# Modeling Climbing Plants as Anisotropic Particles in Houdini

Susan Xie

Advisors: Norm Badler and Adam Mally

University of Pennsylvania

**ABSTRACT**

*Based on the work of Hädrich et al in their 2017 paper* Interactive Modeling and Authoring of Climbing Plants*, I will be implementing a tool for modeling and animating climbing plants in Houdini. The goal of this tool is to provide artists with a better way to quickly model realistic climbing plants in 3D environments. The algorithm presented in this paper treats plants as a collection of anisotropic particles that contain both "internal" parameters, like velocity and orientation, as well as "external" parameters like available surfaces and sunlight. This tool will generate a plant model, and once the user is satisfied with the model, allow for export as an .obj. The user has some degree of control over 1) the leaves on the plant, 2) the density of the shoots, and 3) the variability in growth direction of new shoots.*

## 1. INTRODUCTION

Since the advent of computer graphics in the 1950s, researchers and engineers in the field have worked to create better, more life-like 3D scenes and simulations. In the past decades, hardware and graphics systems have improved dramatically, and as a direct result, computers have been able to produce increasingly more-realistic renders. In applications from architecture to film and games, the goal of 3D scenes is to create as immersive of an environment for the consumer as possible. When modeling outdoor scenes especially, a crucial component of adding realism is the use of vegetation. Not only do organic-looking plants help ground a scene in reality, but they also add a lot of life and interesting visual details to a digital world. Modeling objects that appear organic, however, can be an extremely time-consuming task, and this is very much the case when trying to model individual leaves, bushes, or vines by hand.

This project is an attempt to improve and expedite the process of creating digital plant life. In particular, I will be looking at an algorithm for how vines and climbing plants grow. While the growth of these plants is guided by biological factors like sunlight, nutrients, and available climbing space, their exact growth pattern is still mostly random: no two ivy plants will climb the exact same way over a wall even if they are planted in the same spot and given the same external conditions (temperature, sunlight, rain, etc.). Because of this, as mentioned previously, attempting to model ivy by hand is incredibly time-consuming. Each individual vine and leaf would have to be modeled separately and placed by the artist, and any large duplicated patches will immediately stand out. By taking an algorithmic approach to modeling and animating climbing plants, artists can generate plants in their scenes much quicker and arguably more realistically.

Another problem that emerges with climbing plants in particular is their close interaction with other environmental objects. Since climbing plants literally climb over other surfaces, if one of these surfaces is moved, the location and growth of the plant is different as well. However, when working with plants that are just static models, they will not respond to environmental changes; if the climbing surface is shifted or its geometry is modified (i.e., a brick wall covered with ivy in a video game collapses), the plants retain their initial position, and their growth no longer accurately matches the contour of the surface. When these plants are being modeled strictly by hand, this is a nightmare for the artists. Even if using a more-generic plant procedural tool, a change like this may require the artist to regenerate the plant. Depending on the tool and the size of the plant group, this may still require a rather large amount of time and computing power.

To circumvent these issues, I will be creating a tool to model and animate climbing plants, in which climbing plants are treated as a collection of anisotropic particles. Each particle represents a small segment of the plant's main vine, and new particles are generated from the vine's apical tip(s). Because each of these particles stores its own attributes, as well as a set of its "child" particles, this model for climbing plants is much more dynamic and robust than earlier representations. Particles grow from a "seed", much like how complex L-systems grow from a single initial branch. Creating L-systems that take external factors and parameters into account, however, is complex, and the anisotropic particles are a cleaner solution. Similar to space colonization algorithms, these particles also "compete" for space, but while space colonization uses a set of target attractors, the parameters the growth of these particles is based on are stored internally. Since each particle is so independent, this model allows for much more customization and editability.

This project makes the following contributions:

- Using an algorithm that has already proven to be robust, I will be creating a tool in Houdini. This algorithm has already been implemented in C++ using GLSL and OpenGL, but implementing a version in Houdini will be beneficial in making the tool available to more artists, as Houdini is already an industry-standard tool for technical art and special effects.

· Houdini offers a plug-in called Houdini Engine that allows tools built in Houdini to be exported and integrated into other Pipeline software. As many studios use Unreal, this again allows more people to adapt and use the tool.

### 1.1 Design Goals

The target audience for this project is artists, specifically modelers, animators, and 3D generalists, and technical artists. This tool would eliminate the need to model climbing plants by hand and could replace other procedural tools that are not as robust. Since there is so much built-in functionality for editing and customization, this tool provides a much faster way for artists to achieve any look they want involving climbing plants. This saves them a huge amount of time and improves their quality of life. In a practical industrial sense, this tool can allow companies to produce more realistic climbing plant renders much more quickly and cheaply.

### 1.2 Projects Proposed Features and Functionality

The tool will have the following features:

- Customizable parameter for shoot density
- Customizable parameter for leaf density
- Customizable parameter for variability of shoot growth direction
- Real-time ability to grow additional shoots
- Customizable leaf geometry
- Ability to export .obj of final plant geometry

## 2. RELATED WORK

Vegetation growth is a widely explored research topic in computer graphics, and as a result there are many different algorithms in existence for creating realistic plants. This project will primarily look at the work of one paper by a group of researchers from University of Konstanz, Purdue University, and Stanford University.

### 2.1 Early Methods for Plant Growth: L-Systems

Some of the earliest methods for simulating plant growth involve L-systems, a system of formal grammars where strings correspond to some geometric structure [PL90]. While this was once considered one of the best models for organic structures, this method is now thought to be outdated. While L-systems capture the fractal-like nature of branches and veins, where one large branch splits off repeatedly into multiple smaller branches, they fail to capture the more variable aspects of plant growth. Further, they do not take into consideration how external factors, like sunlight or other geometry in the scene, may affect how the plant grows.

This is not to say that L-systems are no longer relevant; in fact, they can be a good base for more complex plant growth algorithms. In 2009, a paper from Chalmer's University of Technology described a method of using L-systems to model climbing plants. These plants were able to react to gravity and sunlight [Knu09]. However, while it is possible to add additional parameters to L-systems, there are many other methods for modeling plant growth now that lend themselves to customization and variability much better.

### 2.2 Better Methods for Plant Growth: Space Colonization

Another method that became popular for modeling plant growth was the space colonization algorithm. Palubicki et al used the space colonization algorithm to model temperate-climate trees and shrubs, and using this method, were able to generate extremely photorealistic images [PHL*09]. Unlike L-systems, which create plants by repeatedly branching off of a central source branch, agents in a space colonization algorithm move towards a series of attractors. This results in a focus on competition for resources, allowing for much more dynamic and varied growth. While this does yield excellent visual results, there is very little control for more-custom plant growth. Attractors can be manually placed, but there is still little to no control for how branches will actually grow towards these attractors.

To fix this issue, researchers Longay et al. released a tool with a multitouch tablet interface that allows for directing growth using procedural brushes and other parameters [LRBP12]. Since the user has such close control over plant growth, this leads to highly artistic renders; one of the figures shown in the paper is a tree in the shape of swan. However, in contrast to the realistic vegetation growth I am hoping to achieve, this is definitely more of an art-based tool. There is very little focus on any kind of physics and no way to customize physical parameters, which is something important to consider when dealing with animation.

### 2.3 Other Methods

Some of the newest methods for simulating plant growth use a particle-based approach: plant growth is semi-random, and plants grow in segments (particles) based on a set of internal parameters. This model seems to be especially successful when dealing with climbing plants, since their growth relies so heavily on external objects and conditions. In 2008, Thomas Luft released an online ivy generator that is still popular today, with the latest Github repository push dated October of 2017. On the website for this tool, Luft describes his implementation as using the following parameters to determine growth: "a primary growth direction (the weighted average of previous growth directions), a random influence, an adhesion force towards other objects, an up-vector imitating the phototropy of plants, and finally gravity" [Luf08]. While Luft himself states the method is not biologically accurate, it still succeeds in providing visually appealing results. Although this tool works extremely well for static images, it is not conducive to animation; plants from the ivy generator do not respond to wind or other forces.

A newer method, presented by Hädrich et al in 2017, does strive to be mildly more biologically accurate. This is the primary paper I will be referencing for this project.

Climbing plants are treated as a system of anisotropic particles, where particles are affected by both internal parameters (velocity, angular velocity, and orientation) and external parameters (phototropism, surface adaptation, and growth integration) [HBDP17]. One of the biggest advantages of this work over Luft's is the algorithm's robustness: Hädrich et al allow for much more user customization (there are parameters affecting shoot growth and variety) and include real-time editing for very fine-tuned customization. For animation, users can even customize physical and material properties of the plants. I will be using the research conducted by Hädrich et al and implementing it in Houdini.

## 3. PROJECT PROPOSAL

Based on the work of Hädrich et al in their 2017 paper *Interactive Modeling and Authoring of Climbing Plants*, I will be implementing a tool for modeling and animating climbing plants in Houdini. The algorithm presented in this paper treats plants as a collection of anisotropic particles. Once seeds are placed, new particles are spawned from existing particles based on both "internal" particle attributes like velocity and orientation, as well as external influences like surfaces and sunlight. The goal of this project is to, at a minimum, implement a plant growth simulator in Houdini that builds a plant model and allows for export as a .obj. The user should be able to control the density of the leaves on the plant, the density of the shoots, the variability in growth direction of new shoots, and potentially even the shape of the leaves via importing a mesh. In addition to these parametrizations, the user will also have the power to make custom edits to the plants; branches can be added or pruned in real-time for fine-tuning the look of the plant. Finally, plants will react appropriately to external forces like wind, i.e. when under enough stress, branches should bend and break. From Houdini, the tool can also be ported over to Unreal Engine for game developers using the Houdini Engine plug-in.

### 3.1 Anticipated Approach

Since I have never developed a tool in Houdini before, I want to first try implementing the particle data structure in C++ and OpenGL, which I am more familiar with. I will first try to implement this system in 2D, and then I will expand it to 3D. Once I have a particle system that is working, I will bring the code over to Houdini, using spheres as placeholder geometry. After the basic particle system has been implemented in Houdini, I will build out the remainder of the tool there. This includes a UI for editing custom parameters (shoot density, shoot variability, and leaf density), as well as UI tools for custom edits (adding and pruning branches).

### 3.2 Target Platforms

I will be using Houdini FX 17.0.416 (Apprentice version) on a laptop computer with an Intel i7 processor clocked at 2.2 Ghz, 16 GB of RAM, and a NVIDIA GeForce GTX 1070 graphics card.

### 3.3 Evaluation Criteria

Final images rendered in Houdini will be compared with real images of climbing plants, as well as with images from the original Hädrich et al research paper. Since this is not meant to be the development of an entirely new algorithm, but rather an implementation of an algorithm in new software, the goal is to get as close to the original paper's results as possible. I also want to compare this tool to Luft's ivy generator, which is available for free download, both visually (comparison between final generated images) and in terms of speed, efficiency, and final file size.

## 4. RESEARCH TIMELINE

**Project Milestone Report (Alpha Version)**

- Research: understand proposed algorithm and learn how to make custom tools for Houdini
- Implement basic 2D particle system:
  - Particles grow from a single seed
  - Particles generate based on internal factors (velocity, angular velocity, orientation)

**Project Milestone Report (Beta Version)**

- Transfer particle simulation to 3D
  - Particles are affected by external factors (phototropism, surface adaption, growth integration)
- Basic geometric representations for leaves
- Growth affected by custom parameters: shoot density and variability, leaf density

**Project Milestone Report (Final Version)**

- Improve user-interface
- Fine-tune parameters
- Implement features for editing:
  - Adding branches

**Project Final Deliverables**

- Houdini tool
- Rendered images from Houdini
- Documentation

**Project Future Tasks**

I found that while Houdini has a lot of well-developed, built-in tools, there are also limitations that come with it. For instance, while manipulating the particle simulation can be done relatively straight-forwardly from within Houdini's provided node networks, Houdini does not allow modifying any geometry assets from outside the node network. As a result, while Houdini is a great tool for quickly creating a clear user interface with custom simulation, it is not ideal for building a tool where the user can modify scene geometry directly. I initially wanted to create a tool

with more interactivity in Houdini, but I found this interactivity extremely challenging to achieve. If I were to continue this project in the future, I would like to find a way to make the simulation more interaction. Some of the features discussed in the initial paper include:

- The ability to prune plants
- Material properties and elasticity: the user can pull on parts of the plants, and the plants will respond by bending and breaking when under enough stress
- The ability to paint attractors on surfaces to fine-tune simulation

As for improvements in the actual appearance of the vines, more customization can be added to the plants. For example:

- In addition to leaves, the user may also be able to add flowers to the vines. This could be expanded so the user can also specify flower shape (via a user-specified mesh), flower color, flower growth density, and other variability.

## 5. METHOD

This project is primarily built in Houdini, and the tool is presented in the format of an HDA, or Houdini Digital Asset. To approach this tool, I first wanted to better develop the algorithm; while the authors of the initial paper cited discussed the parameters they used and provided a few equations, they did not provide any sample values, and I found it difficult to achieve a particle simulation that gave me the result I wanted. Because I had never built tools for Houdini before, I started by working with a 2D plane in C++ and OpenGL. Not only was this helpful for understanding the equations discussed in the paper, but also for starting to get an understanding for reasonable starting values and time steps. Since the program was written in C++, I had to dt time step and a loop to iterate over the function x number of steps.

The next step was to take the project into Houdini. While I have worked with Houdini's tools in the past, I had no prior experience building custom tools at all. When I initially began working on the tool, I took a completely Python-based approach, using a custom time step and Houdini's hou module to generate geometry. The results yielded by this approach were not ideal—not only was the simulation itself subpar, but this did not take advantage of Houdini's built-in functionality at all, resulting in slow simulations and unorganized, unusable nodes. This was essentially a direct translation of what I had done in C++ to Python.

The following equations were taken from the paper and used to update parameters between steps:

position: $\mathbf{x}_p = \mathbf{x} + \mathbf{v}\Delta t + \frac{(\mathbf{a}_g + \mathbf{a}_e)\Delta t^2}{2}$

orientation: $\mathbf{q}_p = \left[\hat{\boldsymbol{\omega}}\sin(\frac{|\boldsymbol{\omega}|\Delta t}{2}), \cos(\frac{|\boldsymbol{\omega}|\Delta t}{2})\right]$

velocity: $\mathbf{v} = (\mathbf{x}_p - \mathbf{x})/\Delta t$

angular velocity: $\boldsymbol{\omega} = axis(\mathbf{q}_p\mathbf{q}^{-1}) \cdot angle(\mathbf{q}_p\mathbf{q}^{-1})/\Delta t$

These internal parameters were updated within a block of code as follows:

```
def step:
    # update orientation
    accNorm = parent.angVelocity.length()
    accY = math.sin((accNorm * dt) / 2)
    accX = math.cos((accNorm * dt) / 2)

    eulerQuat = Quaternion(angle = accX,
      axis = parent.angVelocity * accY)

    qp = eulerQuat * parent.orientation
    orientation = qp

    # update angular velocity
    orientQ = qp * orientation.inverse()
    acc = orientQ.axis() * orient.angle()
    angularVelocity = acc

    # update position
    position = (parent.position +
      parent.velocity * dt +
      (parent.angularVelocity + gravity) *
      dt * dt) / 2.0

    # update velocity
    self.velocity = (position -
      parent.position) / dt
```

Eventually, as I spent more time in the software however, I started to understand Houdini's node-based coding better. I scrapped everything I had written before and started from scratch using Houdini's node networks. After spawning a series of points with the appropriate attributes, I used a solver to update particle position at each step. This solver acted as the "step" function, and I used an attribute VOP to update position, orientation, and velocity during each step. The pseudocode above still holds; I just implemented it using Houdini's attribute nodes rather than writing lines of code. This new method allowed me to use Houdini's timeline to visualize the simulation occurring in real time, and I was able to expose parameters I wanted the user to be able to edit rather easily.

Afterwards, I worked on implementing the external forces that affect particle growth: surface adaptation, or the ability of the plant to grow over a surface, and phototropism, or the growth of the plant towards a light source. Both of these parameters affect the orientation parameter, which is a quaternion with an axis and angle component. The equations for updating these components provided from Hädrich et al are as follows:

Surface adaptation

$$\mathbf{a}_a = \hat{\mathbf{v}}_s \times \hat{\mathbf{v}}_f$$
$$\alpha_a = (\hat{\mathbf{v}}_s \cdot \hat{\mathbf{v}}_f)\tau\Delta t$$

where $\hat{\mathbf{v}}_s$ represents a vector pointing to the nearest surface, $\hat{\mathbf{v}}_f$ represents the particle's current forward vector, and $\tau$ is a value between 0 and 1 that defines adaptation strength

Phototropism

$$\mathbf{a}_p = \hat{\mathbf{v}}_l \times \hat{\mathbf{v}}_f$$
$$\alpha_p = (1 - O)\eta\Delta t$$

where $\hat{\mathbf{v}}_l$ represents a vector pointing to the light, $\hat{\mathbf{v}}_f$ represents the particle's current forward vector, $O$ is light occlusion, and $\eta$ is a value that controls the strength of the phototropic response.

I found, however, that the effect these parameters had on on my particles was not as strong as I wanted, and so I also updated particle velocity to point more towards the direction of the orientation.

The next issue I ran into was surface collisions: particles would grow towards the surface correctly, but they would also eventually begin growing through the surface unless otherwise controlled. The first method I attempted was basic ray-marching: I used the bounding box of the geometry object to calculate whether if the newly generated particle intersected this bounding box, and then moved it back along the direction of the ray until it was no longer intersecting the surface. However, I found that this failed on more complex geometry. I ended up converting input geometry to VDB, a volume primitive, and this gave the surface an SDF value that could be calculated from within some range of voxels outside the surface of the primitive. I checked geometry collisions using this SDF, and if the particle ended up inside the geometry, I would march back until the SDF was positive once again. While this result still was not perfect due to how the vines were being generated, it gave me the ability to work with more complex input geometry.

After the algorithm was finalized, I began working on constructing vines from the points. I gave each point the following attributes: position, velocity, angular velocity, orientation, and ID. This ID is used to identify the growth direction of a single particle over the course of the simulation; if you start with ten particles for instance, on the next iteration, each of those ten particles will spawn one additional particle, and then each of those new particles will spawn yet another. The ID identifies which particle spawned the new particle—using this same example, we end up with 30 particles after three frames, and we have three particles each that share an ID 1 through 10. I used an Add node to connect all points of the same ID to a single curve. This curve was then extruded into a polywire, smoothed, and cleaned up using a Facet node, and then passed through a Reduce node to further simplify the geometry.

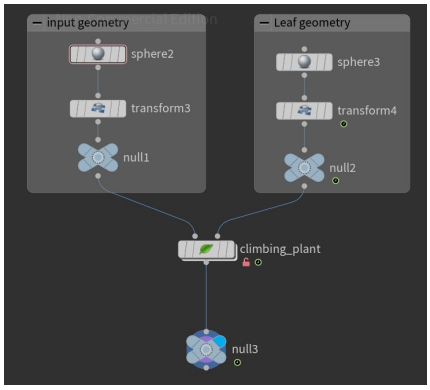
I also wanted the vines to support foliage. To account for this, I added an additional attribute to the points called "leaf" that was simply 0 or 1 based on if a leaf should be placed at that particle. I used a random number function to generate a value for leaf, and then compared this value to a user-controlled parameter for controlling leaf density. I grouped all particles with leaves into a group, and finally used a Copy Stamp node to place leaf geometry at each particle position.

The final feature I implemented was user interactivity. In Houdini, an HDA can have a State View that runs some Python code when the user interacts with the scene. Since one of the inputs to the particle simulation is a series of starting points, I was able to replace that input with an Add node. When the user clicks on the scene, an intersection is calculated with either the scene geometry or construction plane, and a point is appended to the Add node at that position. This allows the user to click anywhere on the scene to "plant" a seed and spawn new plants, even while in the middle of a simulation.

I also attempted to allow the user to prune plants. Hypothetically, I wanted to add a "kill" parameter to the particles: once they were marked to be killed, they would be deleted from the particle simulation. This would in turn destroy that section of the curve, and essentially act like "cutting" the plant. If the user clicked while holding down the Ctrl key, Houdini would calculate an intersection with the vine and find the nearest particle to this intersection from within the particle simulation. This particle, along with any older particles in the simulation along that vine, would then be marked to die and deleted from the scene. I was unable to get this to work, however, because Houdini does not allow for editing scene geometry from outside the node network, including from a State View, meaning I was unable to update any particles or particle attributes without diving into the network myself. There may be some complicated workarounds for this issue, but after reading countless forums and spending hours debugging, I decided Houdini was maybe not the best software to use when trying to design a tool where the user can directly interact with the scene.

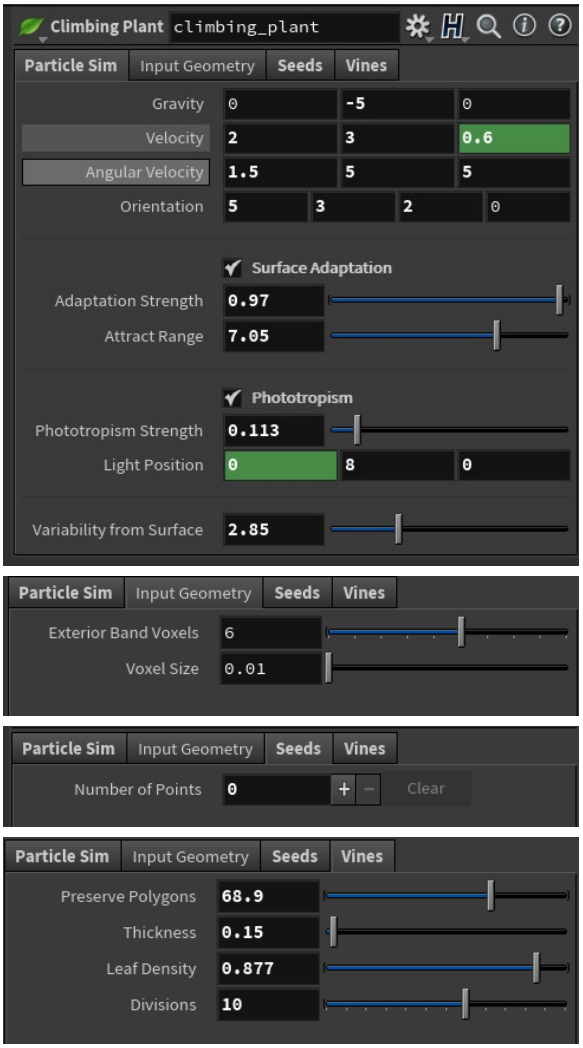
## 6. RESULTS

The final tool is in the form of an HDA which I have named "climbing_plant"—this node can be dropped into any scene and contains the entire subnetwork for the algorithm and plant creation. This node takes in two inputs: geometry for the plant to interact with, and geometry to use for the leaves.



The final tool is in the form of an HDA which I have named "climbing_plant"—this node can be dropped into any scene and contains the entire subnetwork for the algorithm and plant creation. This node takes in two inputs: geometry for the plant to interact with, and geometry to use for the leaves.

The user interface has the following parameters:




The images show the Particle Sim, Input Geometry, Seeds, and Vines tabs respectively. As the tab name would imply, the first set of parameters affect the particle simulation—the user can assign a gravity value as well as a starting velocity, angular velocity, and orientation value for the particles. The next parameter toggles Surface Adaptation: when surface adaptation is being accounted for, the Adaptation Strength slider can be used to control the adaptation strength $\tau$ value, and the Attract Range determines how close the particle must be to the surface to be affected by Surface Adaptation at all. Phototropism can also be toggled, and the user can set the phototropic strength $\eta$ as well as a light position. Variability from surface affects the ray-marching step used when avoiding geometry collisions, so a larger variability will yield vines that do not stick to the surface as closely.

The Input Geometry tab controls how the SDF for the input geometry is generated. Voxel size affects how detailed the VBD conversion is, and "Exterior Band Voxels" affects the range of the VBD's SDF function. In the screenshot, since each voxel has a size of 0.01, the SDF function stretches 6 voxels outside the input geometry, meaning the SDF function evaluates a reliable value at a distance of 0.06 from the surface.

The Seeds tab auto-populates with Point positions after the user clicks on the scene to place points. Points can be deleted and their positions can be updated from within this interface.

Finally, Vines is for controlling the appearance of the output geometry. "Preserve Geometry" is a value used by the Reduce node that cleans up the vines after their creation. The slider controls a value that dictates a percentage of the original polycount to keep. "Thickness" determines the thickness of the vines, and "Leaf Density" determines how densely the leaves on the vines should grow. "Divisions" also deals with geometry and level and detail.

## 7. CONCLUSIONS and FUTURE WORK

Ultimately, I found this method to be an interesting and effective way to simulate plants. L-systems and space colonization algorithms tend to have a somewhat predictable growth pattern—L-systems rely on a series of grammars and rules, and space colonization requires a set of target points. I found the particle system to be more unexpected, which has its pros and cons: the randomness of the growth pattern yields more complex (and perhaps, more realistic) results, but the behavior is also harder to control. I believe this is why the original paper authors also implemented many tools for authoring plants, giving the user the ability to prune the vines and to paint attractors and detractors on surfaces.

I found that while Houdini was a really powerful tool for generating the particles used in the simulation, it was much harder to achieve the level of user interaction I initially envisioned. The user can interact with a scene within an HDA using Houdini Viewer States, but there are still limitations to what this can achieve. While any node parameters within your network are editable, there is no way to directly edit attributes of the geometry itself. Basically, it was easy to get a high level of control over simulation parameters, but there was no straightforward way for the user to interact with the geometry after the simulation had already run its course.

If I were to continue this project and implement the interactivity features I discussed in "Project Future Tasks" (pruning, material properties and elasticity, painting attractors / detractors), I would likely migrate to another software. Specifically, I think the interactivity would work really well within something like a game engine, like Unity or Unreal, since they have many built-in real-time rendering features.

## APPENDIX

**A.** All images and screenshots shown at the end of the document.

**B. Python HDA State View Code Snippet:**

```
def onMouseEvent(self, kwargs):
  ui_event = kwargs["ui_event"]
  device = ui_event.device()
  origin, direction = ui_event.ray()

  if self._geometry:
    hit_geo, position_geo,
    norm_geo, uvw_geo =
    su.sopGeometryIntersection(
    self._geometry, origin, direction)

  if self.vines:
    hit_plant, position_plant,
    norm_plant, uvw_plant =
    su.sopGeometryIntersection(
    self.vines, origin, direction)

  if hit_plant!=-1 and hit_plant!=-1:
     plant_dist = hou.Vector3.length(
         position_plant - origin)
     geo_dist = hou.Vector3.length(
         position_geo - origin)
     if plant_dist <= geo_dist:
        position = position_plant
     else:
        position = position_geo

  if hit_geo==-1 and hit_plant==-1:
     position = su.cplaneIntersection(
         self.scene_viewer,
         origin, direction)

  # Create/move point if LMB is down
  if device.isLeftButton():
     self.start()

     self.node.parm("usept%d" %
         self.index).set(1)

     self.node.parmTuple("pt%d" %
         self.index).set(position)
  else:
     self.finish()

  return True
```

**References**


[HBDP17] HÄDRICH T., BENES B., DEUSSEN O., AND PIRK, S. 2017. Interactive Modeling and Authoring of Climbing Plants. *Computer Graphics Forum* 36, 2 (May 2017), 49–61.

[Knu09] KNUTZEN J.: *Generating Climbing Plants Using L-Systems.* Master's thesis, Chalmer's University of Technology, 2009.

[Luf08] LUFT T: An ivy generator. https://graphics.uni-konstanz.de/~luft/ivy_generator/, 2008.

[LRBP12] LONGAY S., RUNIONS A., BOUDON F., PRUSINKIEWICZ P.: Treesketch: interactive procedural modeling of trees on a tablet. In *Proc. Of the Intl. Symp. On SBIM* (2012), pp. 107-120.

[PHL*09] PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R, PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28,* 3 (2009), 58:1-58:10.

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.,: *The Algorithmic Beauty of Plants.* Springer-Verlag New York, Inc. 1990.

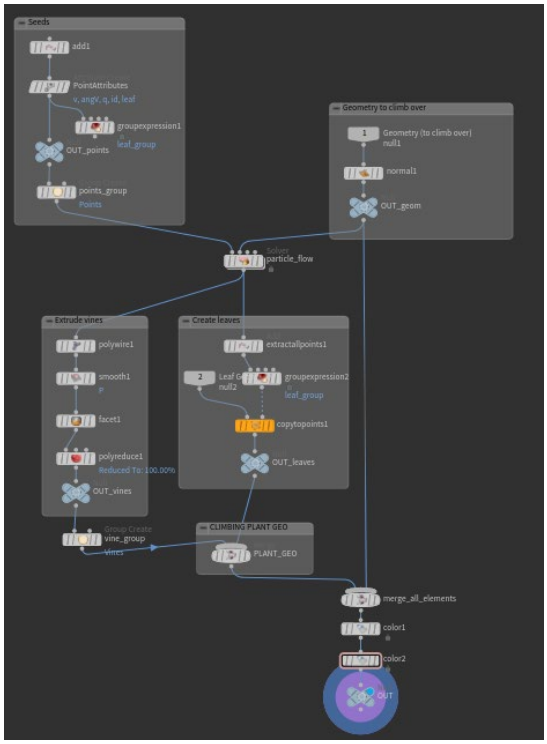



**Figure 1:** *climbing_plant HDA subnet overview*

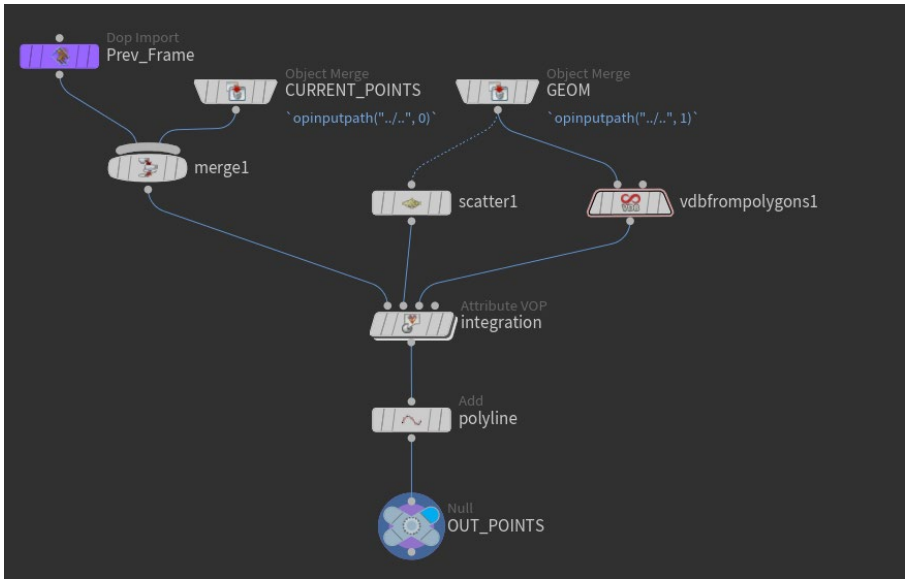



**Figure 2:** *particle_flow subnet overview*

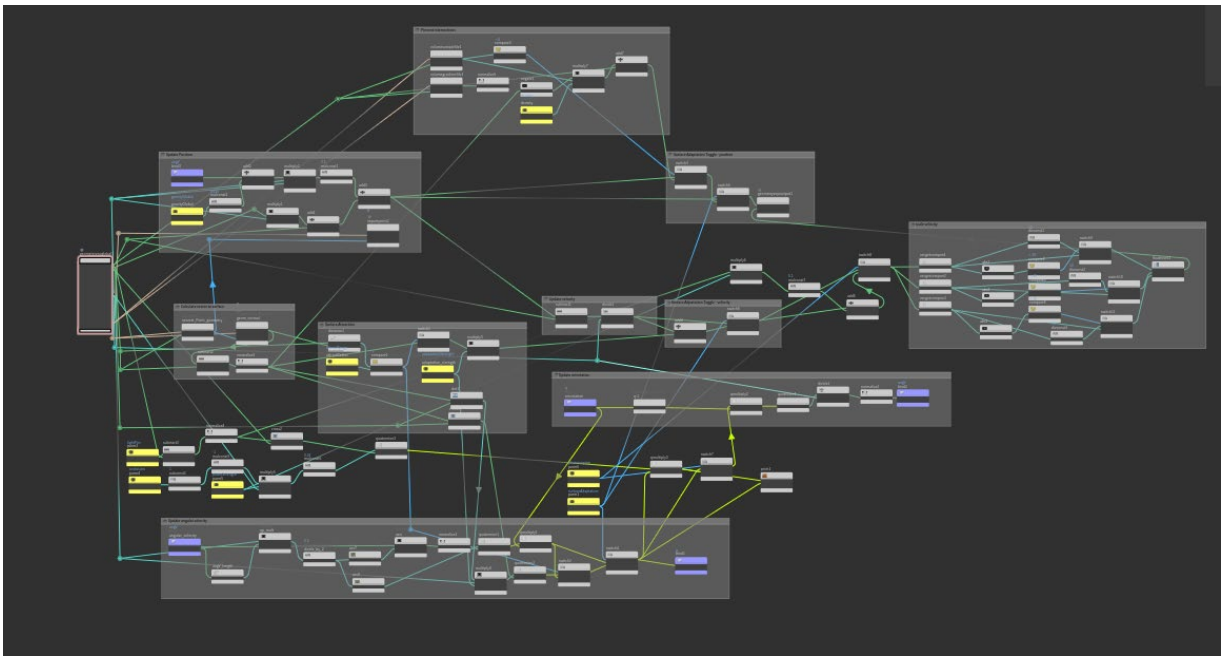


**Figure 3:** *integration AttributeVOP node network: This is essentially the algorithm for calculating all the particle attributes in node form*

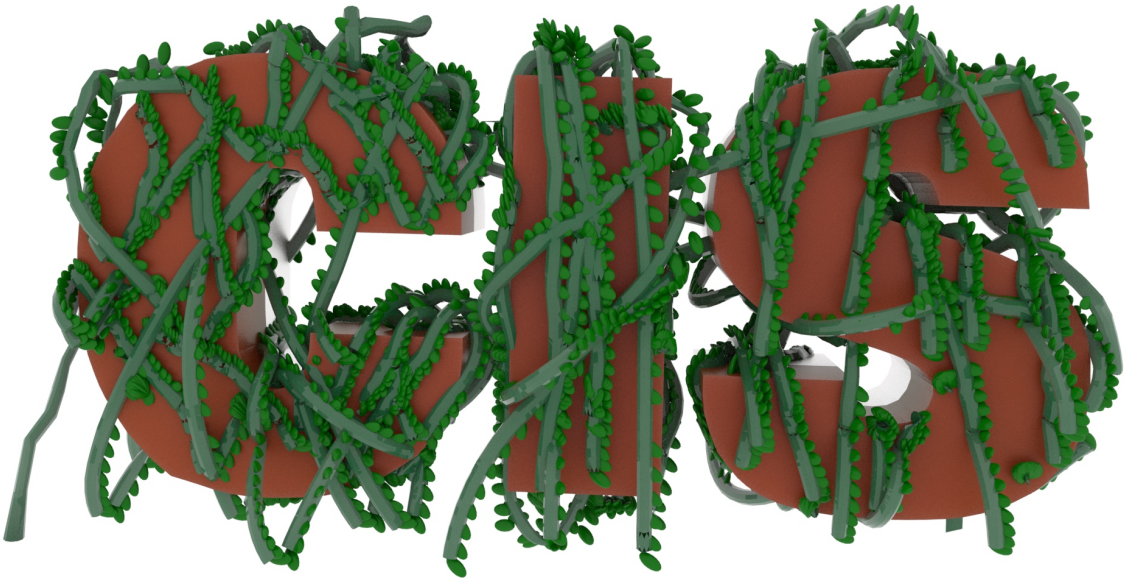


**Figure 4:** *Sample render: CIS*
*Plants were autogenerated from Houdini, exported, and rendered in Arnold for Maya.*