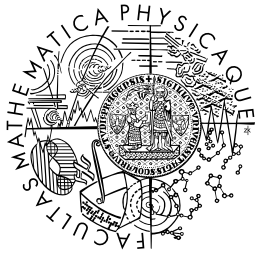

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jakub Pelc

Procedural tree generator with LOD support

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. Josef Pelikán
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank Jan Hovora for his consultation and guidance of this project, and Josef Pelikán for his supervision.

Title: Procedural tree generator with LOD support

Author: Jakub Pelc

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: Trees are a key part of many virtual environments, thus there is a high demand for realistic tree models. Creating such models by hand, including level of detail, is very time consuming. However, the fractal-like structure of a tree may appear complex, but it can be approximated by a set of simple rules. This enables us to construct a tree generator and editor that can produce different tree types using few parameters and produce any number of randomized instances of a tree type. Furthermore, many applications such as games or VR/AR have strict performance budgets, thus level of detail is needed. Different detail levels also must be visually similar to hide runtime transitions between them, further increasing the difficulty of creating such models by hand. We exploit a high-level tree structure the generator provides to create versions of the tree with lower geometrical complexity, mainly by replacing branches with their planar images. This process is fully automatic. The work consists of a generator library capable of creating trees with a highly customizable detail level and a tree editor that can export generated meshes.

Keywords: Tree generator Computer graphics 3D graphics Tree modeling

Contents

1. Introduction

Trees are a common and integral part of the environment around us, and are equally important in virtual environments that mimic the real world, creating a need for high quality tree models [1]. For the specific use case of interactive applications, such as computer games, virtual reality and augmented reality, a detailed model is not enough. Interactivity of these applications severely limits the time that can be spent rendering a single frame, thus setting a limit on the geometric complexity of displayed scenes.

This in turn means that we want to reduce the complexity of tree models for such applications as much as possible, all the while maintaining the impression of a highly detailed object. A key observation is that objects further away from the observer need not be as detailed as nearby objects, and can be swapped for a lower detailed version of themselves. This approach is called *Level of Detail*, or *LOD* for short, and is a common technique in games. The variants of an object with different detail levels are also called *LODs*.

Obtaining high quality tree models including all needed levels of detail requires either a significant time investment to create them by hand, or a significant monetary investment into proprietary tree generation software.

This is not viable for hobbyist game developers or small independent studios. There are other options for obtaining tree models, but they come with their own set of disadvantages:

- **Freeware tree models.** There are plenty of Creative Commons licensed models on the internet, but they usually come only in a single detail level, and the required tree species might not be available.
- **Automatic level of detail generation from existing models.** While it works reasonably well for solid meshes, this is not the case for trees due to their complex structure.
- **Asset collections.** These contain many different trees with LODs, but are usually bound to a specific game engine.
- **Freeware tree generators.** Some generators exist, but they usually lack the ability to generate a wide range of detail levels.

1.1 Our goals

The goal of this work is to provide a source of tree models that is freely available, easy to use, capable of generating a wide range of detail levels of the same tree instance and capable of generating different tree types. The target audience of our application are hobbyist game developers.

A tree generator such as this still relies on realistic and detailed base textures from external sources. However, obtaining free textures from the internet is

[1] In the context of this work a model refers to a collection of meshes and textures that make up a virtual object.

relatively easy, and is bound to become easier with the advancement of free photogrammetry software.

The software will be divided into two main components:

- **The generator**, a library capable of generating tree meshes and producing simple render commands for generation of additional textures [2].
- **The editor**, a program that uses the generator, capable of displaying and modifying the resulting tree in a what-you-see-is-what-you-get manner.

This division allows the main tree generator library to also be used for runtime tree generation, for example in a game with procedural world generation.

The generator will be able to create different tree types based on a small set of input parameters. It will also be able to generate randomized instances of trees defined by the same set of parameters.

Another set of input parameters will control geometric complexity of the resulting meshes. This will be done by reducing the triangle count of base geometric shapes from which the tree mesh will be assembled. Another, more significant way of reducing geometric complexity will be by replacing entire branches with a set of planes onto which the image of the original branch is mapped. This set of planes will be referred to as *polyplanes* in the rest of the work. This approach of simplifying vegetation geometry is commonly used in games, but implementing it automatically is not trivial.

The editor will display the generated tree meshes and provide a user-friendly way of tweaking input parameters of the generator library, with an easy way of seeing the visual results of parameter changes. The editor will also include options for exporting the generated meshes and textures.

1.2 Thesis structure

In chapter 2 we will provide wider context for our work. We will describe various approaches to generating trees, discuss level of detail and describe our technologies of choice. In chapter 3 we will provide details of our tree generator in two parts: in section 3.2 we will describe the **grower** which handles procedural generation of the high level tree structure, and in section 3.3 we will describe the **mesher** which converts it into a tree mesh and handles polyplane generation. In chapter 4 we will provide some details about the editor, which provides user interface for control of our generator and also handles the final stage of polyplane generation [3].

[2] Texture generation itself is not handled in the library in order to decouple it from a specific graphics API. See section 3.1 for more details.

[3] The need to the editor to handle part of tree generation is explained in section 3.1.

2. Analysis

Our tree generator is based on the principles described in the books *The Algorithmic Beauty of Plants* by Prusinkiewicz and Lindenmayer [1] and *Digital Design of Nature* by Deussen and Lintermann [2] and on personal consultation with Jan Hovora [3]. In this chapter we will overview various methods of generating the structure of a tree detailed in those books and discuss the problem of level of detail for trees. We will also describe what form should the resulting tree model take and what technologies will be used to implement the tree generator and editor. This will provide context for the following chapters, where we will discuss the inner workings of our application. Note that we will only provide brief and simplified explanation of the tree generation techniques. For formal definitions, please refer to the referenced materials.

2.1 L-systems

Lindenmayer systems [1, ch.1.1], or L-systems for short, are a technique originally developed to model plant growth. Their key concept is that of *rewriting*, where complex objects are constructed by iteratively replacing parts of an initial simpler object. These replacements are defined by a set of *rewriting rules* or *productions*.

An example of rewriting is the Koch snowflake. The initial object is a simple equilateral triangle. In each step, every edge is replaced with a segmented line forming a triangular spike. In the case of the Koch snowflake, the initial triangle is called the *initiator* and the replaced segmented line the *generator*. See figure 2.1 for an example of the Koch snowflake.

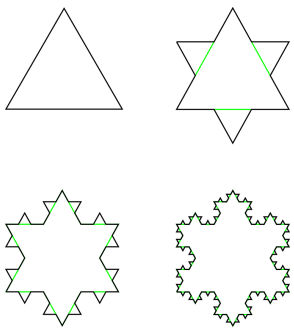

Figure 2.1: The first four iterations of the Koch snowflake [4].

The self-similarity and increasing complexity with each iteration produced by rewriting systems such as the Koch snowflake can also be observed in plants, making them useful for modelling plant development.

L-systems themselves operate on a *string* of *letters*. They are similar to Chomsky grammars, where, in general, each production is defined by pair of letter strings, the first being the original sequence and the second being what this se-

quence is to be replaced with [1]. The difference is that while Chomsky grammars apply a single production in each step, L-systems apply all possible productions in parallel.

In their deterministic context-free variant, called DOL-systems [1, ch.1.2], productions are defined by an original letter and a string that will replace this letter.

Consider this example from The Algorithmic Beauty of Plants with letters a_r, a_l, b_r, b_l, the starting letter a_r and the following productions:

$$
\begin{aligned}
a_r &\to a_l b_r \\
a_l &\to b_l a_r \\
b_r &\to a_r \\
b_l &\to a_l
\end{aligned}
$$

The first few iterations of this L-system will generate these strings:

$$
\begin{aligned}
&a_r \\
&a_l b_r \\
&b_l a_r a_r \\
&a_l a_l b_r a_l b_r \\
&b_l a_r b_l a_r a_r b_l a_r a_r
\end{aligned}
$$

This is not a mere arbitrary example. An L-system such as this one can be used to simulate the development of various algae and the bacteria *Anabaena catenula* [1, ch.1.2], where the letters a and b represent cell states and the subscripts l and r the direction in which daughter cells will be produced.

2.1.1 Other types of L-systems

A major limitation of this basic form of L-systems is their inability to produce sufficiently different instances of the same plant species. We can control the number of iterations of the rewriting rules, however for a given starting letter and iteration count the results will always be identical. For this reason, a randomized extension of L-systems is useful. A *stochastic OL-system* [1, ch.1.7] is an L-system where each production is assigned a probability. A single original letter may be used in multiple productions and the sum of probabilities of these productions is assumed to be one. When rewriting this letter, one of the productions is chosen at random using the probabilities.

Another useful extension is called *context-sensitive L-systems* [1, ch.1.8], where a production may only be applied if the original letter's neighbourhood in the string matches the required context of the production. This can be used to model interactions between plant parts.

The final extension we will discuss is *parametric L-systems* [1, ch.1.10]. They operate on strings of *modules*, where a module consists of a letter and a sequence

[1] We are omitting the full richness of Chomsky grammars for brevity.

of real numbers - the *parameters*. Productions can have a condition associated with them, and are only applied if this condition is fulfilled. Productions can also specify expressions that compute the parameters of resulting modules using the parameters of the original module. Parametric L-systems may be either context free or context sensitive.

2.1.2 Turtle interpretation of L-systems

So far, we have discussed the generation of self-similar strings using rewriting. We will now examine converting the generated strings into a segmented line using a LOGO-style turtle, as described in The Algorithmic Beauty of Plants [1, ch.1.3].

The turtle is an imaginary pen, drawing lines on a plane as it moves. Its state is a position expressed as a 2D vector and orientation expressed as an angle. The turtle can be given commands, such as "move forward one unit" and "rotate by a specified angle". And image can be generated from a string by interpreting the letters as a series of commands. For example, the Koch snowflake and similar curves can be generated this way.

Consider the following L-system, which starts with the string:

$$FRRFRRFRR$$

And has a single production:

$$F \rightarrow FLFRRFLF$$

We may interpret the letters as turtle commands in the following manner: **F** is "move forward one unit", **L** is "rotate 60° to the left", **R** is "rotate 60° to the right". With this interpretation, the L-system draws the Koch snowflake shown in figure 2.1.

A turtle may also be used in three dimensions [1, ch.1.5]. Its state would again include its 3D position and orientation, and instead of drawing lines, the turtle would construct a series of tubes. An example of a structure that can be generated with a 3D turtle is the Hilbert curve.

Our end goal is the branching structure of a tree, but strings are inherently linear. In order to create branching structures with a turtle, we will extend it with a stack, and *push* and *pop* commands for pushing and popping the turtle's state onto the stack. A production in an L-system may generate a branch by inserting letters that represent push and pop commands. See figure 2.2 for an example of turtle interpretation of a branching L-system.

Note that like stochastic L-systems, the turtle interpretation of the resulting string can be randomized to achieve further variation. For example, the letter F may represent the command to move forward by a random distance in a predefined range.

2.2 Limitations of L-systems

The techniques mentioned above tend to operate on each branch separately without taking into account the context of the entire tree. Each branch grows independently of every other branch.

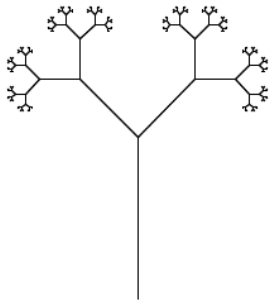

Figure 2.2: An example of a branching L-system [5].

This has the unfortunate side effect of sometimes producing branches that intersect other branches. These self-intersections could be avoided, however implementing such behaviour would be complex and it would likely have great performance impact. In practice, self-intersections are not too visible. Real plants can be observed to fill available space in a mostly even manner, although sometimes two distinct branches may grow very near each other, creating something akin to self-intersection.

Another limitation of omitting the context of the entire tree is that plants tend to grow according to local lighting conditions. A tree will not grow as many leaves and branches in areas that are in shadow. This shadow can be cast by the tree itself. For example we can observe that trees do not generate small branches and leaves near the trunk, where they are overshadowed by the wider crown.

Both of these limitations can be partly remedied by careful tuning of growth parameters.

2.3 Space colonization

Another approach to tree generation, very different from L-systems, is that of *space colonization*, as described by Runions et al. [6].

A volume is filled with *attraction points*, modelling the availability of empty space. The tree itself is constructed iteratively from the root. In each step, branch segments are added in the direction of nearby attraction points. Once an attraction point is reached by the tree, it is removed. The process is stopped when all attraction points are removed, or when another condition is fulfilled, such the maximal number of iterations being reached.

The versatility of this method comes from modifying the distribution of attraction points. We can choose the shape of the filled volume to match our target tree crown shape. We may cut this crown volume with a plane, removing all attraction points on one side, simulating a tree growing next to a wall. Furthermore, the distribution of attraction points need not be uniform. More points can be generated near the surface of the volume, simulating greater availability of light. The set of attraction points may also change over time, for example we can generate new points in the volume, which will cause new small twigs to be generated from already established thick branches. Multiple trees may share the

same attraction point volume to simulate the growth in a forest where trees are competing for space and light.

The downside of this technique is its great computational complexity when compared to L-systems.

2.4 Tree growth principles

In this section we will describe some of the principles behind tree growth. Ideally, our generator will be able to emulate these behaviours in order to produce a realistic tree structure.

Leonardo da Vinci formed a postulate which states that "all the branches of a tree at every stage of its height when put together are equal in thickness to the trunk below them." [1, p.57] We will use this postulate to guide the thickness of outgoing branches in a fork. Specifically, for each fork in the branching structure it should hold that the sum of cross-sectional area of outgoing branches is equal to the area of the original branch. We can think of the branching structure of a tree as a system of tubes transporting some liquid. If the equality did not hold, some part of the tube system would be larger or smaller in diameter than what is optimal.

The book Digital Design of Nature by Deussen and Lintermann [2] describes several other useful principles (based on the Telome Theory [2, ch.2.1]).

Instead of equal branching, a main axis may develop, which then produces most other branches. This behaviour is called *overtopping* and can be observed in most tree species. This can be brought to the extreme with *reduction* of subsequent branches, producing trees with a straight trunk and small branches.

Several other behaviours are described, *planation*, *webbing* and *bending*, however they are mostly relevant to leaves and not the overall tree structure.

Another important behaviour is *budding*, as described in Digital Design of Nature [2, ch.2.2]:

> The bud is a propagation system, since each new branch begins in a bud. We differentiate between terminal main buds and side buds in the axis of carrying leaves (bud axis). The inside of the bud is here protected by fast initial growth of the leaves. In leaf buds, for example, it develops into a branch with leaves, and in the case of flower buds to a bloom or bloom conditions.

Buds are a mechanism through which branches and leaves are generated, so their distribution will have a large impact on the overall tree structure. A branch will create leaves or subsequent branches at so-called *nodes* in regular intervals. These intervals tend to be of a constant length. There are three main ways in which side buds are generated [2, ch.2.2]:

- **Distichy** – a single leaf is created per node, and each subsequent node is rotated by 180°, creating two rows of leaves.
- **Dispersion** – similar to distichy, a single leaf is created per node, however the rotation angle of the next node is smaller, approximating the angle which minimizes the self-shadowing of the leaf formation (also called the golden angle, about 137.5°).

- **Decussation** – two leaves are created per node, opposite of each other, and subsequent node is rotated by 90°.

Two types of branching are described [2, ch.2.3]. In *monopodial* branching, the main axis is dominant, growing longer than the side branches. This behaviour is repeated in a self-similar manner. In *sympodial* branching, side branches are dominant, and the main branch terminates. Note, however, that the side branches need not be equal in strength. Overtopping, or a "main trunk" may be generated by both monopodial and sympodial branching.

These two branching types can be used to describe most tree species found in temperate climate, however they cannot be used to model conifers [3].

Tropisms are another factor contributing to tree shape. In general they describe the tendency of trees to grow or bend branches in a certain direction based on enviromental factors. *Gravitropism* [2, ch.2.5] is the tendency to grow away from gravity, for example observed in the main axis - the trunk. Some parts of the tree may also grow towards gravity, for example roots.

Phototropism describes the tendency to grow in a direction based on incident light direction. Shoots may grow orthogonally to the light direction, or towards the light [2, ch.2.5].

Another behaviour observed in real plants is *apical dominance* [3][7], where the main trunk of a tree tends to grow upwards and dominate over side branches, inhibiting their growth. Side branches dominate over second order side branches, and so on.

2.4.1 Tree roots

Note that we have mostly not discussed generating the roots of a tree. Our work will not cover tree roots at all, instead focusing on the above ground part of a tree, since it is much more visible.

However generating the roots of a tree is in principle not at all different from generating its above-ground body. The very same rules that govern the growth of trunk and branches can be applied to roots, just with a different set of parameters. Their geometry can also be generated in the very same way as trunk geometry.

2.5 Axial trees

The Algorithmic Beauty of Plants describes the concept of *axial trees* [1, ch.1.6], which will be useful for our tree generator.

A *rooted tree* is a tree (graph) where one vertex is the designated *root*, and edges are oriented in such a way that they form paths from the root to leaf vertices. In the context of plants, an edge in such a tree is referred to as a *branch segment*.

An *axial tree* is a rooted tree where for each vertex at most one of its outgoing edges is labeled as a *straight segment*, and the rest are labeled as *lateral* or *side segments* [1, ch.1.6].

An *axis* is a sequence of segments which either originates in the tree root, or its first segment is a lateral segment, and each subsequent segment is a straight

segment and its final vertex does not have an outgoing straight segment. An axis can be thought of as a single branch.

In our generator, we will use a slightly modified concept of an axial tree. Each non-leaf vertex of the tree graph will have *exactly* one outgoing straight segment. The effect of this modification is that every axis must now terminate in a leaf vertex. Decomposing the tree graph into such axes will be used in mesh generation, where each axis will be converted into a bent tube geometry.

2.6 Level of detail

In order to achieve optimal performance, realtime applications that render complex scenes use lower detail variants of far away objects. For example, it is pointless to render several tens of thousands of polygons of something that will be so far away that it only covers a few pixels. This approach is called *Level of Detail*, or *LOD* for short. The same term also refers to the variants of an object with different detail levels.

Let us analyze the ways in which we might reduce the detail level and performance impact of a tree model.

2.6.1 Branch geometry

A tree model consists of two parts: the leaf cards, and the trunk and branches. This geometry is highlighed in figure 2.3. The trunk and branch geometry is mostly made of bent and connected tubes. We can easily customize the conversion of these tubes into a triangle mesh, for example we can use varying number of faces along the diameter of the tube to either increase or decrease its apparent smoothness and triangle count.

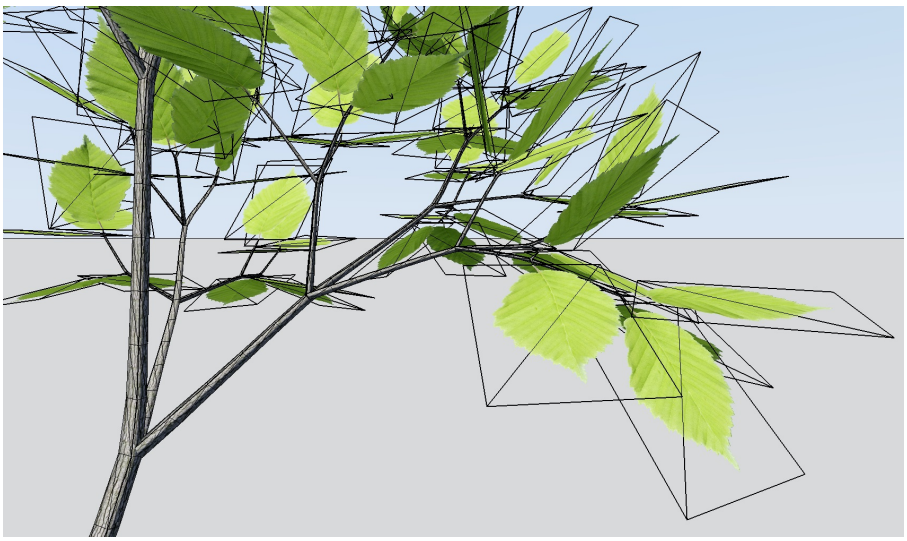

Figure 2.3: A small branch with wireframe, highlighting leaf quads and faceted branches.

Note that a tube or a cylinder does not need a high polygon count to achieve a smooth look. Thanks to surface normal interpolation, even relatively low number of faces can create the impression of a curved surface. The only case when low

face number is plainly visible is when observing the silhouette of the cylinder's upper or lower edge, as can be seen in figure 2.4.

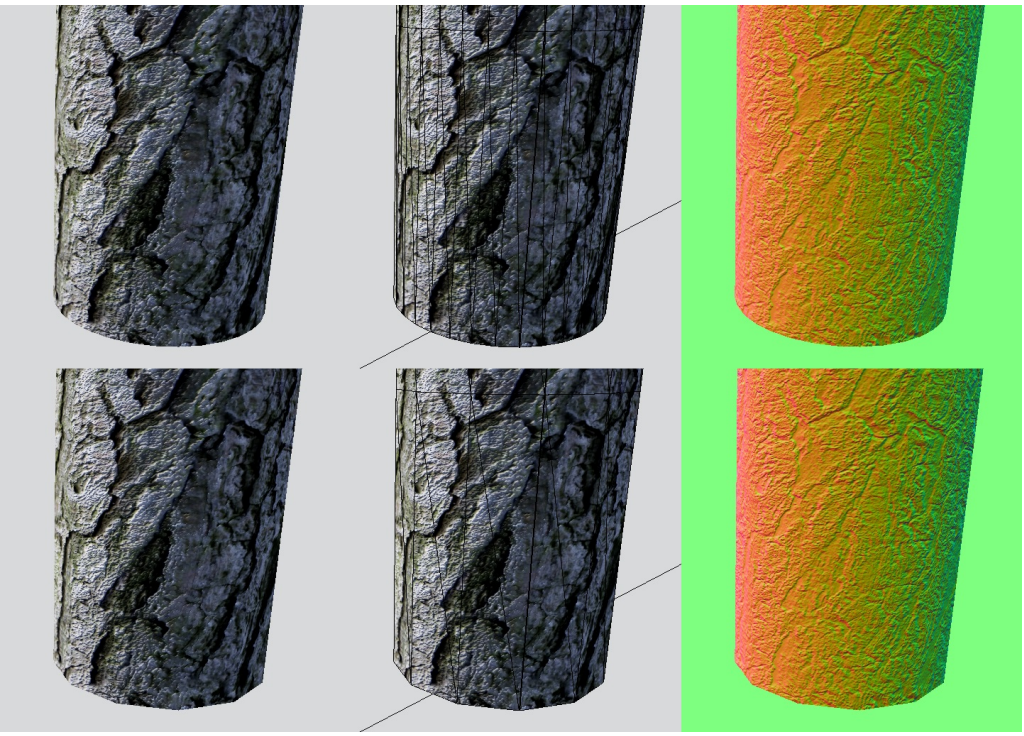

Figure 2.4: Trunk smoothness demonstration.

Trunk cylinder with high face count (top row) and low face count (bottom row), with wireframe (middle column) and normals including a normal map (right column). Note that the only distinguishable difference is at the intersection with the ground.

For trees, the top parts of branches will decrease in radius until they end with a pointy tip and a leaf, hiding the angular nature of the tube geometry. However the bottom end of branches will either intersect the ground, in which case it might be hidden with grass, or intersect another branch, which will visually produce an intersection of two visibly angular polyhedra.

The tree to ground intersection could be hidden with a root system, and branch to branch intersection with a carefully modeled saddle geometry. But in practice, the imperfect branch intersections and ground intersections are not too visible. We omit roots from this work altogether, and we omit saddle generation as well, since it would be a fairly complex task. Both root generation and saddle geometry for branch intersections are a possible future direction to explore.

2.6.2 Leaves and textures

As for leaves, there isn't anything we can do about their geometric complexity. They already only consist of two triangles each. Using a single triangle would be possible, however since we assume that alpha testing will be used for leaves, doing so would lead to wasting many rasterized pixels and pixel shader invocations due to a lot of the triangle surface missing the actual opaque leaf texture [2].

[2]For this same reason it may be beneficial to use more than two triangles for leaves and other alpha tested planes to more closely match the image shape. The benefit of reducing wasted

Reducing the number of leaves would visibly alter the tree, so this is not an appropriate option.

We could also optimize the texture of the leaves (and the trunk) itself, perhaps lower its resolution. This is also not needed in practice, since GPUs and game engines already make heavy use of *mip mapping*, an approach that automatically uses prefiltered lower resolution variants of original textures for surfaces where details of the original would not be visible [3].

Using lower resolution textures could also improve memory usage. However, modern game engines already employ complex techniques in order to only load what is absolutely necessary, such as *virtual texturing*.

2.6.3 Polyplanes

Let us examine leaf cards. It is essentially a plane with an image of a leaf. Why not model the leaf with actual geometry? It would certainly improve the visual appearance of the leaves up close, however the increase in triangle count would be immense, and most of the time the added detail would not even be noticeable. So, instead of complex geometry, we use an image mapped onto a plane made up of two triangles.

Why not apply this very same idea to entire branches? Or the entire tree?

Our primary way of reducing the geometric complexity of a tree will be replacing entire branches with a quad onto which an image of the original branch is mapped. We will call this approach *polyplanes*, and the same term will be used for the planar geometry we replace branches with.

This is a common technique in games, but as far as we can tell there isn't a freeware tree generator that employs it and allows adequate customization of the way polyplanes are generated.

In practice polyplanes are a trade off between geometric complexity and texture memory usage. We will need to store each polyplane's image in some sort of a texture atlas [4], and do so in sufficient resolution for the model to remain visually pleasing at its intended viewing distance.

For this reason, we will analyze ways of reducing polyplane memory usage.

2.6.4 Branch clustering

Let us consider a full-grown tree, where only the tips of branches are replaced with polyplanes. These polyplanes might only contain a few twigs and about a dozen leaves each. This scenario is common for highly detailed LOD levels.

A tree like this could easily contain hundreds of individual branches that are replaced with polyplanes. Storing each one in sufficient resolution would consume large amounts of memory that could be better spent elsewhere.

---

fillrate can outweight the increased face and vertex count.

[3] There is a lot more to mip mapping that what is mentioned here, for example it also improves memory cache utilization and helps reduce aliasing and other artifacts.

[4] An atlas is preferable to multiple smaller textures. Switching textures introduces overhead in most rendering applications, and multiple textures would also introduce complexity into the mesh export process.

Reducing the resolution of each branch image is not desirable, as it would lead to a visually ugly tree model. A good solution would be to instead reduce the number of images needed.

A key observation is that although we are replacing hundreds of unique branches, they are very likely going to be visually very similar to each other, as can be observed in figure 2.5.

We will group branches into *clusters* based on their *similarity*. Every cluster will only generate one set of images (the polyplanes for a single branch may contain multiple images, for example three images from three orthogonal directions). Every branch in a cluster will use these images instead of generating their own.

In order to determine how similar two branches are, we will assign each branch an imaginary position (often more than three-dimensional) based on the growth parameters of its root tree graph vertex. We will then consider the euclidean distance of two branches as their similarity – the closer two branches are, the more similar they are.

Clustering branches based on their root vertex would work poorly if the branches produced from two identical vertices could be vastly different due to randomization, and a different clustering strategy would be required. However root based clustering works reasonably well in our use case.

For a given atlas size, the amount of clusters presents a trade off. Few clusters mean high image resolution, but they may capture the overall shape of the tree poorly due to heavy image reuse. A good cluster count seems to be between 8 and 16.

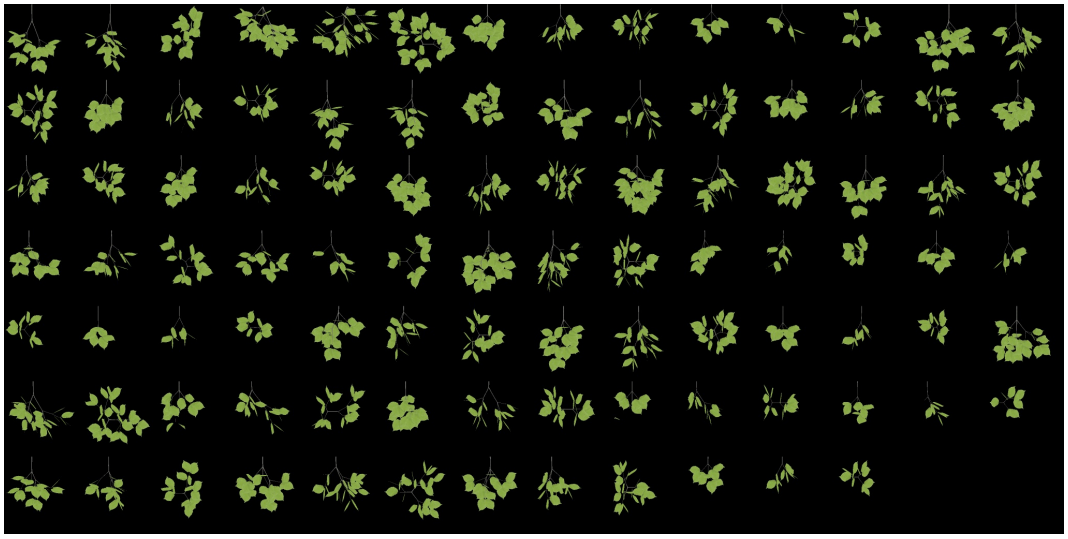

Figure 2.5: An example of polyplane image atlas.

Note the large amount of different branches and their insignificant visual difference.

2.6.5 Sharing polyplane atlases

Since we are reusing images between branches of a tree, why not reuse them across different trees as well?

In video games, a single model of a tree species is usually not enough. When

creating a forest, multiple models of the same species are used for variety. Furthermore, each of the tree models will have several LODs.

We might share the same polyplane atlas across several LODs of the same tree model. This would however not work well in practice. It might ensure that the images for high-detail branches are loaded even if the tree is far away and only using low detail LODs.

But, more significantly, the main difference between LODs is going to be the threshold at which a branch stops generating real geometry and becomes a polyplane. This threshold will be a function of the branch radius. Thicker branches will generate more child branches, which will in turn generate more leaves. Thus branches of different LODs are going to be too dissimilar to place them into a single cluster.

Sharing polyplane atlases across LODs would effectively result in each LOD generating its own images and then combining them into the shared atlas, reducing the effective number of clusters per LOD.

A better way is sharing a polyplane atlas across the same LODs of different tree models of the same tree species. Since the branches are likely to be cut off at the same point, they will also contain similar number of child branches and leaves, and be very similar overall. The main difference between different instances of the same tree species will not be in the ends of tree branches, but in the large-scale structure of the branches, which will still be covered by the unique non-polyplane geometry of the models.

2.6.6 Consistency of different LODs

One major concern when creating LODs for any object, not just trees, is that the different detail levels must still be visually similar. For example, swapping between two levels where each has a different silhouette would be very visible.

Creating consistent tree LODs would be a very hard task for a human, but a tree generator such as ours can handle it automatically, given the proper parameters. See figure 2.6 for an example of consistent tree LODs from out generator.

2.6.7 LOD parameters

Level of detail as we have described it leads us to several parameters we can use to control it.

We will control the number of faces of branches by specifying the number of vertices placed around the circumference of the branch. Furthermore, we will specify this number in relation to the circumference of the branch. This way thicker branches can use smoother geometry while small twigs can use fewer polygons. We will specify:

- Maximal circumference
- Vertex count at maximal circumference
- Minimal circumference
- Vertex count at minimal circumference

Figure 2.6: Tree level of detail demonstration.

An example of several different levels of detail of a single tree, with increasing detail level from left (lowest detail) to right (highest detail). Bottom row displays the same models, but with wireframe view enabled, to highlight their relative complexity. Total face counts of each detail level are (from left to right): 6, 463, 1431, 5541, 20331, 938504.

Branches with higher that maximal circumference will use the vertex count of maximal circumference, similarly for branches with lower than minimal circumference. Branches in between will use a rounded linearly interpolated vertex count value.

To control polyplane generation, we need to specify when a real branch will be replaced with a polyplane. We will specify this with the minimal allowed branch radius. Once a branch radius falls below this value, real branch will be cut off and the rest of it will be replaced with a planar image.

We will also specify the desired number of polyplane clusters, as well as the polyplane atlas size.

2.7 The tree model

In our work, the tree model will be separated into a *trunk mesh*, *leaf mesh* and a *polyplane mesh*, which helps with texturing, since each of these meshes will only use a single set of textures (a color map and a normal map). This maps well to traditional rendering pipelines that cannot use multiple texture sets per mesh. This also helps with mesh export, since we can just generate three files and state which file uses which textures instead of specifying this in the files themselves.

2.7.1 Leaf mesh

Generating the leaf mesh is straightforward: for every leaf, we append a leaf card to the mesh. A leaf card consists of two triangles forming a rectangular shape, and the leaf texture is mapped to its surface.

The leaf texture is assumed to contain an image of a single leaf, including the alpha channel, which will be used along with alpha testing to only render pixels that actually lie within the leaf and discard the rest of the pixels in the leaf quad.

Note that if polyplanes are used, the final model will most likely not contain any leaves at all, since they will all be replaced with (and included in) polyplanes.

2.7.2 Polyplane mesh

So far we have defined polyplanes as a branch replaced with a quad that is textured with an image of the original branch. However, approximating a three-dimensional structure of a branch with a single plane is not sufficient from the aesthetic perspective, especially if the camera is near co-planar with the polyplane, which reduces it to a thin line from the camera's point of view.

For this reason, we will use three perpendicular planes instead, capturing the original branch from three view points. This ensures that whatever the camera's position in relation to the polyplanes may be, at least one plane will always be facing roughly in the camera's direction, partly hiding the planar nature of the structure.

Another key element of maintaining an illusion of a 3D branch is *normals*. In the context of computer graphics, normals describe the orientation of a surface, in particular the vector that is perpendicular to the surface. Normals are the main property affecting lighting.

Normal vectors are usually stored along with each vertex of a mesh, and pixels inside a triangle use interpolated normal calculated using the normals of the triangle's three vertices. To add further details to an object, textures that encode normal vectors, usually called *normal maps*, are used. These textures store relative normals in relation to the surface's original normal. This enables us to have varied lighting even on flat geometry.

For this reason normals can be used to greatly improve the appearance of polyplanes, thus we will generate a normal map along with the color map of each branch. These normal maps are demonstrated in figure 2.7.

2.8 Technologies

Both the generator and the editor will be implemented in C# using .NET Core 3 [5]. Rendering will use the OpenGL graphics API through the OpenTK library [8]. We have chosen these technologies primarily due to our familiarity with them. Using .NET Core and OpenGL also enables us to run the resulting application on non-Windows operations systems, at least in theory [6].

The editor will use the Dear ImGui library [9] for user interface, through the ImGuiNET bindings [10]. ImGui is widely used in rendering-related applications (though mostly for debugging purposes), and this project was a good opportunity to familiarize ourselves with this library.

We will use Newtonsoft.Json library [11] for JSON serialization and deserialization. We have chosen this library over the built-in solution in .NET Core because it provides better control of the serialization and deserialization process.

Our GlobCore library [12] will be used for rendering, which is an OpenGL helper library we have originally created for personal projects.

[5].NET 5 was not available when we started working on this project.

[6]We have only tested our application on Windows 10.

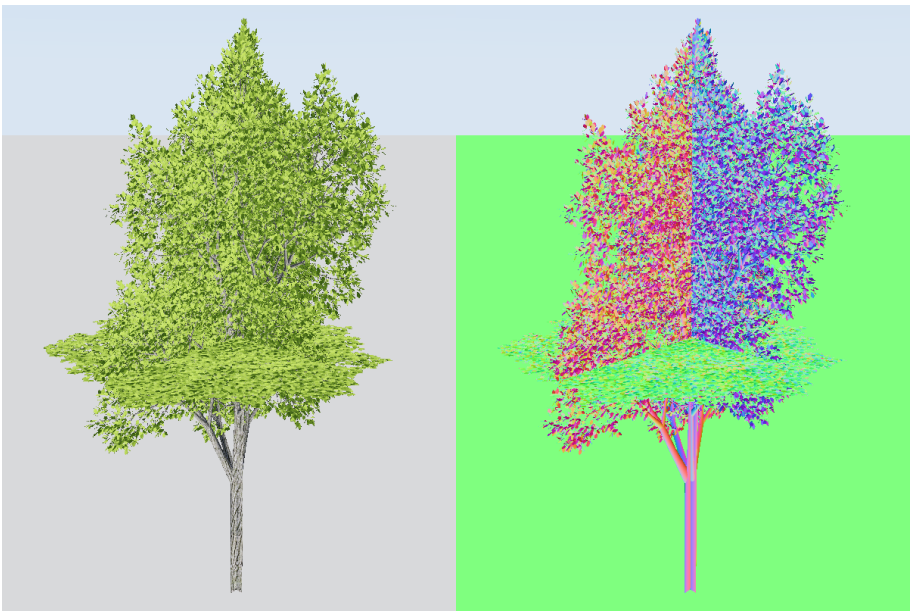

Figure 2.7: Polyplane consisting of three planes.

A demonstration of the three polyplane planes in the case where the entire tree is replaced with a single polyplane set. Note that due to the use of normal maps, the planar nature of the model is almost indistinguishable in a still image (left). Normals are displayed on the right.

3. Tree generator

In this chapter we will provide details about the design of our tree generator. First, we will discuss the generation of the high-level tree structure, then we will describe the process of converting it to a mesh.

3.1 Overview

Our generator should ultimately output finished tree meshes and polyplane textures for a set of input parameters.

When creating an object with an L-System, one would first generate an abstract representation of the object, a string of letters, and then produce the object itself (an image or a mesh) using a turtle interpretation of said string. We choose a similar approach. We separate tree generation into a *grower* that generates an abstract tree representation and a *mesher* that interprets this representation and creates the final tree model. Another advantage is that developing and debugging a tree growth algorithm is easier when it is separated from mesh generation.

Rendering polyplanes will require access to the finished meshes of the branches we are replacing with images, so it also makes sense to design the polyplane renderer as a separate part of the generation pipeline.

Since one of our goals is for the generator library to be usable at run time by other applications, all of the above steps should be implemented in a way that is as portable as possible. This mostly limits the polyplane renderer. The task of rendering a mesh into texture is a typical graphics workload and could be implemented on the GPU. However this would require us to bind our library to a single graphics API. OpenGL would be suitable for its portability and its familiarity to the author, however many games choose Direct3D instead, or use a third party engine.

Another option would be to avoid the GPU altogether and use a software renderer, however implementing one that supports all required features, such as texture mapping and shading, would be very time consuming. Software rendering on the CPU is also not ideal performance wise.

For these reasons we have decided to not handle polyplane rendering in the generator at all. Instead, the generator will provide its user with a set of *render commands* that describe how the polyplane texture should be generated.

These commands carry enough information to make their execution a matter of one for-loop and a few calls to the user's preferred graphics API to set up the required GPU state. In our work, the editor will handle render command execution.

Thus, our generator will be divided into two parts:

- Generating the tree structure using a *grower*.
- Generating the mesh using a *mesher*.

In the first step, we will produce a high level representation of the tree. This will also produce geometric information, such as the layout of tree branches and their thickness. We will refer to this high level tree structure as the *tree graph*,

since a graph is an ideal way of representing a tree. The vertices of this graph will be referred to as *tree vertices* to distinguish them from *mesh vertices*.

The second step will consume the tree graph and produce the final tree mesh, or rather several meshes, split according to texture usage, as described in section 2.7. One mesh will include all geometry with bark texture and will be referred to as *trunk mesh*, another mesh containing leaf cards will be called *leaf mesh* and a final mesh for polyplanes will be called *polyplane mesh*.

The third part of the generation pipeline, polyplane rendering, will be handled by the application which uses the generator.

3.2 Grower

In our generator, we will use an approach that mostly matches a context-free parametric L-system with a single complex rewriting rule (which includes some randomization). However, we will use a different viewpoint. Instead of letters in a string, we will operate on vertices in a tree (graph). In each step, we will apply the production on every new vertex produced in the previous step. We begin with a graph containing a single vertex, the root, and apply the production on this vertex. If it creates any child vertices, we will apply the production on them, then on their children and so on. The branching of the tree graph describes the branching structure of the resulting tree model.

We chose this approach because a graph maps well to the structure of a tree, unlike a string, and we don't constrain ourselves to any particular type of L-system. Instead we generate vertices with arbitrary code, which makes implementation easier and gives us greater freedom than L-systems.

Note that every vertex is processed exactly once, and that the result of processing a vertex is only dependent on its predecessors. This allows us to process different branches of the tree in parallel.

The high-level tree growing code is in `Treegen/Growing/Grower.cs`.

Our production will consist of executing a sequence of *growth modules*. A module takes an input vertex, modifies it and passes it to the next module. During modification, it may also create child vertices. A module itself has a number of parameters affecting its operation, and modules have a strictly defined order in which they execute. These parameters are set before tree generation and are not modified in the process. Tuning these parameters is the primary mechanism through which the appearance and structure of generated trees is controlled.

A vertex in this tree graph will be referred to as a *tree vertex*. Its code can be found in `Treegen/Growing/TreeVertex.cs` and it consists of the following data:

- An array of generic floating point parameters called *attributes*.
- A collection of random number generators.
- Geometric information: the forward vector, the right vector (together describing the orientation of the branch at this point), the length of the branch segment from the previous vertex and current branch radius.
- An array of child vertices.

The vertices of a tree graph will form *branches*. The first child of a vertex is always considered to be the continuation of the current branch, while other children are the beginnings of new side branches. In other words, the edge to the first child will always be considered to be the straight segment, as defined in axial trees (described in section 2.5). A branch is similar to the concept of an axis in axial trees.

The attributes correspond to various parameters used during tree generation. They are used by growth modules, together with the random number generators. Geometric information is used when converting the tree graph to a mesh.

Each leaf vertex of the tree graph will generate a leaf in the tree model.

3.2.1 Random number generation

In order to easily generate different instances of the same tree species, the generation process has to be randomized. However it is also desirable for the process to remain deterministic. For this reason, we use pseudo-random number generators that are ultimately controlled by a single *seed*, which is set once per tree instance.

Using a single random number generator per tree would have a number of disadvantages. A single generator would produce a sequence of pseudo-random numbers, and growth modules would consume this sequence as the tree is generated. This would mean that the order in which we process vertices affects the pseudo-random numbers. While vertices in a single branch have a well defined order of execution (from the root towards the leaf vertex), vertices of neighbouring branches do not. And, as stated earlier, they may even be processed in parallel.

For this reason, each branch will use its own local random number generator.

Another requirement we have for randomization of tree growth is *consistency*. Modifying one randomized aspect of tree generation should only affect this one aspect, and everything else should still look the same. For example, modifying the probability of a branch fork should not affect how many new branches are randomly generated per fork. In a naive implementation, decreasing fork probability would lead to more vertices being generated between forks, and thus more numbers from the pseudo-random sequence being consumed. This would mean that once a fork is generated and a random number is queried to get the new branch count, it would be read from a different location in the pseudo-random sequence, thus possibly returning a different number, resulting in a visually very different tree.

For this reason, a branch will not just have a single local random number generator, but multiple generators, one for each purpose a random number could fulfill. Each purpose has a value in the `TreeRandomType` enum.

As stated earlier, we want to control the random generation of a tree with a single seed value. However, tree generation will use many generators that need to be seeded.

To solve this problem, we introduced a `TreeRandomRepository` class, which stores random generators for a single branch. The seed of a tree is used to seed the repository for the main tree branch, starting with the tree root.

When a new repository is seeded, the seed is used to initialize a special random generator that will only be used to seed new repositories (branches) that fork from

the current one.

Purpose-specific random generators are seeded using a variation of the repository seed.

The code for seeding a repository is as follows:

```
public static TreeRandomRepository CreateNew(int seed)
{
    TreeRandomRepository repo = new TreeRandomRepository();

    // Seeding generator uses the seed directly
    repo._seeding = new SimpleRandom(seed);

    // Every other generator uses the seed
    // XORed with the type name hash
    // That way, even if new generator types are added,
    // the old types remain unaffected
    for (int i = 0; i < RandomCount; i++)
    {
        int localseed = seed;

        unchecked
        {
            localseed = localseed ^ SimpleRandom.HashString(
                ((TreeRandomType)i).ToString());
        }

        repo._randoms[i] = new SimpleRandom(localseed);
    }

    return repo;
}
```

This method is used both to initialize the root repository and a repository for each new branch.

Note that seeds for purpose-specific random generators are generated using the original seed and a hash of their purpose's name. We could use the integer enum value instead, but that could change if we inserted a new item into the enum. Also note that we are using a custom string hash [13] instead of the built-in .NET hash, which is not guaranteed to be stable over time.

The random generator itself (called `SimpleRandom` in code) is a linear congruential generator adapted from a StackOverflow code snippet which is in turn based on an article from Wikipedia [14].

Code related to random number generation can be found in the following files:

- `Treegen/SimpleRandom.cs`
- `Treegen/Growing/TreeRandomRepository.cs`

3.2.2 Tree vertex attributes

Our goal is to algorithmically implement tree growth principles as they are described in section 2.4. For this purpose we will store several attributes in each vertex. Note that most of them are rather arbitrary and not based on real plants. They are however useful in controlling tree structure.

- **Age** can be interpreted in multiple ways. It is the relative age of the tree vertex, where the maximal value of 1.0 is a full-grown node, and values close to zero are very young nodes. It can also be viewed as the nutrient flow through a node. Ages of branches in a fork will fulfill the da Vinci postulate described in section 2.4. Age is also directly related to branch cross-sectional area at a given point, and thus also affects branch radius. The effect of different starting age can be seen in figure 3.1.
- **Dominance** determines how "trunky" a branch is. Dominant branches are affected by the imaginary "antigravitropism" described in section 3.2.3. This behaviour is inspired by *apical dominance* described in section 2.4.
- **Stiffness** determines how easily the branch can bend. This only affects gravitropism.
- **BranchHormone** is the intensity of an imaginary hormone which controls when new branches are generated.
- **Roll** is a helper attribute used when orienting new branches.
- **Gravitropism** determines how much a branch bends towards gravity, or away from gravity if it is negative.
- **LastBranchRadius** is a helper attribute used to smooth out branch forks.

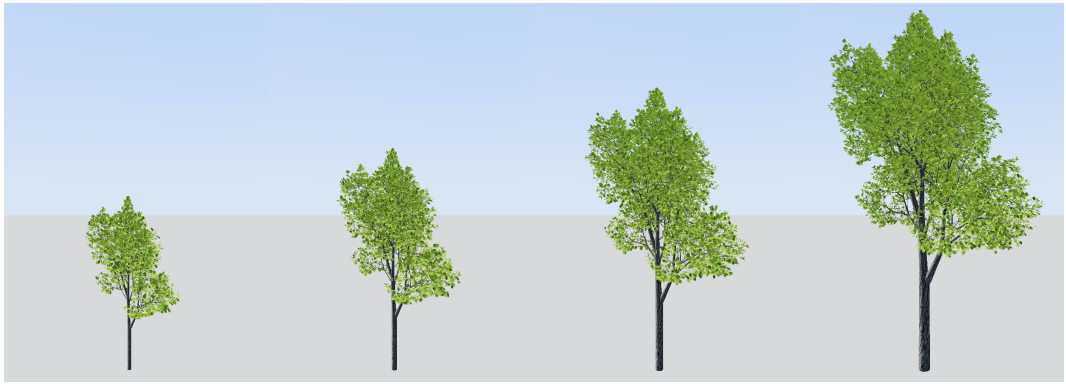

Figure 3.1: A demonstration of the age vertex attribute.
The trees are identical except for their starting age, which is (from left to right) 0.125, 0.25, 0.5, 1.0.

3.2.3 Growth modules

We will now list all growth modules in the order in which they are executed.

1. **Root module** is the first module to execute. Its only purpose is to set the initial dominance parameter to a user specified value.

2. **Gravitropism module** computes the gravitropism attribute. Growth is directed away from gravity for dominant branches, and towards it for non-dominant branches.

3. **Geometry module** computes branch radius, branch segment length and applies the gravitropism attribute.

4. **Branching module** has the greatest effect on tree structure. It controls when and how branches are generated, when they are terminated, how dominance and age are distributed in a branch fork and in what directions new branches will grow. This module is responsible for creating new tree vertices for both branch forks and normal branch growth.

Source code of all growth modules can be found in the `Treegen/Growing` directory.

Note that in the source code of growth modules some parameters (properties) have a `GrowParamSlider` attribute. This is used to specify that this parameter should be controlled by a slider in the editor user interface, and minimal and maximal values are specified as well. Properties without this attribute are controlled by a text box that accepts the appropriate number types (integer or floating point). An example of this attribute in `GeometryModule` is given below.

```
public float RadiusScale { get; set; } = 0.2f;
public float LengthScale { get; set; } = 0.5f;

[GrowParamSlider(0.05, 0.5)]
public float LengthPow { get; set; } = 0.35f;

[GrowParamSlider(0f, 0.999f)]
public float BranchRadiusTrailing { get; set; } = 0.5f;
```

3.2.4 Branching module

We will explain the inner workings of the branching module in greater detail, as it is most important in determining the tree structure.

Parameters **HormoneBuildUpMin**, **HormoneBuildUpMax** and **HormoneThreshold** control how often new branches are generated. This uses the BranchHormone tree vertex attribute mentioned earlier. This attribute starts at zero, and each new vertex added to the current branch increases this value by a random number from the range specified by *HormoneBuildUpMin* and *HormoneBuildUpMax*. Once this value reaches or exceeds the *HormoneThreshold* value, new branches are generated and BranchHormone value is set to zero for each of the child vertices, including the vertex that is the continuation of the current branch.

TripleBranchProbability controls the probability of a fork with three outgoing branches (including the current branch) being generated instead of a fork with two outgoing branches.

BranchSlant is used to control the angle between the direction of original branch growth and the direction of child branches. Note that the actual slant angle of each child branch is dependent on their relative ages. Branches with higher age will grow straighter and others will be pushed away.

BranchRoll is used to rotate each new set of child branches relative to the previous set. This can be used to emulate distichy, dispersion and decussation mentioned in section 2.4.

MaxLeafAge controls the smallest age allowed for a tree vertex. When age falls below this threshold, the branch is terminated and a leaf is generated by the mesher. Thus, no leaf can have an age larger than this value. This parameter effectively controls how many leaves will the tree have. Lower values lead to more leaves.

OriginalBranchAge controls what portion of the original age is kept in the main branch at a fork. **OriginalBranchAgeDeviation** controls how much this value will randomly deviate. This parameter can be used to control overtopping mentioned in section 2.4.

DominanceInheritance controls how much dominance child branches inherit from the main branch during a fork.

DominanceForwardnessFactor controls the tendency of dominant branches to continue growing in their original direction during a fork. If this value is low, the original branch will be "pushed" to the side by its child branches.

MainTrunkForwardnessFactor works similarly to *DominanceForwardnessFactor*, but is only applied to the main trunk, or more specifically, any branch that has dominance above 0.99.

AgeSapping is used to simulate stress [3]. Stress causes imaginary nutrients (the age attribute) to be partly consumed in each node.

3.3 Mesher

At this point in the generator pipeline we have "grown" a tree graph and are ready to generate the meshes themselves.

A mesh will consist of a *vertex buffer* – an array of vertices, and an *index buffer* – an array of indices, indexing into the vertex buffer. Each three succeeding indices will form a triangle from the vertices they reference.

A single vertex will consist of a position, normal, texture coordinates and tangent vector. The tangent vector is only required for rendering with normal maps and is not present in exported meshes, as it can be easily computed from normals and texture coordinates.

All code related to meshing can be found in the `Treegen/Mesh` directory.

3.3.1 Pipeline overview

If we were to only generate the trunk and leaf mesh with no polyplanes, the mesh generation pipeline would be simple. We would first decompose the tree graph into branches (similar to axes in axial trees described in section 2.5) and

then generate tube geometry for each branch. Introducing polyplanes adds a great deal of complexity to the mesh generation process, however this simplified pipeline is still used to generate the meshes that are rendered into polyplanes.

The full pipeline is as follows:

1. Split the tree graph into branches. Terminate each branch at the point where it should be replaced with a polyplane.
2. Gather the *origin vertex* of each polyplane – the tree graph vertex at which the cutoff to a polyplane occured.
3. Generate clusters (described in section 2.6.4) from polyplane origin vertices.
4. Allocate polyplane images in the polyplane atlas.
5. Generate polyplane render commands and meshes.
6. Place polyplanes at polyplane origin vertex locations.
7. Generate tube geometry for each branch.

This is the pipeline for generating a single tree model. However the actual generator is capable of generating multiple tree models at once. This is done to enable sharing a single polyplane atlas among multiple tree instances of the same species and the same LOD.

This only affects the pipeline slightly. The steps 3, 4, and 5 are simply done for all trees at once, while the rest are done for each one separately.

Most code related to mesh generation can be found in the file `Treegen/Mesh/Mesher.cs`.

3.3.2 Branch decomposition

A branch, as described in section 3.2, is a path through the tree graph, similar to an axis of an axial tree, as described in section 2.5. For every vertex in a branch except the first one it must hold that the next vertex in this branch must be the main child of the current vertex. Breaking the tree graph down into branches assists us with mesh generation, since each branch can be converted into a continuous bent tube geometry. The tubes of child branches will simply originate inside the geometry of their parent branch.

Note that tree vertices only store their position in relation to the previous vertex. However we will need to obtain absolute world positions of each vertex for mesh generation. For this reason, branches will also store the absolute position of their starting vertex. This position will be zero for the root branch and vertex.

Decomposing the tree graph into branches is a simple recursive function. To construct a branch, we begin at its starting vertex, add it to the branch and move to the main child of this vertex. We repeat until we move into a leaf vertex, then we terminate. We will also call this function recursively for every non-main child vertex we encounter to construct their respective branches. We will also keep track of the radius and absolute position of each processed vertex.

As we have specified in section 2.6.7, branches below a certain radius will be cut off and replaced with polyplanes. Branch decomposition needs to be aware of

this and will stop generating a branch if this polyplane condition is fulfilled. In that case it will also generate a polyplane origin vertex. Note that the first vertex the radius of which falls below the threshold is still included in the branch in order to avoid a visually abrupt transition between real geometry and a polyplane.

Polyplane origin vertex consists of the first branch vertex the radius of which fell below the threshold and of the absolute position of this vertex.

3.3.3 Clustering

Once we have gathered all the polyplane origin vertices, we will group them into clusters based on their similarity.

A "position" vector (an array of floating point numbers) is generated for each vertex. The lower the distance between position vectors of two vertices, the more similar they are. This vector is directly constructed from the values of the different tree vertex attributes, however some attributes have more weight than others. For example the value of age is multiplied by a large number to increase its influence on clustering. See the method `GetClusteringVector` in `Treegen/Growing/TreeVertex.cs` for more details.

For clustering itself, we have chosen to implement the K-Means++ algorithm [15] due to our familiarity with it. This algorithm produces cluster centers and every vertex is assigned a cluster based on its nearest cluster center. However these centers need not match any of the input vertices. For this reason, after running K-Means++ we find the nearest real vertex to every cluster center, proclaim these vertices as the centers of their respective clusters and then reassign the clusters of every vertex based on their proximity to the new cluster centers. See figure 3.2 for an example of polyplane clustering.

3.3.4 Polyplane generation

At this point in the meshing pipeline we have already determined which polyplane origin vertices will actually generate polyplane images and which will just reuse images from other origin vertices. We will now generate render commands and polyplane card meshes. The process can be broken down into three steps:

1. Generate meshes for each cluster center and initialize render commands.
2. Allocate atlas space for each image.
3. Finalize render commands and generate polyplane card meshes.

In the first step, we generate meshes that will be rendered into images for each polyplane. We use the same branch decomposition and branch mesh generation that is used for full tree models, but we do not terminate branches if their radius falls below polyplane threshold, like we would in normal mesh generation.

We will eventually pass those meshes to the polyplane renderer inside a *render command*, which also consists of a transform matrix, the rendered region size and viewport information about where the meshes are to be rendered in the polyplane atlas.

We set the transform so that the mesh is oriented vertically in the direction of growth at the polyplane cluster origin tree vertex. We also compute its rotated

Figure 3.2: A tree with two polyplane clusters, visualized using yellow and purple colors.

bounding box extents. Each polyplane consists of three images from the three orthogonal directions. We use the bounding box extents to set the relative size of the images.

In the second step, we pass the relative sizes of each polyplane image to an *atlaser*, which will then allocate actual texture space for these images. This process is described in section 3.3.5.

In the third step, we finish the render commands by including the atlas locations and sizes of the final images. We also generate a mesh containing three cards for each polyplane cluster origin with the proper texture coordinates. This mesh is oriented so that the polyplane origin is at zero in its coordinate space, the increasing Y axis points along growth direction and X axis points along the right vector of the original tree vertex.

When placing polyplane cards for each actual polyplane origin (not just cluster origins), we simply append the proper polyplane card mesh to the final polyplane mesh, rotated and translated according to the origin's position.

3.3.5 Allocating atlas space

To allocate atlas space, we essentially need to pack several smaller rectangles into a larger rectangle. This is known to be a NP-hard problem [16].

Furthermore, we may rotate the rectangles and we may also resize the larger

atlas rectangle (uniformly in both directions). Changing the size of the atlas rectangle can be thought of as resizing all the packed rectangles at once. This is a sensible action, since we don't care about the exact sizes of the packed rectangles, but their relative sizes should be maintained. Otherwise, one branch polyplane could have a high resolution image, while the next would be just a few pixels, causing a visible discrepancy.

We implemented the following heuristic. We split the atlas space into rectangular tiles of identical size, and place each image into one of these tiles. These tiles need to be large enough to fit the largest image, but also small enough so we can fit enough tiles into the atlas to cover all images. Due to our ability to resize the tiles (or the atlas), we can simply find the largest image, and then find the largest possible tile size that has the same aspect ratio and that can fit into the atlas enough times to cover all images. We use a binary search algorithm to do this. See figure 3.3 for an example of the result of our clustering algorithm.

This approach was originally intended as a quick and simple solution that would allow us to develop and debug the polyplane generation process as a whole, and was intended to be replaced with something more sophisticated. However, this approach works reasonably well in our use case since polyplane branches are cut off at the same radius, thus all polyplane images are likely to be of a similar size. For this reason we have decided to keep using this approach although a better solution could certainly be implemented in the future.

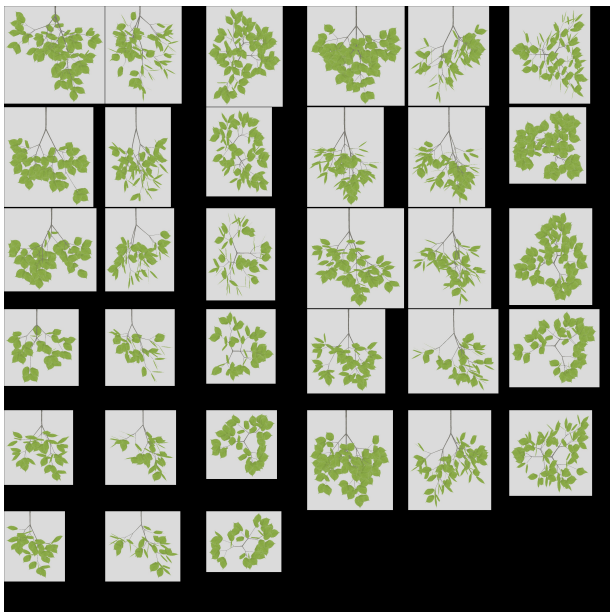

Figure 3.3: A visualization of the result of our atlaser.

Grey regions are included in actual polyplane images, black regions are unused.

3.3.6 Branch mesh generation

First, the branch vertices are passed through a filter that interprets them as a control polygon of a cubic Bezier curve, and moves each vertex to its corresponding position on the curve. This smooths the branch geometry, avoiding abrupt growth direction changes that the grower produces. To generate a branch mesh

we generate a vertex ring for each tree vertex of a branch, rotated according to the right and forward vectors of the vertex. We then connect subsequent rings with faces. However, each subsequent vertex ring is slightly rotated around the axis of growth in order to place its vertices as near their counterparts in the previous ring as possible. This helps us avoid twisting artifacts in the mesh.

Trunk texture coordinates are generated so that the texture always wraps around the entire branch in the X axis. The texture X coordinate is also shifted slightly along the branch length into a spiral pattern to hide texture repetition. The Y texture coordinate is increased with unit of branch length, however the distance the Y coordinate is moved is also divided by the current branch radius. This eliminates stretching artifacts for thin branches. See figure 3.4 for a demonstration of trunk texture coordinates.

At the end of each branch we generate a leaf card, unless it is a branch that ends with a polyplane.

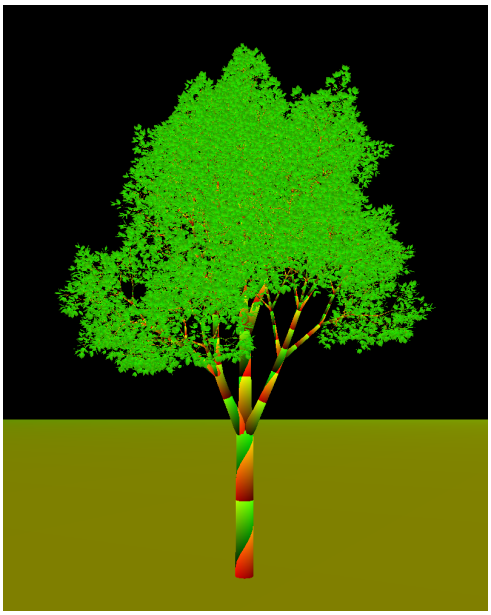

Figure 3.4: A visualization of texture coordinates of the final mesh.

Red color channel corresponds to the X texture coordinate axis, green to the Y axis. Note the twisting pattern visible in the trunk. Also note that the texture repeats more often in the growth direction on thinner branches.

4. Editor

The editor serves as a frontend for the tree generator, allowing the user to edit trees interactively, save and load grower parameters or export final tree models. The editor displays the currently edited tree including all its levels of detail. It can also display multiple instances of the tree, where each instance can have a different seed and a different starting age. These instances will share polyplane atlases, as described in section 2.6.5.

The user can change growth parameters using several sliders and text boxes. The regeneration of the tree must be triggered manually, either using an option in the menu or a keyboard shortcut. Automatic regeneration would not be appropriate, since generating the trees can take several seconds or more.

The editor also contains a set of default trunk and leaf textures from ambientGC [17].

A full user guide is included in `docs/guide.md`.

The `TreeEditorState` class stores everything about the edited trees. It stores the current "project". This class is serialized into JSON in order to save current tree parameters. It consists of:

- Grower parameters
- Mesher parameters, including LOD parameters
- List of tree instances
- File names of used textures

Another notable class is the `TreeBatch`, which contains the state, manages its loading and saving, model exporting and setting texture filenames. Most of the editor logic itself is implemented in `Editor.cs` and menus are defined in `UserInterface/MenuGenerator.cs`.

A notable feature is that user interface for growth modules is generated automatically using reflection. As mentioned in section 3.2.3, parameters of a growth modules may be decorated with an attribute specifying that the parameters should be represented with a slider. See figure 4.1 for an example of the auto-generated UI.

Selecting textures uses a file picker adapted from GitHub user prime31 [18].

4.1 Rendering

Editor uses a simple forward renderer implemented in OpenGL. Its features include a simple diffuse lighting (with no shadows), normal mapping, fake backscattering for leaves, tonemapping [19] and gamma correction [20].

During rendering of alpha-tested geometry, a separate alpha texture is used [1] instead of the alpha channel of the diffuse texture.

[1] Having separate alpha makes it easier to process it separately from color, for example by converting it to a signed distance field. In the end we decided not to do this as such preprocessing is better left for the game engine that would consume our trees.

The renderer uses a module based architecture, where each module takes care of one part of the rendering pipeline. This is done mostly out of habit, as we use this architecture in some of our other projects which have much more complex renderers.

Shaders can be found in the `Shaders` directory. Note that shaders can be edited at runtime and are automatically reloaded by the application. This behaviour is handled by our GlobCore library.

4.1.1 Polyplane renderer

Rendering polyplanes is similar to filling a g-buffer during deferred shading. Diffuse, normal and alpha textures are all written to at once by the pixel shader using multiple render targets.

First, a frame buffer object is bound, to which polyplane atlas textures are attached (diffuse, normal and alpha textures). This framebuffer is cleared and then render commands are executed. An orthogonal projection matrix is set so that the viewport includes the command's rendered region extents, and the command's transform is set as the world matrix. Command's meshes are then rendered, including the proper trunk and leaf textures.

The code is in `TreeEditor/Rendering/PolyplaneRenderer.cs`.

4.1.2 ImGui renderer

The ImGui renderer is adapted from a sample program for the ImGui.NET library [21]. ImGui produces a list of render commands, similar to our polyplane generator, that need to be executed by the application. The code can be found in `TreeEditor/Rendering/ImguiRenderer.cs`. This class also handles passing keyboard and mouse input to ImGui.

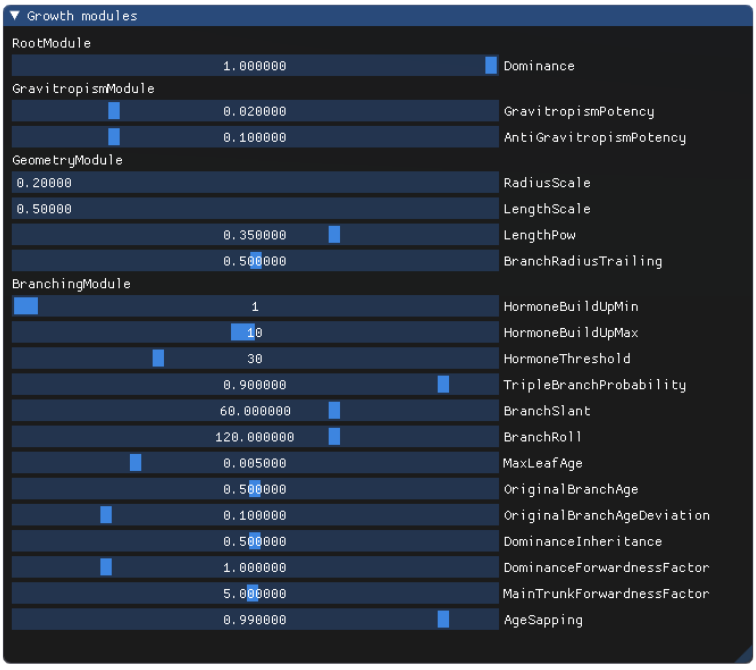

Figure 4.1: Auto-generated ImGui user interface for growth modules.

5. Documentation

The user may interact with our work in several ways. For the user that wants to use the editor to generate tree models we have prepared the user guide described below in section 5.1. A brief overview of the inner workings of the generator can also be found in `docs/docs.md`.

For the user that wants to integrate the library into their own program we provide an overview of using our library in section 5.2 and we have also prepared a sample program that uses the generator library.

The user that wants to modify or extend our library should refer to chapters 3 and 4 which describe the high level structure and workings of our generator. Low level details can be found in the source code in the form of many comments.

5.1 User guide

An application such as this is ultimately aimed at hobbyist game developers. These users may not necessarily be programmers, but will in general have a good understanding of the principles and inner workings of video games. Thus there is no need to explain concepts such as a "texture" or "vertex count" and a quick tutorial with many descriptive images is sufficient. This tutorial can be found in `docs/guide.md`.

The application comes with several demo tree "projects" included in the `trees` directory.

5.2 Using the generator library

5.2.1 Public interface

Public interface of the library consists of:

- **All growth modules**, their base class, `TreeVertex`, `TreeRandomRepository` and `SimpleRandom`. This allows the user to create their own growth modules.
- **Mesher** and all related classes: `MesherParams`, `MesherResult`, `RenderCommand`, `Mesh`, `MeshVertex`.
- **ObjWriter** to allow easy export of generated meshes.

5.2.2 Generating a tree

In order to generate a tree, one would:

1. Create an instance of the `Grower` class.
2. Add all required growth modules to the `Modules` list in the grower. This step can be merged with the previous one by using the `Grower.CreateWithAllModules` method, which creates a `Grower` instance with all growth modules from the library already included.

3. Create a root vertex for the tree using the `TreeVertex.CreateRoot` method.

4. Grow a tree from this vertex using the `Grower.Grow` method. Set the root vertex into the `Root` property of the `GrowContext` parameter.

5. Generate meshes from this vertex using `Mesher.GenerateBatch` with the appropriate parameters. This method returns a `MesherResult` instance.

6. Execute the render commands provided in the `MesherResult`.

See the project `TreeSample` for a very simple example application that generates the default tree and saves it into a Wavefront .obj file.

The editor itself also serves as an example integration of the tree generation library into an application. Please refer to the `TreeBatch` class which interfaces with the generator, especially its constructor and the `UpdateAll` method.

An example of polyplane render command execution can be found in the file `TreeEditor/Rendering/PolyplaneRenderer.cs`. Also refer to section 4.1.1.

6. Conclusion

We have provided a generator capable of producing different tree types and highly customizable LODs, scaling from just several polygons to detailed trees with individual leaves. The generator may also be included in another application to enable runtime tree generation. Furthermore, it is written in a multi platform way, allowing it to run on non-Windows systems.

Our tree generation is based on "growing" a tree graph and mimics real plant growth. It is inspired with L-systems and recreates self-similarity that can be observed in real trees. Extensibility was not our primary goal, however the design of the generator and editor allows for relatively easy extension of tree graph generation with the addition of new growth modules. User interface of these new modules is generated automatically as long as its parameters are of the supported types.

The complexity of tree models comes mainly from their many individual leaves and tiny twigs that connect them to the greater branch system. Our main strategy is to replace this complex geometry with a card onto which the image of the original branch is mapped. This process is fully automatic and can be controlled with just four easy to understand parameters [1]. Furthermore, we generate level of detail in a way that preserves the tree silhouette, which helps with hiding LOD transitions.

We feel that we have created a complete and useful tool despite the many possible improvements listed in the next section.

See figure 6.1 for an example of our generated level of detail, figure 6.2 for a screenshot of the editor with user interface and figure 6.3 for a screenshot of a set of generated trees.

6.1 Future improvements

There are many ways our work could be extended in the future. We will list some of these extensions below.

- **Normal smoothing** – using fake smoothed normals instead of real normals can improve the appearance of trees. Tree crowns tend to have a spherical shape which appears smooth from a distance, and both leaf and polyplane normals could be adjusted to resemble the smoother surface of a sphere.
- **Branch intersection improvements** – branch intersections are just two intersecting polyhedra in the mesh. A smoother saddle-like geometry could be generated instead.
- **Roots** – support for generating tree roots could be added, likely using the same growth modules, but with a different set of parameters.
- **Voxelizer** – voxel based video games and art are becoming increasingly popular, likely due to the popularity of *Minecraft* and voxel editing software

[1] Branch radius at which it is replaced with a polyplane, number of polyplane clusters, width and height of the image atlas.

MagicaVoxel. By having a high level representation of a tree in the form of the tree graph, our generator could produce accurate voxel tree models with better results than voxelizing a tree mesh could produce, at least for large voxel sizes. There is already an experimental implementation of a voxelizer included in the source code, it is however not complete and not made available to the user.

- **Extended tree generation** – the versatility of our tree grower could be improved. We could provide much more control over tree growth to the user by replacing scalar parameters with curves mapping current age to a parameter value. Support for generating conifer trees is also missing.
- **Improved atlaser** – allocation of polyplane images from the atlas could be improved by reducing the amount of wasted space, thus increasing the resolution of polyplanes without increasing memory consumption. This could be done by implementing a better algorithm, perhaps one that can also resize polyplane images or rotate them.
- **Animation** – motion from wind is key in bringing a nature scene to life. We could generate animation data during the mesh creation process automatically. However creating an efficient tree animation system is not something that can be done at the asset generation level, since it would need to handle entire forests with hundreds of trees. A custom rendering pipeline would be required to achieve good performance, but that would tie us to a single rendering engine.
- **Tree geometry editing** – the editor could allow the user to modify the tree graph by hand, for example by rotating growth direction, by translating vertices or by inserting entirely new vertices. This could be used to produce customized tree shapes, where the user would create the high-level branch structure by hand, but everything else would still be generated procedurally. This would combine the strengths of manual and procedural tree creation while retaining automatic level of detail.

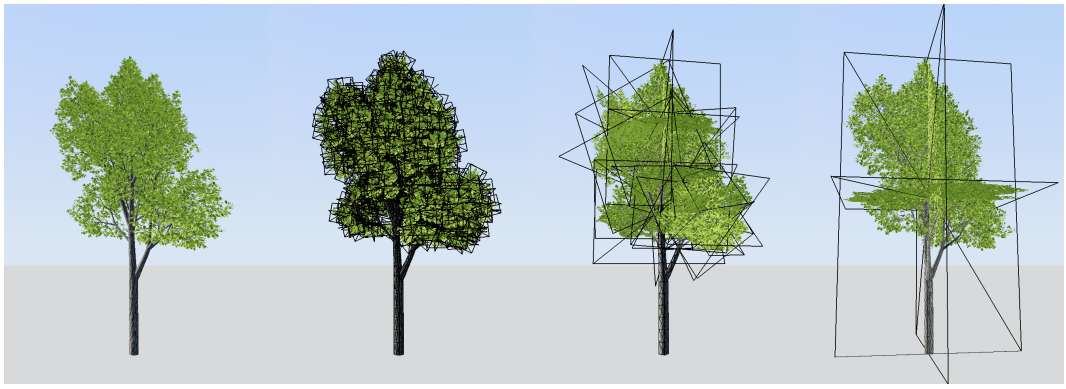

Figure 6.1: A demonstration of level of detail.
On the left there is a tree with individual leaves, other trees use polyplanes with different radius cutoff. Wireframe is enabled for better visibility of polyplane cards.

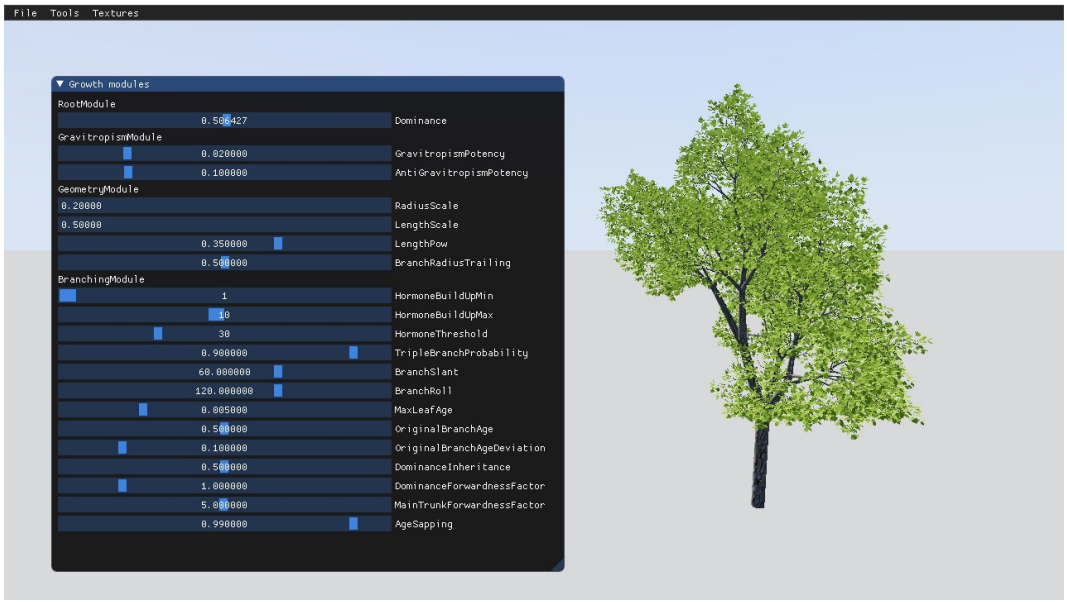

Figure 6.2: A demonstration of the tree editor including UI.

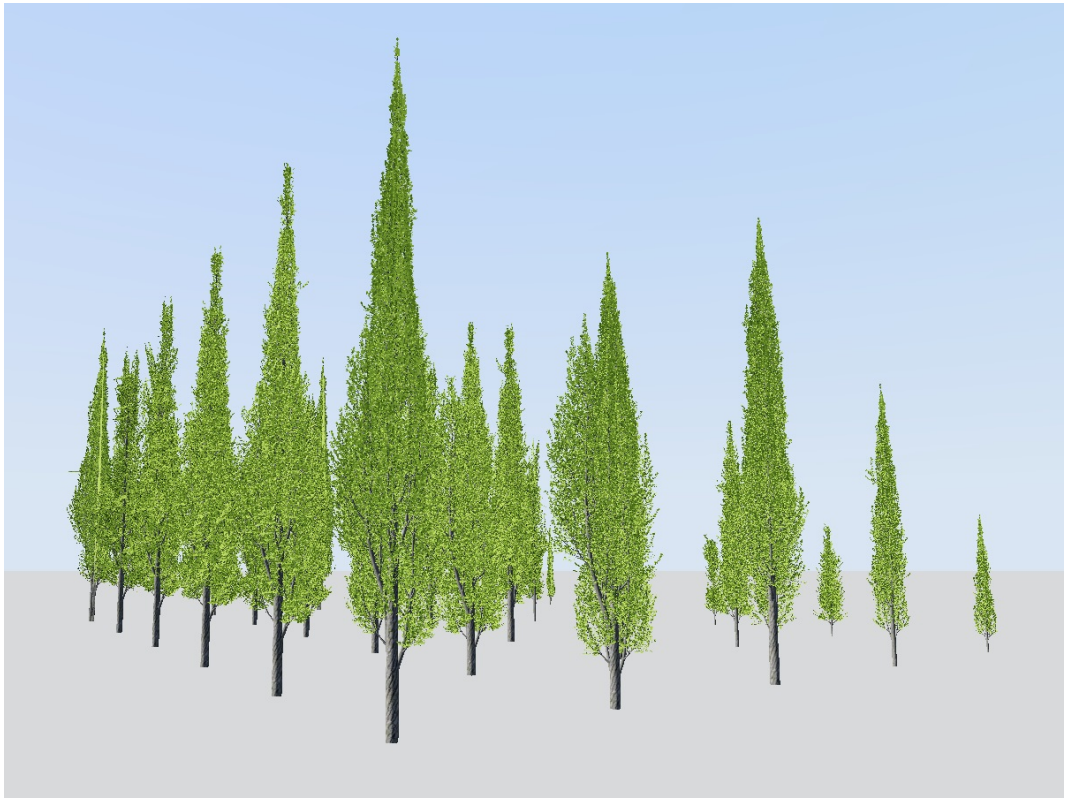

Figure 6.3: A demonstration of the generated trees.

References

[1] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. 1996. Available online at `http://algorithmicbotany.org/papers/#abop` as of 2021-05-20.

[2] O. Deussen and B. Lintermann. *Digital Design of Nature*. Springer, 2005.

[3] Jan Hovora, 2021. A consultation.

[4] Wxs, 2007. Image of the first four iterations of the Koch snowflake, by Wxs, licensed under the CC-BY-SA 3.0 License, available online at `https://commons.wikimedia.org/wiki/File:KochFlake.svg` as of 2021-05-20.

[5] Svick, 2008. An example of a branching L-system, by Svick, licensed under the CC-BY-SA 3.0 License, available online at `https://en.wikipedia.org/wiki/File:Graftal7.png` as of 2021-05-20.

[6] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. pages 63–70, 01 2007. doi: 10.2312/NPH/NPH07/063-070. Available online at `https://www.researchgate.net/publication/221314843_Modeling_Trees_with_a_Space_Colonization_Algorithm` as of 2021-05-20.

[7] Apical dominance. Available at `https://en.wikipedia.org/wiki/Apical_dominance` as of 2021-05-15.

[8] The OpenTK library, available at `https://opentk.net/` as of 2021-05-11.

[9] Omar Cornut. The Dear ImGui library, available at `https://github.com/ocornut/imgui` as of 2021-05-11.

[10] mellinoe, 2017. The ImGui.NET library, available at `https://github.com/mellinoe/ImGui.NET` as of 2021-05-11.

[11] The Newtonsoft.Json library, available at `https://www.newtonsoft.com/json` as of 2021-05-11.

[12] The GlobCore library, available at `https://gitlab.com/Kvaleya/globcore` as of 2021-05-11.

[13] Jon Skeet, 2011. Code for a simple string hash, available at `https://stackoverflow.com/a/5155015` as of 2021-05-09.

[14] xeranic, 2013. Code for a pseudorandom number generator, available at `https://stackoverflow.com/a/17094254` as of 2021-05-20. The code itself is adapted from a Wikipedia article, available at `https://en.wikipedia.org/wiki/Linear_congruential_generator` as of 2021-05-20.

[15] Milan Straka. Machine learning for greenhorns, lecture 11 (pca, k-means, gaussian mixture). Lecture at Faculty of Mathematics and Physics, Charles University. Available at `https://ufal.mff.cuni.cz/courses/npfl129/2021-winter` as of 2021-05-11.

[16] Rectangle packing. Available at https://en.wikipedia.org/wiki/Rectangle_packing as of 2021-05-11.

[17] Lennart Demes. ambientcg. Default trunk and leaf textures are from https://ambientcg.com/ (formerly cc0textures), licensed under CC0 1.0 Universal. Available at https://ambientcg.com/view?id=Bark001 and https://ambientcg.com/view?id=LeafSet014 as of 2021-05-23.

[18] prime31. File picker for ImGui.NET. Available at https://gist.github.com/prime31/91d1582624eb26353954173930180l6e as of 2021-05-12.

[19] Jim Hejl. Filmic tonemapping function, available at https://twitter.com/jimhejl/status/633777619998130176 as of 2021-05-12.

[20] Sébastien Lagarde and Charles de Rousiers. Moving frostbite to physically based rendering 3.0. Page 88, available at https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf as of 2021-05-12.

[21] mellinoe, 2017. ImGui renderer from ImGui.NET sample program, available at https://github.com/mellinoe/ImGui.NET/blob/master/src/ImGui.NET.SampleProgram.XNA/ImGuiRenderer.cs as of 2021-05-12.

List of Figures

A. Attachments

File or directory name	Contents
`docs/`	Contains user guide and a brief documentation.
`src/`	The complete source code of the project, including a Visual Studio solution.
`textures/`	Default bark and leaf textures.
`trees/`	Example tree files.
`README.md`	Brief readme with basic instructions, including compiling.

The repository with the same contents can also be found at `https://gitlab.com/Kvaleya/treegen` at the time of writing [1].

The user tutorial can also be viewed at:
`https://gitlab.com/Kvaleya/treegen/-/blob/master/docs/guide.md`.

The project is compilable with Visual Studio 2019. Note that the final executable **must be run in the root directory of the repository**. For example it will look for shaders in `src/Shaders`.

[1] 2021-05-17. The repository contents may be updated in the future.