Procedural Ivy Generator Houdini Digital Asset

by

Caoimhe McCarthy
s5204473

Masters Degree Project
MSc Computer Animation + Visual Effects

BOURNEMOUTH UNIVERSITY

August 2020

Abstract

Vegetation growth has always been an important aspect of computer graphics. In many instances, overgrowth of ivy and vines have played a key role in achieving a realistic aesthetic scene. Though the artistic modelling of 3D ivy is somewhat dated, the industry is constantly driving to be more and more procedural. This paper demonstrates the generation of a Houdini Digital Asset for ivy growth. This procedural tool can generate a variety of different scenes based on user defined parameter values. The ivy is generated using a sort of particle based method first introduced in 1985 by Reeves and Blau (1985). Using this as the basis for the growth algorithm, the HDA also simulates external tropisms such as gravitropism and phototropism.

Along with the ability to automatically generate ivy, this tool allows the user the more stylistic approach of drawing ivy directly onto the environment geometry. This art-directable procedural tool demonstrates an easy way to generate climbing ivy.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 The Problem . . . 2
1.2.1 Note . . . 2
1.3 Ivy . . . 3
1.3.1 Tropisms . . . 4

2 Relevant Work **5**
2.1 L-Systems . . . 5
2.2 Particle Based . . . 6

3 Technical Background **8**
3.1 VEX . . . 8
3.2 Houdini Digital Assets . . . 9
3.3 Signed Distance Fields . . . 9

4 Implementation **11**
4.1 Growth . . . 11
4.2 Curve Growth . . . 14
4.3 Tendrils . . . 14
4.4 Stem Scaling . . . 15
4.5 Leaves . . . 15
4.6 Collision Detection . . . 17

5 Known Issues + Solutions **19**
5.1 Bunching . . . 19
5.2 Merged Geometry . . . 19
5.3 Gravity . . . 20
5.4 Uniform Scaling . . . 20

6 Performance **21**

7 Further Work **24**

8 Conclusion + Discussion **25**

Bibliography **28**

1. Introduction

1.1 Motivation

Plant growth simulations have been present in this industry for a significant length of time. Initially, this content had to be created manually by the artist in an attempt to recreate a real world environment. The film and games industry continue to require vast amounts of vegetation and overgrowth throughout scenes as important background assets. Among the demands of modern requirements such as hyper-realism and fast real-time rendering, these industries have an ever growing need for procedural production tools that can be used and implemented with relative ease. Typically, a procedural approach to content generation is far less time consuming and potentially lower in cost. Vegetation is an ideal candidate for a procedural approach as many natural patterns and phenomena can be recreated using mathematics and can be generated in vast quantities using specific programmable software. Having a procedural approach can greatly shorten development times and generate an incredible amount of variance entirely computer driven or even allow for intricate artistic direction.

Vegetation such as vines and shrubs can add fine detail to scenes making them appear more realistic as well as overgrowth bringing life to an otherwise dull scene. Having a scene filled with plant life and growth can imply a passing of time and can convey lively interaction between objects in the scene, without which an audience would never consider. The methods in which all types of growth are recreated on screen are fascinating, however in this instance this project aims to demonstrate only a fraction of what can be achieved.

1.2 The Problem

The aim of this project is to create a system which generates climbing plants that interact with the surrounding environment. The system framework should be easily modified as per user preference.

The following is a list of statements to aid in drawing focus to the main tasks set out by this project.

1. Given seed points, automatically generate climbing ivy. Should branches be required, consider how they are then generated from the main stems. Similarly, integrate a method of drawing curves directly into the scene for full artistic control.

2. Introduce the main factors in plant growth, tropisms. The effects of gravity and light exposure are have great influence over the growth of plants. Depending on where the light source geometry is, a plant should be able to direct its growth toward it.

3. Regarding collisions there are two priorities, avoiding the plant intersecting the environment in an unnatural way and avoiding leaves colliding with each other. Should collisions be detected there should be an appropriate reaction to counter this.

4. Customisation plays a key role in the overall goals of this project and should allow the user the versatility to swap out geometries and alter shapes should they wish. With the final product aiming to be a compact, easy to use tool.

1.2.1 Note

It is important to note certain limitations of this tool. The tool was created with the premise of adding growth and life to a scene though static. The ivy is not simulated over time and instead the focus is on the overall shape and impression. Though realism is typically always an important goal in vegetation recreation, this tool aims to also allow the artist manipulate real world environments as they see fit.

1.3 Ivy

There are many different types of Ivy out there but generally all are climbing vines. Ivy can grow vigorously, making it a great plant to give quick results when trying to create a busy scene. The most common Ivy found in Europe is known as English Ivy, or hedera helix. As well as being a vigorous climber, hedera helix is also a trailing plant meaning with nothing to cling to it droops with gravity, making it common in hanging baskets. This plant is very resilient and can even survive winters with heavy frost, however due to it's impressive resilience some parts of North America have classified this plant an invasive species (RHS, 2020).

FIGURE 1.1: Ivy growing on a tree at Queens University Belfast (Personal collection).

Climbing plants such as English ivy or grapevine (vitis vinifera) generally produce two lateral buds. One which forms a lateral branch, a similar structure to the main stem, whereas the other develops into a tendril. Tendrils are twisting threadlike strands typically protruding from the vine stem. In English ivy, the ends of the tendrils flatten and extrude an adhesive liquid upon contacting the surface allowing the ivy to climb and support the weight of the vine.

The tool created during this project aims to recreate a general ivy vine close to English ivy with the ability to alter various parameters resulting in the appearance of different ivies.

Conditions such as specific characteristics of different plants i.e leaf size and shape or direction of branching can be altered easily to generate visually different growth. However depending on the parameters chosen, the tool is always sensitive to the environment. Parameters may need to be altered for realistic growth depending on what the plant is climbing and the size of the scene. Geometry such as walls have no big obstacle for the ivy to overcome compared to a more complex structure where direction of growth is not so obvious.

FIGURE 1.2: Ivy branching attached to a wall at Queens University Belfast (Personal collection)

1.3.1 Tropisms

A tropism is a biological response of a plants growth in response to external environmental stimuli. Though there are many different types of tropism few are considered during this project. Phototropism is a plants response to sunlight and the effect it has on it's directional growth. By adding a sun object to the scene the direction of growth can be persuaded towards it.

Geotropism refers to oriented growth in reaction to gravity. In certain species of ivy this occurs when there are no surrounding objects to cling to and instead the ivy hangs. Geotropism is also implemented in this tool with options to change it's strength.

2. Relevant Work

2.1 L-Systems

A widely used approach considered when aiming to recreate growth patterns is the L-system. This method follows a set of repetitive replacement rules to create fractal-like structures and realistic plant models (Prusinkiewicz and Hanan 1989). L-systems were introduced by Hungarian Biologist Aristid Lindenmayer. He used L-systems to model the growth processes of plants. L- systems are a type of formal grammar in that the system consists of rules, an alphabet and an axiom (1990).

The production rules governing L-systems are versatile and a variety of structures can be created. These methods have gone beyond modeling trees and plants and have found uses in documenting the morphology of many organisms found in nature. L-systems are recursive by nature meaning intricate fractal structures can be easily generated. This recursive nature can bring up problems however as it can often lead to a distinct self similarity not typically found in plants and trees. L-system's can also have the issue of intersecting branches, as the recursive nature dictates a newly created branch only considers the branch before it regardless of the rest of the system (Kaandorp and Kubler 2001).

These systems are typically chosen due to the simplicity of their structure and with the ability to expand existing L-systems, a variety of different plant characteristics can be shown. Though the initial L-system structure proposed by Lindenmayer was relatively simple, some have developed it further to consider external factors and tropisms (Měch and Prusinkiewicz 1996).

Kniemeyer et al. (2008) introduced Relational Growth Grammars, a change from the traditional L-system method of string rewriting grammars extending the notations and semantics into graph grammars. This new method allowed for more a more dynamic approach to L-systems and was embedded in the language XL, bringing together object-oriented and rule-based programming (Kniemeyer *et al.* 2008).

Though there are many applications of L-systems, this project has chosen to instead focus on particle based growth algorithms that create less known examples of growth but are also frequently used in the industry.

2.2 Particle Based

Another method introduced when modelling plant growth is by using a particle based system to generate points along the plant stems. Reeves and Blau (1985) first introduced this method in 1985. Initially this method did not account for environment interaction, meaning the systems were essentially closed. This being very limiting, it inspired further work. Both Arvo et al. (1988) and Green (1989) implemented different techniques to allow for interaction between particles and the environment. Arvo et al. used ray casting to detect the proximity of objects while Greene used the 3D grid values, voxels, to detect collisions. Arvo's method using raycasting limits the means of sensing the environment which limits the type of information that can be obtained (Benes and Millán 2002). The method applied in Greene's technique allows the plants to grow based on predefined rules i.e relationships such as intersection and proximity. These elements can be evaluated faster using voxel space rather than analytic geometry (Greene 1989).

Each particle based method has the same foundation, the plants are generated from given seed points. From there, particles are grown each depending on some local variables. Beyond the simple versions, a more intricate version of a particle based growth system, AMAP, was introduced in 1988 (de Reffye *et al.* 1988). AMAP is a biologically based system that uses so-called "intelligent" particles. This system considers even more environmental factors such as sufficient food for the plants to grow among other important resources.

In 2002, Benes and Milan (2002) extended the particle based system even further, introducing the concept of traumatic reiteration. This technique kills a particle if no further growth is possible. The growth then filters down the plant to the next available particle and begins to stimulate new lateral growth from there. This method introduces obstacle avoidance, furthering the realistic interaction between the plant growth and the environment.

The tool described here uses a particle based approach to model the climbing plants. By using Houdini's VDB volumes, voxel space is used to aid in collision detection and to direct the growth of the ivy.

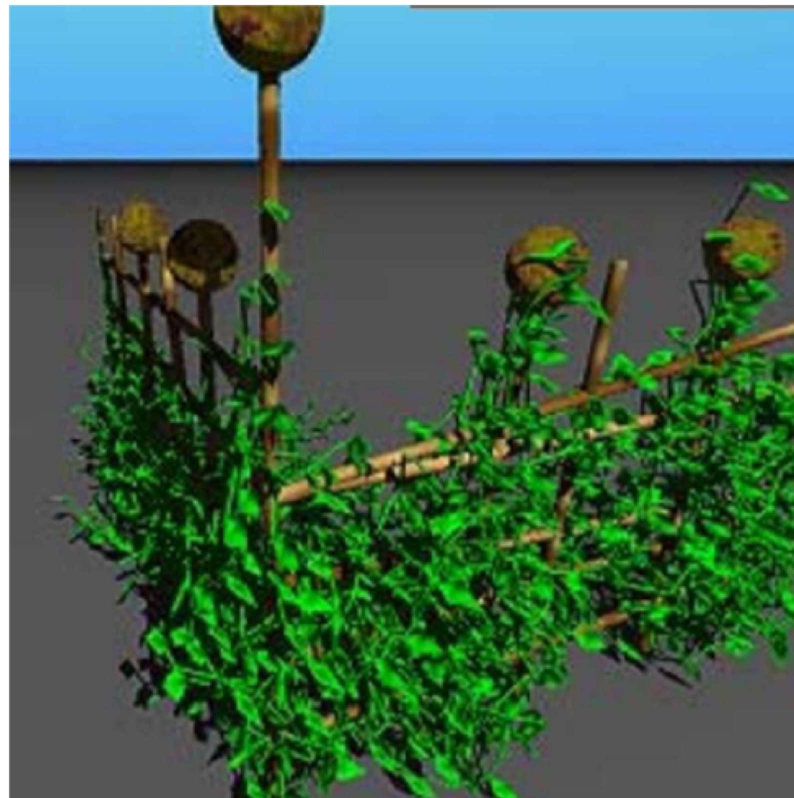

FIGURE 2.1: Ivy generated by Benes and Milan (2002) using traumatic reiteration.

3. Technical Background

3.1 VEX

The tool described in this paper was entirely made in the 3D procedural software, Houdini. Houdini's ability to handle different scripting languages makes it versatile and makes geometry among others relatively easy to manipulate. Houdini's native scripting language is known as VEX. This expression language can be used throughout Houdini with the option to be applied to rendering, compositing, modelling or particle systems. Though clearly not as widely known and used across multiple DCC tools such as python, within Houdini VEX is an incredibly versatile and extremely efficient language for writing custom shaders and nodes etc. The VEX language is somewhat based on C, C++ and RSL and is relatively easy to understand given any prior knowledge of these languages. (SideFX, 2020)

Having knowledge of VEX is very useful when it comes to customising nodes and tweaking attributes. The *Attribute Wrangle* node is a powerful node used for implementing VEX code on different geometry types. The wrangle node's use of scripting can allow the network to be more versatile and faster to implement. By using channel functions, we can add custom sliders to the code that allow for procedural manipulation of the scene. *Attribute Wrangle* has been used multiple times throughout this tool, it's use primarily in forming the growth algorithm though also simply to create and manipulate different attributes in the system as will be explained later in Section 4.1 in greater detail.

3.2 Houdini Digital Assets

Houdini's procedural power enables a user to convert their custom nodes into reusable digital assets. These Houdini Digital Assets allow the user to package complex networks and build custom interfaces into a node that users can install and use similar to any other node within Houdini. The method to build a slick HDA can be tedious in linking parameters and ensuring everything is portable and contained within the asset though the convenience of the resulting tool makes the process always worth it. These nodes are used at both object and geometry level depending on the creators intentions. HDA's also have built in help cards, aiding the user each step of the way.

3.3 Signed Distance Fields

The particle based approach used in this tool would not be possible without the use of Houdini volumes. Standard Houdini volumes store values for voxels i.e 3D pixels. The values stored can be position, size and orientation, etc. Depending on the type of volume different information is stored. For this project the necessary information needed from the voxels is a signed distance field. SDF's are typically used for fluids though work equally well for polygons. An SDF stores the distance from a point to the surface in each voxel.

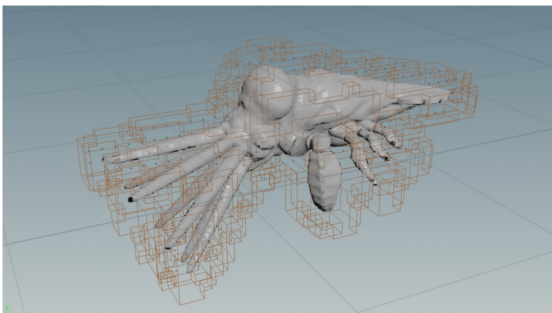

FIGURE 3.1: Convex hull surrounding "squab" geometry showing the exterior band.

The *VDB from Polygons* SOP converts polygonal surfaces into VDB volume primitives. This node can create a signed distance field encapsulating the newly converted volume. To understand the SDF, first consider VDB volumes, they have active regions or what is called interior and exterior bands. Increasing these bands increases the active regions i.e spreading out more voxels surrounding the volume containing SDF values. For the signed distance field, it is convention that any value contained within the surface has a negative value and voxels

outside the surface have a positive value. Furthermore, points on the surface of the geometry have a zero value. This surface is regarded as an isocontour. See Figure 3.2.

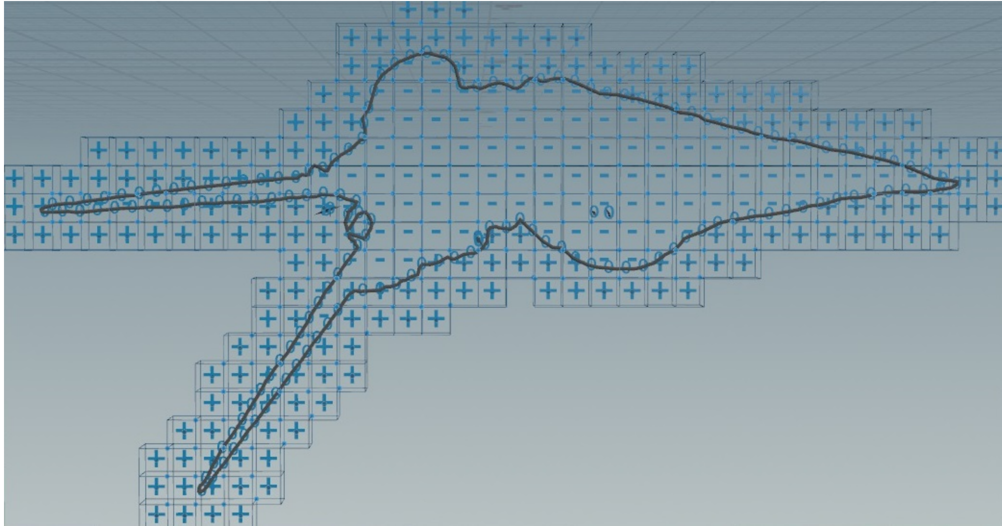

FIGURE 3.2: Depiction of a signed distance field encapsulating the "squab" geometry. Areas outside of the volume have a value of 1. While within the interior band each voxel has a negative value. Values on the isocontour are zero.

The VEX function `volumesample()` returns a float value which is the signed distance from a given point to the volume surface. The `volumegradient()` function returns a directional vector describing the direction of a low value to a high value. This is not normalised upon return of the value so depending on intentions, it is generally good to normalise it after it's been computed. Having access to both the distance and gradient allows for the necessary computations throughout this HDA described in Section 4.1.

4. Implementation

The foundation of the particle based approach chosen for this paper is based on a method described in a publication by Johannes Richter (Richter 2017). Richter approaches the generation of the Ivy stems using points and determining the placement of each point based on the previous point position. He describes the points having main forces controlling where they will be placed. An upwards direction enabling the plant to climb, a wander vector allowing the plant to grow in a random direction each step and the final being a wall direction which aims towards the closest object in the environment from each point.

The geometry in the scene is first inputted into a HDA called IvyGuides where it is converted into a VDB using a *VDB From Polygons* SOP, this generates a signed distance field around the geometry. Choosing the distance field to fill the interior band is necessary to avoid potential incorrect values further in the implementation. Filling the interior ensures each voxel contained within the geometry will have a negative signed value. Having this conversion contained within its own HDA makes the entire ivy generation quicker as once the geometry is loaded in and converted it need not be altered again and Houdini will not attempt to update it further.

4.1 Growth

Once the seed points are chosen and inputted into the IvyGenerator HDA the growth algorithm can begin. In the *attribute wrangle* node named stem_growth the main variables are initialized. This wrangle takes in three separate inputs, the first being the relevant points to work over, the second is the VDB from which the signed distance field is generated and the third is the optional sun object. Within the node a for loop begins with the max amount of steps the vine

stem can grow as its limit. A random vector is generated using the loops increment counter to ensure variation. This random vector is used as the wander vector and will be added to the overall direction calculated to allow the stem to develop in any direction each step.

Using the signed distance field created earlier both the distance of the position from the surface and the gradient to the surface are calculated using the functions `volumesample()` and `volumegradient()`. As the gradient points from a low value to a higher value, pushing out from the geometry, the value is negated for use as an attraction factor instead.

Should the user want a sun effect influencing the growth pattern, the new geometry is brought into the wrangle, independent of the environment using a `point()` function. This can access an attribute of the geometry in question, in this case the sun's position. The sun's influence on each point is calculated by normalizing the difference between the perspective position of the point and the sun object's position. The initial direction is then calculated. It is found by a summation of the different tropisms along with user specified influences.

This direction is then updated depending on certain conditions. Should the SDF distance of a position be greater than the defined threshold i.e no longer close enough to the geometry, the direction vector is changed. A clear issue would be if the vines continued to grow upwards without any object to cling to therefore the upwards growth is removed. Similarly if gravity is activated, the downwards direction and it's influence are added to the accumulated direction vector. This direction is then normalised and fit into specified bounds using the `fit01()` function.

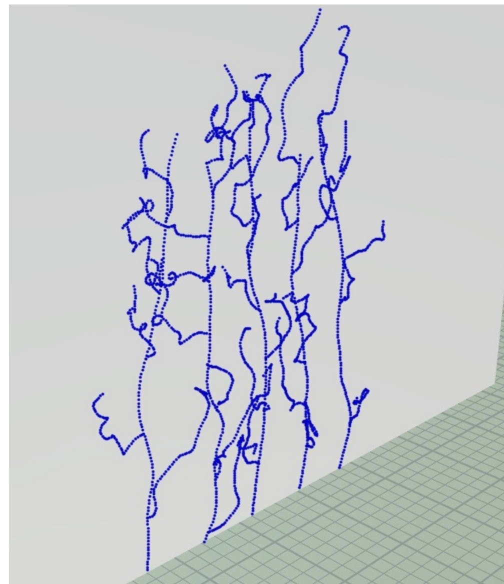

FIGURE 4.1: Stem and branch generation. Lines have been resampled after growth.

The functions `volumegradient()` and `volumesample()` are used once again on the newly calculated position evaluated by the summation of the previous

position and the direction vector. Should the `volumesample()` return a negative value the newly generated position is inside the interior band voxels, implying it is within the geometry. To correct this the position is pushed out along the gradient vector away from the volume. A point is then added using `addpoint()` and set to the newly found position using `setpointattrib()`.

In case a seed point was generated too far away from geometry and the gravity factor is turned off a condition is set using the `removepoint()` function. This removes any seed points on the condition the user has chosen to do so.

The resulting points are connected using an *Add* SOP. Since the points are contained within the for each loop the lines form individually regardless of the increasing point number, `@ptnum`.

Branches are generated in a similar fashion where the seed points are instead scattered points along the stem. Scattering points rather than using the generated points allows for more custom variation and control. The algorithm is also updated to allow extra customisation i.e choosing the height at which branches should grow. This allows for more stylized branches.

To address some potential collisions between points and the geometry, an *Attribute Transfer* SOP is applied. This transfers the normals of the scene geometry onto the points of the curve. Doing this allows for a slider to push the curve out away from the geometry along the normals to avoid any potential intersections created during the smoothing process. The wrangle containing this runs over all points and updates their *@P* position attributes by adding the geometry's normal.

Houdini's convenient *Smooth* SOP allows to remove any overly jaggy point positions on the stem or branches before adding a slight, more realistic distortion. This distortion is added using an *Attribute VOP* with a curl noise. This noise is particularly useful for drawn curves as the growth appears more natural. Parameters of the noise have been promoted to allow for customisation though very little change is typically needed.

4.2 Curve Growth

The curve growth is quite simple in comparison to the automation. Using a *Draw Curve* or *Curve* SOP and inputting them into the IvyGenerator HDA the stem is generated. In the network, *Copy* and *Point Jitter* nodes are added to allow for further customisations.

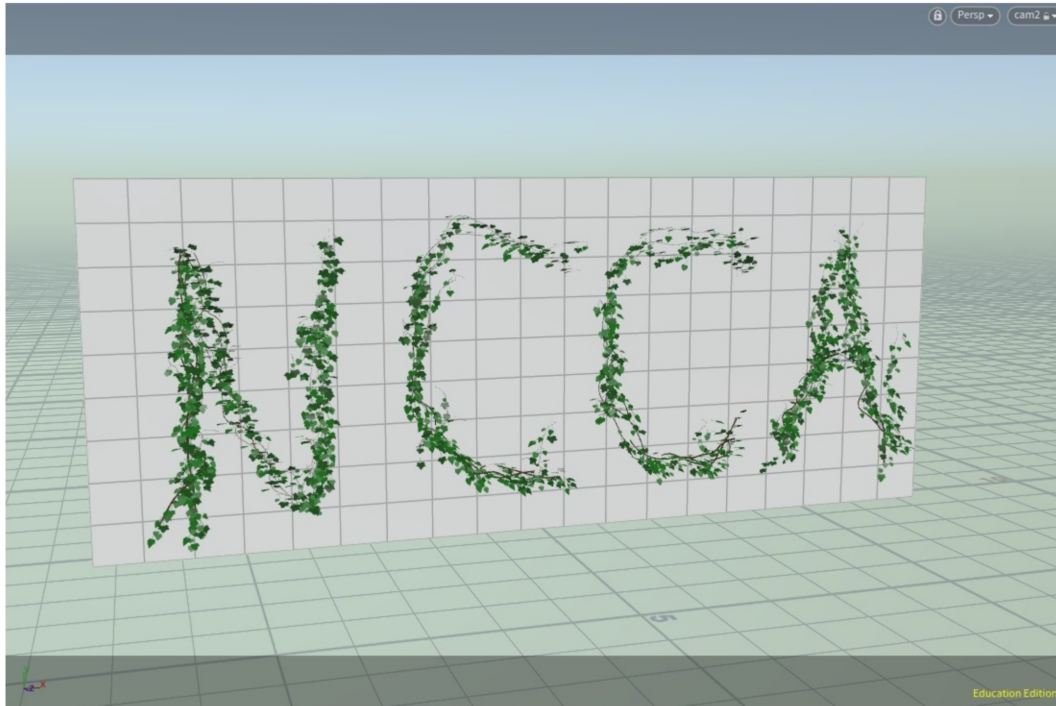

FIGURE 4.2: "NCCA" drawn on geometry using Draw Curve SOP with multiple strokes.

4.3 Tendrils

The tendrils are generated by projecting points onto the geometry. This approach is reminiscent of the method behind ray-casting though it only used for the tendrils as can be more computationally heavy. In a *wrangle* node with two inputs, the tendril seed points and the mesh geometry, the projected points are created.

Given positions on the stem, a normalised random vector is generated using `fit01()` and `rand()` functions. Using the current point position and the random vector, a normalised position is found radiating from the stem. A spread

factor is added to manipulate how far the tendrils may spread out along the geometry. Once the new positions are found, a projected position is found on the geometry using the `minpos()` function. This function finds the closest position on the surface of a geometry and returns that position. For a line to be created between points in Houdini it is first necessary to add vertices at each point. This is achieved using `addprim()` and `addvertex()` functions, creating the line between the vertices. As the shape of tendrils in actuality are curved and tend to be wider at the point of contact with an object, a ramp and curl noise are applied to the lines. This is achieved within an attribute VOP node.

4.4 Stem Scaling

For converting the lines into actual tube geometry a *Polywire* SOP is used. As with most vegetation, ivy is thicker at the base of the stem, where the plant is older and thins out along the branches. L-systems in Houdini have a built in parameter called the `@lage`, this is a float value determining the position of each generation in comparison to the root. Unfortunately, the nature of the system designed here does not have access to such an attribute nor would one be easily generated. Instead the primitives are grouped off upon generation, stemPrims and branchPrims and a *Resample* node is used to create the `@curveu` attribute. This attribute stores measurements of a curve from one edge to the other. This is ideal for determining the scale of each curve based on the `@curveu`. In two separate wrangles, one running over stemPrims and the other over branchPrims the `@pscale` attribute is defined using a ramp channel, created by `chramp()`. These ramps enable the user to vary the thickness of the ivy vines.

4.5 Leaves

In order to have realistic placement of the leaves the curve tangents of each point were found. This was done using a *Polyframe* SOP and activating the tangent attribute. Similarly to earlier in the network the normals of the environment geometry are found and also transferred onto the points. Combining these together in an attribute wrangle it is possible to manipulate the normals

FIGURE 4.3: Figure shows varying thickness of stems and branches using a ramp.

to preference. Using simple vector algebra the new normal direction is calculated by finding the difference between the geometry normals and the tangent normal. To customise this further an upwards direction vector is added to aim the points even more.

Since the resulting vectors are relatively conforming to a single direction, a jitter is added. In another wrangle node a random vector is generated along with an influencing factor which are added to the normal directions. The random vector allows for variation on each point.

In other wrangles more options are made available to the user. Using an if statement and a `removepoints()` function, potential leaf points are removed. In one wrangle it is solely based on height `@P.y`, however in the other these points are removed depending on a direction set by the user. This could be used to portray areas of seemingly dead branches where leaves no longer grow.

Using an *Attribute Randomize* SOP an attribute `@pt` is created. This attribute will be used to determine which objects get copied to particular points and with what weighting, adding further artistic control for the user. This is achieved using a custom discrete distribution type within the *Attribute Randomize*. Contained within a for each loop is a switch node, this node has been modified to

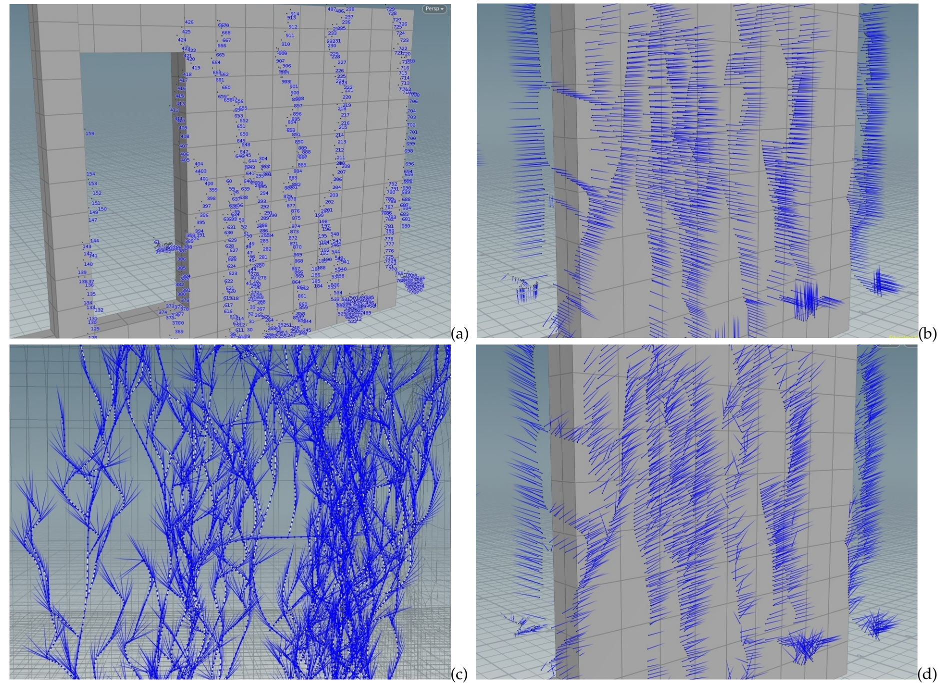

FIGURE 4.4: (a) Points scattered along ivy stem, (b) Normals transferred from the wall geometry to the scattered points (c) Curve tangent attributes per point (d) New computed normal direction for on each point for whitch the leaves will be copied according to.

change input depending on the `@pt` attribute. The *Copy to Points* SOP packs and instances the geometry before copying making the process faster.

4.6 Collision Detection

Dealing with a high amount of leaves an obvious issue that arises is leaf collisions. For these collisions the copying leaves to points approach is modified. Various tests were carried out to determine the most efficient method of intersection analysis in Houdini and are described in Section 6.

Where every for each loop prior to this in the network merges on each iteration, instead this method manually merges the geometry. With the for each begin loop set to "fetch piece or point" and a second to "fetch feedback", the collision detection can begin using intersection analysis. Houdini provides a SOP enabling this. During each iteration of the for each loop we must determine which leaves to copy and which to not. This is done using a *Switch* node with two inputs, the *Copy to Points* SOP and a *Null*. The *Intersection Analysis* node checks if there is an intersection between geometries and returns the number of

intersecting points. If the number returned is greater than zero then the switch node changes to have the null as its input, preventing the copy from occurring on that iteration.

To speed this process up the for each loop is contained within a *Compile Block*. In geometry networks, a compile block allows for multithreaded for each loops, meaning each iteration is spread across multiple cores. These benefits come with certain restrictions however and in this case, the issue of referencing other nodes by name, compile blocks do not allow this. Compile blocks require all inputs to be static i.e they are the same for every operation. Due to this, spare inputs were instead created to access the information necessary in the intersection analysis. It is also required to fully encapsulate the for each loop within the compile block. Every input into the for each loop requires a *Compile Begin* and a *For Each Begin* node. The added for each begin blocks are set to ”fetch input” to make the block fully compilable.

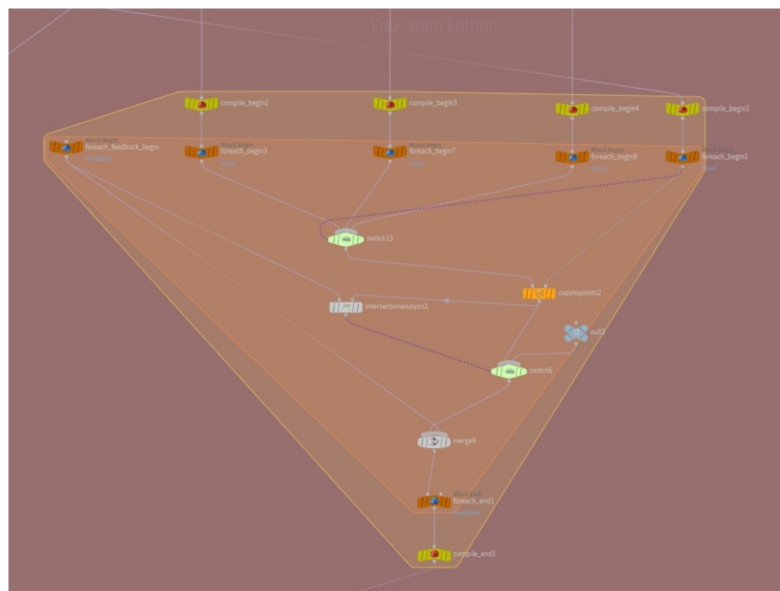

FIGURE 4.5: Image shows the collision detection method implemented in this HDA. The for each loop is encapsulated within a compile block to help speed. (Personal collection)

5. Known Issues + Solutions

5.1 Bunching

One issue that arises during the ivy generation are certain branching collisions. In general, with ideal parameters set this issue becomes unnoticable in a scene with the use of a *Fuse* SOP. However, for some cases should the max amount of steps the ivy is allowed to grow be too large and the geometry in the scene is small, bunching tends to occur at the highest point of the geometry. With the *Fuse* SOP added to counteract this as well as the ability to trim away points outside the SDF the HDA can be tweaked to prevent this. It can still however cause unfortunate bunching should the steps be too high.

5.2 Merged Geometry

When converting the polygonal geometry into the VDB volume the interior and exterior bands are generated. For geometry that is simply merged together with intersecting meshes the SDF appears to disregard the intersecting cavity and points could be created inside the interior band of the SDF.

These cavities cause issues during the sampling of the SDF values. The function does not return a negative value causing points to be generated inside the geometry. The reason behind this occurrence is unknown at this point however efforts have been made to resolve it. In the IvyGeoGuides HDA the polygonal geometry is first converted into a fog volume then into an SDF using a *Convert VDB* SOP, for the SDF to register the interior and exterior bands it is put through a *VDB From Polygons* SOP once again to result in a functioning signed

distance field. This solves the problem satisfactorily however the multiple conversions take longer to cook. It is recommended this option is chosen however should there be intersecting meshes in the environment geometry and after any changes are made to the geometry.

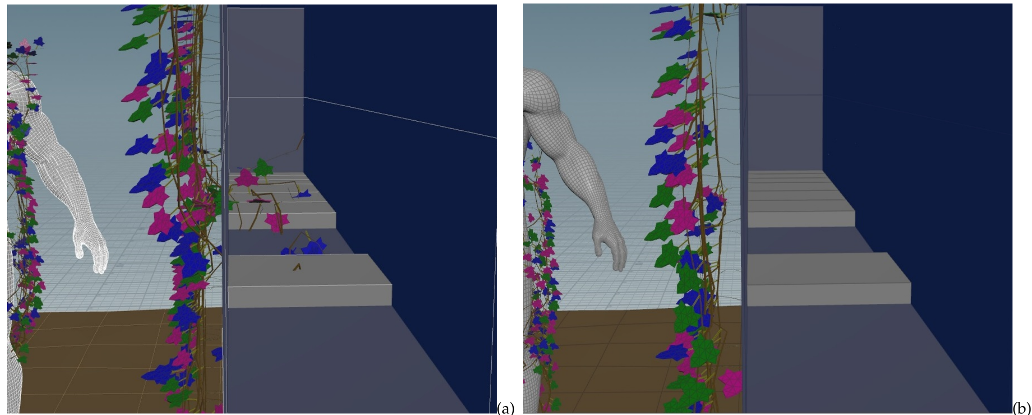

FIGURE 5.1: (a) Image shows geometry forming inside the environment due to intersecting faces of the wall. (b) After correction by adding a fog volume, stem geometry no longer generates in the cavities created by intersecting geometry.

5.3 Gravity

The effect of gravity in this tool has limitations and only works well with specific parameters et. If there is no ground plane present in the *VDB to Polygons* and unless the seed points are within the SDF range initially gravity's effect will just make ivy growth fall downward without attaching to anything.

5.4 Uniform Scaling

Unfortunately there is no quick way for uniform scaling of the entire ivy plant. To fit the geometry in the scene depends entirely on the parameters chosen. Add a unit box or sphere to the IvyGenerator HDA would initially yield strange results. To counter this appropriate values must be chosen i.e. max steps, step size and leaf size etc.

6. Performance

As this project aimed to create a quick, user friendly tool, necessary attention was given to the speed of the set-up. The process in which this tool works is highly iterative, which can undoubtedly result in costly time consumption. This first modification was made during the growth phase. Initially the branch growth was generated inside a for each loop as adding lines is done by the point number `@ptnum`. The foreach loop was necessary to separate each branching point so that individual primitives could be generated. This was altered to generate primitives using an `@id` attribute created in a point wrangle. Creating this attribute on the seed points of the branches allowed for the removal of the for each loop.

A lot of focus for speeding up the tool was aimed at the leaves. Houdini 18's updated *Copy to Points* SOP has a built in piece attribute which initially seemed the fastest way to copy the geometry. To use the piece attribute and for the fastest results the geometry needs to be packed before copying using a *Pack* SOP. While this did speed the process up, it did not work with the collision detection method. In order for the intersection analysis to work the geometry must have point attributes. Unpacking these attributes before copying made little to no difference making the approach redundant as though the geometry had never been packed at all.

As the intersection analysis node is computationally heavy, the entire for each loop was contained within a compile block. As mentioned earlier, in Houdini compile blocks enable a for each loop to become multithreaded allowing the program to run over multiple cores. Unfortunately, given a very high volume of points to copy to this process can still be exceedingly slow if the leaf geometry has a high poly count. A possible solution to this would be to alter the

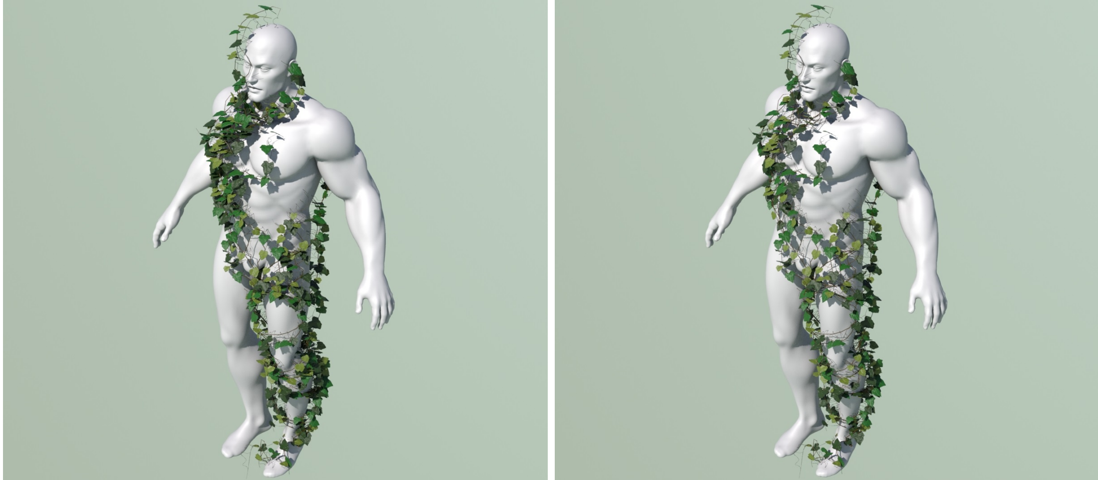

FIGURE 6.1: Figures show scene with and without leaf collision detection activated.

network so a low poly version of the leaf geometry is passed into the intersection analysis, then whichever points are chosen to copy, these could be fed into the simple for each loop along with the high poly geometry.

Another significant factor to note is the actual computer the simulation is working on. One early use of collision detection ran at 52 minutes with low poly geometry for 2663 points on a personal laptop compared to 6 minutes on a powerful desktop computer.

Should the leaf geometry have a relatively high poly count, the speed severely slows. Figure 6.1 shows the same scene of 504 scattered points. With collision detection activated it took over 3 minutes to evaluate compared to seconds for no collision detection. The pack and instance option available in the Copy to Pints SOP had to be turned off due to the material applied to the leaf objects.

To aid in the interactive performance of the HDA it is advised to disable "leaves", "branches" or "tendrils" in the main tab of the IvyGenerator HDA when manipulating seed points or drawing in the viewport. As the HDA will attempt to update the growth each time these relevant factors are moved, disabling these allow for smooth, fast drawing/moving in the viewport. Simply renable the toggles again when ready.

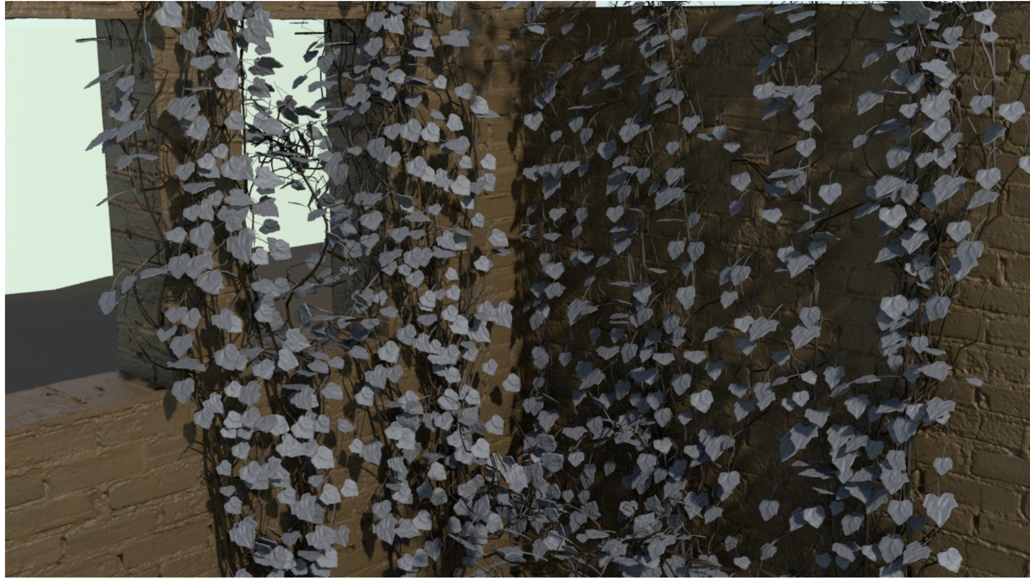

FIGURE 6.2: Ivy scene with collision detection without textures. A textured version is shown in Figure 8.4.

Name	Time
Total Statistics	1h 08m 42.3
Nodes	1h 08m 42.1
obj	1h 08m 42.1
scene	1h 08m 42.1
IvyGenerator1 (Sop/IvyGenerator)	1h 08m 42.1
compile_end1 (Sop/compile_end)	1h 08m 41.3
normal5 (Sop/normal)	0.579s
merge10 (Sop/merge)	0.172s
merge5 (Sop/merge)	0.082s
polydoctor1 (Sop/polydoctor)	0.020s
switch16 (Sop/switch)	0.005s
attribdelete2 (Sop/attribdelete)	0.003s
compile_begin1 (Sop/compile_begin)	< 0.001s
switch7 (Sop/switch)	< 0.001s

Education

FIGURE 6.3: Performance monitor showing over an hour to process collision detection for the rendered scene shown in Figure 6.2.

7. Further Work

Though the tool is wrapped up nicely in a portable HDA, the potential for further work is always present. Extra factors could be added, for example pseudotropisms and animation. Initially, another aim of this tool was to allow for easy animation of the growth using multiple *Carve* SOPs and relevant timeshifts or even a wire solver. However, due to time constraints the final method of animation for each part of the ivy growth was not completed and instead removed from the network.

Taking from Benes and Mill (2002) the idea of traumatic reiteration could be introduced. The phenomena occurs when the plant is unable to grow along its current path any more. If traumatic reiteration were to be added to this HDA, obstacle avoidance could potentially be implemented.

Another attempted approach for obstacle avoidance was by using heat maps and a gradient driving the direction of growth, should certain points of the geometry have a low heat value the growth would ignore that part of the geometry, instead finding it's way along an increasing path. Having obstacle avoidance implemented in this system would have allowed for even more customisation and potentially more realistic development in the ivy.

As it stands there is a limitation on the amount of different geometries that can be used as leaves. This is relatively simple to adjust however if the user is familiar with Houdini beyond the top level interface. Though access would need to be available to dive inside the network and create another merge node.

8. Conclusion + Discussion

The HDA created alongside this thesis paper successfully generates procedural ivy. It is an easy to use tool and allows for various customisation's. A variety of parameters are available to manipulate depending on the artists preferences. The final result can produce realistic environments that are fully art-directable in line with the original objectives of this paper.

The procedure used to generate ivy using this tool begins with the use of the IvyGeoGuides HDA, where the VDB and SDF are created. The outputs of IvyGuides input directly into the IvyGenerator HDA and seed points can be generated by manually adding points to the third input of the HDA or simply using a scatter node. There are options to copy these points with varying translations in order to generate more seed points in similar locations.

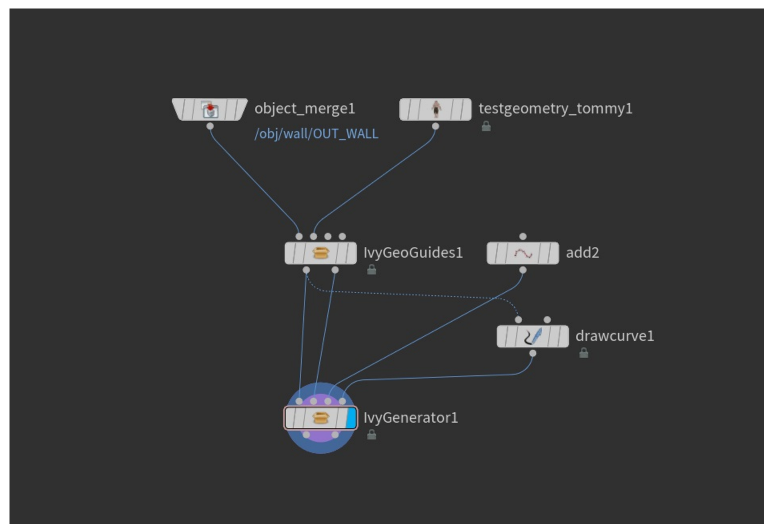

FIGURE 8.1: Typical network set-up using both IvyGeoGuides and IvyGenerator at geometry level.

The HDA also provides the ability to draw curves, taking curves into its fourth input. These curves can be projected directly onto the geometry or randomly

positioned in space for a significantly stylised approach to the ivy. This ability allows for highly art directed generation of plant growth.

The automatic growth begins with the seed points, the tool considers different tropisms that will affect growth and allow the user to vary the influence of these using appropriate sliders within the user interface. Should the user wish, branching can occur for an even more realistic look and similar to the stem there are various factors that can be applied to the branches as well as options to manipulate the growth. Including whether or not a branch should grow based on the height of each potential branching point.

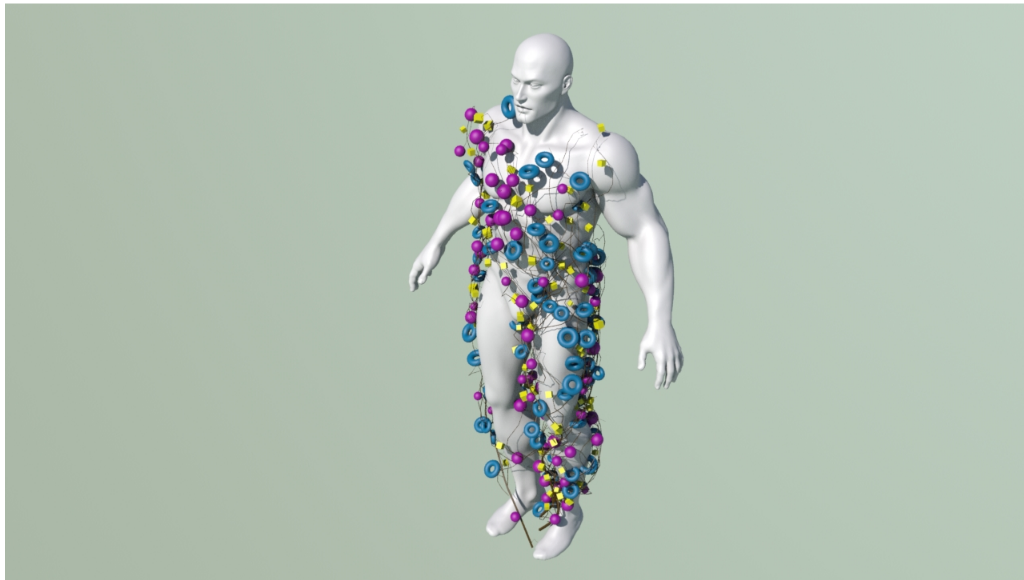

FIGURE 8.2: Leaves swapped out for custom geometry.

As some ivies do not continue their climbing growth once they have reached the top of their supporting environment, there is the option to trim the ivies in which case regardless of the maximum amount of iterations of growth, once the ivy has begun to grow too far away from the geometry it is trimmed.

The HDA provides the option of adding tendrils to the ivy plant, showing how the ivy sticks to the wall, parameters for these can be tweaked as per user preference such as the maximum distance the stem can be before the tendrils stop trying to attach to the object's surface or how far they spread out from the stem.

The leaf parameters also allow variation in distribution of the leaf geometry and orientation. There are default geometries built into the HDA for ease of use

along with options to use other objects in the scene file or files on disk such as bgeo or obj files.

Both HDAs come with a help card to aid the user experience. The over-all result of the IvyGenerator HDA can provide both realistic ivy growth or even surreal scenes while using custom geometry.

FIGURE 8.3: Render of Ivy scene without collision detection using high poly leaf geometry.

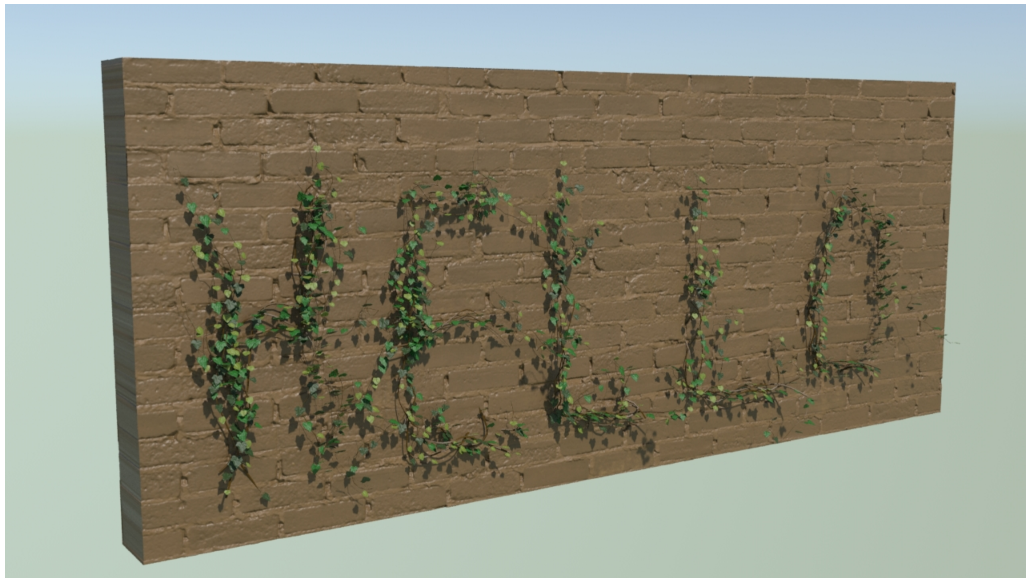

FIGURE 8.4: "Hello" drawn on wall using Draw Curve SOP set to preject onto geometry.

Bibliography

Arvo J. and Kirk D., 1988. Modeling plants with environment-sensitive automata. In *In Proceedings of Ausgraph'88*, 27–33.

Benes B. and Millán E. U., 2002. Virtual climbing plants competing for space. *Proceedings of Computer Animation 2002 (CA 2002)*, 33–42.

de Reffye P., Edelin C., Françon J., Jaeger M. and Puech C., 1988. Plant models faithful to botanical structure and development. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, New York, NY, USA. Association for Computing Machinery, 151–158.

Greene N., July 1989. Voxel space automata: Modeling with stochastic growth processes in voxel space. *SIGGRAPH Comput. Graph.*, **23**(3), 175–184.

Kaandorp J. A. and Kubler J., 2001. *The Algorithmic Beauty of Seaweeds, Sponges, and Corals*. Springer-Verlag, Berlin, Heidelberg.

Kniemeyer O., Barczik G., Hemmerling R. and Kurth W., 2008. Relational growth grammars – a parallel graph transformation approach with applications in biology and architecture. *Applications of Graph Transformations with Industrial Relevance Lecture Notes in Computer Science*, 152–167.

Měch R. and Prusinkiewicz P., 1996. Visual models of plants interacting with their environment. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH 96*.

Prusinkiewicz P. and Hanan J., 1989. *Lindenmayer Systems, Fractals and Plants*. Springer-Verlag, Berlin, Heidelberg.

Prusinkiewicz P. and Lindenmayer A., 1990. The algorithmic beauty of plants. *The Virtual Laboratory*.

Reeves W. T. and Blau R., 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. SIGGRAPH '85, New York, NY, USA. Association for Computing Machinery, 313–322.

RHS , 2020. Hedera helix [online]. Available at https://www.rhs.org.uk/plants/43091/Hedera-helix/Details, [Accessed 7 August 2020].

Richter J., Oct 2017. Plants in houdini. *3D World*, (225).

SideFX , 2020. Vex [online]. Available at https://www.sidefx.com/docs/houdini/vex/index.html, [Accessed 17 August 2020].