Procedural Forest Generation with L-System Instancing

Masters Project Report

Ben Carey
MSc Computer Animation and Visual Effects

August 2019

Contents

Abstract

Producing densely populated believable forests can be a challenging process in CG, due to the size and complexity of the geometry required. This problem is exacerbated when trying to render in real time, for example in gaming or interactive projects. It is often necessary therefore to employ procedural methods to generate the tree geometry and to employ techniques to improve the efficiency of the rendering. This paper is concerned with the implementation of one such method, by Kenwood et al (2014) [8], which uses carefully constructed instancing of L-Systems at shader level to efficiently produce a forest of varied trees. The implementation is built as a stand-alone tool in C++, using OpenGL and Qt Creator.

1. Introduction

The creation of large forest assets is one of many problems in computer graphics that cannot realistically be solved by 3D artists alone: while modellers and texture-artists can create an individual CG tree to a far greater degree of detail and realism than any procedural method, they run into two major issues when trying to replicate this work at the level of forests. The first is of course the amount of work required: producing each tree asset will take an artist a considerable amount of time, and individually creating the hundreds or even thousands required for a large forest scene is simply not plausible in the timeframe of any professional 3D project. Secondly, if each tree is created separately, the amount of memory required for the forest as a whole will likely be prohibitively large, potentially too large to be stored on most PCs; and even if doesn't exceed the computer's memory capacity, the render times for such a forest would probably be unacceptably long.

Consequently, procedural generation of vegetation has been the subject of considerable study, giving rise to methods like L-systems and Diffuse Limited Aggregation. Such techniques have been developed by studying the growth of plants and the many factors that influence them, and finding algorithms that mimic some part of this process. By employing stochastic elements, these methods can easily generate a large number of distinct trees, and can offer creators familiar with the procedure a number of ways to modify these trees' shapes to fit artistic direction.

Using these methods, memory usage is often still a problem however: if every tree is individually produced using some procedural method, this still requires storing a large amount of data for each one, and sending a large number of vertices to the renderer. This is a particular problem in computer games, or other cases where real-time rendering is required. One solution is to simply produce a small number of hero trees and then instance them on the shader; but this repetition is likely to be noticed, particularly in interactive media.

In this paper, we look at a method to produce a large forest scene with distinct trees in a memory-efficient way that allows for real-time rendering, following a procedure introduced by Kenwood et al in their paper '*Efficient Procedural Generation of Forests*' (2014) [8]. Their technique uses L-systems to fill up a cache of branches, and then performs an algorithm to instance these branches on the shader to form a forest. In this way we can take advantage of the efficiency of the GPU for instancing, but because the instancing is performed at the level of branches, we can still build up a unique set of trees. This project implements Kenwood et al's method as a C++ tool built using OpenGL and Qt Creator, and additionally performs various other techniques in the shader to further distinguish the trees and improve the final look of the forest, demonstrating the efficiency of the method by showing how much can be done in the shader without too greatly affecting the speed of the program.

2. Previous Work

2.1 L-Systems

L-systems are perhaps the most famous technique for procedural digital plant generation, created by biologist Aristid Lindenmayer as a mathematical model of plant growth (1968) [1]. The various systems and practices that have developed around them are explored in depth by Prusinkiewicz and Lindenmayer in '*the Algorithmic Beauty of Plants*' (1990) [5]. L-systems are a type of formal language with a 'rewriting system', consisting of an alphabet, an initial word in that alphabet called an 'axiom', and a series of 'replacement rules', defining a replacement of some letter or sub-word with some other word in the alphabet. An output string is then formed by repeated applications of the replacement rules on the initial axiom. For uses in computer graphics, certain letters in the alphabet can then be interpreted as commands to a 'turtle geometry' drawing system (Prusinkiewicz, 1986) [4], allowing us to produce a visual representation of the plant. The use of recursion allows L-systems to develop exponentially more complexity with each new application of the rules, mimicking the increase in complexity of plant life as it grows.

We can also introduce stochastic processes into L-system generation to allow the system to produce random variation in the plant structures it creates. This has been proposed by a number of papers, for example by Yokomori (1980) [2], who formalises a stochastic L-system as an L-system along with a probability mapping assigning each replacement rule a probability, with the requirement that the sum of the probabilities of all rules with the same precedent is 1. In this way, we can assign multiple replacement rules for the same variable and at each stage the system applies one of these rules with a given probability; this allows us to generate a large number of distinct trees from the same system.

Figure 1: plants produced by L-systems

2.2 Other Methods

In spite of the versatility of L-systems, there are a large number of alternative methods for procedurally generating plants in 3D applications. One criticism of L-systems is their inability to account for surrounding environment: while L-systems can describe individual plants well, they cannot simulate the interaction of such plants with outside objects, each other, or even themselves. This is obviously an important consideration when generating forest scenes.

One alternative system is 'Diffuse Limited Aggregation' (DLA): this is a 'kinetic phenomenon' first proposed by Witten and Sander (1981) [3], whereby an initial seed or set of seeds (called an aggregate), is placed somewhere on a lattice, then particles are spawned in random areas around the edge of the lattice and undergo Brownian motion until they make contact with the aggregate. When they touch the aggregate, they are added to it and stay in this position. In this way, the aggregate grows, and due to the nature of the construction it develops branching structures. By assigning constraints to the particle spawning, seed positions and Brownian walk, we can get some level of control over the structure produced; for example, we can avoid intersection of the aggregate with other objects by declaring that no particle inside another object can be added to the aggregate.

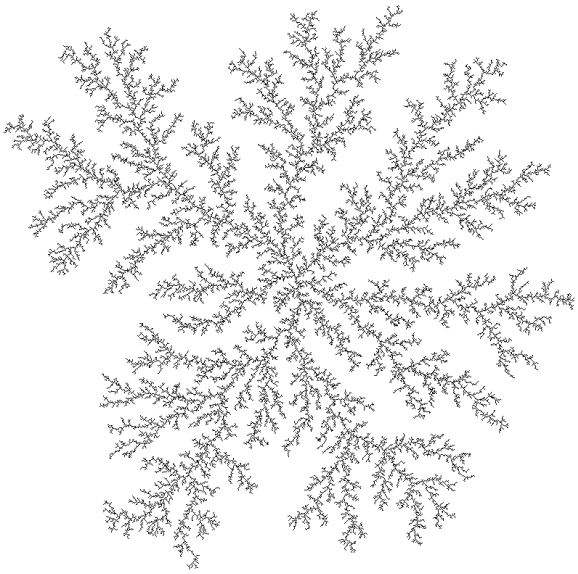

Figure 2: an example of Diffuse Limited Aggregation

More complicated methods for forest generation often involve some form of ecological simulation, or similar techniques, to determine the growth and interaction of trees in the forest environment. For example, in '*Modelling Asymmetric Growth in Crowded Plant Communities'* (Damgaard, 2009) [7], a set of mathematical models is proposed to allow us to quantify the relative growth rates of various plant species of different sizes due to competition in a crowded environment. The recent paper '*Synthetic Silviculture: Multi-scale Modeling of Plant Ecosystems'* (Makowski et al, 2019) [10] presents a method to efficiently mimic things like tropism and competition for resources by considering a 'multi-scale representation for plant ecosystems', considering growth separately at the level of branches, plants and the ecosystem as a whole.

However, despite the benefits these more complex systems offer when it comes to describing and simulating plant interactions, L-systems have the advantage of simplicity: the recursive nature of their construction gives us a clear understanding of the make-up of the overall structure and of the points where branching occurs, which allows us to manipulate the structure in a way we couldn't with, for example, DLA. This is exploited by Kenwood et al in '*Efficient Procedural Generation of Forests*' (2014) [8] to allow us to build up a large number of trees by copying instances of branches, which we will explore in more detail in the next section. Nonetheless, the aforementioned papers provide an important context to consider this project in, as they suggest a number of possible avenues for future extensions to the program.

3. Technical Background

3.1 L-System Formalisation

Before looking at the instancing algorithm, a more formal understanding of L-system production is required. For the purposes of this paper, we will be using stochastic L-systems, for the most part following the definition by Prusinkiewicz in '*Algorithmic Beauty of Plants*' (2019) [5], though with some changes specific to our implementation:

A stochastic L-system is an ordered quadruplet $L =< V, \omega, P, \pi >$ where

- V is a finite alphabet, V^* represents the set of all words over V, and V^+ represents the set of all non-empty words
- $\omega \in V^+$ is a nonempty word called the *axiom*
- $P \subset V^+ \times V^*$ is a set of *productions*, each written as $\mu = \chi$ for some $\mu \in V^+$, $\chi \in V^*$, with μ known as a *precedent*, or *non-terminal*, and χ referred to as the *replacement* or just the *RHS*
- $\pi: P \to (0,1]$ is a function from each production to the range $(0,1]$, called the *probability distribution*, with the requirement that $\sum_p \pi(p) = 1$ for all rules p that share a precedent.

We then define a *rule* in the system as a set of all productions that share a given precedent (although within the code of the C++ program, *rule* is also used interchangeably with *production*). An *application* of a rule r with precedent a to some word w over V consists of exchanging all occurrences of a in w with a replacement from one of the productions in r, with the probability of a given production p being chosen equal to $\pi(p)$. If there are multiple overlapping occurrences of a in w, the process of replacing is performed from left to right: the first occurrence, a_1, of a in w is replaced, and the next occurrence of a to be replaced will be the first subsequent occurrence that did not overlap with a_1.

For a given ordering of the rules in the system, a *derivation* in L consists of a finite number of applications of the rules to the initial axiom, performed in the given order (cycling round to the first rule again when we've got through all of them). The resultant word is called the *derived* or *generated* word, and its *age* or *generation* is defined as the number of rule applications required to reach it.

To turn a derived word into a 3D asset we just need to define an interpretation of various letters of the alphabet for the turtle graphics system to use to draw, which we will call the *semantics* of the language: in general, certain symbols will be interpreted as commands to do things like moving forward and drawing a line, rotating the current position, and saving or retrieving information to/from the stack. The specific interpretations employed in this project are covered in the 'L-System Syntax and Semantics' section of the appendix; they are heavily based on the ones used by the Houdini L-System node [11]. For the purposes of this paper, the most important symbols are the *branching commands*, '[' *and* ']': they effectively tell the turtle to create a branch for the tree by saving the current position at '[', then returning to it when the corresponding ']' is reached.

3.2 Instancing Branches

3.2.1 Definitions

The instancing method proposed by Kenwood et al (2014) [8] works by modifying the initial axiom and rules of an L-system to keep track of branches as they're introduced to the system and use them to fill an *instance cache*. A few definitions are required first to supplement the L-system formalization above.

Consider an L-system $L =< V, \omega, P, \pi >$ containing the symbols '[' and ']' as part of its alphabet with the semantics defined above. We will call the sub-word enclosed by a '[' symbol and its corresponding ']' a *branch*. We will say it is a *valid branch* if it contains a letter that is a precedent of some rule in L (this helps to rule out branches that don't offer any stochastic variation to the system, as such branches will not be useful for our method). Kenwood et al assign each valid branch an *id*, with two branches given the same id if and only if they contain the same letters.

In a derived word, we call the sub-word between a '[' and its corresponding ']' a *derived branch,* and call it a *valid derived branch* if it was valid in the production that introduced it. Valid derived branches inherit the id from the branch that introduced them. We also define the *age* of each valid derived branch to be the age of the derived word when the branch was introduced to it.

The key concept behind Kenwood et al's method is that for any two derived words of the same generation, w and v, replacing any valid derived branch in w with any valid derived branch in v of the same age and id produces another correctly derived word in L. When the system is used to describe trees with stochastic variation, this gives us a way to build up new correct trees from a small number of precalculated 'hero' trees by randomly selecting branches from each one.

3.2.2 Extending the L-System

To utilise this property of L-systems, we will create an extension of the alphabet $V' \supset V$, containing new symbols that we can add to the semantics. We introduce two symbols that represent commands to record the current branch (and all subbranches in it) as an instance, called the *startInstance* and *endInstance* commands, and one representing a command to get an instance, called the *getInstance* command. For now, we will refer to them respectively as s, e and g. The *s* and *g* commands each take two parameter inputs, representing the id and age of the branch.

Kenwood et al's procedure involves adding these new symbols to the axiom and rules of L. First, we replace the axiom ω with an extended axiom $\omega' = s(0,0)\omega e$, which represents a command to record the whole tree as an instance, treating it as a branch with id 0 introduced at age 0. Then for each production p in L, if p contains n valid branches, we replace p with 2^n new productions, representing the possibility for each branch of either recording it as an instance or calling to replace it with another instance of the same id and age. For example, the production

$$A = ![B]\backslash\backslash\backslash[B]\backslash\backslash\backslash B$$

would be replaced with

$$(1)\ A = !s(id, age)[B]e\backslash\backslash\backslash s(id, age)[B]e\backslash\backslash\backslash B$$
$$(2)\ A = !g(id, age)\backslash\backslash\backslash s(id, age)[B]e\backslash\backslash\backslash B$$
$$(3)\ A = !s(id, age)[B]e\backslash\backslash\backslash g(id, age)\backslash\backslash\backslash B$$
$$(4)\ A = !g(id, age)\backslash\backslash\backslash g(id, age)\backslash\backslash\backslash B$$

Once these new productions have been introduced, we are left with a new set P' of productions, so we need to define a modified probability distribution $\pi' : P' \to (0,1]$ satisfying the required conditions to create a new L-system $L' =< V', \omega', P', \pi' >$. We do this by using the old probability distribution

and a global instancing probability ρ: for a production $p' \in P'$ formed from an original production $p \in P$ with n valid branches, we define

$$\pi'(p') = \pi(p) \times \rho^i \times (1-\rho)^{n-i}$$

where i is an index representing the number of occurrences of the *getInstance* command in p'. In the example above then, if the production had a probability of 0.3, and the instancing probability is 0.4, the new productions would respectively have probabilities

$$\begin{aligned}(1)\quad & 0.3 \times 0.6^2 = 0.108\\ (2)\quad & 0.3 \times 0.4 \times 0.6 = 0.072\\ (3)\quad & 0.3 \times 0.4 \times 0.6 = 0.072\\ (4)\quad & 0.3 \times 0.4^2 = 0.048\end{aligned}$$

which sum to 0.3 as required.

When we create the new productions, we immediately assign the first parameter of each newly added symbol as the id of the relevant branch. We assign the second parameter when applying the rule to a derived word to reflect the current age of the word when the rule is applied.

3.2.3 The Instance Cache

This process leaves us with a new L-System L' where derived words contain the symbols s, e and g around or in place of the branching symbols '[' and ']'. We now need to define the interpretation of these symbols in the semantics of L' so the turtle graphics system knows how to deal with them. While we've so far considered this extended L-system in very general terms, for the rest of this section we'll assume that the interpretations of symbols in L by the turtle graphics system involve building up buffers of vertices and indices representing geometry to send to the renderer, and that at each stage of the interpretation process the turtle has a saved position and orientation, that can be represented as a single transformation.

To add our new symbols to this semantic system, we require an *instance cache* structure. This is a device that stores transformation data for branch instances according to their id and age and allows us to retrieve a random instance of a given id and age. An *instance* consists of a transformation determining the position and orientation of the start of the branch relative to the base of the tree, data determining the vertex and index buffers required to draw the branch, and a list of *exit points* with associated id, age and transformation, representing points where another branch instance is required to complete the current branch. Note that an instance can stretch over several generations.

We define the interpretations as follows:

$\boldsymbol{s(i,a)}$ Create an instance using the current transformation, mark it as active, and add it to the instance cache with id i and age a. Assign the instance a pointer to the end of the index buffer to mark the beginning of the render data.

$\boldsymbol{e}$ Deactivate the instance started by the corresponding s command and mark the length of the index buffer required to draw it *(note that just like with '[' and ']', each e symbol will correspond to exactly one s symbol; for example, in the word* $Fs(1,1)[s(2,1)[FF]\backslash Fe]e$ *the first s is paired with the second e).*

$\boldsymbol{g(i,a)}$ Mark the current position as an *exit point* with id i, age a, and transformation equal to the current transformation, for all currently active instances. This will be used later in the instancing algorithm to tell the program to retrieve an instance from the cache of that id and age.

3.2.4 The Instancing Algorithm

Once this extended L-system L' has been defined and created, we can perform the main algorithm to produce a forest from the instance cache. First, Kenwood et al fill the instance cache by running the L-System a predetermined number of times to produce a small number of derived words representing the 'hero trees'. These words are interpreted according to the L-system syntax, and the *startInstance*, *endInstance* and *getInstance* symbols mean that instances are added to the instance cache during interpretation.

Next, a large number of points are scattered across a grid to represent initial tree positions for the forest (the method of scattering is beyond the scope of Kenwood et al's paper). Each point position is viewed as a transformation and assigned an id and age of 0, then fed into a *create()* function to produce an instanced tree. This is a recursive function that receives id, age and transformation inputs. It retrieves a random instance from the instance cache of the given id and age, then computes the transformation needed to place that branch in the required position in world-space based on the input transformation and the relative transformation of the instance. The resultant transformation and data for the instance's vertex and index buffer are added to an output variable to be sent to the renderer. The function then iterates over all exit points in the current instance and computes the world-space transformation corresponding to that exit-point from the input transformation and the exit-point's relative transformation. It uses this, alongside the exit-point's id and age, as input to recall the function and retrieve more branch instances to complete the structure. Pseudo-code for the algorithm from the paper is below.

```
create(output, direction, age, cache)
instance = cache.getInstance(age)
T = direction × instance.direction.inverse
output.addTreeInstance(instance.buffers, T)
for i = 1 → length(instance.exitPoints)
   newAge = instance.exitPoints[i].exitAge
   exit = instance.exitPoints[i].direction
   newDirection = direction × exit
   create(output, newDirection, newAge, cache)
```

Figure 3: the pseudocode from Kenwood et al's paper (2014) [8]

The result of this process is a list of transformations alongside pointers to index buffer ranges, representing geometry for the entire forest. However, while the trees have been built from randomly selected branch instances, and so should for the most part be distinct, we have only had to store a relatively small number of vertices in memory – the vertices making up the initial hero trees. This data can be sent to the renderer, where the transformations can be applied on the GPU, resulting in a much faster render than if the trees had each been created individually.

4. Implementation

4.1 OpenGL

The aim of this project was to implement Kenwood et al's algorithm as a forest creation tool. The first hurdle to overcome when beginning the implementation was deciding on the type of platform to use. Working within a DCC would have had a few notable advantages: firstly, many programs, like Houdini for example, come with pre-made L-system tools with a large number of user parameters, that would save a lot of work at the start of the project. Secondly, if the procedure could be implemented within a package like Houdini, this would automatically allow it to fit into many professional pipelines, thereby majorly increasing its utility.

However, the project was ultimately created as a standalone C++ tool using OpenGL (with the NCCA NGL library) in Qt Creator, for a number of reasons. To start with, the algorithm requires manipulation of the L-system syntax and semantics, which would have been hard to do within the confines of an existing L-system tool; despite the extra work required initially, coding the L-system generation and interpretations from scratch offered more control over the process. Additionally, the efficiency of Kenwood et al's method comes from the ability to send all the transformation data to the shader or renderer, to be performed on the GPU. This could prove tricky within many DCC packages where the user's control over how data is rendered in the viewport can be limited; whereas it is a procedure well-suited to the workflow of OpenGL, which supports instanced draw methods and allows a large degree of manipulation on the shaders, with the ability to transform vertex positions and add geometry all on the GPU.

4.2 User Interface

The user interface for the tool was created as a UI form in Qt Creator. It is split into two main sections: the L-System tab and the Forest tab. The L-System tab contains 3 subtabs allowing the user to define the axioms and rules of the forest's different L-systems, as well as various L-system parameters. The Forest tab displays the result of the instancing algorithm, and allows users to manipulate the terrain of the forest as well as the scattering of the trees.

The form contains an OpenGL window that renders the data for the above tabs, through an object of class NGLScene. This class contains members representing all the key elements of the program: a set of LSystem objects that hold the data for each L-system subtab, $m\_LSystems$, two Forest objects, $m\_scatteredForest$ and $m\_paintedForest$, and TerrainData and TerrainGenerator objects, $m\_terrainData$ and $m\_terrainGen$ that collectively determine the data for the terrain. The interface is implemented through signals and slots that allow the user to alter the internal states of these member objects as well as other members of NGLScene; the details of these member objects will be covered in the following sections.

4.3 L-System Generation

The LSystem class takes care of both the generation of a derived L-system word from a given axiom and set of rules, and the creation of geometry data from that word using the program's 'turtle graphics' semantics (the details of which are covered in 'L-System Syntax and Semantics' in the appendix). For string generation, an internal Rule struct was created to group together productions and probabilities in an easily accessible format; when a user calls the generate command on the LSystem tab, this sends a call to the $breakDownRules()$ method for the currently active L-system object, which fills the member variable m_rules according to the user-defined productions and probability ratios. The $generateTreeString()$ method then returns a derived word for the L-system of age $m\_generation$ by repeated string replacement, using the random engine member m_gen to help determine which replacement we use at each point. To help with string parsing, this project makes use of the external Boost library for splitting strings and simple replacements, and the standard library regex for string searching.

The $createGeometry()$ function performs the turtle graphics interpretation of this string, filling vertex, index and other buffers to be used by NGLScene for rendering. This method is implemented as a large switch statement, applied to each character of the string in turn. At each stage of the loop, the function keeps track of the current position with the variable $lastVertex$, the direction the turtle is facing, dir, and the vector representing the turtle's right, $right$. It also keeps track of a number of other variables like the current step size, angle and thickness. Starting a branch with the '[' command causes all these variables to be pushed to stacks of saved values, and ending that branch with ']' retrieves the top-most value from each stack. Symbols that permit bracketed parameters use the function $parseBrackets()$ to determine if the subsequent characters of the string represent parameters for the command; if not they use default values.

Because $createGeometry()$ is used to make geometry both for the individual L-systems in the L-System tab and for the hero trees used in the creation of the forest, the LSystem class contains a Boolean member $m\_forestMode$. For every buffer vector member of the LSystem class there is a corresponding hero buffer member, and the start of $createGeometry()$ declares a number of pointers to buffers; the pointers are then each assigned to either a standard buffer member or the corresponding hero buffer member depending on whether or not $m\_forestMode$ is true. This way the same code allows us to fill up either the hero buffers or the standard ones as required. When using standard buffers, they're reassigned to empty at the start of the function, but hero buffers aren't because we want to be able to assign vertices from multiple trees to a single hero buffer.

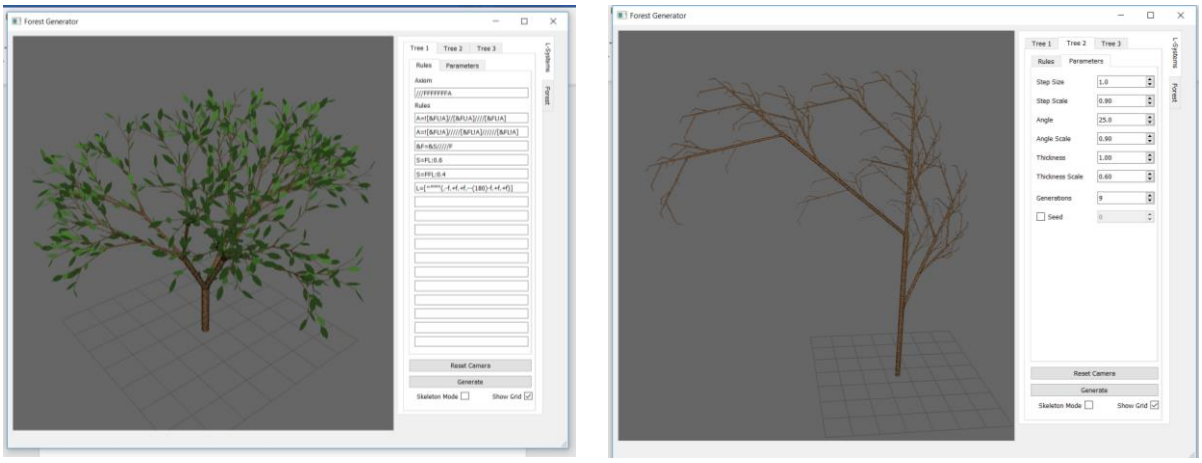

Figure 4: L-systems created using the tool

4.4 Instance Methods

Before we can apply $createGeometry()$ in forest mode to an L-system object, we need to rewrite the rules of the L-system to add in the instancing symbols required for Kenwood et al's algorithm. For this project, the symbols '@' and '$' were chosen to represent calls to *startInstance* and *endInstance* respectively, and '<' was chosen to represent a call to *getInstance.* A fourth symbol '>' was also added, which we will call the *endGetInstance* command: in this implementation *getInstance* no longer replaces a branch but instead *getInstance* and *endGetInstance* are placed around a branch in the same way as with *startInstance* and *endInstance*. This is to deal with the rare but possible case of the instance cache being empty for a particular (id, age) combination when we perform the instancing algorithm: in most cases, after a '<' command has been interpreted by $createGeometry$, the function is told to simply skip to the corresponding '>' and move on, treating the string as if the branch had been removed; but if the instance cache has no entries for that id and age when the '<' symbol is reached, it is treated as if it is an '@' symbol, in addition to its other interpretation, and the corresponding '>' then takes the role of '$'.

These symbols are added by the $addInstancingCommands()$ method, which adds @(0,0) and $ around m_axiom and then refills m_rules by replacing each production with a number of alternate productions including the symbols '@', '$', '<' and '>', and altering the probabilities accordingly using $m\_instancingProb$, as described in section 3.3.2. To aid this method, each rule is given a variable $m\_numBranches$ which represents the number of valid branches in the rule; this is assigned to each rule by $countBranches()$ (called in the $breakDownRules()$ method), which determines whether a branch is valid or not by using the regex expression $m\_nonTerminals$ (a string describing the precedents of each rule) to check whether or not a non-terminal occurs in the branch.

$countBranches()$ also fills a list $m\_branches$ of valid branches of the rules in the system (which includes the axiom as the first element) and the index of each branch in this list is used as the id parameter for each '<' and '@' symbol by $addInstancingCommands()$. The age parameter is initially replaced with the symbol '#'. At each stage of rule application, $generateTreeString()$ will switch each '#' in the replacement of any rule it is applying with the current age of the derived word it is applying it to.

Consequently, after a call to $addInstancingCommands()$, we have created a new set of rules that include instancing symbols, and a subsequent call to $createGeometry()$ with $m\_forestMode$ set to true allows us to start filling the instance cache. The implementation of the instance cache uses the Instance class, which contains as members all the elements discussed in section 4.2.3: a transform matrix, indexes representing the start and end points of the relevant hero index buffers for rendering, and a list of ExitPoint objects, which each contain an id, an age and another transform. The instance cache itself is represented by a triple nested std::vector of Instance objects, $m\_instanceCache$, where the outer index corresponds to the id of the instance, the middle index represents the age, and the inner index separates multiple instances of the same age and id. This nested vector structure is in fact employed multiple times throughout the project, so a set of macros were created in InstanceCacheMacros.h to improve readability and reduce repetition in the code. Using this, the instance cache is defined as a $CACHE\_STRUCTURE(Instance)$.

The interpretations of the symbols '@', '$' and '<' are then defined in $createGeometry()$ as stated in 3.2.3, with the additional interpretations of '<' and '>' when $m\_instanceCache$ is empty at the given id and age defined as described above.

4.5 Forest Generation

The NGLScene Forest member $m\_scatteredForest$ deals with the actual generation of the forest from branch instances. It contains a list of LSystem objects, $m\_treeTypes$ (one for each tree tab) and when the user renders the forest scene, m_forest calls the LSystem method $fillInstanceCache()$ for each member of $m\_treeType$. $fillInstanceCache()$ calls $addInstancingCommands()$ for the L-system, and resizes its instance cache according to the number of unique branch ids and the generation number. Then it sets $m\_forestMode$ to true, clears all hero buffers and calls $createGeometry()$ $m\_numHeroTrees$ times where $m\_numHeroTrees$ is a member of the Forest class passed into $fillInstanceCache()$ as a parameter.

With this done, the instance cache of each L-system member of the forest is filled, and $m\_scatteredForest$ calls $createForest()$. This in turn calls $scatterForest()$ which scatters points across the terrain to fill $m\_treeData$, a list of initial tree transforms and tree types. Then for each tree in $m\_treeData$, it calls $createTree()$ which implements the pseudocode from Kenwood et al (2014) presented in section 3.2.4, iterating through the exit points of instances to build the tree from branches taken at random from the given (id, age) position of the given tree types' instance cache.

In this implementation however, the output data takes a different form to the one presented in figure 3. To speed up the rendering in NGLScene, by allowing us to use the command $glDrawElementsInstanced$, it was necessary to group together all occurrences of the same instance in the output. To achieve this, the instance cache structure was re-used, letting us create an output member variable of type $std::vector < CACHE\_STRUCTURE(std::vector < ngl::Mat4 >) >$, $m\_transformCache$. Here the $CACHE\_STRUCTURE$ as usual separates elements based on id, then age, then different instances of the same id and age; meanwhile the outer std::vector of $m\_transformCache$ separates instances taken from the instance caches of different tree types, while the innermost index separates different branches using identical instances. When $createTree()$ adds data to $m\_transformCache$, it simply adds the current transformation to the list of transformations in $m\_transformCache$ of the given instance (specified by tree type, age, id and inner index). This leaves us with a nested list of transformations to send to the renderer, each of whose positions in the nested structure determines the branch instance that it corresponds to; in NGLScene this means for each t, id, age and $innerIndex$ we can create a transform buffer from

$$m\_scatteredForest\,.m\_transform\_cache[t][id][age][innerIndex]$$

and vertex and index buffers using

$$m\_LSystems[t].m\_heroVertices$$

$$m\_LSystems[t].m\_heroIndices$$

with the start and end points of the index buffer determined by

$$m\_LSystems[t].m\_instanceCache[id][age][innerIndex].m\_startInstance$$

$$m\_LSystems[t].m\_instanceCache[id][age][innerIndex].m\_endInstance$$

We then send these buffers to the shaders to draw the instances and perform the transformations per instance in the vertex shader.

4.6 Terrain

The terrain for the forest is created with the variables $m\_terrainGen$ and $m\_terrain$, objects of classes TerrainGenerator and TerrainData. These classes were imported from a previous project and the details of how they work is beyond the scope of this report, but in short TerrainGenerator uses a Perlin noise module from the library Libnoise to generate a heightmap, which is then passed into TerrainData. TerrainData takes this heightmap and performs a LOD reduction algorithm on it, following a method presented by Lindstrom and Pascucci (2009) to speed up the rendering of the terrain. Whenever trees are placed on the grid, care is taken to use the Perlin noise with same parameters to determine their height.

For this project, a few additional methods were added to each class to allow us to pass more data from the terrain to the shaders: in TerrainGenerator, $computeNormals()$ allows us to use the heightmap data to find approximate normals, tangents and bitangents for each vertex in the terrain. This data is passed on to the TerrainData class, along with UVs for each point on the terrain, and these are put into buffers to be sent to the renderer during $fillVerticesAndIndicesForRendering()$. This data allows us to bring in texture maps and normal maps and link these up to a diffuse lighting model in the shader.

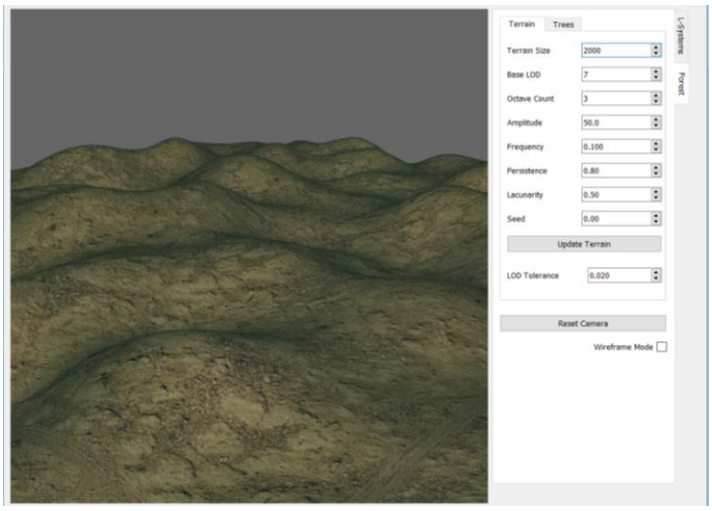

Figure 5: the terrain

4.7 Rendering and Shaders

In *'Efficient Procedural Generation of Forests*', Kenwood et al state that their aim is merely to show the efficiency of the method, and consequently no attempt is made to improve the aesthetic appearance of the trees. In this project, we additionally want to demonstrate how the technique could be built upon with shader work to produce more fleshed-out scenes. Consequently, a lot of shader techniques were used; while the final effect is still far from the quality of a production level scene, it illustrates how a company may be able to use such a method in its pipeline.

The first challenge to sort out was how to send all the relevant data to the shaders; this can often be a bottleneck in animation pipelines so it was important to find a way to minimise the amount of data sent. This was all done in the shader methods and VAO building methods in NGL scene; in VAOBuilding, NGL VAO objects are created and bound to buffers to be sent to the shader; in ShaderMethods, textures and uniform variables are loaded to the shaders. Because binding buffers to a VAO can be costly, multiple VAO objects were created, each dealing with a different object and hence a different set of vertex and index buffers – this means that we can reduce the amount of times an individual VAO must be bound and unbound.

Two types of NGL VAO class were used for rendering: for most of the scene, the standard NGL SimpleIndexVAO was used; but this doesn't support instanced drawing, as was necessary for the forest algorithm, hence it was necessary to write a new VAO class, InstanceCacheVAO. When setting data for this VAO, in addition to sending it vertex and index buffers, transform data from the forest output $m\_transformCache$ is added, as well as indexes representing the start and end of the current instance in the index buffer; these transforms are then applied each to a different instance using $glDrawElementsInstanced$. To make use of this, all VAOs used in forest rendering are stored in cache structures reflecting the structure of $m\_transformCache$ and each one is initialised as an InstanceCacheVAO. Building them then just involves iterating through the cache structure, initialising each VAO as an InstanceCacheVAO and sending it the corresponding transform data from $m\_transformCache$ and instance data from the relevant L-System's instance cache, as discussed in section 4.6. All VAOs of either class type can also be given additional buffers with the function $addBufferToBoundVAO()$.

Once this is set up, any number of methods can be applied on the shader to improve the look of the scene. For this project, we use geometry shaders to add thickness to the trees, then apply textures and normal maps to them, using a TBN matrix to link the normal map to a basic diffuse lighting model. We also add commands to create polygons and default leaves to the L-System semantics, which are created with separate sets of buffers and hero buffers and rendered using separate VAOs. The leaves are rendered as points, which are turned into planes in a geometry shader and assigned leaf textures with alpha maps. All of this results in relatively low overhead since the calculations are not computationally complex and it is all done on the GPU; it is considerably faster, for example, to create the extra geometry for the trees on the geometry shader than to create additional vertices for each branch to be sent into the shader.

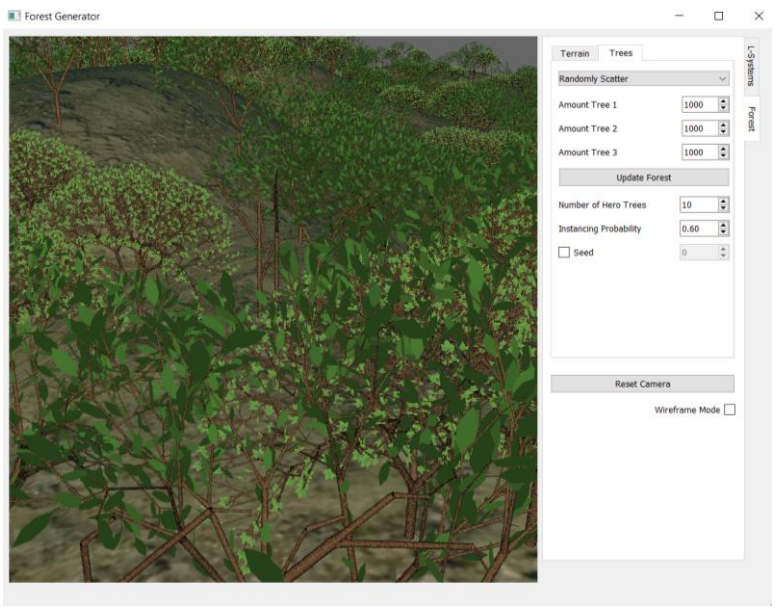

Figure 6: close-up showing some of the results of the shader work

4.8 Tree Painting Tool

As a final improvement to the tool, a painting method was added to allow users to paint trees directly onto the terrain. A user can select one of the 3 basic L-Systems created in the LSystems tab as a 'brush' and use the mouse to paint these across the terrain. This is done by the function $getProjectedPointOnTerrain()$ in NGLScene, which uses ray casting to determine a ray in world space pointing from the cursor into the screen using projection, view and model matrices (following a tutorial by Anton Gerdelan, 2016 [12]). The method then moves along this ray at regular intervals and compares the y-value at each point with the corresponding value from the terrain Perlin noise module; once this reaches below a specified threshold, we have found the point of intersection with the plane, and can draw a tree at this point.

The trees are added to $m\_paintedForest$, a second Forest member that's used to take care of these painted trees. This object uses the instancing algorithm $createTree()$ just like $m\_scatteredForest$ does, and is drawn using its own sets of instance cache VAOs; but trees are added to it one at time with the internal method $addTreeToForest()$, rather than all at once with $createForest()$. This addition extends the tool to make it much more versatile from an artist's perspective, allowing them to choose the positions of various parts of the forest.

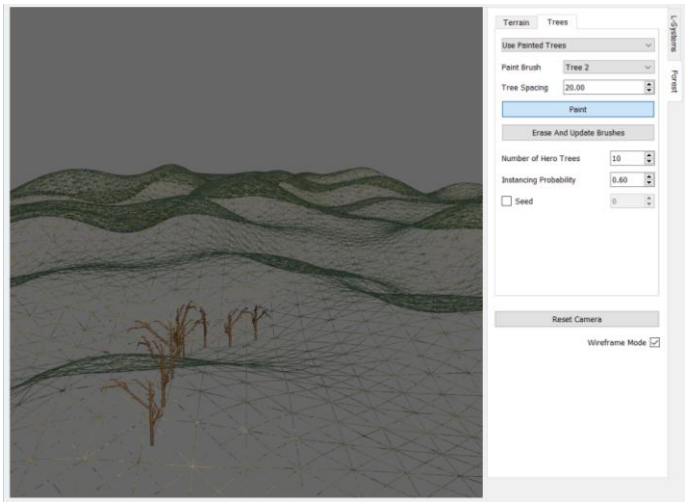

Figure 7: the painting tool in action

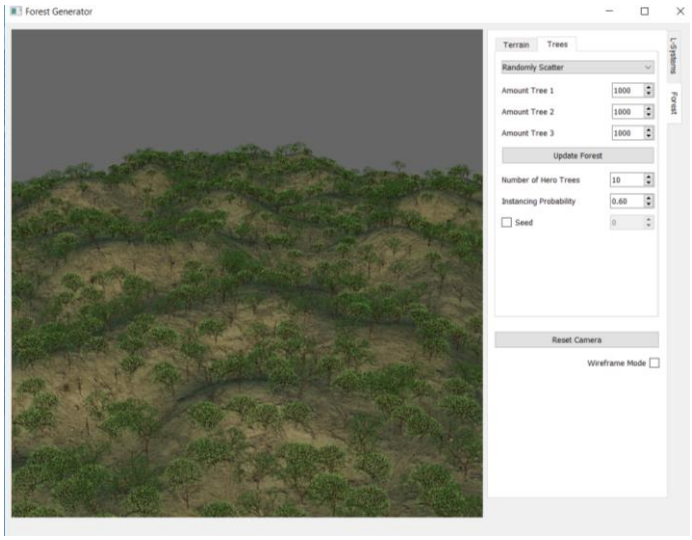

Figure 8: a scattered forest

5. Results and Future Work

The project has successfully recreated the algorithm introduced by Kenwood et al in '*Efficient Procedural Generation of Forests' (2014)*, using instancing of L-Systems at the level of branches to produce a forest with a varied range of trees at a relatively speedy rate. Kenwood et al said that they aimed to produce forests with trees in the range of around 10,000 with rendering times of a few seconds; our tool achieves this, though it slows down considerably as the generations of the LSystems increases and thus the size of the instance cache increases.

There are user controls that can help to combat this problem: the number of hero trees used for each L-system in a forest and the instancing probability can both be changed. There was also a limit added to the instance cache in $createGeometry()$ in the LSystem class, where no instance can be added to a cache at certain id and age if that entry of the cache is already too full; adding this as a user interface option would be another way to improve user control over the data flow. Additionally, Kenwood et al suggest the use of hash tables for the instance cache, which would produce a much more efficient way of storing and accessing the data than the nested std::vector structure used here.

The painting function is a good addition to the tool from the perspective of artistic direction and versatility, and leaves a lot of fairly straightforward avenues for future expansion; for example, allowing the painting of multiple trees at once for speed, or allowing us to erase individual trees. Unfortunately, as it currently stands, the paint tool isn't able to fully take advantage of the instancing algorithm's efficiency: this is because adding new trees to the forest as the user moves the mouse requires updating $m\_paintedForest$'s $m\_transformCache$ member, which in turn requires the instance cache VAOs to be rebuilt, which is a relatively slow process. This problem has been partially resolved by giving $m\_paintedForest$ the member variable $m\_adjustedCacheIndexes$ which marks which bits of the transform cache have been changed at each point of the painting process and allows us to only rebuild the corresponding VAOs; but the function still slows down very quickly when using trees with large numbers of generation as the paint brush.

An easy solution would be to just paint dots onto the terrain, then create trees from the dot positions only after painting has been finished; however, this loses the advantages of real-time feedback on the painting process. Finding a solution that allows the painting tool to make proper use of the speed of the instancing technique to create real-time painting is a subject for further work.

It would also be good to find ways to incorporate the tool more readily into other pipelines: at the moment, it would perhaps fit into the pipeline of a companies that, for example, have their own game engines hard coded in C++, allowing them to easily fit this code into it; but porting it into a DCC would be a trickier prospect, and one worth looking into. Additionally, it would be worthwhile expanding the tool to incorporate other types of L-systems or even other plant simulations: while L-systems aren't generally able to interact with external environments, it would be entirely possible to incorporate some form of ecological simulation to inform the initial placement of trees, for instance.

6. Conclusion

This project has presented a C++ tool that efficiently generates and renders forests from L-system instancing, in a manner that could allow for usage in real-time media. It addresses two of the major issues faced by CG artist in forest creation, speed and variation: the trees are distinct but are still rendered in a short space of time. Additionally, the paint tool introduced demonstrates a way that artists could use such a method to produce scenes in a more controlled way. The method used for the algorithm makes integration with other software potentially challenging, but coding it from scratch gives it the project a large degree of flexibility in terms of improvements to the user interface and possibilities for extensions.

Bibliography

[1] Lindenmayer, A., 1968. *Mathematical models for cellular interaction in development, Parts I and II*. Journal of Theoretical Biology, 18:280–315.

[2] Yokomori, T., 1980. *Stochastic characterizations of EOL languages*. Information and Control, 45:26–33.

[3] Witten Jr., T. A., Sander, L. M., 1981. *Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon*. Phys. Rev. Lett. 47, 1400 – Published 9 November.

[4] Prusinkiewicz, P., 1986. *Graphical applications of L-systems.* In Proceedings of Graphics Interface '86 — Vision Interface '86, pages 247–253. CIPS.

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

[6] Lindstrom, P., Pascucci, V., 2001. *Visualization of Large Terrains Made Easy*, Proceedings of the IEEE Visualization Conference, October 24-26

[7] Damgaard, C., 2009. Invited Talk: *Modelling Asymmetric Growth in Crowded Plant Communities*. Third International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications, Beijing , pp. 267-269.

[8] Kenwood, J., Gain, J., Marais, P., 2014. *Efficient Procedural Generation of Forests*. Journal of WSCG 22(1).

[9] Gerdelan, A., 2016. *Mouse Picking Ray Casting.* Anton's OpenGL 4 Tutorials, website, http://antongerdelan.net/opengl/raycasting.html

[10] Makowski, M., Hädrich, T., Scheffczyk, J., Michels, Dominik L., Pirk, S., Palubicki, W., 2019. *Synthetic Silviculture: Multi-scale Modeling of Plant Ecosystems*. ACM Transactions on Graphics (TOG), Volume 38 Issue 4, July 2019, Article No. 131

[11] Side effects software, 2019. *Houdini Engine 17.5 Documentation: L-System geometry node*. https://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html

[12] De Vries, J., 2019. *Learn OpenGL Website*. https://learnopengl.com/

Appendix

L-System Syntax and Semantics

The interpretations for the L-systems created in this project are based heavily on the ones used by the Houdini L-System node [11]:

Every production must contain exactly one '**=**' character. The characters preceding the '**=**' sign constitute the non-terminal to be replaced, the characters after are the string that will replace it.

Optionally, a production can contain one (and only one) '**:**' symbol. This must appear after the equals sign and marks the end of the replacement string. The symbols after the '**:**' represent the probability ratio of the production being applied; if these symbols cannot be parsed to a float, they are discarded and the program assumes that they represented a 1. Similarly, if no '**:**' symbol is given, the production is automatically assigned a probability ratio of 1. Note that when the L-system is generated, all productions with same precedent are grouped together and their probability ratios are normalised to produce a value between 0 and 1 for each rule, with their sum adding to 1. Thus, 3 productions with the same precedent that are all concluded with the string ":1" will each be assigned a probability of $\frac{1}{3}$.

The full syntax for a production is then:

$$precedent\_string = replacement\_string : probability\_ratio$$

In addition to the restrictions on the symbols '**=**' and '**:**', the following characters cannot be used in a production since they are used in the implementation of the instancing commands in the program: '**$**', '**@**', '**<**' and '**>**'. Attempting to use these, or incorrectly using '**=**' and '**:**' will result in the rule being discarded. It is also forbidden to use the symbol '**{**' after a previous use of '**{**' that hasn't been concluded by a corresponding '**}**', since this would mean creating a polygon within a polygon (see below), which the program doesn't know how to deal with.

All other symbols are allowed, in any order. After generation of the final L-system string, all unknown symbols will be ignored and known symbols will be interpreted as defined below (all brackets following characters are optional and if no bracket is given the specified default value will be used).

F(p) Move forwards a distance of p (default: current step size), creating geometry
f(p) Move forwards a distance of p (default: current step size), creating no geometry
+(p) Turn right p degrees (default: current angle)
-(p) Turn left p degrees (default: current angle)
&(p) Pitch up p degrees (default: current angle)
^(p) Pitch down p degrees (default: current angle)
/(p) Roll clockwise p degrees (default: current angle)
\(p) Roll anticlockwise p degrees (default: current angle)
"(p) Multiply current length by p (default: step size scale)
;(p) Multiply current angle by p (default: angle scale)
!(p) Multiply current thickness by p (default: thickness scale)
[Start branch: save all data at this point
] End branch: revert all data to the saved values from the corresponding branch begin point
{ Start polygon
} End polygon
. Add the current position as a vertex of the current polygon
J Add the default leaf object to the current position
This symbol is replaced at each stage of the L-system generation with the age that it is introduced at. Consequently, it cannot be used as a precedent for a rule.