for the node: a node dies whenever it reaches the maximum amount of children or in degenerate cases such as the position of the child node being coincident with its own position.

As the information needed on a per-node basis are quite important in terms of quantities, whenever an attribute is not needed any longer it is removed from the geometry.

4.3 Parameter initialization

In order for the colonization algorithm to start, some parameters have to be set such as the tree crown envelope, the attraction points position and the roots points.

4.3.1 Tree crown

The network provides up to three different ways to specify the crown volume for the tree. All of them take as an input polygonal shapes that are later converted to a volume. In its very basic implementation, the user-specified tree crown is converted to a volume using the *IsoOffset* node. The conversion is necessary in order to populate the tree crown volume with the attraction points. Moving from the base implementation, the user can specify a collision object for the tree crown to interact with. This options makes it possible to simulate a tree growth process where the final shape is affected by surrounding obstacles. The collision object is first converted to a volume. To identify the viable area for the tree to grow, the *volume mix* node performs a subtraction operation between the two input volumes. The resulting volume is then used for the tree to grow. Another option allows the user not to specify tree crown volume directly, but it instead calculates the viable space for growth from a given object. The input geometry is converted to volume twice: the first conversion replicates the input geometry closely whilst the second one will inflate the volume with a user-defined offset. The difference between the volumes will create a layer wrapped around the initial object. This option can convey the illusion of a plant clinging and growing finding support on an object, like vines do.

Unfortunately the conversion from polygon to volume can produce artefacts due to the conversion method used by the *IsoOffset* node. Different workarounds

to this problem have been tried but none of them proved to be successful for all kinds of input geometry.

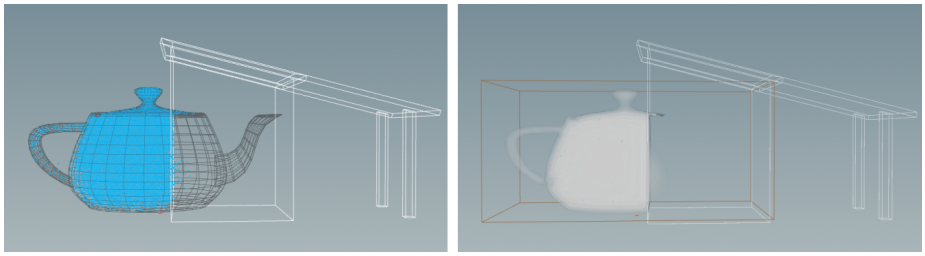

Figure 6: Collision Object
Left side geometry objects, right side resulting volume.

4.3.2 Attraction Points

At early stages of the production, the volume obtained from the tree crown shape was seeded with uniformly distributed attraction points. The resulting trees, as described by Runions, show a uniform density in the branches distribution. In many real trees and shrubs, however, the density increases near the crown surface due to better light exposure. To achieve the same result, following Runions suggestion, the attraction points in the network are seeded using a user-defined density profile, function of the distance from the outer surface of the tree crown. The steps to achieve such behaviour are explained below.

As a first step, the tree crown is seeded with attraction points using a uniform distribution. A *Wrangle* node "`distance_from_bounding_object`" evaluates, for each attraction point, the distance from the bounding object provided as a second input to the node. The resulting value is stored to an attribute named `@dist`. The *Promote* Houdini node is used twice to find respectively the maximum `@max_dist` and the minimum `@min_dist` value for the `@dist` attribute. This information feeds a second *Wrangle* node "`radial_distribution`": each node's `@dist` value is taken from its original range (`@min_dist`, `@max_dist`) and re-scaled to its corresponding value in the new range $(0, 1)$. The resulting variable is linked to a user-customizable ramp. The $y-$value of the ramp represents the probability for the attraction point to survive. A random number based on the `@ptnum` of the current attraction point is calculated and compared to the probability value. If it results to be less, the attraction point gets removed. The ramp extends the potential of Runions idea by giving the users not only control over the density near the crown surface but providing them with a greater art-directability over the density of branches from the centroid of the tree crown shape to its boundaries.

The same principles are used to define a second custom distribution that uses the Y coordinate of the point position instead of the distance from the object (in this case the nodes involved are called "`AP_height`" and "`vertical_distribution`").

The newly generated attraction points are then initialized.

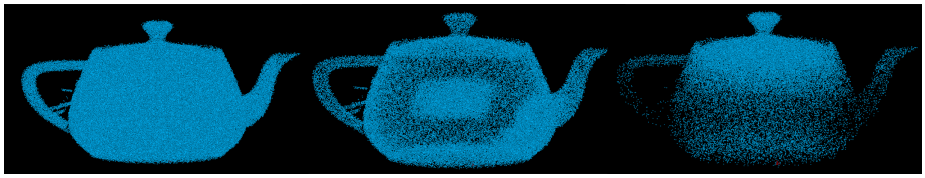

Figure 7: Attraction Points density distribution

4.3.3 Roots

The network provides three different ways to specify the roots for the colonization algorithm. A user can either create single root points by specifying the coordinates, input a pre-made set of points, or a trunk-and-branches curves structure.

For the first two options the initialization of the nodes attributes follows the same process. All the root points' `@parent` is set to -1 to signal that they do not have any parent. As said before, the colour `@Cd` is initialized to yellow $1, 0.5, 0$, `@kids` is declared as an empty array, and `@type` is initialized to "node". The parameters `@growdir`, `@n_dead`, and `@attr_dead` are initialized to their default values, respectively $0, 0, 0$, 0 (meaning that the node is alive) and -1.

The initialization of some attributes such as `@parent`, `@kids` and `@type` differs in the case where a custom trunk structure is fed in the system. This kind of structure already contains primitive information that links together points in a precise hierarchy. The most likely scenario is that each branch is represented as a curve primitive. The fork points of the structure will then belong to two different primitives at once, for example to both the trunk and a branch. The first attribute to be initialized is the `@kids` array using the "`trunk_kids`" snippet. This *Wrangle* node is encapsulated in a `for` loop nodes block that will iterate on each primitive of the input geometry. The *Wrangle* will therefore process the points of one primitive at a time. In the *Wrangle* interface, the snippet is set to "run over details" so that the iteration process over the points is explicitly declared in the code. The snippet starts by retrieving an ordered array of the current primitive points. A `for` loop then iterates over this array reading the point's `@ptnum` value and assigning the next point in the array to itself as `@kids` element. By using this approach, the snippet successfully handles specific cases in which the children to the point belong to different primitives.

After having initialized the `@kids` attribute, the `@parent` is straightforward: the "`trunk_parent`" *Wrangle* iterates over each point of the input geometry regardless of the primitive they belong to. Each point will read its own `@kids` list and proceeds to set itself as `@parent` attribute to these points. One can refer to this process as "parent injection". It is important to highlight that at this stage the `@id` attribute described in section 4.2 has still not been initialized, and the parent-children relation is still built on the `@ptnum`. It is also important

to understand that in this particular part of the network all the geometry being processed belongs to the tree structure. The `@id` can be therefore initialized as a plain copy of the `@ptnum`. Being `@ptnum` and `@id` coincident at this stage, the initialization process described results to be correct. As a next step, the `@type` has to be set. The implementation process makes sure that the user is given a certain extent of controls over the choice of root points for the growth algorithm. The network highlights as candidate roots all the points contained within the volume of the tree crown shape. The user can then decide which of them to keep. For the selected points, the `@type` is set to "node" while the remaining ones are set to "trunk". During the growth algorithm, only the nodes will interact while the others will be lifeless. A final attribute `@generation` is initialized for all root points regardless of whether they are single points or points selected from a custom structure. This attribute plays a key role in a later part of the network to set and scale the cross section of the branches according to their age.

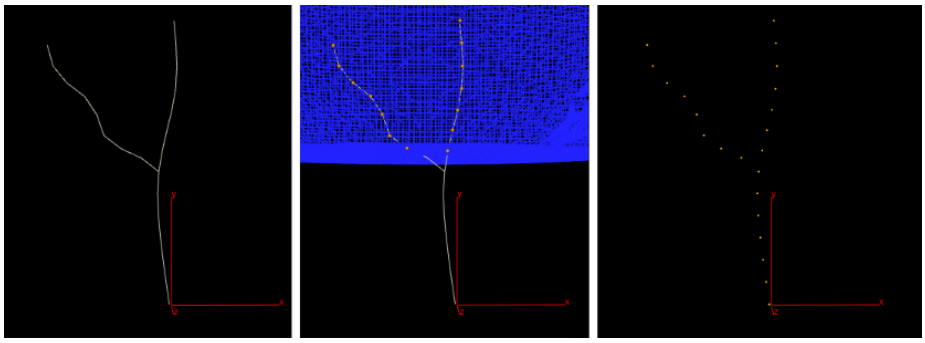

Figure 8: Roots initialization
From left to right: custom trunk structure, root selection, conversion to points.

4.4 Space colonization

The space colonization main algorithm lies inside a dedicated subnetwork "`growing_alg`". This subnetwork mainly consists of a `for` loop that determines the number of time the geometry is processed. Contrary to the stop conditions presented in section 3.1, the loop will only terminate after a user-defined number of iterations. It can happen that the user requires the algorithm to run more times than the ones required to meet the stop condition. In this case, even though the network will continue iterating, no new geometry is created. When running the algorithm, the user has also control over the values for D, dk and di.

Before delving in the colonization loop, the *Wrangle* "`AD_treepts`" sets a detail attribute over the geometry that specifies the total number of tree nodes based on the `@type` attribute of the points. This is later used to produce the `@id` for the newborn nodes.

At this stage as well, points with type "trunk" are filtered out of the growth process and rejoined with the rest of the geometry when the process ends.

Shortly, at each iteration the colonization algorithm needs to first find the set of attraction points (if any) that influence each fertile tree node, create new nodes in the direction determined by the influencing attraction points, and decide which attraction points will survive to the next iteration.

4.4.1 Finding influencing attraction points

The tree nodes and the attraction points are divided based on their `@type` value. In order to keep the calculation as light as possible, a preparation step is undertaken using the "`find_influencing_attr`" before finding the closest tree node of each attraction point. The *Wrangle* receives as first input geometry the set N of the attraction points and as second input the set T of tree nodes. The node purpose is to isolate the attraction points that will influence the tree growth for the current iteration from the inert ones. The snippet is set to "run over details". Iterating over each tree node, a function `_setInfluencing()` is invoked. The function takes as an input `a` the position of the tree node. Based on that position, the function retrieves all the attraction points that are closer than the user-defined *di* distance. Every attraction point that satisfies this condition is flagged as an influencing node. The set of influencing nodes is therefore a subset of the set N of all the attraction points. Using the *Blast* node, the influencing nodes are separated from the other inactive attraction points.

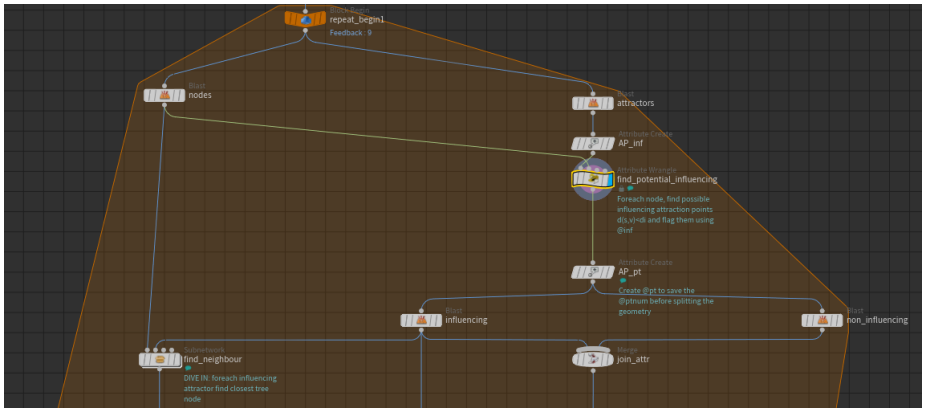

Figure 9: Influencing nodes network

The tree nodes and the influencing nodes are wired into the "`find_neighbour`" subnetwork. Each attraction point only influences the tree node closest to it. The search for the closest tree node is implemented in "`fast_search`": the *Wrangle* takes as a first input geometry the tree nodes and as a second input geometry the influencing points. At this stage not only each influencing point will look for its closest tree point but the sets $S(v)$ are populated as well. In this implementation the set is stored as a tree node array attribute `@nbrs` and will contain the `@ptnum` for all the influencing points for which the node is the closest

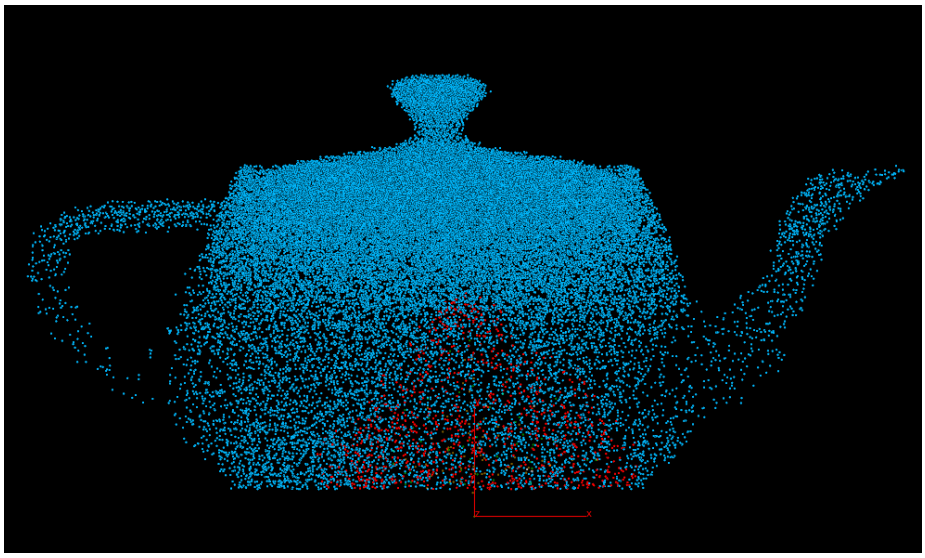

Figure 10: Visualization of the influencing nodes

of the tree nodes. This operation can be computationally heavy, therefore the following algorithm has been developed aiming to optimize such operations.

The algorithm is divided in three parts:

- an array `closest[]` is declared to have the same size as the total count of influencing points. Starting from the index 0, the influencing point with `@ptnum` 0 is processed. The point will search for the closest tree node and will store its `@ptnum` value in the array. The process moves to index 1 and processes the influencing point with `@ptnum` 1 and so on until the `closest[]` array is completely populated. Given how the array was populated, the indices of the `closest[]` array univocally identify the influencing points.

- two new arrays, `ordered_attr[]` and `ordered_closest[]`, are created:

  - the `argsort` function applied to the `closest[]` array will return a list of indices that, if applied to the `closest[]` will give a sorted sequence in an increasing order. In this particular case, the indices represent the influencing points. The newly created `ordered_attr[]` array holds therefore a sorted sequence for the influencing points.

  - the `sort` function applied to the `closest[]` will return a sorted array in increasing order. This new array is `ordered_closest[]`.

- `ordered_attr[]` now stores the influencing points, sorted by increasing `@id` of the tree node they influence (stored in `ordered_closest[]`). The algorithm now proceeds to locate the sets $S(v)$ and store them in the

proper tree nodes. The sets are consecutive chunks of the `ordered_attr[]` array. To tokenize it properly, the `ordered_closest[]` is used. A `for` loop iterates over the length of the array. The value of `ordered_attr[i]` is stored in a new temporary array `nbrs[]`. A check is performed to compare the current element for `ordered_closest[]` and its consecutive one. If the two do not match, it means that the end of the current set $S(v)$ has been reached. The temporary array is assigned as `@nbrs` attribute to the tree node specified in `ordered_closest[i]`. The temporary array is cleared for the next set.

Algorithm 1 `fast_search` pseudo-code

```
1  foreach influencing point
2    find closest tree node
3    append the value in an array
4
5  //each tree node can appear more than once.
6  sorted[] = sorted array, contains the closest tree nodes
7  sorted_attr[] = get indices sorted sequence
8
9  for(int i = 0; i < len(sorted); i++)
10   read i_th value of sorted_attr[]
11   append to a support array
12
13   if(sorted[i] != sorted[i+1])
14     /the support array contains the full list
15     //of influencing point for the sorted[i] tree node
16     set support array as @nbrs list for sorted[i] tree node
17     empty support array for next tree node
```

The algorithm proved to be dramatically faster compared to its basic implementation. The reader can compare the performances by switching the input for the switch node in the "`grow_alg/find_neighbour`" subnetwork. The approach undertaken here also improved Runions algorithm by reducing the number of computation required to achieve the result.

4.4.2 Set fertility for the tree nodes

Extending Runions' algorithm further, the user can specify the maximum count of branches that a node can generate, and from which generation to start the branching process. This node gives a better control over the final tree structure as it allows to achieve a sparser result when reducing the count of children per node. By choosing the generation from which to start branching, the user can enhance the visual impact of the main trunk structure. The *Wrangle* "`set_fertility`" operates over this constraints: the eligible nodes for the production of a new tree node are flagged as fertile. Another check is here performed: in the case the node has already reached the maximum count of children

allowed, the node is flagged as dead and becomes inactive for all the remaining iterations of the colonization algorithm.

4.4.3 Calculate new born direction

The "`newborn_dir`" *Wrangle* runs over each tree point that is fed as a first input geometry. If the node results to be fertile, the snippet proceeds to retrieve the identifiers of the influencing points to that tree node stored in the `@nbrs` array attribute. The set of all the influencing points is fed to the *Wrangle* as a second input geometry. Having the identifiers for the influencing points and the influencing points geometry as input, the algorithm can use the identifiers as lookup keys to retrieve the position of the influencing points. The growth direction for the future newborn is then calculated using equation (7). The normalized equation (8) result is stored in the vector attribute `@dir` of the tree node. This way, the parent node is in charge of holding all the information needed to later create and initialize correctly the newborn nodes. Before moving on to the next step of the network, the snippet checks for degenerate cases for which the magnitude of the `@dir` vector is 0, which happens whenever the `@nbrs` list of influencing points is empty. These nodes are declared dead and their fertility flag reset to 0.

Algorithm 2 Newborn direction

```
1 foreach tree node
2   if node is fertile
3     fetch list of influencing points
4     foreach influencing point
5       direction += normalize(inf_point_pos - tree_node_pos)
6       new_direction = normalize(direction)
```

4.4.4 Creation of the new node

At this stage, the network has gathered all the information needed to create the new generation of tree nodes. The *Wrangle* "`create_new_node`" runs over each fertile node of the tree structure. The position of the new tree node is determined and the point created using the `addpoint()` Houdini VEX function. The `_initializeAttribute()` function is then invoked to take care of all the parameters of the newborn that will make it a viable new tree node. The function takes as parameters the `@ptnum` of the newborn and the `@ptnum`, `@dir` and `@gen` of the parent node. With these information it populates the attributes for the newborn as described in 3. The colour is initialized as green because the new node does not have any children yet. This information is later used in the network to find the points on which to scatter leaves. The `@id` is temporarily set to a default value and set to its proper value later on in the network. Compared to Runions' algorithm, the project offers a new approach.

Algorithm 3 Initialization of newborn tree node

```
attr_dead = -1;
Cd = {0, 1, 0};
fertile = 0;
generation = parent_generation + 1;
growdir = parent_dir;
n_dead = 0;
parent = parent_id;
type = "node";
id = -1;
```

The original algorithm kept the distance D between consecutive tree nodes as a constant, but in nature younger shrubs grow shorter than the previous ones. In an attempt to reproduce this natural phenomena, this projects implements a scaling algorithm that progressively reduces the D length value across generations. The scaling factor is represented as a ramp that the user can manipulate to achieve the desired look.

To set the newborns' `@id` correctly, these points are separated from the other tree nodes. To set the `@id`, the network relies on the total count of tree points as showed in 4.

Algorithm 4 Set newborn id

```
foreach newborn
  id = treepointcount + current iteration value
  update treepoint count as treepointcount + 1
```

The reason behind this particular implementation can be found in the peculiar nature of the Houdini wrangle nodes. The geometry the snippet can access is only the one that is wired in as input to the node. As explained in section 4.1, any changes the snippet produces to the geometry will not be reflected on the geometry until the node is cooked and fed as input to the next node in the network. Following this logic, the update of the tree point count, even if it gets changed every iteration, will only be written on the geometry when the snippet has iterated over all the points. The tree point count value used to assign the new `@id` is therefore the same for each newborn.

As a final step, the parents' `@kids` list attribute has to be updated with the newborns' `@id`. The separated tree nodes are fed as first input geometry to the "`update_parent`" *Wrangle*. The *Wrangle* will also take as a second input geometry the newborn points. For each of the newborns, the snippet will retrieve the `@parent` (which is the `@id` of that point) value. Given the parent's `@id`, the `_findPt()` function returns a handle to the parent node for which the currently processed newborn is appended to the list of children. Since the parent now has at least one child, its colour is set to yellow. To speed up the performances

further, the old tree points are divided further so that only the fertile ones, therefore the parents, are fed in the "`update_parent`" *Wrangle*.

Once this setup is completed, the tree nodes are joined back together.

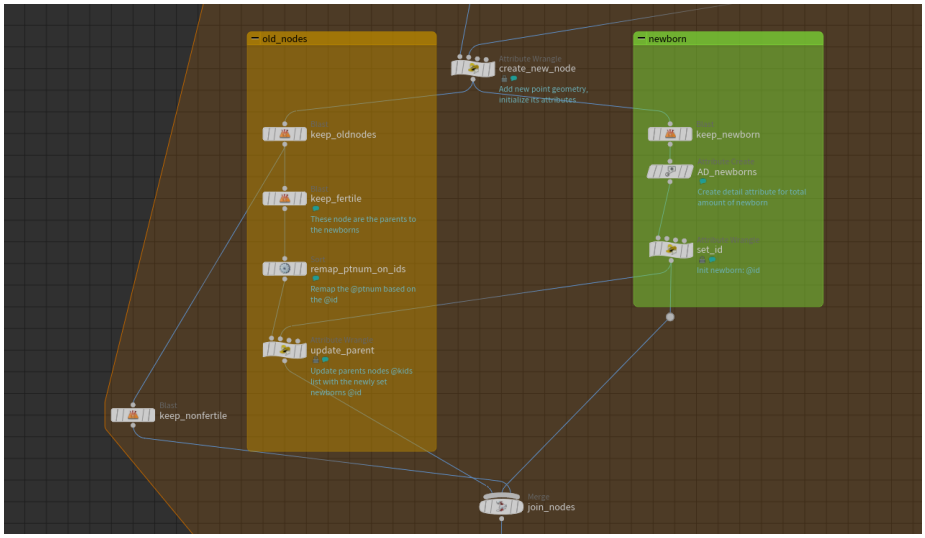

Figure 11: Newborn network

4.4.5 Remove attraction points

As a final step of the growth algorithm, as described in section 3.2.2, the network has to remove the attraction points that are closer to the tree points than the kill distance dk. Runions explains that greater values for this parameter yield to increasingly sparse crowns. In his algorithm, the dk is a constant of the whole growth process. The tree branching structure presented in his research resulted to be too sparse when it comes to small twigs close to the tree crown shape. In an attempt to improve the appearance of such small twigs, the algorithm presented in this project scales the value for dk based on a ramp. The user can therefore manipulate its value and decrease it for younger twigs to achieve better density and more details near the crown surface.

4.4.6 Preparation to next iteration

Some attributes of the tree nodes have to be reset to default values in order for the next iteration to start in a clean state.

- `@dir` is set back to $0, 0, 0$
- `@nbrs` is emptied
- `@fertile` flag is set to 0

4.4.7 Extra controls

Even though the overall algorithm can produce a wide range of different branching structures, the project provides some additional controls:

- **pruning probability**: starting from a user-defined generation and killing probability threshold value, a random number is calculated for each tree node and compared to the killing probability. If the random number is smaller than the threshold, the node is set to non fertile. This small check gives more controls over the density of the branches.
- **pruning by distance**: the user can define a minimum distance from the tree crown geometry. Whenever the distance of a tree node from the crown shape is less than that value, the node is set to dead.
- **fix growing angles**: it may happen that siblings branches grow too close one to each other. A new control checks the angle between siblings branches and modifies the younger sibling's position if the angle is less than the specified threshold.

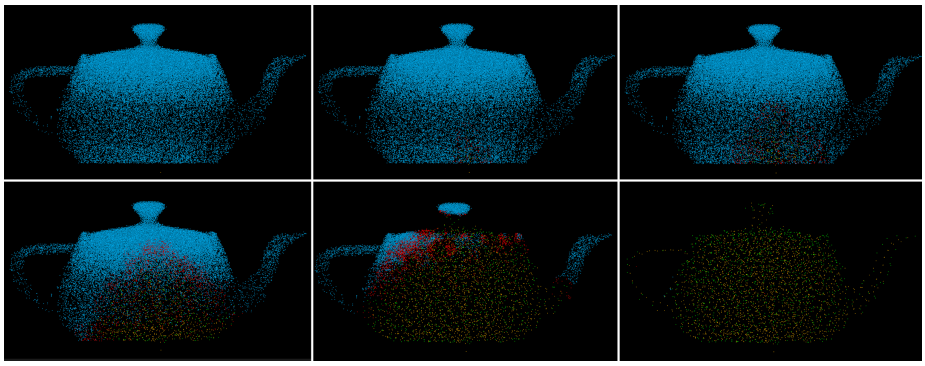

Figure 12: Colonization status after 0, 5, 10, 15, 20, 30 iterations

4.5 Linking the points

The relation between parent and children created during the creation of new points makes it trivial to link the point together to create the tree skeleton. The *Wrangle* "`connect_points`" fetches each point's parent. Using the `addprim()` VEX function a new empty primitive is created. The two points are added to that primitive and the link between them is created.

4.6 Cross section evaluation

In order to create a polygonal mesh around the tree skeleton, the cross section has to be evaluated on a per point basis. The network implements two different algorithms to fulfil this purpose.

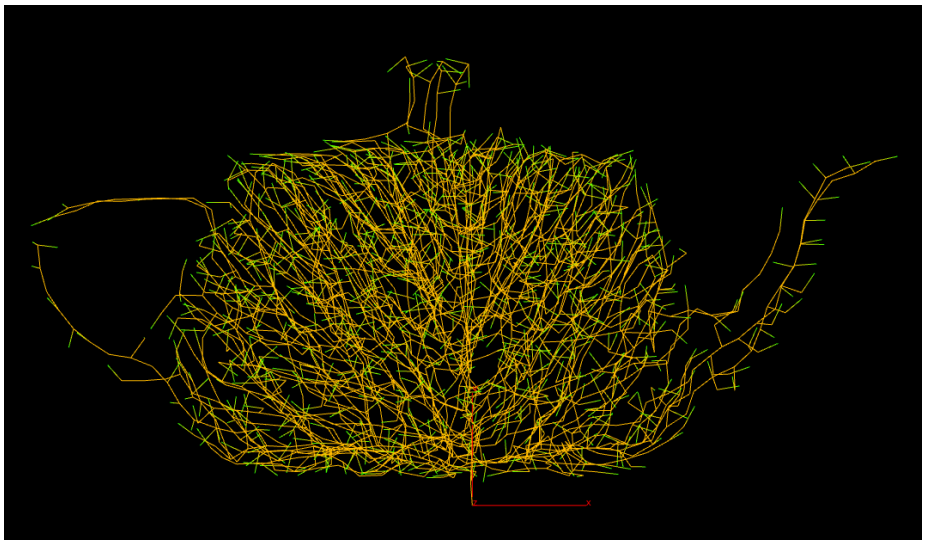

Figure 13: Skeleton creation

4.6.1 Ramp scale

The user is provided with a ramp and a multiplier slider. The multiplier scales the amplitude for the whole ramp while the curve on the ramp controls the scaling factor for each generation of tree nodes. The "`ramp_scale`" relies on the `fit()` function to rescale the generation value for the current point to a value in the range $(0, 1)$. The new width is then computed as the product between the ramp value and the multiplier.

4.6.2 Leonardo's Rule

In his research, Runions relies on Shinozaki et al (1964) to determine the cross-section of the limbs. In this model, known as Leonardo's Rule, the diameter of a limb below a branching point is determined by the combined cross-section of the limbs above. The process assumes that all tip points share the same radius r_0 and proceeds assigning the radii for the other points from the branches' tip towards the tree base. When two branches with different radius r_1, r_2 join at a branching point, the radius r of the supporting limb is given by $r^n = r_1^n + r_2^n$ where n varies between 2 and 3. This work implements the same main formula twice to distinguish the scenario where the roots are loose points and where the roots are points of the user defined trunk structure.

For the loose roots, the tree points are sorted based on the `@generation` value and processed from the highest generation to the smallest one and assigns the @width value. This way, the algorithm manages to traverse the tree from the tip branches to the root.

The implementation of the Leonardo's scale for the second scenario is ad-

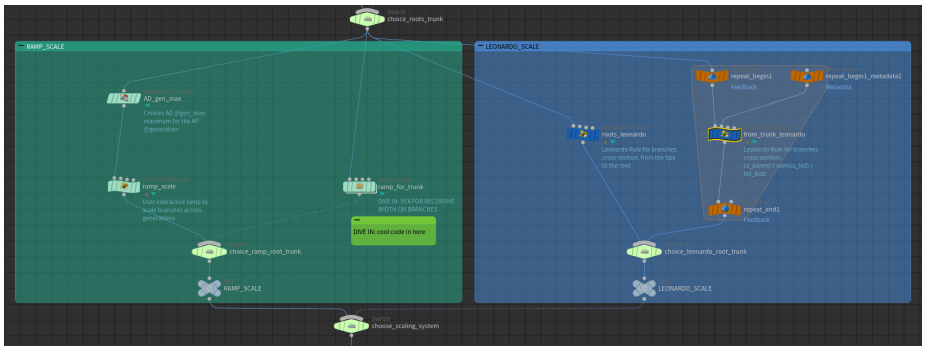

Figure 14: Width setup network

Algorithm 5 Leonardo's Scale for loose roots

```
1  sort tree points based on @generation in an array
2  reverse array to have highest gen values for smaller indices
3  foreach point in array
4    if point has no children
5      @width = user defined value for tip branches
6    else
7      @width = sqrt(sum pow(@width) of all children in @kids)
```

dressed in the following section 4.6.3

4.6.3 Special case: trunk

For a custom trunk input the original ramp algorithm cannot be used as the scaling factor is calculated based on the generation number populated during the growth algorithm. Unfortunately, the trunk points do not have a generation number to rely onto. Furthermore, the user might want to have a better control over the main branches structure as they play a key role in the final appearance of the tree. As a consequence, the network first sets the width for the trunk and on a later stage takes care of the smaller branches generated by the growth algorithm. The trunk structure is fetched again from its original source.

As for initializing the `@kids` attribute as discussed in section 4.3.3 the *Wrangle* node works on the points of each primitive at the time. The Houdini loop blocks are in charge of iterating over the primitives of the trunk structure. The *Wrangle* creates a ramp and a multiplier for the user to manipulate the width at will. Before assigning the `@width` value, a check is performed on the `@width` value for the first point of the primitive. If the `@width` value is -1 (the default one), the primitive being processed is the main trunk. The algorithm can assign the `@width` for the next points of the primitive simply by multiplying the ramp value and the multiplier factor. If the `@width` is different than -1, the primitive represents a secondary branch and its first point is the branching joint

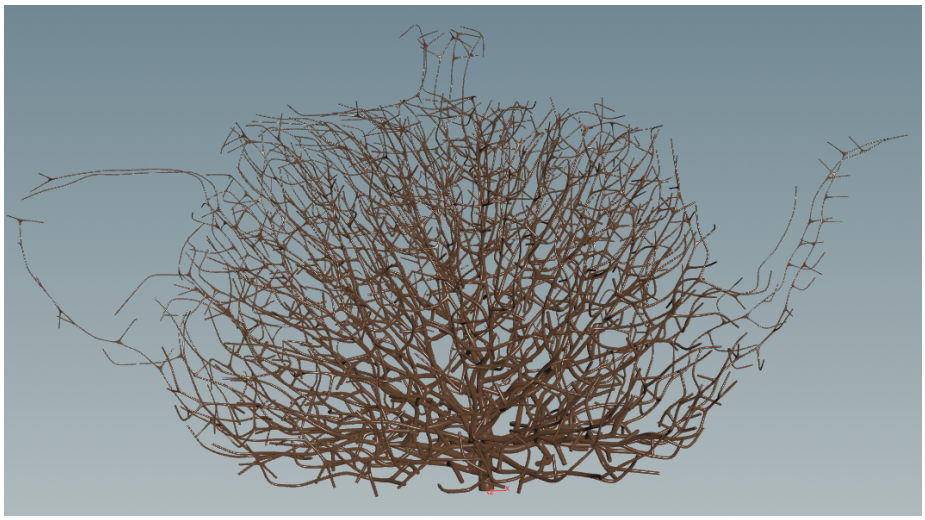

Figure 15: Ramp scale

between the current branch and its supporting limb. Its width is therefore been already set as part of the other branch. To blend one branch into its secondary one, the next nodes in the primitive are set similarly to the main trunk except that the user-defined multiplier is substituted by the first node's width.

Algorithm 6 Trunk `@width` initialization

```
1  foreach primitive of the user input trunk structure
2    fetch all the points of the primitive
3    if the primitive is the main trunk
4      foreach point in the primitive
5        assign @width value based on user defined ramp
6    else (is a branch)
7      foreach point in the primitive
8        //joint = point between curr branch and parent limb
9        @width = ramp * @width_jointof the joint
```

As next step the width for the branches generated by the growth algorithm has to be set.

For the ramp scale, the main idea of the artistic-directable ramp is maintained although blending the branches with the trunk structure requires some further manipulation. If for the previous case all the root points shared the same user-defined `@width`, in this case the width of each node acting as a root has already been set. The initial condition for this algorithm is represented by a set of tree structures with trunk points as roots. Each tree has to scale the `@width` of the branches according to the ramp but each of them should use the

Figure 16: Leonardo's scale

`@width` of its own root point as a multiplier to achieve the desired blending effect. This would be trivial with a recursive algorithm: ideally one should iterate on each child of the root and set the `@width`. Each child then acts as root of its own sub-tree and the operation is repeated. Unfortunately Houdini VEX does not support recursive function calls at the current time. The algorithm has therefore been converted from recursive to iterative by using a stack data structure as a support. The root node stores its children in the stack. A `while` loop iterates over the stack. For each iteration, the last added node is removed from the stack and is processed. After setting its own `@width` value, the node pushes its children in the stack and a new iteration begins. The `while` loop only stops when the stack is empty. Using the stack as support data structure, the described process successfully simulates a depth-first recursive algorithm and overcomes the updating geometry problem explained in section 4.1.

Algorithm 7 Trunk `@width` non-recursive initialization

```
1  foreach root point
2    set @width of root point based on user ramp
3    find children of root point
4    push children on the stack
5
6    while(stack not empty)
7      pop last element from the stack
8      set its @width based on user defined ramp
9      find its children
10     push children on the stack
```

For the Leonardo's scale, traversing the tree from the tip branches to the roots is not a viable option since trunk points already have a `@width` set. This work tries instead to implement the same main formula presented in section 4.6.2 by processing the tree limbs from the roots to the tip branches. The initial value of `@width` for the roots points is initialized alongside all the other points of the trunk structure. Having the first nodes' width set, the algorithm iterates over every point, in order, finding the children of the node and setting their width based on its own width and the total count of children. This approach works in this specific implementation because the tree nodes have been created in order with increasing `@id` numbers. Therefore, when iterating on them, each node's width has always already been set by a previous node. Once again, the nature of the *Wrangle* node does not allow to easily propagate the width from parents to children due to the cooking and updating issue explained in section 4.1. To overcome this problem, the *Wrangle* node only processes one point at the time while the iteration is implemented with the loop blocks native in Houdini. Since the blocks are in charge of the iteration, the *Wrangle* gets cooked and the geometry updated.

Algorithm 8 Leonardo's Scale for roots based on trunk structure

```
1  foreach tree point
2    //non root points
3    if @width has not been set already
4      find parent @width
5      find total number of siblings
6      calculate @width with Leonardo formula
7      set tree node @width
```

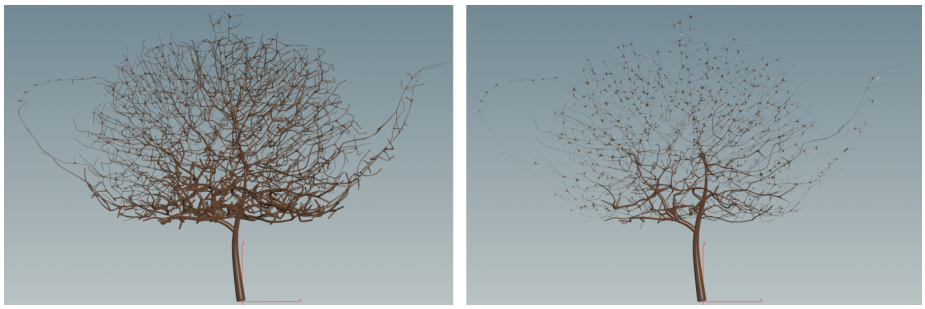

Figure 17: Ramp scale and Leonardo's scale for custom trunk structure

4.7 Trunk Skinning

After building the tree structure, each branch is re-sampled to gain more control over the final look. The *PointJitter* SOP helps moving from straight branches

curves to a more gnarled and varied silhouette. Its effect gets easily out of control causing the geometry to change too drastically. To counterbalance the sharp angles generated by the *Jitter*, the *Smooth* SOP interpolates these angles and achieves a more natural look. The *PolyWire* SOP creates the polygonal mesh around the tree skeleton using the `@width` attribute set in section 4.6 as cross-section value. A second *Smooth* SOP interpolates the thickness values at the branching nodes. As a final step, the user can decide whether to color the trunk with a *Color* SOP or input a custom shader.

4.8 Leaves

In nature, trees produce leaves only at the very tip of the branching structure. To replicate this phenomenon the network isolates all those nodes of the tree who do not have any children. The leaves are then copied on those points by using the *Copy_Stamp* node. For *Copy_Stamp* to work properly, the normal direction has to be set for the tree nodes. The *Point* SOP activates an edge force on these point. The edge force is directed along the edge direction which is stored in the `edge_dir` built-in variable for points in Houdini. With a *VOP Subnetwork* the `edge_dir` is copied as value for the normal direction of the points.

The user can decide to either use the default geometry or to input a custom model for the leaves. For the default ones, some *Copy_Stamping* expressions randomize the roll and pitch within a set range of values on a per instance basis. For the custom geometry instead, a control is given to specify the number of leaves per tree node. The *Copy_and_Transform* evenly spreads the leaves radially by dividing 360 degrees by the number of leaf instances specified by the user.

The user can choose whether to use a plain color or apply a custom shader. While the default geometry is ready to shade, the user has to make sure to prepare the custom geometry for the shading process using the *UV_Project* Houdini node before feeding the leaf object as input to the network.

5 Problems and attempted solutions

5.1 Determine S(v) set

At very early stages of production, the $S(v)$ set has been implemented as a Houdini group. Groups in Houdini are similar to a boolean variable associated to each point of the geometry that specifies the membership of the point to the group. Having n tree nodes, the system would have to store n groups. After a few couple of iteration however, the number of different groups resulted to be inefficient to use. At this stage the sets were also processed in a *ForEach* subnetwork that unfortunately caused issues of duplicated geometry which lead to frequent crashes of the scene. As a solution, the current data system relying on the attribute `@nbrs` instead of the groups was adopted.

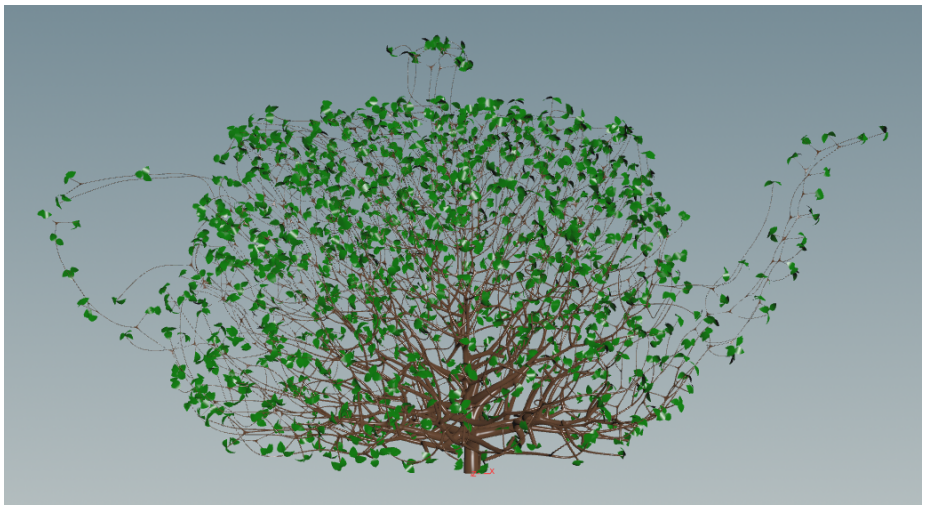

Figure 18: Leaves scattering

5.2 Branching collisions

For some cases, especially for the very first generations, it may happen that branches tend to grow too clustered together. The `fix_newborn_dir`, as explained in section 4.4.7, takes care of these particular scenario moving the newborns to a minimum distance angle from their closest sibling. To achieve this result, the algorithm presented in section §7.1 was used.

5.2.1 *IsoOffset*: from polygons to volumes

The *IsoOffset* node converts the user-input crown shape to a volume that can be seeded with attraction points. Unfortunately the conversion leads to some artefacts such as horizontal lines exceeding or carving the original geometry. Different solution have been tried such as changing the volume conversion method but none of them proved to be universally successful.

5.2.2 Scaling

The approach to Leonardo's scaling system adopted in this work for the user defined trunk scenario can lead to chopped branches. With the scaling algorithm starting from the roots and proceeding to the top from a user-defined initial `width` value, branches can looked truncated when they die due to lack of space or because of the pruning system. Unfortunately replicating Runions approach and traverse the tree from the tip branches to the roots would not work properly because the trunk already has a custom thickness set. The blending between automatically generated branches and user-input ones would probably have to be achieved manually.

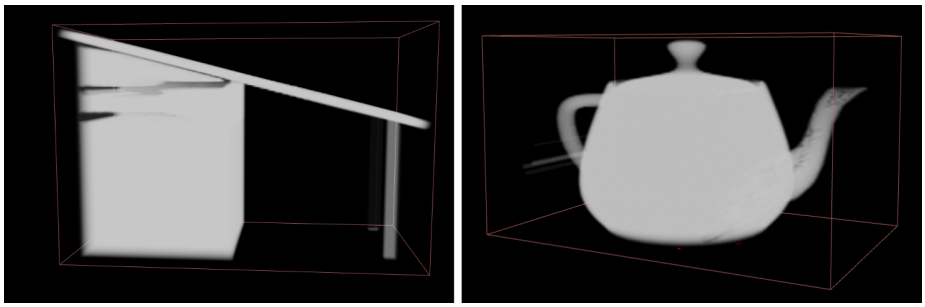

Figure 19: Artefacts caused by the *IsoOffset* node

A similar issue arises for the ramp scaling system: as each branch within a generation will have the same cross-section, the terminal segment of branches dying in early generations will be of the same width as non-terminal segments of more long-lived branches. As far as the ramp scale is concerned, a solution could not be found during this project and the problem remains open for future work.

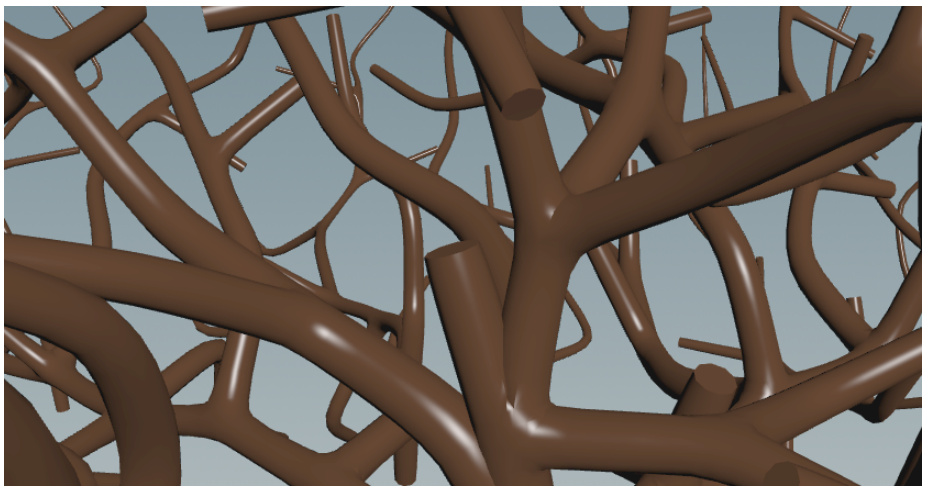

Figure 20: Scaling chopped branches

6 Performances

An iterative system that handles a huge amount of geometries can result to be extremely slow. Even though the performances were not the primary focus of this project, a considerable effort has been made to make the network as fast as possible. The reason for investing in the performances relies in the nature of the

tool: the project presented aims to be an easy and quick-to-setup tool to create procedural and art-directable trees. If the tool were to require several minutes to produce a result, the user would find it hard to compare two successive attempts at tuning the parameters to get the desired look. At very early stages of development, cooking a 30-iterations tree took up to 40 minutes. Grouping the geometry points involved in the space colonization algorithm into nodes and attraction points helped speeding up the execution time.

Even though this change improved the performances, the network still spent an excessive amount of time processing the `find_neighbours` section 4.4.1 and the `setting_id` section 4.4.4 algorithms. Dividing the tree nodes respectively into newborns, parents and old tree nodes improved the performances dramatically since the search loop was only to iterate over a few hundred points instead of a few thousand.

The biggest speedup was however achieved by optimizing the algorithm for the research of neighbours. In its first version, each iteration of the growth algorithm required the same node to be cooked over 20000 times in order to update the geometry points. The new approach presented in section 4.4.1 made the real difference in terms of the final execution time.

Even though the execution time still heavily depends on parameters such as the total number of attraction points, iterations, D, di, dk, the average cooking time stabilized around 15 seconds for 40-iterations trees.

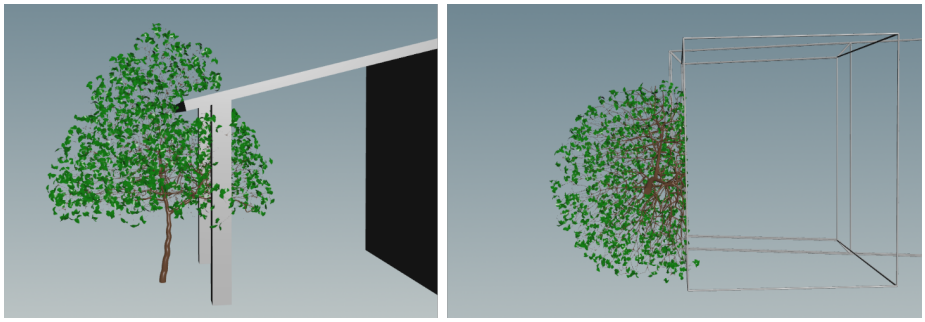

Figure 21: Tree object avoidance

7 Conclusions

The described implementation successfully reproduces the approach presented by Runions. The volume of the tree crown is represented as a set of attraction points which get removed as the branches approach them. Parameters specify the granularity of the empty space, the search distance from which the branches can sense it and how deep the empty space sensed can be penetrated by the branches. Further controls manipulate the distribution of the attraction points allowing for a greater density of branches near the surface of the tree crown.

Information of the hierarchy of branches offers control over their different sizes. The final result produced a realistic branching structure even without any post-processing of the skeleton. In some cases, post-processing operations such as resampling the branches or applying random noise to the tree points can lead to a more natural look of the final artefact. The project builds on Runion's research to add more controls over the branching structure by varying the parameter values on-the-fly and introducing small corrections such as the pruning system and the angle correction. A fully art-directable scaling system for the branch size has been developed alongside the one proposed in the original paper.

The tool successfully integrates a user-defined main skeleton with the geometry produced by the space colonization algorithm fulfilling the purpose of having a tool for art-directable procedural vegetation. The implementation also proved to be able to handle a wide range of scenarios: it can grow trees from one or multiple sources at once, a custom trunk can be used as starting point for the space colonization algorithm, the generated geometry reacts to the surrounding obstacles by either avoiding them or using them as objects to crawl onto in a vine-like behaviour. When tested for rendering, the produced mesh unwraps correctly.

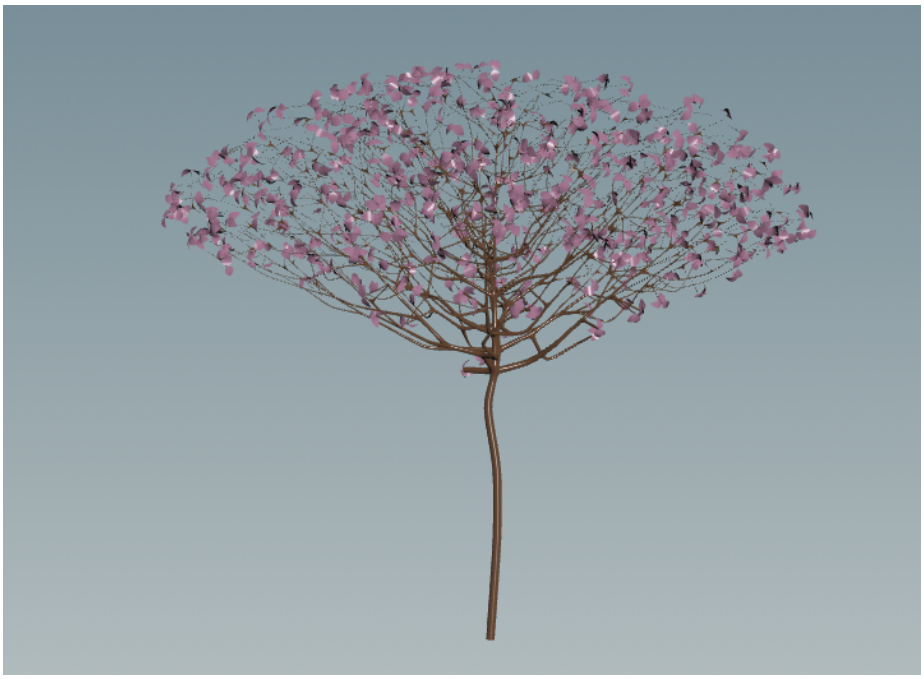

Figure 22: Tree with conical crown shape

7.1 Future work

The system successfully simulates the competition for space that characterizes tree structures. As a future enhancement, competition for light could be introduced as well. The light influence could probably solve the issue of having a too sparse foliage. To address the same problem, the system could use a proxy or billboards instead of actual geometry that expands only at render time. That would maintain the viewport responsive while improving the final look.

Other problems remain open for future research. The method proposed in this project creates a believable mature tree. The project could be significantly improved by being able to simulate the growth process that leads from young shrubs to the mature trees.

A simulation of breeze through the leaves or wind interacting with the tree would improve the overall result and could be accomplished using the Houdini *Wire* solver.

For scenes populated by a large number of plants, the current geometry might be too heavy. Implementing a procedural LOD for the polygonal mesh and an automatic pruning of the small twigs based on the distance from the camera could make the viewport more responsive and the rendering quicker.

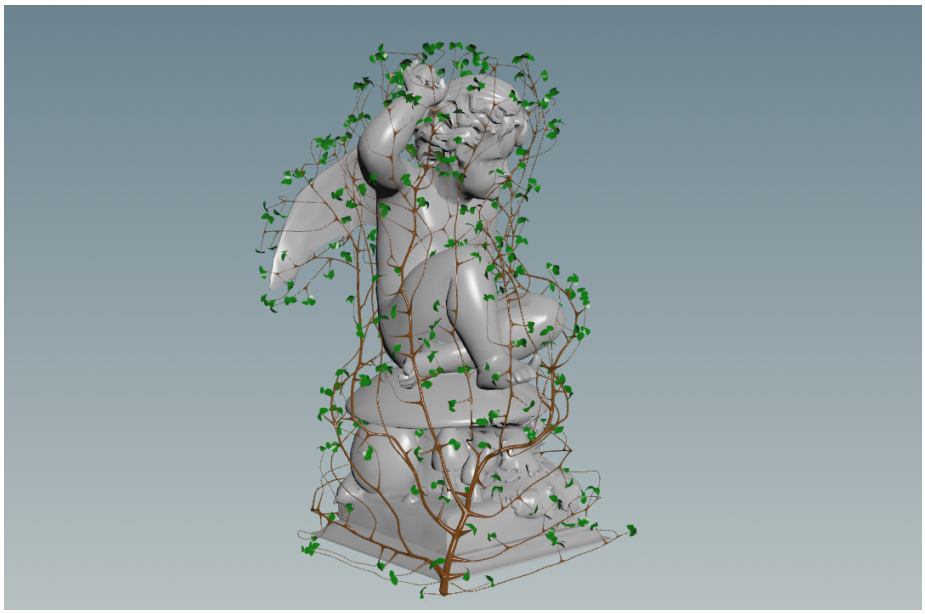

Figure 23: Vine-like growth

Appendix

L-Systems

L-Systems (Lindenmayer Systems) were conceived as a theory for plant development. The key idea of L-System is rewriting. An L-System is a rewriting system and a formal grammar. A formal grammar is a set of rules for rewriting strings in a formal language. A rewriting system defines, in this case, methods for replacing substrings with other symbols. An L-System consists of:

- Alphabet: set of symbols used to create strings. These symbols can either be replaceable or constant.
- Axiom: string or symbol to start the production, made of elements of the alphabet.
- Production rules: rules to replace and expand symbols or substrings with other sequences of symbols.

In L-Systems, production rules are applied in parallel and simultaneously in each generation. In this sense, L-System differ from formal languages in which only one rule is applied per iteration.

L-Systems can be classified as:

- Context-free: if every production rule only refer to an individual symbol.
- Context sensitive: production rules refer not only on individual symbols but also to groups of symbols.
- Deterministic: the production set provides one and one only production rule for each symbol. An L-System that is deterministic and context-free is usually referred as D0L System.
- Stochastic: several production rules are given for each symbol, each iteration one of them is chosen based on a certain probability value.

A visual representation of the produced string can be achieved by employing the "turtle" interpretation, that translates strings into graphical commands. The turtle interpretation is based on the following commands:

- F: move forward a step of length d. Draw a line between current and previous positions.
- f: move forward a step of length d without drawing a line.
- +: turn left by an angle of δ.
- -: turn right by an angle of δ.

Given their recursive nature, L-Systems can be used to generate self-similar fractals.

Fractals

The term "fractal" derives from the latin *fractus*, which means "broken". The word was firstly introduced by Mandelbrot (1982). In Mandelbrot's words, a fractal is

> "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole."

The main features of a fractal are then, as per definition:

- Self-similarity: each part is a scaled size copy of the whole.
- Fine structure at small scale.
- Recursive definition.

Angle correction

Whenever two sibling branches grow too close together, the network provides a way of moving the youngest one from the older sibling so that the angle between them is bigger or equal than a certain user defined value. We call $\hat{a}$ the "moving" vector and $\hat{b}$ the "still" vector. $\hat{a}$, $\hat{b}$ are defined as follows.

$$\hat{a} = (a_x, a_y, a_z) \tag{11}$$

$$\hat{b} = (b_x, b_y, b_z) \tag{12}$$

$$\mid \hat{a} \mid = \mid \hat{b} \mid = 1 \tag{13}$$

It is known that $\cos\vartheta = \frac{\hat{a}\cdot\hat{b}}{|\hat{a}||\hat{b}|} = \hat{a}\cdot\hat{b}$ from which follows that

$$\vartheta = \cos^{-1}(\hat{a}\cdot\hat{b}) \tag{14}$$

One can now decompose $\hat{a}$ as

$$\overrightarrow{a}_{\parallel} = \cos\vartheta\hat{b} \tag{15}$$

$$\overrightarrow{a}_{\perp} = \hat{a} - \overrightarrow{a}_{\parallel} \tag{16}$$

It is also known that

$$\tan\vartheta = \frac{a_{\perp}}{a_{\parallel}} \tag{17}$$

The user defines the new angle between the two vector as ϑ_0. One can therefore calculate a new $a'_{\perp}$ that satisfies the following equation

$$a'_{\perp} = a_{\parallel} \tan \vartheta_0 \tag{18}$$

By combining equation (17) and equation (18)

$$\overrightarrow{a}'_{\perp} = \frac{a_{\parallel} \tan \vartheta_0}{a_{\perp}} \overrightarrow{a}_{\perp} = \frac{\tan \vartheta_0}{\tan \vartheta} \vec{a}_{\perp} \tag{19}$$

One can therefore define the new position $\vec{a}'$ as

$$\overrightarrow{a}' = \overrightarrow{a}_{\parallel} + \overrightarrow{a}'_{\perp} \tag{20}$$

and its unit vector as

$$\hat{a}' = \frac{\overrightarrow{a}'}{a'} \tag{21}$$

Flowers

To enrich this project another small and simple tool has been developed to create procedural flowers., taking inspiration by Perez (2017) work on fern plants and Dalvi (2015) work on vines. The network for this tool is divided in three big areas, respectively for the stem, the petals and the stamen.

Stem

The network provides a full customization on the stem geometry. The user can either modify the default line geometry that acts as a skeleton for the stem or input his own line. The *Jitter* node is used to manipulate the stem points by adding custom noise on their position so that the final artefacts looks more natural. Since the *Jitter* displacement is based on a random seed, that seed can be used to create different models for the same flower specie when populating a scene. A control is given so that the stem can be bent at will. In a process similar to the one described in section 4.6.3 the cross-section of the stem is function of the `@ptnum`, a ramp then determines the value for the cross-section of the specific point. The *PolyReduce* node can change the total count of polygon for the stem geometry. In a future, that node could be used to generate procedural LOD based on the distance from the camera looking at the scene. As a final refinement, the user can benefit from a *Sculpt* tool to add more refinement to the stem polygonal mesh.

Stamen Base

The stamen base has been modelled as a general shape that can produce visually appealing results for a wide variety of flower species. If in need of more refinement, the user can either sculpt more subtitle details with the *Sculpt* tool or use a custom geometry. As for the stem, a *PolyReduce* node is provided to allow for a future procedural LOD.

Stamen Filaments

The anatomy of flowers can vary greatly if we compare, for example, a sunflower with a lily. Whereas a sunflower model can be easily achieved using the modelling of the stamen base discussed in the previous paragraph, a lily-like flower has very long stamens at the centre of the petal crown. To reproduce such feature the stamen filaments have been modelled alongside the stamen base. The geometry, as for the stem, starts from a simple curve. Custom bending is also available to the user to change the stamen appearance at will. A *Copy_Stamping* expression manages to create slightly different instances of such filament for a total amount defined by the user.

Petals

As for every part of the network, the user can decide to either use the default geometry or select a custom model. Petals are scattered on top of the stamen base. The user, based on the stamen base geometry, can decide a minimum level and a maximum level for the petals to be scattered on based on the angles as follows. A value ϑ on a per point basis is calculated as

$$\vartheta = tan^{-1} \frac{p_y}{\sqrt{p_x^2 + p_z^2}} \tag{22}$$

The angle is compared to a minimum and maximum value defined by the user. The points with ϑ falling in the specified range are used as sources for the petals.

References

Cieri, S., Muraca, A., Schwank, A., Preti, F., Micilotta, T., 2016. The Jungle Book: Art-Directing Procedural Scatters in Rich Environments. *DigiPro '16*, 23 - 23 July Anaheim, CA. New York: ACM, 57-59.

Dalvi, R., 2015. *Creating animated ivy in Houdini - Tutorial 001.* Video. youtube.
Available from: https://www.youtube.com/watch?v=Uhr3HjHW_oU
[Accessed 21 May 2017].

Feriani, M., 2017, *Background Research for Ice Growth Simulation CGI Technologies*, MSc CGITools Report.

Filmmakerperez, 2017. *Houdini Tutorial - Procedurally Modeling 3D Plants.* Video. youtube.
Available from: https://www.youtube.com/watch?v=K0qKWRBgwCY&t=1198s
[Accessed 16 May 2017].

Honda, H., 1971. Description of the form of trees by the parameters of the tree-like body : effects of the branching angle and the branch length on the shape of the tree-like body. J. Theoret. Biol, 31, 331-338.

Power Up Your Houdini Skills with SideFX, 2017. Making the Last Bastion. *3D World Magazine*, 20 June 2017,
Available from: https://issuu.com/futurepublishing/docs/houdini_issuu
[Accessed 24 July 2017].

Ijiri, T., Owada, S., Igarashi, T., 2006. The Sketch L-System: Global Control of Tree Modeling Using Free-form Strokes. *6th International Symposium Smart Graphics*, 23-25 July, Vancouver, Canada. 138-146.

Keim, H., Simmons, M., Teece, D., Reisweber, J., Drakeley, S., 2016. Art-Directable Procedural Vegetation in Disney's Zootopia. *SIGGRAPH '16*, 24-28 July 2016. Anaheim, CA: ACM

Mandelbrot, B. B., 1982. *The fractal geometry of nature.* W. H. Freeman and Company.

Melson, T., 2016. Can't See The Jungle For The Trees. *SIGGRAPH '16 Talks*, 24 - 28 July Anaheim, CA. New York: ACM

Oppenheimer, P., 1986. Real Time Design and Animation of Fractal Plants and Trees. Computer Graphics Proceedings, Annual Conference Series, 18-22 August, Dallas. Dallas, Texas: ACM SIGGRAPH, 55-64.

Prusinkiewicz, P., Lindenmayer, A., 1990. *The Algorithmic Beauty of Plants.* New York: Springer-Verlag.

Rodkaew, Y., Chongstitvatana, P., Siripani, S., Lursinsap, S., 2003. Particle Systems for Plant Modeling.

Runions A., Fuhrer M., Lane B., Federl P., Rolland-Lagan A.-G., Prusinkiewicz P., 2005. Modeling and visualization of leaf venation patterns. *ACM Transactions on Graphics*, 24(3), 705-711.

Runions, A., Lane, B., Prusinkiewicz, P., 2007. Modeling Trees with a Space Colonization Algorithm. In: Ebert, D., Mérillou, S., *NPH'07 Proceedings of the Third Eurographics conference on Natural Phenomena*, 4 September 2007, Prague. Aire-la-Ville: Eurographics Association, 63-70.

Schwank, A., James, C. J., Milciotta, T., 2016. The Trees of The Jungle Book. *SIGGRAPH '16 Talks*, 24-28 July 2016. Anaheim, CA: ACM

Shek, A., Lacewell, D., Selle, A., Teece, D., Thompson, T., 2010. Art-directing Disney's Tangled procedural trees. *SIGGRAPH 2010 Talks*, 26 - 30 July Los Angeles. New York: ACM, 53.

Shinozaki, K., Yoda, K., Hozumi, K., Kira, T., 1964. A quantitative analysis of plant form — the pipe model theory. *Japanese Journal of Ecology*, 14 (3), 97–104.

Soares, O., Moser, M., Aalbers, F., 2016. Vegetation Choreography in The Good Dinosaur. *SIGGRAPH '16 Talks*, 24 - 28 July Anaheim, CA. New York: ACM

SideFX, 2017. *Attribute Wrangle.* sidefx.com,
Available from: http://www.sidefx.com/docs/houdini/nodes/sop/attribwrangle [Accessed 14 August 2017].

Wang, C., Yang, K., Han, D., 2008. New Modeling Method for Trees. *2008 International Conference on Advanced Computer Theory and Engineering*, 20-22 December 2008 Phuket. Los Vaqueros Circle: Institute of Electrical and Electronics Engineers (IEEE), 633-637.

Weber, J., Penn, J., 1995. Creation and Rendering of Realistic Trees. *22nd International ACM Conference on Computer Graphics and Interactive Techniques*, 6 - 11 August 1995 Los Angeles. New York: ACM, 119 - 128.

Wikipedia, 2017. *Graph theory.* wikipedia.org,
Available from: https://en.wikipedia.org/wiki/Graph_theory
[Accessed 14 August 2017].

Wikipedia, 2017. *Formal grammar.* wikipedia.org,
Available from: https://en.wikipedia.org/wiki/Formal_grammar
[Accessed 14 August 2017].

Code

Algorithm 9 distance_from_bounding_object

```
// calculate distance from input1 (bounding object)
// assign to attribute @dist
f@dist = xyzdist(1, @P);
```

Algorithm 10 radial_distribution

```
// density as function of the radial distance
// of bounding object
int min_dist = detail(0, "min_dist");
int max_dist = detail(0, "max_dist");

float fit = fit(f@dist, min_dist, max_dist, 0, 1);
float probability = chramp("probability", fit);

i@dead = 0;

// if the rand number less than the user defined probability
// node is set to dead and removed the next node
if(rand(@ptnum) < probability)
{
  i@dead = 1;
}
```

Algorithm 11 AP_height

```
// save y coordinate for the point to the @height attribute
f@height = @P.y;
```

Algorithm 12 vertical_distribution

```
// density as function of the Y coordinate
int min_Y = detail(0, "min_Y");
int max_Y = detail(0, "max_Y");

float fit = fit(@P.y, min_Y, max_Y, 0, 1);
float probability = chramp("probability", fit);
i@dead = 0;

// if the rand number less than the user defined probability
// node is set to dead and removed the next node
if(rand(@ptnum) < probability)
{
  i@dead = 1;
}
```

Algorithm 13 trunk_kids

```
// fetching info about what primitive are we working on
int prim = detail(1, "iteration");
int points[];
int kids[];
int pt;
int child;

points = primpoints(0, prim);
// the last node will not have any children to set
for(int i = 0; i < len(points) - 1; i++)
{
  // set as child for current node
  // the next one in the list of points
  pt = points[i];
  child = points[i+1];
  kids = point(0, "kids", pt);
  append(kids, child);
  setpointattrib(0, "kids", pt, kids, "set");
}
```

Algorithm 14 trunk_parent

```
// find who the kids are for the current node
// and set for them the current node as parent
// PARENT INJECTION
int kids[] = point(0, "kids", @ptnum);
int child;

for(int i = 0; i < len(kids); i++)
{
  child = kids[i];
  setpointattrib(0, "parent", child, @ptnum, "set");
}
```

Algorithm 15 trunk_type

```
// only choose some of the points in the "node" groups
// to be marked as starting nodes for space colonization alg
// the remaining ones are marked as "trunk": they will not
// play any part in the space colonization algorithm
if((inpointgroup(0, "nodes", @ptnum)) &&
   !(@ptnum % int(ch("discretization"))))
{
  s@type = "node";
}
else
{
  s@type = "trunk";
}
```

Algorithm 16 trunk_generation

```
if(s@type == "node")
{
  i@generation = 0;
}
else
{
  i@generation = -1;
}
```

Algorithm 17 trunk_id

```
i@id = @ptnum;
```

Algorithm 18 create_AP_len_nodes

```
f@len = ch("D");
```

Algorithm 19 find_potential_influencing

```
// Set influencing inf attribute to attractors =============
void _setInfluencing(vector p)
{
  int influencing[];
  int attr;

  influencing = nearpoints(0, p, ch("di"));

  for(int i = 0; i < len(influencing); i++)
  {
      attr = influencing[i];
      setpointattrib(0, "inf", attr, 1, "set");
      setpointattrib(0, "Cd", attr, {1, 0, 0}, "set");
  }
}

// MAIN CODE ================================================
int nodes_count = npoints(1);
int influencing[];
vector p;

for(int i = 0; i < nodes_count; i++)
{
  p = point(1, "P", i);
  _setInfluencing(p);

}
```

Algorithm 20 fast_search

```
// find closest tree node to given attr point and return it=
int _findClosestNode(int attr)
{
  vector p = point(1, "P", attr);
  return nearpoint(0, p);
}

// MAIN CODE ================================================
int influencing_count = npoints(1);
int node_pt;
int closest[];
int ordered_attr[];
int ordered_closest[];
int nbrs[];

for(int attr = 0; attr < influencing_count; attr++)
{
  node_pt = _findClosestNode(attr);
  append(closest, node_pt);
}

ordered_attr = argsort(closest);
ordered_closest = sort(closest);

for(int i = 0; i < len(ordered_attr); i++)
{
  append(nbrs, ordered_attr[i]);

  // we have to update the whole list of nbrs at once or
  // or the geometry will only see the last added point
  // (cooking problem)
  if((i == len(ordered_attr) - 1) ||
     (ordered_closest[i] != ordered_closest[i + 1]))
  {
    setpointattrib(0, "nbrs", ordered_closest[i],
                   nbrs, "set");
    nbrs = {};
  }
}
```

Algorithm 21 find_neighbour

```
// find closest tree node to given attr point and return it=
int _findClosestNode(int attr)
{
  vector p = point(1, "P", attr);
  return nearpoint(0, p);
}

// append attractor to nbrs list for the given tree node ===
void _addNearAttractor(int node; int attr)
{
  int nbrs[] = point(0, "nbrs", node);
  append(nbrs, attr);
  setpointattrib(0, "nbrs", node, nbrs, "set");
}

// MAIN CODE ======================================================
int node;
int attr = detail(2, "iteration");

node = _findClosestNode(attr);
_addNearAttractor(node, attr);
```

Algorithm 22 set_n_dead_probability

```
if((i@n_dead == 0))
{
  if(i@generation > ch("kill_gen"))
  {
    if(rand(@ptnum) < ch("probability"))
    {
      i@n_dead = 1;
    }
  }
}
```

Algorithm 23 set_fertility

```
if(i@n_dead == 0)
{
  int kids[] = point(0, "kids", @ptnum);
  if(int(point(0, "generation", @ptnum)) <
     ch("branching_start"))
  {
    if(len(kids) == 0)
    {
      i@fertile = 1;
    }
    else
    {
      i@fertile = 0;
    }
  }
  else if(len(kids) < ch("kids"))
  {
    i@fertile = 1;
  }
  else
  {
    i@fertile = 0;
    i@n_dead = 1;
  }
}
```

Algorithm 24 set_n_dead_distance

```
if(i@n_dead == 0)
{
  float distance = xyzdist(1, @P);
  if(distance < ch("min_dist") &&
     i@generation > ch("generation"))
  {
    i@n_dead = 1;
    i@fertile = 0;
  }
}
```

Algorithm 25 newborn_dir

```
if(i@fertile == 1)
{
  int nbrs[] = point(0, "nbrs", @ptnum);
  foreach(int pt; nbrs)
  {
    v@dir += normalize(point(1, "P", pt) - @P);
  }
  v@dir = normalize(v@dir);

  // if dir == 0 the point will not reproduce anymore
  // it means that its @nbrs list was empty
  if(length(v@dir) == 0)
  {
    i@fertile = 0;
    i@n_dead = 0;
  }
}
```

Algorithm 26 fix_newborn_dir functions

```
// Find ptnum from id ======================================
int _findPt(int id; int numpt)
{
  for(int pt = 0; pt < numpt; pt++)
  {
    if(point(0, "id", pt) == id)
    {
      return pt;
    }
  }
  return -1;
}

// Find sibling growdirection ==============================
vector _getSibling(int pt; int numpt)
{
  int kids[] = point(0, "kids", pt);
  int eldest_id = kids[0];
  int eldest_pt = _findPt(eldest_id, numpt);
  return vector(point(0, "growdir", eldest_pt));
}

// Correct the angle between two siblings ==================
vector _correctAngle(vector a; vector b;
                     float theta, n_theta)
{
  // assumes dir and sib to be both normalized vectors
  vector a_para = cos(theta) * b;
  vector a_perp = a - a_para;
  vector new_aperp = a_perp *
         ((length(a_para) * tan(n_theta)) / length(a_perp));
  vector new_dir = a_para + new_aperp;
  return normalize(new_dir);
}
```

Algorithm 27 fix_newborn_dir main

```
// MAIN CODE ================================================
int kids[];
vector sibling;
vector dir;
float deg_angle;
float rad_angle;
if(i@fertile == 1)
{
  if(int(point(0, "generation", @ptnum)) <
                                        ch("gen_correction"))
  {
    kids = point(0, "kids", @ptnum);
    if(len(kids) != 0)
    {
      sibling = vector(_getSibling(@ptnum, @numpt));
      dir = vector(point(0, "dir", @ptnum));
      rad_angle = acos(dot(dir, sibling));
      deg_angle = degrees(rad_angle);
      if(deg_angle < ch("angle"))
      {
        vector new_dir;
        new_dir = _correctAngle(dir, sibling, rad_angle,
                                radians(ch("angle")));
        setpointattrib(0, "dir", @ptnum, new_dir, "set");
      }
    }
  }
}
```

Algorithm 28 create_new_node functions

```
// Find parent id ==========================================
int _findId(int point)
{
    return point(0, "id", point);
}

// Initialize new point attributes =========================
void _initializeAttributes(int son, father; vector dir;
                           float len; int gen)
{
  setpointattrib(0, "attr_dead", son,-1,              "set");
  setpointattrib(0, "Cd",        son,{0, 1, 0},       "set");
  setpointattrib(0, "fertile",   son,0,               "set");
  setpointattrib(0, "generation",son,gen,             "set");
  setpointattrib(0, "growdir",   son,dir,             "set");
  setpointattrib(0, "id",        son,-1,              "set");
  setpointattrib(0, "len",       son,len,             "set");
  setpointattrib(0, "n_dead",    son,0,               "set");
  setpointattrib(0, "parent",    son,_findId(father),"set");
  setpointattrib(0, "type",      son,"node",          "set");
}
```

Algorithm 29 create_new_node main

```
// Main Code ================================================
if(i@fertile == 1)
{
  int newpoint;
  float len = point(0, "len", @ptnum);
  int gen = point(0, "generation", @ptnum);
  int max_gen = detail(1, "numiteration");
  float new_len;

  // switch between user defined ramp for internodes length
  // and constant length through all internodes
  // create and place the new node, initialize its attribs
  if(int(ch("enable_ramp_scaling")))
  {
    float fit = fit(i@generation, 0, max_gen, 0, 1);
    new_len = chramp("D", fit) * ch("mult");
    newpoint = addpoint(0, @P + vector(new_len * v@dir));
    _initializeAttributes(newpoint, @ptnum, v@dir,
                          new_len, ++gen);
  }
  else
  {
    if(gen == 0)
    {
      newpoint = addpoint(0, @P + vector(len * v@dir));
      _initializeAttributes(newpoint, @ptnum, v@dir,
                            len, ++gen);
    }
    else
    {
      newpoint = addpoint(0, @P + vector(len *
                                  ch("scaling") * v@dir));
      _initializeAttributes(newpoint, @ptnum, v@dir,
                            len * ch("scaling"), ++gen);
    }
  }
}
```

Algorithm 30 set_id

```
// MAIN CODE ================================================
int treecount = detail(0, "treepts");
int newid = 0;

// cycles over all the newborns, set the @id
// using the previously created @treepts
// then updates the @treepts value for next
// iteration of the space colonization algorithm
for(int i = 0; i < @numpt; i++)
{
  newid = treecount + i;
  setpointattrib(0, "id", i, newid, "set");
  setdetailattrib(0, "treepts", 1, "add");
}
```

Algorithm 31 update_parent

```
// Find parent ptnum from id ================================
int _findPt(int id; int numpt)
{
  for(int i = 0; i < numpt; i++)
  {
    if(point(0, "id", i) == id)
    {
      return i;
    }
  }
  return -1;
}

// Update father children ===================================
void _updateKids(int parent; int child_id)
{
  int kids[] = point(0, "kids", parent);
  append(kids, child_id);
  setpointattrib(0, "kids", parent, kids, "set");
}

// Update Parent ============================================
void _updateParent(int child_pt; int child_id; int numpt)
{
  int parent_id = point(1, "parent", child_pt);
  int parent_pt = _findPt(parent_id, numpt);
  _updateKids(parent_pt, child_id);

  //color to yellowish: node has now at least one child
  setpointattrib(0, "Cd", parent_pt, {1, 0.5, 0});
}

// MAIN CODE ================================================
int newborns = detail(1, "newborns");
int child_id;
i@treepts = detail(1, "treepts");

for(int i = 0; i < newborns; i++)
{
  child_id = point(1, "id", i);
  _updateParent(i, child_id, @numpt);
}
```

Algorithm 32 set_attr_dead

```
// Set attr_dead based on fixed dk value ===================
void _setDead(vector p)
{
  int dead[] = nearpoints(0, p, ch("dk"));
  foreach(int attr; dead)
  {
    setpointattrib(0, "attr_dead", attr, 1, "set");
  }
}

// Set attr_dead to attractor based on ramp ================
void _rampSetDead(vector p; int max_iteration, generation)
{
  int dead[];
  float fit = fit(generation, 0, max_iteration, 0, 1);
  float dk = chramp("dk_ramp", fit) * ch("mult");
  dead = nearpoints(0, p, dk);
  foreach(int attr; dead)
  {
    setpointattrib(0, "attr_dead", attr, 1, "set");
  }
}

// MAIN CODE ===============================================
int nodes_count = npoints(1);
vector p;
int max_iteration = detail(2, "numiteration");
int curr_iteration = detail(2, "iteration");

for(int i = 0; i < nodes_count; i++)
{
  p = point(1, "P", i);
  if(int(ch("enable_scaling")))
  {
    _rampSetDead(p, max_iteration, i@generation);
  }
  else
  {
    _setDead(p);
  }
}
```

Algorithm 33 reset_AP_dir_fertile

```
v@dir = {0, 0, 0};
i@fertile = 0;
```

Algorithm 34 connect_points

```
// Create a line between the 2 points ======================
void _createLine(int father; int son)
{
  int line = addprim(geoself(), "polyline");
  addvertex(geoself(), line, father);
  addvertex(geoself(), line, son);
}

// MAIN CODE ================================================
int parent_pt = point(0, "parent", @ptnum);
_createLine(parent_pt, @ptnum);
```

Algorithm 35 trunk_width

```
// fetching info about primitives
int prim = detail(1, "iteration");
int points[];
int pt;
float width;
float fit;

points = primpoints(0, prim);
if(point(0, "width", points[0]) == -1)
{
  // main trunk case
  for(int i = 0; i < len(points); i++)
  {
    pt = points[i];
    fit = fit(i, 0, len(points), 0, 1);
    width = chramp("scale", fit)* ch("multip");
    setpointattrib(0, "width", pt, width, "set");
  }
}
else
{
  // other branches case
  float mult = point(0, "width", points[0]);
  for(int i = 1; i < len(points); i++)
  {
    pt = points[i];
    fit = fit(i, 0, len(points), 0, 1);
    width = chramp("scale", fit) * mult;
    setpointattrib(0, "width", pt, width, "set");
  }
}
```

Algorithm 36 transfer_width

```
int numpt = npoints(1);
float width;

for(int i = 0; i < numpt; i++)
{
  width = point(1, "width", i);
  setpointattrib(0, "width", i, width, "set");
}
```

Algorithm 37 ramp_scale

```
// @width as function of the @generation value of the node
int gen_max;
getattribute("opinput:0", gen_max,
             "detail", "gen_max", 0, 0);

float fit = fit(i@generation, 0, gen_max, 0, 1);
f@width = chramp("scale", fit) * ch("mult");
```

Algorithm 38 ramp_for_trunk functions

```
// Find parent id ==========================================
// Populate the @width using a stack =======================
void _setWidth(int pt; float mult; int gen_max)
{
  int stack[], int kids[];
  int curr_pt, generation;
  float fit, width;

  //find my children
  kids = point(0, "kids", pt);

  //push them in the stack
  for(int i = 0; i < len(kids); i++)
  {
    //check if the kid already has a width set
    if(point(0, "width", kids[i]) == -1.0)
    {
      push(stack, kids[i]);
    }
  }

  while(len(stack) > 0)
  {
    //pop child
    curr_pt = pop(stack);

    //set its width
    generation = point(0, "generation", curr_pt);
    fit = fit(generation, 0, gen_max, 0, 1);
    width = chramp("scale", fit) * mult;
    setpointattrib(0, "width", curr_pt, width, "set");

    //find its children
    kids = point(0, "kids", curr_pt);

    //push the children
    for(int i = 0; i < len(kids); i++)
    {
      //check if the kid already has a width set
      if(point(0, "width", kids[i]) == -1.0)
      {
        push(stack, kids[i]);
      }
    }
  }
  return;
}
```

Algorithm 39 ramp_for_trunk main

```
// MAIN CODE ================================================
int gen_max;
getattribute("opinput:0", gen_max,
             "detail", "gen_max", 0, 0);
int root_count = npoints(1);
int pt;
float mult;

for(int i = 0; i < root_count; i++)
{
    //find id from input1, use it as ptnum for points
    //in input0 (where id and ptnum match)
    pt = point(1, "id", i);
    mult = point(1, "width", i);
    _setWidth(pt, mult, gen_max);
}
```

Algorithm 40 roots_leonardo

```
// index is the @ptnum for the tree points,
// content is the width of that point. Init to all 0.
float widths[];
for(int i = 0; i < @numpt; i++)
{
  append(widths, 0);
}

// array where the intex is the @ptnum for the tree points
// and the content is the generation value for that point.
int generations[];
int gen;
for(int i = 0; i < @numpt; i++)
{
  gen = point(0, "generation", i);
  append(generations, gen);
}

// array for reordered @ptnum for the tree points by gen,
// reverse order (from pts with greater gen back to root)
int sorted_by_gen[] = reverse(argsort(generations));

// populate widths[] from the tips to root. Iterate over
// sorted_by_gen[] to pick points from newest to oldest
int kids[];
for(int i = 0; i < len(widths); i++)
{
  // retrieve children list for current point
  kids = point(0, "kids", sorted_by_gen[i]);

  if(len(kids))
  {
    float accum = 0;
    foreach(int kid; kids)
    {
      accum += pow(widths[kid], 2);
    }
    float new_width = sqrt(accum);
    widths[sorted_by_gen[i]] = new_width;
  }
  else
    widths[sorted_by_gen[i]] = ch("tips_width");
}

// finally update the geometry with the new values
for(int pt = 0; pt < len(widths); pt++)
{
  setpointattrib(0, "width", pt, widths[pt]);
}
```

Algorithm 41 from_trunk_leonardo functions

```
// Find parent for a given node ============================
int _findParent(int pt)
{
  return point(0, "parent", pt);
}

// Find kids count for a given node ========================
int _kidsCount(int pt)
{
  int kids[] = point(0, "kids", pt);
  return len(kids);
}

// Find width for a given point ============================
float _findWidth(int pt)
{
  return point(0, "width", pt);
}

// Calculate new width attribute value =====================
float _calculateWidth(int pt)
{
  int parent = _findParent(pt);
  int kids_count = _kidsCount(parent);
  float parent_width = _findWidth(parent);

  if(kids_count == 0)
  {
    return 0;
  }
  float pow_width_parent = pow(parent_width, 2);
  float mywidth = sqrt(pow_width_parent/kids_count);
  return mywidth;
}
```

Algorithm 42 from_trunk_leonardo main

```
// MAIN CODE ================================================
// trunk already has a @width set, this Leonardo proceeds
// from roots (the ones belonging to the trunk)
// to the tip branches

int pt = detail(1, "iteration");
float width;
if(point(0, "generation", pt) >= 0)
{
  if (point(0, "generation", pt) == 0 &&
      point(0, "width", pt) == -1.0)
  {
    width = ch("width");
    setpointattrib(0, "width", pt, width, "set");
  }
  else
  {
    setpointattrib(0, "width",
                   pt, _calculateWidth(pt), "set");
  }
}
```