Space Colonization Vines

Adam Khaddaj and Karim Tantawy

COMP4900

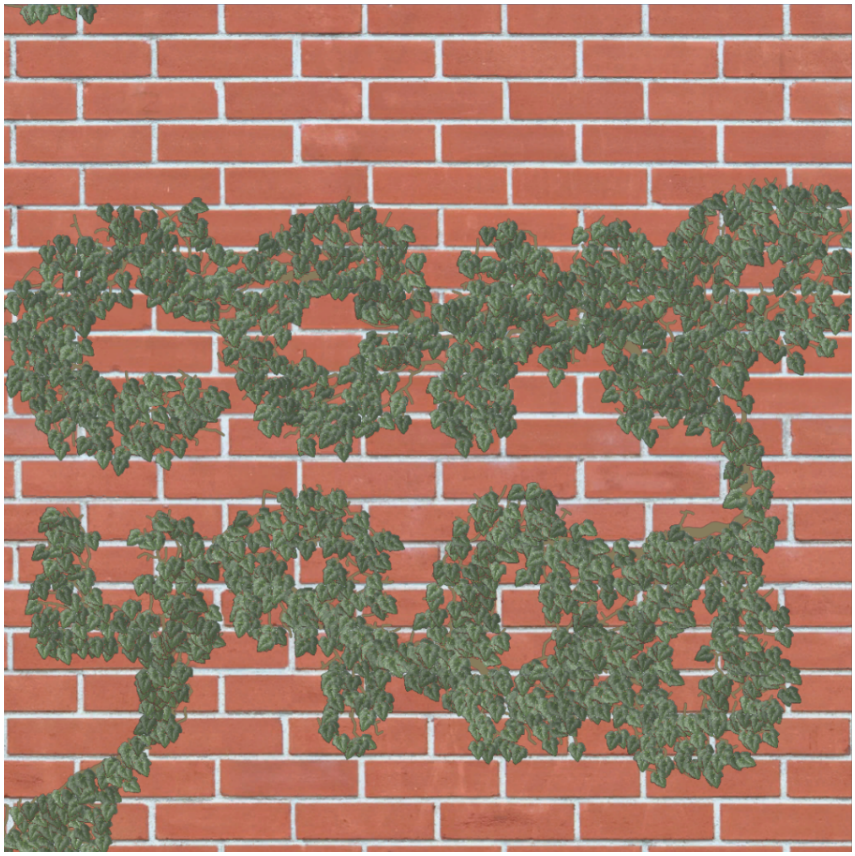

Figure 1: COMP 4900 course code drawn onto a texture using the vine generation tool. Vines rendered at 350 vine resolution.

Abstract

Vine growths are visually striking phenomena in that occur in natural vegetation, and can play a large role in transforming the atmosphere of an outdoor environment. This project presents a 2D vine growth generator and renderer that stores the traversal of vines in UV space, and renders these vines to a texture in real-time. This is primarily achieved through an implementation of Space Colonization Algorithms, first introduced by Runions. The project also implements an interface in Unity that allows users to easily fine tune several parameters, allowing them to determine where and how vines grow.

1. Introduction

1.1 Motivation

Several tools exist to help artists generate 3D vines as meshes. However, not many 2D options exist to help in the generation of vine growths. A 2D alternative would be beneficial for several reasons such as the performance benefits of rendering a 2D texture as opposed to a 3D mesh. This would also be preferred to the alternative of artists manually creating vine textures by hand which is a tedious and time-consuming process that does not allow for much experimentation or variation. This is important because vine plants, unlike trees, can not be copy-and-pasted in a scene as their growth is highly dependent on the surface and underlying object that it is growing on. This means that many variations of vine growths are required for different contexts to create a convincing scene. Many of the tools on offer also vary in terms of their realism and adherence to the vine plant's biological behavior. A more realistic and visually consistent method that provides artists with a great amount of control over the result would be preferred in this context. This would help artists quickly iterate and generate several textures using a more automated approach without sacrificing their input.

1.2 Goals

The primary goal was to create a tool that allows users to generate vines that grow on top of textures, with several parameters for changing vine growth behaviour. This was achieved by implementing a space colonization algorithm similar to the one described by Runions et. al [1], but modified and optimized for use in 2D space. The main idea is to populate the UV coordinate space of a texture with *attraction points* and *growth nodes*, represented as C# objects, that interact with one another to guide vine growth traversal throughout 2D space in a way that simulates "space competition" typically seen in vines and vegetation.

Users should be able to modify several parameters that impact how vines grow. Inclusion and exclusion zones can be created that determine where vine growths can spread to, allowing users to exclude or include certain areas of a texture or 3D model. Attraction points and growth nodes can be assigned "tags" that determine how they interact with one another, allowing multiple vine growths to overlap with one another while others compete for the same 2D space. Attraction points can be "wobbled" around, and certain parts of vine growths can be randomly halted, both of which cause interesting growth patterns to occur.

All of this is implemented while taking advantage of Unity's editor interface. This not only allows users to toggle and alter parameters easily, but also makes use of Unity's raycasting and RenderTexture features to allow users to manually draw the aforementioned inclusion and exclusion zones on top of object surfaces.

Finally, vine path generation will happen separately from vine rendering, to make real-time vine generation possible. This requires simply storing the growth nodes for later use in a custom shader.

2. Previous Work

The foundations of this project is built off the work by Runions et. al [1] on Space Colonization Algorithms, which is a method for simulating the realistic growth of vegetation in 2D or 3D space. The algorithm achieves this by populating a space with *attraction points*, which influence the spreading and growth of *growth nodes* that will eventually determine the path that vegetation grows onto. Space colonization is particularly effective at simulating "space competition" between different growth node branches, since growth nodes remove attraction points upon reaching a certain proximity threshold, halting other growth nodes from growing on top of one another. Runions primarily uses this to generate trees in 3D space, since it guides the growth of branches without having them intersect one another, however the algorithm can be applied to any kind of growth where natural spreading and space-competing occurs. This does propose a unique problem with our projects goal, however, which is that vines do not always perfectly compete for space, and oftentimes grow on top of one another. A solution to this problem is explained in section 3.6.

The algorithm is relatively straightforward. First, root nodes and attraction points are placed throughout the space. Then, determine which attraction points are influencing which growth nodes, based on a set "influence radius". For each growth node, get the average direction of all attraction points that are influencing it. Using this direction, determine the position of new growth nodes, and create them. Then, check whether any of these new growth nodes are in the "kill radius" of an attraction point, indicating that the attraction point should be removed. Remove these attraction points, and then perform the same steps again. Figure x shows each step of this algorithm in a 2D example.

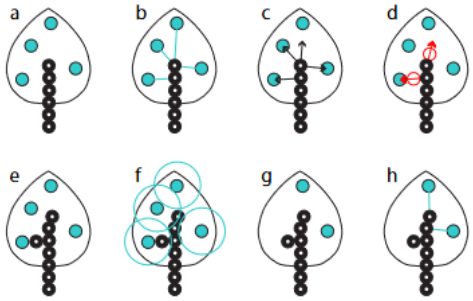

Figure 2: The steps of the space colonization algorithm. From Runions et. al

Several vegetation generation methods that do not use space colonization have found success as well. These methods typically make use of L-systems, or Lindenmayer systems, which are ways to represent complex growing rules of shapes found in nature (typically

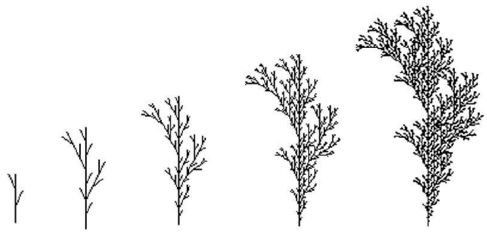

Figure 3: Vegetation growth with L-systems. From Gardeaux

vegetation). L-systems are composed of a unique alphabet, an initial axiom, and a list of production rules. Starting with the initial axiom (a starting set of symbols), during each iteration of growth, the production rules dictate how each symbol in the alphabet should be replaced (the letter A may be replaced with two A's, for example). This is repeated to create natural branching, fractal-like behaviour. While L-systems are effective for modelling many kinds of vegetation, they were not implemented in this project for several reasons. Firstly, as previously discussed, a large focus of this project is being able to give users fine control over growth parameters, which is easier to achieve with space colonization as the rules for growth are less rigidly defined. Furthermore, this project intends to explore a mixture of having vines compete for space and grow on top of one another, something that space colonization is particularly effective at modelling and controlling. For these reasons, L-systems were not sued.

3. Implementation

3.1 Work Environment

This project was created using Unity, for several reasons. The editor interface makes receiving visual-feedback easier and also acts as a nice template for the eventual vine growth interface that this project implements. Furthermore, several Unity features are used by the tool such as the RayCast class to determine where a user clicks on the surface of a mesh and determine the resulting UV coordinates and the RenderTexture class which is fundamental to our method for drawing and animating the texture. We decided on Unity because it is an engine that handles the importing of meshes and textures as well as 3D rendering while providing us with the previously mentioned libraries. This gave us an environment that allowed us to focus on the project itself rather than any boilerplate code required to run it.

3.2 Space Colonization Algorithm

To reiterate, the Space Colonization Algorithm proposed by Runion et. al involves populating a space with attraction points that influence the direction of growth for nearby growth nodes, who's positions will eventually form the paths for vines to be rendered onto. This project's goal is to use this algorithm to generate a path and store it in a data

structure that can be easily be read and traversed for later use in real-time. There are three classes that are used during both vine path generation and vine rendering. The first is an abstract parent class *Node*, which simply stores a Vector2 position, and a boolean indicating whether the node is 'active' or not (used later by its children classes).

This is inherited by the *Grower* class, which represents growth nodes that will eventually spread throughout the UV space to form the paths that vines will render onto. As path generation occurs, *Growers* will spawn new *Grower* nodes at adjacent positions, determined by a *growDirection* property that updates as nearby attraction points influence it. *Growers* store pointers to their parent and child nodes, essentially treating each vine growth as a doubly linked list. This type of data representation is especially beneficial for the purposes of this project, as it also allows for easily storing and updating values such as node depth (distance from the root growth node), and thickness (incrementing value based on how many nodes exist among children nodes). Several other benefits to storing growth nodes this way is elaborated on in later sections.

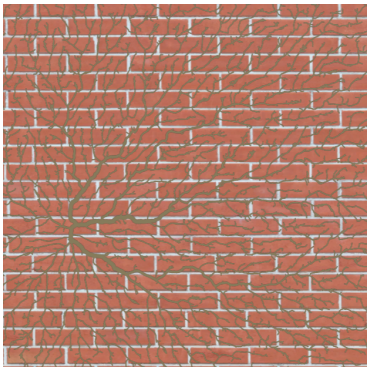

Figure 4: Vine texture generated at a resolution of 164. A limit is applied to the number of *Attractor* kills a *Grower* may have, with child *Growers* inheriting the number of kills of their parent *Grower* +-3

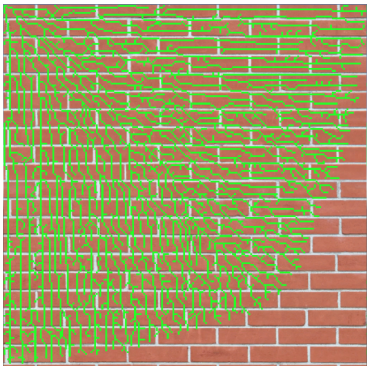

Figure 5: Vine texture generated with nodes existing on a grid rather than continuous 2D space

The *Attractor* class represents attraction points that influence the growth direction of nearby *Growers*, and eventually disappear should a *Grower* exist too close. Relevant attributes in this class include *influenceRadius* and *killRadius*, which determines how

close a *Grower* must be in order for the *Attractor* to influence it or for the *Attractor* to be removed by it, respectively. Several important functions are defined in the *Attractor* class as well, namely a function which determines the nearest *Grower* in the *Attractor's influenceRadius*. The specifics of this function are explained in section 3.4.

The relationship between node positions and their eventual "real" positions in UV space is an important consideration. Initially, all nodes were stored in a 2D array of length and width equal to the resolution size of the final vine texture, where their indices indicate their UV coordinates (after being mapped back to the range of 0-1). This made optimization trivial, since *Attractors* only needed to check indices within their *influenceRadius* during path generation. While this worked, it resulted in vine textures that were noticeably pixelated, since *Growers* could only exist in spots on this grid. This style may be beneficial for certain uses, but did not fit the visual style that our project aimed for, and so node positions instead began being stored in continuous 2D space, with the upper limit still being the resolution of the final vine texture such that it can be remapped back to 0-1 for eventual use in UV space.

Before the main loop of the algorithm begins, root nodes (the starting *Growers*) and *Attractor* nodes must be distributed across the UV space. Our project allows for both a random distribution of these nodes, as well as for users to manually determine "inclusion" and "exclusion" zones for where they should exist via a drawing tool. This implementation of these zones is explained in section 3.3. In either case, typically only 1 to 3 root nodes are placed, often at the corners of a UV map. *Attractors* are distributed through simple random distribution by iterating over the 2D space and having a 1/3 chance for an *Attractor* to exist at each location. More sophisticated distribution methods were considered for achieving better uniformity (such as blue noise sampling), however this random distribution seemed to perform very strongly, and did not require improvement, at least in the testing done for this project.

Now, the main loop of the algorithm can begin. During each iteration, all *Attractors* are iterated through. For each Attractor, if there exists a *Grower* within its *influenceRadius*, that *Grower* is set to be 'active', and has its *growDirection* updated based on its position relative to the *Attractor*. If that *Grower* is also in the *Attractor's killRadius*, the *Attractor* is killed (set to inactive). Recall that each *Attractor* can only influence one *Grower*, however a *Grower* can be influenced by many *Attractors*, and will have its *growDirection* eventually be the normalized sum of all nearby *Attractors* influencing it.

After this, all active *Growers* are iterated through, meaning all *Growers* that will be spawning new *Grower* nodes. For each such *Grower*, their child node position is determined based on the *growDirection*, and the child *Grower* is created, with the pointers of the parent and child updating accordingly.

This summarizes the core of the path generation, however several additional steps are included. Firstly, to determine when traversal ends, a condition is included that checks whether an *Attractor* has been killed within the last ten iterations of the main

loop. If this is not the case, we can safely assume path creation has ended, and break the loop. Secondly, several important values used during rendering and optimization are also updated during this path generation. A thickness value is recursively incremented whenever a child *Grower* is created, to influence vine thickness during rendering. *Growers* can also keep track of how many *Attractors* they have killed, which can be used in a variety of ways to promote branching and interesting behaviors (see figure x). Other additional steps are explained later in this chapter.

3.3 Inclusion/Exclusion Zones

The use of inclusion or exclusion zones is an optional step in our method that allows the user to define the areas of the texture where attraction points, and subsequently vines, can be generated. In [1], the authors describe this approach by explaining that the algorithm can "account for the presence of obstacles to growth, by eliminating the attraction points beyond the surfaces of collision." This is what we call an "exclusion zone" in our tool which defines where the attraction points can not be created. Inversely, there are also "inclusion zones" which define areas where attraction points can be placed. Any area not defined as an inclusion zone will not have attraction points in it. During this step, the user also defines where growth nodes will be placed in the texture and how many there will be.

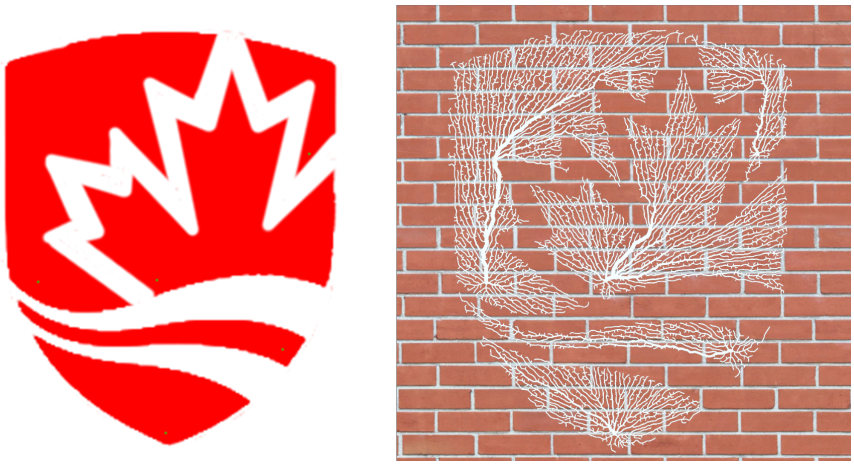

Figure 6: Carleton University logo as inclusion map(left), Resulting vine texture generated based off of the inclusion/exclusion map at a vine resolution of 450(right)

In our tool, these zones are stored as a simple map. That is, they are stored as a texture with red portions describing the inclusion/exclusion zones and green points representing the position of the root growth nodes. Figure 6 is an example of an inclusion map that has the Carleton logo in red defining where the attraction points can be placed and green points inside all of the separate parts of the logo defining where the root growth nodes should be positioned. How this is implemented is a simple check where the inclusion/exclusion map is sampled at the position where the potential attraction point will be placed. If the mode is set to inclusion and the potential position is red in the map

then an attraction point can be placed there otherwise that position is skipped. Inversely, if the mode is set to exclusion, then an attraction point can be placed at the potential position if and only if the position is not red in the map.

Figure 7: Exclusion zone defined using our simple drawing tool(left), Resulting vine texture generated based off of the exclusion map at a vine resolution of 150(right)

Figure 7 shows an example of a map being used to exclude the window area from having any attraction points placed in it. This is one of several use cases for the inclusion/exclusion step in the tool. It allows the user to precisely specify where the vines will grow, giving them more control over the final result. One can imagine several other examples such as painting an inclusion zone on the side of a building so the vines only grow on that wall or painting an exclusion zone around a fountain in a garden where the vines are regularly cut or pruned.

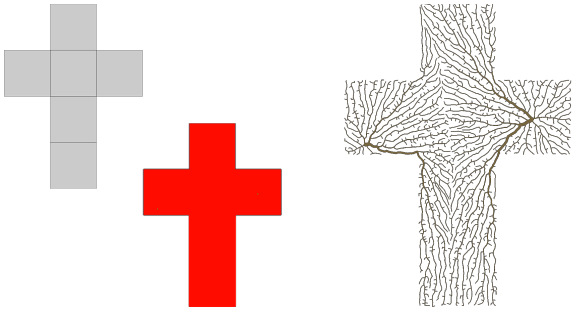

Figure 8: UV map of cube model(left), Inclusion zone texture created using a UV map(middle), Resulting vine texture generated based off of the inclusion map at a vine resolution of 250(right)

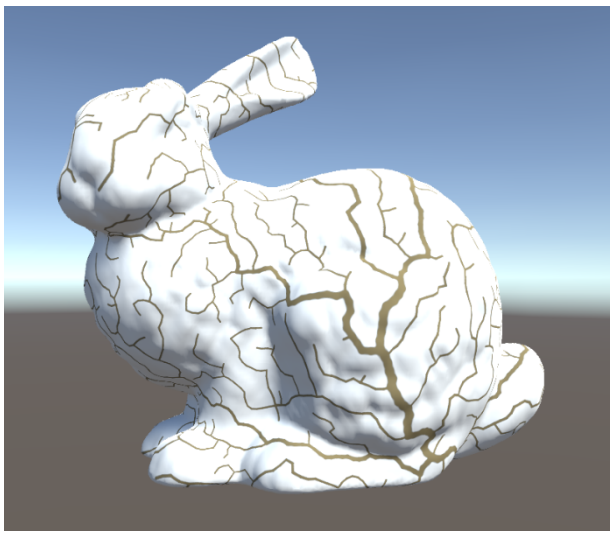

Figure 9: Bunny mesh with vine texture generated on it

Another benefit of the inclusion/exclusion texture is that when it is defined by the UV map of a mesh, it can ensure that vines only grow where they will be visible on the model. This can be done by exporting the UV layout of a mesh as a texture and then processing the image so the UV layout's regions are red(as is shown in Figure 8). However, this also has several limitations. Firstly, There will be discontinuities at the edges of the UV map where the vines will be abruptly cut off. Secondly, the scaling is not automated in our method so if the UV map is relatively small in the texture containing the UV layout then the vines will appear larger on the model itself which would require manually reducing the vine thickness, increasing the vine resolution, and increasing the resolution of the final texture to correct. Most importantly, we assume that the UV map is either continuous and/or there is no tiling or overlapping of the UV coordinates on the mesh otherwise the effect is not as effective or breaks altogether.

For the creation of the boundary texture our tool either takes a texture passed into it that was created in some image editing software such as Photoshop or it allows the user to draw the zones and growth node points directly onto the mesh using a simple painting tool. The tool consists of casting a ray from the point that the user clicks on the screen to determine the point on the mesh they are attempting to paint on. The UV coordinate of the point the user is trying to paint on is determined(using Unity's very useful RaycastHit class) and that is used to paint a circle onto a texture. The method of drawing onto the texture is the same as the method of drawing the vines onto a texture discussed in the *Rendering* section. This process is the same for drawing both the zones and the growth node points with the only difference being that the brush size for drawing the zones is adjustable while the other is fixed in size. An example of the tool is shown in the left image in Figure 7.

3.4 Texture Drawing

Drawing of the texture is accomplished by using "render textures", a feature in Unity that allows a rendering to be output directly to a texture [5]. The main benefit of using render textures in our case is the ability to draw to it using shaders. This provides two advantages, namely: drawing onto the texture is not resolution-dependent (since we are working in UV space and the texture resolution can be determined by the render resolution), allowing us to quickly change between resolutions on the fly, and it allows us to offload some of the work of generating the texture onto the GPU thereby improving performance. The alternative would be drawing to a texture by processing individual pixels which would require the implementation of scaling and rasterization for drawing onto the texture. This alternative method would also be much slower as it is done entirely on the CPU. Performance was a serious consideration for us as one of the project's goals was the dynamic effect of a real-time vine growth animation.

The process of drawing the vines involves creating a render texture that stores the final texture and iterating through an array containing the generated Grower nodes sorted according to a preorder traversal. In each iteration, a segment is drawn onto the render texture which is a line with its endpoint at the position of the current Grower node in the iteration and its start point at the position of the current node's parent. It is drawn using a simple shader that draws a line between the two positions passed into it as uniforms. The thickness and color of the line are also determined by uniforms passed into the shader. This is accomplished using the "Graphics" class's "Blit" method(provided by Unity) which takes two textures stored on the GPU and copies the contents of one onto the other using a provided shader[6]. By creating a temporary texture to store the current result and then "copying" the temporary texture back onto the final texture using the specified shader we can iteratively draw or "paste" the different segments onto the final texture.

This is the same procedure used to draw the leaves onto the texture with the only difference being the shader that is used during the blitting operation. The shader used for drawing the leaves takes a leaf texture, position, rotation, and scale to essentially "paste" that leaf texture onto the final result. Importantly, the leaf texture's wrapping mode must be set to clamp and the texture itself must be padded with transparent pixels around the border to prevent any visual artifacts when pasting it onto the texture. This is because if the wrapping mode is set to clamp and the bordering pixels are transparent then the texture can be transformed and pasted on top of the texture. For variation, the leaf texture's color is multiplied by a value within the range of $[0.8, 1.2]$ in the shader to vary its brightness. It is also translated by a slight offset along the Y-axis so that the stem of the leaf is what connects to the vine.

After this process is complete, the result is a rendered texture with the vine growth drawn onto it. This texture can then be used as a regular texture in unity or moved onto the CPU and saved as an image with the desired resolution. A simple material is used

in our project that takes two diffuse textures. One is the mesh's texture and the other is the vine texture and it draws the vine texture over top of the object's original texture. This is the preferred alternative to the more destructive approach of changing the object's diffuse texture by pasting the vine texture on it. We use this material to visualize the actual vine texture on objects in the scene.

As previously mentioned, the vines are iteratively drawn onto the texture and are stored in a pre-order. This means that drawing the segments with a delay between each iteration will result in an animation where the vines grow over time. The speed of render textures and the "blit" method allow us to play this animation in real-time on any mesh if the vine resolution is at a reasonable size. This is achieved using a Coroutine which can pause the execution of a method at a particular line and then return to it in the next update or after a set delay[7]. Pausing the execution of the drawing after every segment or every segment at height h of the tree is drawn allows the vines to be drawn over time.

This method of blitting to a render texture was inspired by "Peer Play" who demonstrates this concept in the context of creating a texture to be used as a map for tracks in a snow deformation shader effect[8]. Interestingly, this blitting technique provides us with the freedom to render the vines in different styles. Different shaders can be used to define how the vines and leaves will be rendered. For example, instead of drawing lines between Grower nodes a simple circle can be placed at their positions. This presents the opportunity to use the generated vine structure to create different effects such as a "corruption" effect that spreads over time or anything else.

3.5 Leaf Placement

Leaf placement is calculated by using the Grower nodes position in UV space. The leaf texture is scaled and translated to that position with a slight translation in the Y-axis to offset the texture so that the leaf stem is what is connected to the vine. During the iteration over the Grower nodes, we use a simple counter to create an interval between the leaves to be blitted onto the texture. This allows us to control the density of the leaves with a greater interval resulting in fewer leaves being generated. Optionally, you can also have leaves only grow at the ends of the vines by including a simple check that the Grower node does not have any children. The leaf scale is simply set to a fixed number +- a random offset to vary the leaves' size similar to how leaves grow to different sizes in real life.

The rotation of the leaves on the texture is determined by, firstly, finding a $startPoint$ and $endPoint$ along the vine path. These points are simply a parent Grower node's position for the $startPoint$ and a child Grower node's position for the $endPoint$. The $startPoint - endPoint$ vector is normalized and used as the direction vector to calculate the leaf's rotation. This is called the $segmentDirection$. This is done so that the leaf's direction can be along the vines. For the leaf texture, no rotation means that it is facing downwards and an angle $\theta > 0$ means a clockwise rotation. Performing the following

calculation results in getting the leaf's rotation:

$$\theta = \arccos((0, -1) \cdot segmentDirection)$$

In the above equation, we are calculating the dot product between the vector $(0, -1)$ which is the direction vector when the leaf has not been rotated and the normalized vector $segmentDirection$ which is the direction vector along the vine path. Getting the arccos of this value returns the angle between them and, therefore, the rotation amount for the leaf. Before this step, there is an optional step which is controllable by the user to add gravity to influence the leaf's rotation. This simply involves adding the $segmentDirection$ vector and a gravity vector $(0, -1)$ multiplied by some coefficient to increase its influence then normalizing that vector. The resulting vector is the new direction vector for the leaf.

Optionally, the rotation can also be offset +- a random angle within a range that the user can set within the tool to allow for more variety.

3.6 Optimizations

As explained in section 3.2, attractors must determine whether there exists growers in it's radius of influence. The naive solution is to have attractors compare positions with all growers during each iteration of the growth algorithm, however this unsurprisingly results in extremely slow path generation. Our project's solution to this issue is to implement the same optimization introduced by Worley [4]. The 2D space where nodes exist is segmented into a grid, where each index on this grid is an array that stores references to all nodes inside of that tile. The size of the tiles on this grid are set to be as close as possible to the size of attractor influence radii. This way, it is guaranteed that any attractor must only compare distances with growers that exist within a distance of one grid tile from it's own. This drastically reduces the number of distance comparisons attractors must make, and results in very a large boost in performance.

3.7 Additional Parameters

Our implementation includes many parameters that can influence growth behaviour to produce a variety of interesting visual results. Aside from those already mentioned, several notable ones will be discussed in this section.

An effect can be toggled that "wiggles" attractors in a circular motion as vine path generation occurs, as was recommended by Dr. David Mould. This promotes vine growths to curl in interesting ways, often creating more organic-looking vines (see Figure 11). The strength of this wiggle effect can be increased to enhance this effect, although, at high levels, it no longer preserves the structure of vines. To accommodate for this effect while still using the grid optimization method, grid tiles are always set to be at least the size of the attractor influence radius plus the maximum distance that the wiggle effect can displace them from their starting position, meaning that even if an attractor escapes the

borders of its designated grid tile, the only growers it can influence will still be within a distance of one tile from its original grid.

A naturally occurring effect when generating vine paths with multiple growth nodes is the presence of borders where opposing growths meet. If a user wants to control how vines interact with other growths (overlap, border, a mix of both, etc.), an option is provided to "tag" attractors and growers. These tags can be used to determine whether an attractor is able to interact with a grower of the same tag. For examples, one root node and its children may be given tag A, while another root node at its children can be given tag B. Then, when distributing attractors, they can be randomly assigned either tag A or B, resulting in vines being able to overlap with one another, as commonly occurs in many real-life instances. Furthermore, attractors can be assigned any number of tags, meaning that some will interact with all growers while others will only interact with fewer ones. This creates a mix of overlap and space-competition (see Figure 10).

Many other parameters can be toyed with, including attractor influence radius, attractor kill radius, attractor density, limitations on growers' kill count, etc.

Besides the previously mentioned vine parameters, there are several parameters to influence the generation and placement of the leaves on the vine. The user can toggle drawing the leaves or drawing them at the ends of the vines only. They can swap the leaf texture with other textures similar to other tools such as Speed Tree. The user can scale the influence of gravity on the leaves' rotations and increase the interval at which the leaves are placed resulting in fewer leaves on the vine. They can also specify a range for the offset that will be randomly added to the rotation and scale of the leaves.

Additionally, There are inclusion/exclusion parameters that allow the user to draw the inclusion/exclusion zones using our simple drawing tool as well as place root Grower nodes on the texture. Optionally, the user can just provide a texture that includes the zones and the growth points. This option is there if the user wishes to use some image editing software for creating the texture with the inclusion/exclusion zones.

4. Results

4.1 Analysis

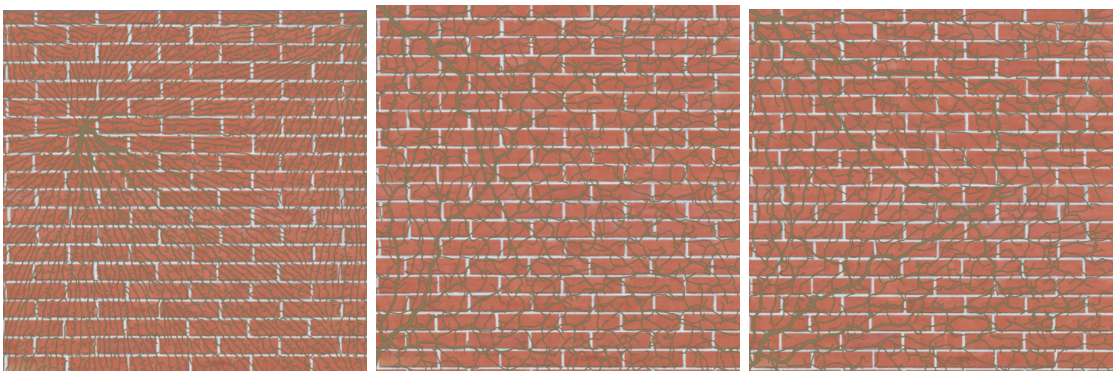

Figure 10: Vines rendered in 128 resolution, showing the use of tagged attraction points. On the left, tags are unused, causing natural borders between vine growths. In the center, tags are setup such that vines can freely grow on top of one another, by having tag-exclusive Attractors. On the right, Attractors can possess both tags or only one tag, cause a mix of overlap and space competition.

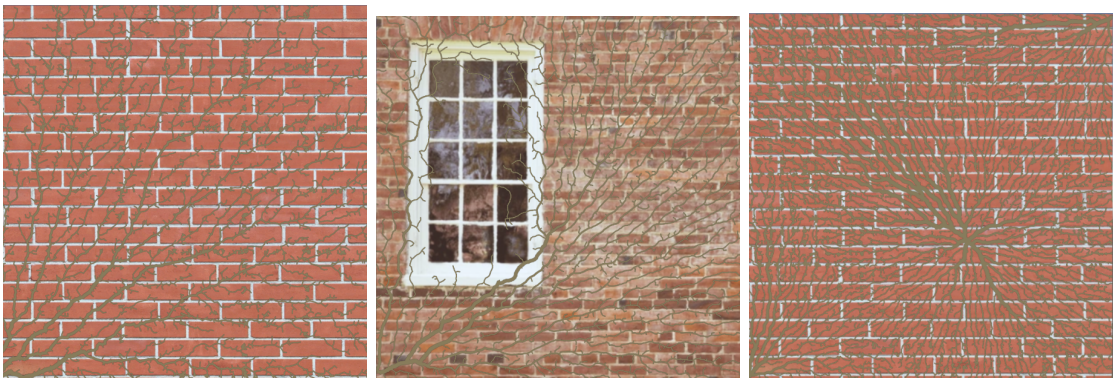

Figure 11: Vines rendered with a wiggle intensity of 1. On the left shows how wiggling promotes interesting growth patterns. In the center, wiggling causes vines to slightly trespass exclusion boundaries. On the right, wiggling causes Growers to be pulled past one another, causing a mix of overlap and competition, even when tags aren't used

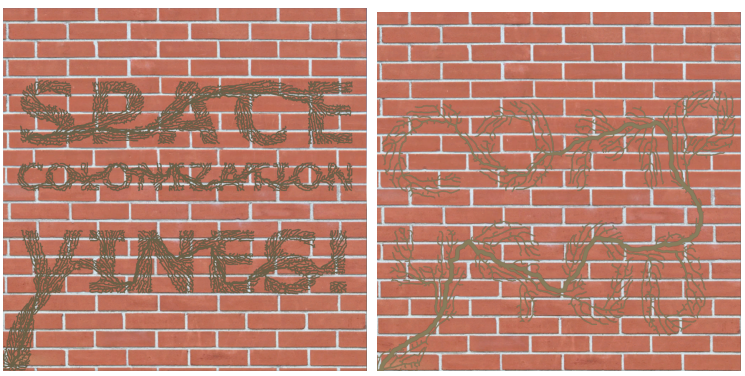

Figure 12: Vines rendered using *inclusion* mode to produce text

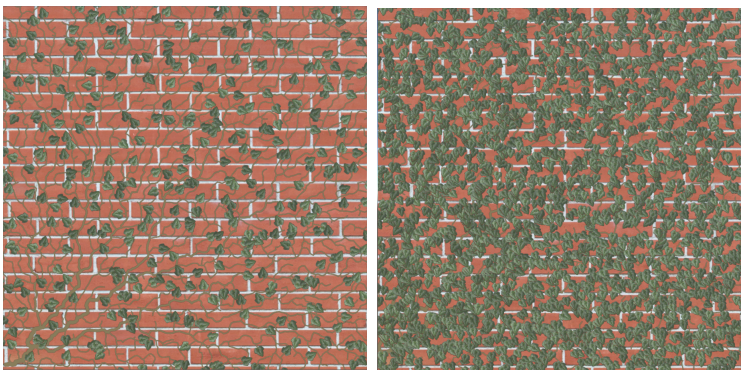

Figure 13: Vine with a single root Grower node at the bottom left with leaves only being generated at the ends of the vines with a vine resolution = 164(left), Vine with a single root Grower node at the bottom left with leaves being generated all along the vines at a leaf density interval = 2, leaf gravity scale = 0.8, Random Leaf Rotation Offset = 45 degrees, and vine resolution = 164 (right)

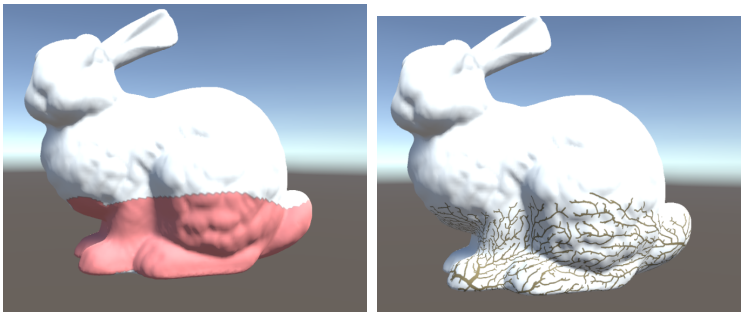

Figure 14: Bunny mesh with inclusion zone painted along the bottom of the mesh(left), Vines generated using a wiggle = 0.5 and a grower kill limit = 20 at a vine resolution = 400

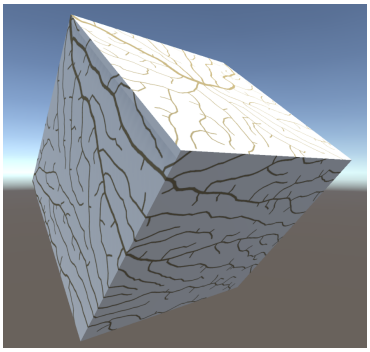

Figure 15: Cube mesh with a vine texture generated using its UV layout as an inclusion map. vines growing to or from the top face exhibit discontinuities.

Figure 9 shows how different vine growths interact with one another with and without the use of tags. The overlapping vines create a very nice "overgrowth" effect, however when combining overlap and space-competition together, the visual difference isn't terribly noticeable.

The Attractor "wiggle" effect not only promotes more interesting growth patterns, but also has a variety of very interesting applications. Because wiggling Attractors allows them

to trespass inclusion/exclusion zones, this ends up in them only slightly breaching certain areas, mimicking how vines often slightly grow onto certain surfaces before halting, as seen in Figure 10. Furthermore, wiggling Attractors provides a secondary way for creating a mix of vine bordering/overlap with other vine growths, as the wiggling Attractors will draw Growers past one another. Wiggling does unfortunately sometimes cause vines to "collapse" on themselves, likely caused by the Grower chasing the Attractor in a circle and spawning children on top of itself.

By limiting having it so that Growers deactivate after killing a certain number of Attractors, and having child Growers inherit their parents' kill count +- some offset, it causes longer branches of Growers to halt, giving other smaller branches of Growers the opportunity to grow towards the now uncontested Attractors. This is a great way of promoting branching in the vines, as shown in Figure 3, rather than having fewer, dominating long strands of Growers, as shown in most other figures.

Generally, when adding leaves we can get appealing results especially when increasing the density and the influence of gravity on the leaves' rotations such as in Figure 13. This exhibits a similar appearance to a vine dense with leaves in reality as with more leaves the weight becomes heavier and they tend to start drooping downwards due to gravity. The fact that our implementation only uses a single leaf texture, however, does leave a lot to be desired. The repetitive stamping of the leaves is obvious even with the variations in color, scale, and rotation are applied.

Figure 14 shows a good result in terms of applying the texture to a mesh. Since the bunny has a single continuous UV map for the body, continuous vines can grow along the feet of the bunny realistically as if the vines are climbing onto it. You can see a not so effective application in Figure 15 where discontinuities appear for the vines that travel from the sides of the cube to the top face. This is due to the UV map of the cube mesh which has an edge along the top face. This demonstrates a limitation of our method in that discontinuities, distortion and different scales in UV maps are not processed or accounted for.

4.2 Performance

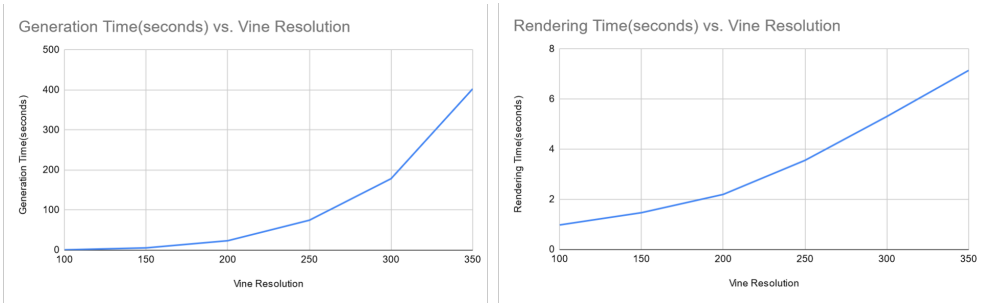

Figure 16: Performance charts: Vine generation time in seconds vs. vine resolution(left), rendering time in seconds vs. vine resolution(right)

The performance metrics in 16 were collected by running our implementation on a PC with an Intel Core 15-11600KF @ 3.90 Ghz processor, 16.0 GB of RAM and an Nvidia Geforce RTX 3060 desktop graphics card.

As the left graph in 16 shows, the time taken to generate the vines grows fast when increasing the vine resolution. The vines generated are simple vines with two root Grower nodes at the top-left and bottom-left corners of the texture. Additionally, performance actually improves with the use of inclusion/exclusion maps. That is because they limit the number of attraction points being generated, therefore, requiring less processing when running the algorithm since the speed of the algorithm is bound to the number of attraction points in the space.

The right graph in 16 shows the time taken to render the texture. That is, to draw it onto a texture using the blitting operation mentioned. The final texture resolution is being rendered at 1024x1024 pixels. The vines rendered also include the leaves at the highest density possible by our tool. Because most of the work is done on the GPU using Unity's render textures and the blitting operation, this is a fairly fast process. However, as shown in the graph the time taken also grows rather fast when increasing the vine resolution.

5. Postmortem

We believe that the greatest strength of this project is how well it managed to implement and combine many different parameters. It's very easy to toy with different configurations of vine growth, and there's a lot of ways a user can creatively apply the vine generation tool.

Generally, we are very happy with how the generated vines look. The space colonization algorithm does a great job of maintaining the core structural components of growing vines, even when a large amount of randomness and strange parameters are thrown at it. The textures that can be generated using our tool also convey vines in a variety of conditions such as overgrown and unkept vines with a dense amount of leaves or vines that have lost all of their leaves. This is great as one of our goals was to be able to generate vine textures that could then be used in games that include lots of vines such as post-apocalyptic games where buildings have been taken over by vegetation.

We are also satisfied with the results of the dynamic element of the texture. We can get the vines to animate and grow in real-time which was a goal we were uncertain we could achieve.

The recommendation to wiggle attraction points had great results and gave a lot of ideas for other creative ways to toy with the space colonization algorithm. Specifically, it was interesting to see how a simple change was able to have so many different unexpected applications. This really shows how important it is to spend time toying around with generation.

The use of tagging Attractors and Growers produced fairly good results but was overall a bit underwhelming. It looks great when vines grow overtop of one another, but mixing overlap and space competition together doesn't produce particularly unique results.

A weak point of our approach is that we did not integrate lighting. A normal map for the vine texture can be generated after the texture is drawn by determining the normals from the height field, for example, as well as blitting the normal map of the provided leaf texture at their same positions. This is a potential solution that we did not have enough time to implement but would have greatly improved the final result. The lack of any lighting results in a flat-looking texture.

Another weak point is that the vines themselves lack any real detail. They consist of lines that are a solid color and don't exhibit any of the textures that vines have. If we had more time we would have wanted to create some texture on the vines by using some type of noise in the shader that draws the line segment.

An important goal of ours was to have the vines grow on meshes realistically and uninterruptedly. If we had more time, we would want to implement an algorithm that allows the vine texture to appear continuous on meshes with different UV maps such as those that consist of several islands as well as account for the distortion that occurs as a result of UV unwrapping. This would have seriously benefited our project and allowed it to apply to a larger set of meshes. Currently, our project just places the texture onto the mesh and makes several limiting assumptions such as that the mesh's UV layout is one continuous island with minimal distortion. It also does not account for wrapping to prevent discontinuities at the edges of UV layouts.

An extension that we had hoped to implement is that of negative phototropism. We wanted the vines to tend towards shade when growing on a mesh as the ivy plant does in real life. We had a fairly naive implementation in mind that highlights a limitation of our approach. The implementation we had in mind was that for every potential attraction point to be placed we would take the UV coordinates of its position and by determining the normal at that point on the mesh we could use it to calculate the dot product between the normal and the vector calculated by taking the sun's position minus the position of the point in world space. If the point is not a point that is sufficiently in shade then an attraction point would not be placed there. This is the same process as determining how much light is reaching a point in the Phong lighting model. However, during the drawing of the vines on the texture, the blitting operation does not provide the shader with the vertex data of the mesh so that would mean that an otherwise easily parallelizable process that could be done on the GPU would have to be done on the CPU by iterating through the triangles of the mesh and calculating the Barycentric coordinates until the triangle that contains that UV coordinate is found(which could be many triangles or none). This would be a very slow process and would not be worthwhile. An alternative approach would have to be found to move this work onto the GPU to improve the performance which would mean getting the vertex data of the mesh that we are drawing the vines

onto.

6. Conclusions

This project achieved an interesting application of Runion's space colonization algorithm by applying it in UV space to allow for intricate vine growths to be applied to the surfaces of objects. Several parameters can be changed within the interface, allowing users to finely control how and where vines grow and promote interesting growth patterns. Choosing to separate vine path generation from vine rendering allows for real-time vine growths that spread across surfaces, and the addition of applying leaf textures across the vines adds new texturing opportunities.

There are still many areas for improvement. Even with optimizations, vine path generation is still fairly slow at higher resolutions, and animating vine growth becomes slow when many triangles are being drawn at once. When applying vine growth to 3D objects, the seams of the UV map cause noticeable discontinuities. There are also many extensions that would drastically improve the results of the project, such as incorporating lighting.

Regardless, the project was successful in creating the visual style that we sought out at the beginning of the semester, and we hope to continue iterating on it in the future.

7. References

[1] A. Runions, B. Lane, and P. Prusinkiewicz, "Modeling Trees with a Space Colonization Algorithm," in Proceedings of the Eurographics Workshop on Natural Phenomena, 2007

[2] P. Prusinkiewicz, J. Hanan, M. Hammel, R. Mech, "L-systems: from the Theory to Visual Models of Plants", CSIRO-Cooperative Research Center, 2003

[3] https://www.researchgate.net/figure/The-first-five-growth-steps-of-a-plant-like-L-System_fig2_2

[4] S. Worley, "A Cellular Texture Basis Function", 1996

[5] "RenderTexture," Unity, https://docs.unity3d.com/ScriptReference/RenderTexture.html (accessed Apr. 17, 2024).

[6] "Graphics.Blit," Unity, https://docs.unity3d.com/ScriptReference/Graphics.Blit.html (accessed Apr. 17, 2024).

[7] "MonoBehaviour.StartCoroutine," Unity, https://docs.unity3d.com/ScriptReference/MonoBehavi (accessed Apr. 17, 2024).

[8] "Snowtracks shader - unity CG/C tutorial [part 3 - draw to splatmap]," YouTube, https://www.youtube.com/watch?v=-yaqhzX-7qo (accessed Apr. 17, 2024).