TU WIEN Informatics

Generieren ästhetischer Pflanzenmodellen aus einem offenen Datenformat des Projekts für nachhaltige Agrarökosysteme in Godot

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Moritz Leander Großfurtner
Matrikelnummer 00271409

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Christian Freude

Wien, 4. Dezember 2023

Moritz Leander Großfurtner	Michael Wimmer

TU WIEN Informatics

Generating Aesthetic Plant Models from an Open Data Format of the Project for Sustainable Agroecosystems in Godot

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Moritz Leander Großfurtner
Registration Number 00271409

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Christian Freude

Vienna, 4th December, 2023

Moritz Leander Großfurtner	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Moritz Leander Großfurtner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Dezember 2023

Moritz Leander Großfurtner

Danksagung

Ich möchte meinen herzlichen Dank an all diejenigen ausdrücken, die mich während meiner Arbeit an dieser Bachelorarbeit unterstützt und geleitet haben. Ohne ihre Hilfe wäre das alles nicht möglich gewesen.

Zuallererst möchte ich meinem Betreuer, Univ. Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer, sowie Dipl.-Ing. Christian Freude, meinen tiefsten Dank aussprechen. Deren Hilfe, Geduld und Ratschläge haben eine maßgebende Rolle bei der Gestaltung der Arbeit gespielt und mir geholfen, sie erfolgreich abzuschließen.

Auch bei der Fakultät der Technischen Universität Wien und allen Lehrern, die ich während meiner akademischen Laufbahn getroffen habe, möchte ich mich herzlich bedanken. Des Weiteren bin ich dankbar dafür, dass die Bildung, die ich erhalten habe, durch den Staat und die Gemeinschaft um mich herum erschwinglich und dadurch für mich und viele andere möglich gemacht wurde.

Weiters spreche ich meinen tiefen Dank auch meiner Familie aus, für all die Unterstützung und das Verständnis ohne welchem ich nie so weit gekommen wäre. Die Ermutigungen, so wie die schwankende Geduld und der Glaube an meine Fähigkeiten, sie abzuschließen, waren entscheidend, um mich bis zum Ende motiviert zu halten.

Ebenso einen großen Dank an meine Freunde und Mitstudierenden, deren moralische Unterstützung und anregende Diskussionen mich durch alle die Höhen und Tiefen der Arbeit begleitet haben.

Abschließend ist diese Arbeit nur möglich dank der Arbeit all derer, die vor mir kamen und auf der ich aufbauen konnte. Ich bin euch allen zutiefst dankbar.

Moritz Großfurtner

Acknowledgements

I would like to express my heartfelt gratitude to all those who have supported and guided me throughout the journey of completing this bachelor's thesis. Without their help, this endeavour would not have been possible.

First and foremost, I extend my deepest appreciation to my thesis advisor, Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer, as well as Dipl.-Ing. Christian Freude. Their invaluable guidance, patience, and insightful advice played a pivotal role in shaping the direction of this research and helped me finish it in the end.

I am also indebted to the faculty of the Technical University of Vienna, as well as all the guidance I received from all the teachers I encountered in my academic journey. For the fact that the education I've received was made affordable by the State and the community around me, I am deeply grateful.

Further, I extend my sincere thanks to my family for their love, support, and understanding throughout my academic journey. Their encouragement as well as their wavering patience and belief in my abilities to finish it were instrumental in keeping me motivated till the end.

I would like to acknowledge my friends and classmates who provided moral support, engaging discussions, and moments of joy during the ups and downs of this thesis. Your friendship has made this academic pursuit all the more enjoyable.

Lastly, this thesis would not have been possible without the foundation I was able to build upon, and I am deeply appreciative of each and every one of those who came before me.

Moritz Großfurtner

Kurzfassung

Diese Arbeit beschäftigt sich mit der Umsetzung eines Plugins für die Open-Source Game Engine Godot 3.5, mit dem Ziel einfach grafische Darstellungen von Pflanzen zu erzeugen. Konkret sollen dadurch Pflanzen aus dem Agroecosystem Projekt modelliert werden. Dafür werden die Daten aus Dateien, die einem vordefinierten Dateiformat entsprechen, geladen, welches die abstrakte Struktur der Pflanzen beschreibt. Nach dem Laden erzeugt das Plugin 3D Oberflächen, zum Darstellen der Zweige bzw. der Äste und Stämme und verwendet Instancing, um die Blätter effizient darzustellen. Eine der Kernfunktionen dabei ist die adaptive Anpassung des Detailgrads der Oberflächen anhand der Distanz der Kamera zu der jeweiligen Pflanze.

In der Umsetzung des Plugins wurde die GDPlugin Funktionalität Godots verwendet, um das Plugin möglichst reibungslos in Godot einzubauen. Das prozedurale Erstellen der Oberfläche wird mithilfe eines Algorithmus umgesetzt, welcher aus "Tree Skeletons"(Baum Skelette) die besagten Oberflächen erstellt. Durch einige Einschränkungen in Godot 3 war es nur möglich den Algorithmus auf der CPU und nicht auf der GPU umzusetzen. In Tests stellte sich heraus, dass Modelle, die aus dem Plugin statisch exportiert wurden, zu besserer Leistung führten, als Modelle mit adaptivem Detailgrad.

Abstract

This bachelor's thesis explores the development of a plugin for the open-source game engine Godot 3.5, aimed at providing an easy way for procedurally creating pleasing plant visualizations, specifically in the frame of the Sustainable Agroecosystem project. This is achieved by importing data conforming to a predefined format that abstractly describes the structure of plant organisms. Upon import, the plugin generates 3D surfaces for the branching structures and employs instancing for rendering leaves efficiently. One of the key features of the plugin is its adaptive surface subdivision mechanism, which dynamically generates the surface at different levels of detail based on the proximity to the camera.

The plugin's implementation leverages Godot's GDPlugin feature to seamlessly integrate into the engine's workflow. The procedural generation of plant structures is achieved through algorithmic processes that translate "tree skeletons" into 3D surfaces. However, due to limitations inherent in Godot 3, the adaptive subdivision mechanism is implemented on the CPU. In tests, this resulted in the following: Exports of models in the highest level of detail yielded better performance than models with adaptive subdivision.

The thesis covers the design, implementation, and theory behind the plugin. An evaluation of the plugin's functionality and performance is conducted, highlighting its capability to dynamically adapt the mesh at runtime. Performance comparisons between the adaptive subdivision approach and using the exported surface are presented, revealing the issues with the implementation on the CPU.

In conclusion, the developed plugin presents a novel approach to procedurally generate and render complex plant structures within the Godot 3.5 game engine. It extends the capabilities of the engine in creating realistic virtual environments while addressing the challenges of adaptive subdivision on the CPU. The thesis explores the intricacies of integrating such plugins into game engines and opens avenues for further optimizations.

Contents

CHAPTER 1

Introduction

In the field of computer graphics, the generation of plant models has been a topic of study since the early days. It poses various difficulties, like the high complexity of models, often with a large amount of triangles and possibilities such as self similarity, instancing for smaller plants which might aid in performance. All the while finding application in multiple industries. From the entertainment sector via movies and games, to serious fields, e.g. architecture, city planning or in the agriculture sector.

Sustainable Agroecosystems (SusAgro) falls into the last category, exploring how different environments and vegetation combinations interact with each other and influence the growth of plants. This thesis aims to implement a performant extension for the Godot game engine, to provide high fidelity visualizations for the simulations in the frame of that project, as well as publicly providing it as an open source extensions to enable the Godot community to use the same concepts for performant plants in 3D Games. Thus, this thesis falls into an overlap between the entertainment and serious sector.

To achieve something to that end, an open format was created together with SusAgro that can be adopted and generated by others should they want to.

The main contribution of this paper is an extension of Godot 3.5 for generating high fidelity geometry from vegetation data generated by the AgroSus Project. This is done by converting the AgroSus Data into the skeleton graph made up of chains based on the work of Pirk [Pir13]. Secondly an adaptation of Bloomenthal's approach [Blo85] for modelling plants from the skeleton is implemented, using Hanson & Ma's [HM95] Parallel Transport Frame algorithm instead of the Frenet Frames as suggested by Pirk as well as Runions et al. [RLP]. Lastly, the on the fly subdivision approach of Pirk is implemented using Godot 3.5's capabilities and evaluated.

1.1 Motivation

The data generated by simulations of SusAgro is an abstract description of the skeleton and rough shape of plants. In order to facilitate intuitive understanding of the data, this projects aims to provide adequate visualization of said abstract data. In addition to aiming to be of help in gaining understanding, this can also be useful in communicating the findings without the need of submersing oneself deeply in the material. Visualizing large amounts of geometry can significantly impact performance, in turn hindering the process of visualizing and engaging with the data. Therefore, the aim is to provide a performant visualization.

1.2 Godot

Godot is an open source game engine, that's been in publicly available since 2014 [Engb]. Since then, it's received many updates and is now at version 4. The thesis project uses version 3.5 as that was the newest version at the start of the project.

Game engines are tools that provide various functionalities to the developers using them. They are the interfaces that bring together the artefacts of the various departments involved in the development, from 2D or 3D art, Audio and the Code that handles the logic of the games and provide ready-made functionality that the developers can use, like physics engines, animation tools, and many more [BW20]. There are a multitude of game engines [TE19], each with various functionalities and focus, supporting different programming languages and target platforms [BW20].

1.2.1 Godot compared to other Engines

To provide a context for the area in which Godot operates, the two most common game engines for games released on steam are used for comparison, Unity Engine & Unreal Engine [Ste]. Like Unity & Unreal, Godot provides the tools to develop both 2D and 3D games/scenes, as well as providing a marketplace/a community where people can add plugins to enhance the functionality of the core engines [BW20].

While Godot is open source [Engb], Unity & Unreal are both proprietary software. They differ in various aspects as well, and one key aspect that is relevant for this work will be discussed here briefly. Most game engines take care of the rendering process so that it does not have to be implemented from the ground up by developers. However, since the needs for how the games should be rendered often differs from project to project, engines like Godot, Unity & Unreal expose some of the graphical API to the developers. Both Unity & Unreal allow developers to write their own shaders in High Level Shader Language (HLSL), while Godot provides its own shader Language based on GLSL. Further, they differ in what aspects of the rendering pipeline are exposed to the developers. This has also changed over the releases of different versions. With Unity introducing different rendering pipelines for different granularity of control as of 2018.1, and Godot upgrading the used OpenGL version from OpenGL ES 2 with Godot 3 to OpenGL ES3 in Godot

4 (though the main Focus of Godot 4 is the Vulkan based renderer) [Enga] [Uni]. The functionalities that are exposed determine what tools the developers have at hand to implement their desired solutions, something that will be discussed later on and was a limitation in the development of this project.

CHAPTER 2

Related Work

This chapter will showcase and introduce some of the developments and methods in the field of procedural generation of plants, as well as important existing tools.

The field of Computer Graphics has long been examining ways of (efficiently) bringing plants to the screen via models for scientific, industrial and entertainment purposes. Some of the earliest work in visually modelling plant life stems from Honda [Hon71]. Later on, the work of Lindenmayer & Prusinkiewicz [PL90] paved the way for many a method through their works with L-Systems, a rule based approach for describing the way plants grow. Both considered the process of plant generation as recursive structures which lend themselves well to the rule based generative approach, as noted Palubicki et al. [PHL+09] who based their work on the approach of Ulam [Ula62]. Ulam, in contrast, considered plant growth as a self organizing process where the patterns emerge, rather than being defined up front. Other works have explored ways to make the process of generating plant models more accessible. Such as the work of Deussen and Lintermann [LD99] who introduce a graph based workflow made up of different components describing structural and geometrical elements to generate the plant geometry.

In the following sections will explore various approaches of both the recursive and self-organizing kind. Afterwards the issue of creating the 3D surfaces and techniques for handling them will be discussed, as it poses its own set of challenges depending on the demands of the application of the models, from highly detailed to simplified.

2.1 Plant Generation

The process of generating plants for computer graphics can generally be split into two steps:

1. Creating the abstract description of the form of the plant (e.g. words in the case of L-Systems, more on that later)

2. The production of actual 3D graphical representations of plants. In most mainstream applications these will be 3D meshes, however they could also be renderings of point clouds (e.g. when visualizing 3D Scans)

Further the process of generating plants can generally speaking be categorized into reconstruction of real world data (static modelling) and procedural modelling (which can be further subdivided into various different categories) this thesis focuses on procedural methods.

2.2 Static (Data driven) modelling

Static, data driven, methods usually create representations from real world data like pictures or 3D scans of plants.

In the case of pictures a further processing step is needed to generate representation in 3D space, e.g. Lopez et al. [LDY10] use pictures of trees to create a 3D skeleton graph of the trees. These skeleton graphs can in turn be used to generate 3D surfaces. The concept of skeleton graphs will introduce and discussed later on in more detail.

3D scans can be visualized directly and convincingly given that the resolution is high enough, by directly visualizing the resulting point clouds. However, techniques for generating polygonal meshes were also developed, e.g. Xu et al. [XGC07] introduced a method that generates a tree skeleton from the point clouds and results in polygonal models with minimal user interaction.

In that same vein, Hu et al. [HLZ+17] tackled the issue of generating trees from airborne LiDAR point clouds which usually have few samples on tree branches making the geometry generation difficult. To remedy this, they segment the different trees using a normalized cut segmentation. And introduce a process of retrieving a natural tree skeleton via the addition of trunk points and refinement through the use of direction fields and angle constraints.

Many more techniques were developed in this field but, as this is not the focus of this thesis, will not be elaborated on further. Interested readers can find further methods mentioned in the works of Yi et al. [YLG+18] & Pirk et al. [Pir13].

2.3 L-Systems

One of the frontiers of procedural plant generation in computer graphics were Lindenmayer & Prusinkiewicz with their work on L-Systems. In their initial form, conceived by Lindenmayer in 1968, L-Systems were abstract descriptions of plant organs and their growth behaviours in development. At their core they are rewriting systems, rewriting

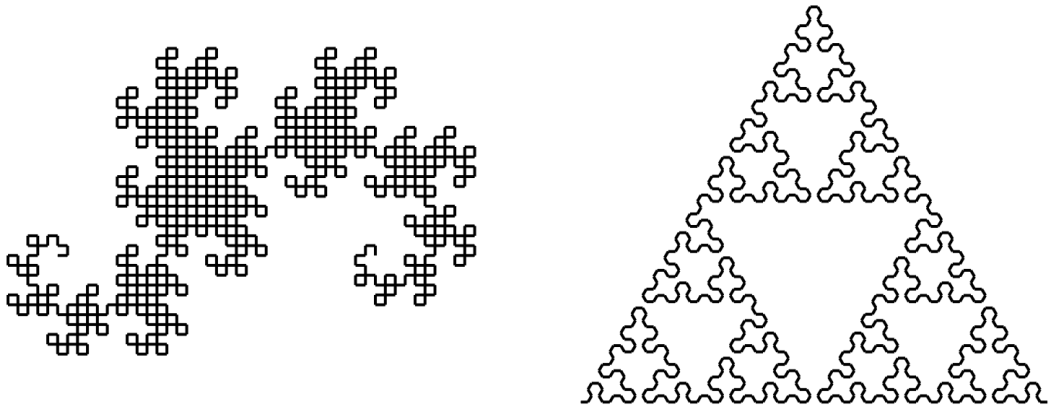


Figure 2.1: Example of curves generated by basic OL-Systems and visualized using Turtle interpretation [PL90, p. 11]

words to create new ones, similar to Chomsky's context free grammars, however the rewriting happens in parallel for each character of the word instead of sequentially [PL90, p. 1-3]. An L-system consists of an alphabet of characters, production rules which define rewriting for a given character, as well as a starting word called the Axiom [PL90, p. 4]. The characters can represent different organs of the plants, e.g. stem, flower or leaf.

2.3.1 Turtle Interpretation

To interpret the words produced by the L-Systems in a graphical manner, Prusinkiewicz & Lindenmayer [PL90, CH1, p. 6-7] use the turtle interpretation for drawing lines in 2D space by interpreting the symbols as instructions for a "turtle" moving through space see figure2.1 for an example. This approach is based on the ideas of diSessa and Abelson [dA]. Later on extending it to the 3rd dimension by introducing three vectors for defining the turtles heading [PL90, CH1, p. 18].

2.3.2 Tree like Structures

To allow for branched (tree-like) structures, adaptations to the original system were needed and the **Bracketed OL-System** was introduced [PL90, CH1, p. 24], which added "[" & "]" as means to create a stack like functionality for delimiting branches in an L-System word see figure 2.2 for a visualization.

2.3.3 Context Sensitivity

To account for the context which surrounds the individual organs in a plant context-sensitive (IL-)Systems were added to the toolbox of L-Systems, making it possible for rules to take into account the characters before and/or after the one examined by the production rule [PL90, CH1, p. 30-31], see figure 2.3. Context sensitivity within as well

Figure 2.2: Example of structures generated by bracketed OL-System and visualized using Turtle interpretation [PL90, p. 25]

as without the plant is also of vital focus in many later works that do not make use of L-Systems, e.g. the work of Palubicki et al. [PHL+09] views trees as self organizing organisms and takes into account the context of each bud to determine its fate.

2.3.4 Parametric L-Systems

The development of plants as well as its parts is made up of different processes, one of which is the elongation of its segments over the course of its life. This process can be described by growth equations [PL90, CH 1.9, p36-40] which can have different forms, and only some of them can be modelled or approximated by DOL-Systems or IL-Systems, thus leading to the introduction of Parametric L-Systems. Parametric L-Systems extend L-Systems through the addition of parameters of real numbers to letters, as well as parametric production rules that apply to them. These enable the modelling of the various growth functions a plant might have, from elongation to its branching angle [PL90, CH 1, p. 40-46]. To expand on this, later on a technique by Yi et al. [YLG+18] will be introduced, which uses growth equations to limit and model the growth of self-organizing

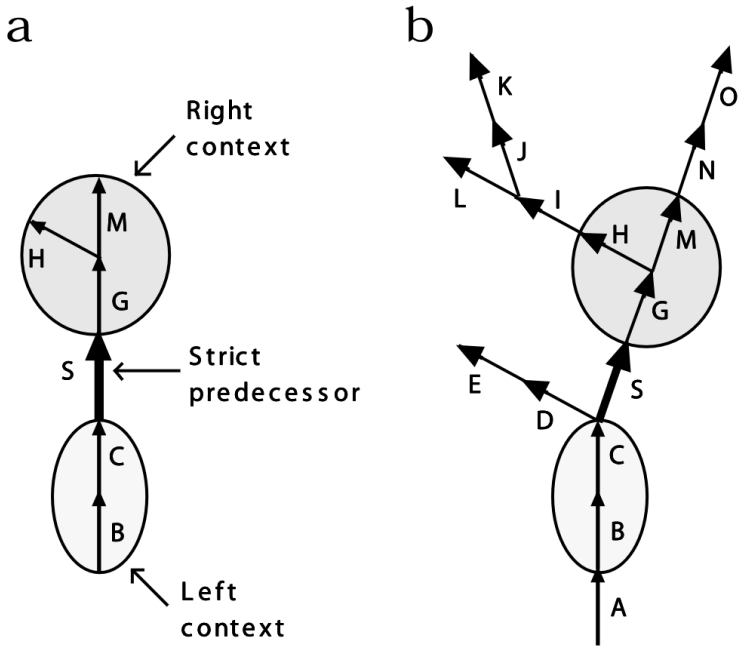


Figure 2.3: Matching in a context sensitive L-Systems [PL90, p. 31]

trees.

2.3.5 Methods built on L-Systems

As mentioned before, many methods were developed using the L-System approach, a few of which will be introduced in the next paragraphs.

Briefly touching on the topic of meshing which will be explored in more detail later on in section 2.5, Lluch et al. [LVM04] used L-Systems and the turtle interpretation to generate contours at each displacement of the turtle. These contours are saved into nodes with some additional information and later used to generate the geometry. This is similar to the approach of using **Coordinate Frames** to generate generalized cylinders, which are also used in this thesis as well as by Pirk [Pir13] based on Bloomenthal's [Blo85] work, which will be explored in more detail in section 2.5.

Manipulation of L-Systems & Improving the usability of plant modelling

As L-Systems can get quite complex, the manipulation of them can become difficult for non-experts. Further, even small changes require the whole System to be re-run, as well as the result having to be interpreted again. To enable modellers to dynamically make changes to the plants generated by L-Systems, Rynkiewicz [RN16] introduced an approach that allows users to make changes to the 3D model without the need to change the underlying system. To achieve this, they parse the string produced by the L-System

to create a skeleton of the tree, consisting of separate Bézier curves for each branch. The geometry is then generated by using points on the Bézier curve as centre points for a cylindrical mesh. Again an approach similar to the one of Bloomenthal [Blo85] which will be expanded upon in section 2.5. The ease of manipulation is then achieved by exposing handles for the Bézier curves, enabling the adaptation of each branch segment individually.

There have been many other techniques developed for improving the usability of plant modelling, e.g. Sun et al. [SJJ09] exposed few explicitly named parameters to increase the ease of generating plants without the need for deep understanding of the functionality of L-Systems. While Ijiri et al. [IOI06], a few years earlier, allowed users to control the development of L-Systems by providing them with an interface to **a)** modify the generating rules responsible for the local structures and **b)** draw a line controlling the axis and depth along which the fractal structure will grow. However, as they hold little relevance to this thesis, this area won't be expanded upon further.

Tackling the problem of data size with L-Systems

Meshes of plant's can become highly detailed, posing challenges for both the transmission of data over the network and the rendering of such detailed geometry.

To tackle the issue of displaying the geometry, multiresolution is often used to reduce the complexity of the objects. To that end different Levels of Detail (LOD) are created where each level has the same model at a different resolution. Various geometry based simplification methods exist, however, as Lluch et al. [LCV03] argue, they may fail to simplify the geometry of trees properly, as they do not take into account the structure of the tree. To remedy this issue, they propose generating the LOD not based on the geometry, but on the structure of the tree using parametric L-Systems. In this "procedural multiresolution" approach, they create a "weighted tree" which holds the modules (branches) weighted by how much they contribute to visual structure. They achieve this by analysing the structure of the trees for various quantifiable features like length or texture and using those for the creation of the weighted tree. The result is a structure that holds the different LOD's as the branches from most to least important, and can thus be used to efficiently render the trees. In addition to efficient rendering, they also propose using this for the progressive transmission.

In comparison, Jaeger et al. [JSJC10] make use of the inherent self similarity of rule based generators to reduce the amount of data needed to encode a plant. They observe that at a given cycle of generating a model, there will be similar patterns. These similarities are then used by referencing them instead of explicitly saving each instance. Key differences of the patterns like branching and phyllotaxy angles do not need to be stored as they can be reproduced from the physiological age (branching order). Using this technique they are able to reproduce the plants later on, requiring less storage, which consequently requires less bandwidth. Further, it is possible to generate a variety of models by using different ages in the generation. However, a caveat of this technique is, that it will only

work with rule based generators (and not for example with self organizing techniques) due to required self similarity and reliance on the physiological age as parameter for the development. In addition, it is also not possible to take into account changes caused by the environment or interaction, like e.g. pruning by a user.

Parallel L-System processing

Lipp et al. [LWW] in turn make use of the structure inherent to L-Systems to create highly parallel algorithms for both the derivation and the interpretation of arbitrary L-Systems.

To achieve this in the derivation they first analyse the modules for their output length and resulting offsets in parallel, before doing the actual rewriting in parallel, using the length and offset calculated in the previous steps. For the production, they noticed that most of the turtle states & commands can be represented as matrices and leverage the associativity of matrix multiplications by accumulating them in parallel before combining them in a separate pass. Finally, they found push & pop commands used for branching can easily split the work into two separate threads.

By employing these strategies they showed, that for L-Systems which grow quickly their algorithms are superior to a highly optimized Single CPU implementation. Further, they argue that the advantages of a GPU based solution become even more pronounced when taking into account CPU-GPU transfer times.

2.4 Self organizing plants

Arguing that the repetitive character diminishes in more mature trees, Runions et al. [RLP] take an approach that views plants as self organizing organisms instead of recursive structures pioneered by Ulam [Ula62]. Runions et al. "Space Colonisation Algorithm" is based on a previous work by Runions et al. [RFL$^+$05] for generating leaf veins in 2D space. At its core, the algorithm works by iteratively adding new nodes onto the existing tree structure, growing the plant step by step. The growth of these new elements is guided by points marking the available space. To populate the space with these points, a three-dimensional shape is used as input. The shape is then filled with *attraction points* that guide the growth of new nodes. They signify the existence of empty (available) space and are removed when they are within a defined distance of a branch (kill distance) signifying that the space is no longer available, see figure 2.4 for a step by step visualization of the algorithm. The algorithm terminates when all attraction points have been erased or there are no more nodes in the reach of the remaining attraction points [RLP, p 2-3]. After the termination, the resulting structure of nodes is processed further, redundant nodes are removed and the remaining once relocated to keep the structure of the skeleton graph. Additionally, the graph can be smoothed out further by creating curve subdivisions [RLP, p 2]. The resulting skeleton graph akin to methods described later on in section 2.5 is then used for the creation of the 3D model of the plant [RLP, p 2].

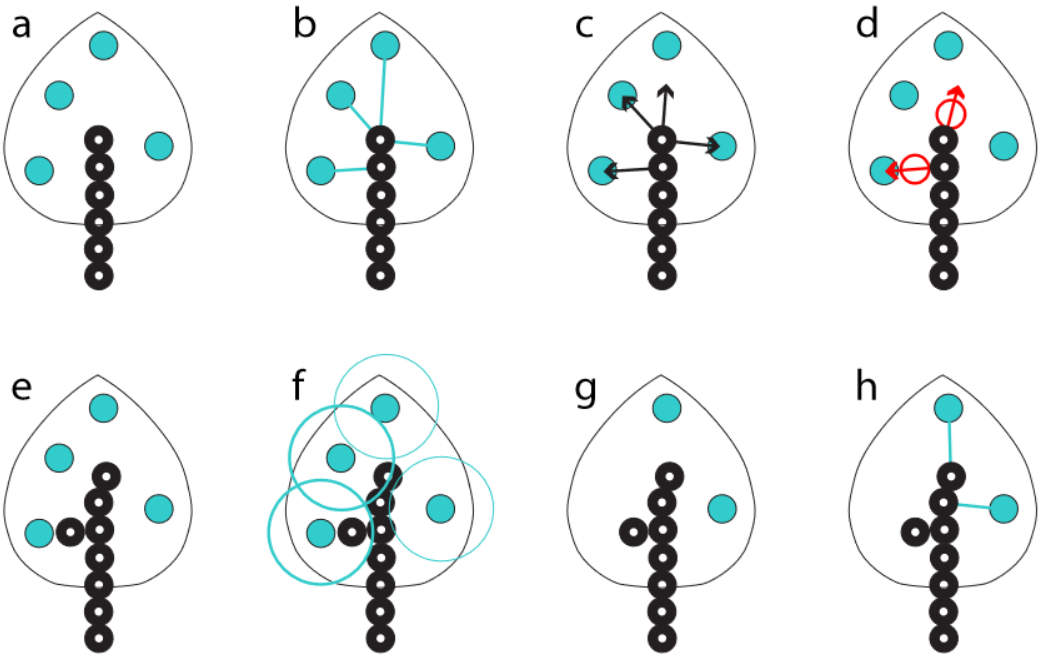


Figure 2.4: Step by step visualization of the Space colonization algorithm [RLP, p. 3]

This novel method was adapted soon after by Palubicki et al. [PHL+09] in 2009 expanding on the concept of self organizing plants by doing two main things:

1. They introduce the internal allocation of resources to determine the fate of buds. Proposing two different models: the extended Borchert-Honda (BH) model (considering each branching point one at a time) and the priority model (taking into account entire axes).

2. In addition to competition for space, they also take into account the competition for light through an approximation using a voxel grid to propagate the shadows cast by buds and branches.

Expanding on this idea, Yi et al. [YLG+15] achieve finer control over tree species properties by integrating the BH as well as the priority model, utilizing their different approaches of allocation for different aspects of the modelling.

As mentioned previously in Sect 2.3.4 plants growth patterns can be described by growth functions. Yi et al. [YLG+18] noted this lack of realism important for applications like agriculture, forestry and co, in the modelling via self-organizing plants and addressed that issue in their 2019 paper. They propose integrating growth modelling by constraining the resources allocated in each iteration of the algorithm through a growth equation, see

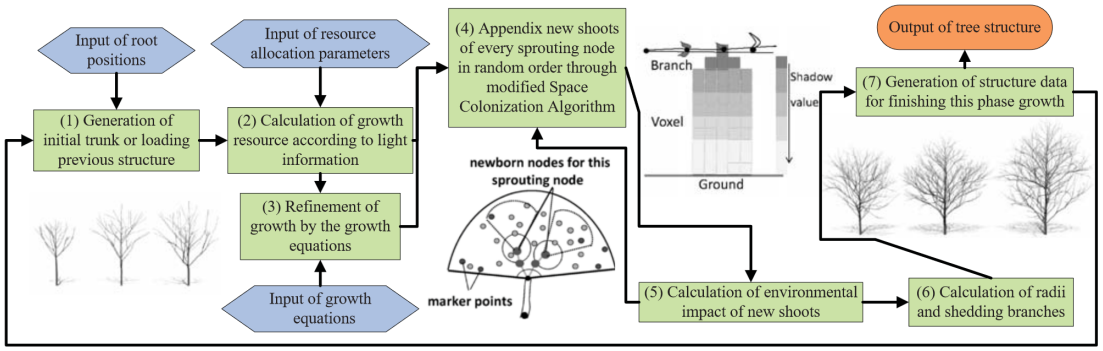


Figure 2.5: Pipeline of modeling trees using growth equations [YLG+18, p. 3]

figure 2.5. This approach not only leads to more realistic growth speed and thus models at the different stages of development, it also has the advantage of limiting the height a plant can grow to, consistent with the behaviour of real plants.

2.5 Meshing

As mentioned in the beginning of section 2.1, most (procedural) plant generation methods can be split into two processes. This section will discuss the second step of this process: Generating a visual representation of plants in the form of a 3D mesh.

2.5.1 Core of Meshing

One of the seminal methods for generating a mesh for a branching plant, in that case a tree, stems from Bloomenthal [Blo85], who represents the tree's branching pattern as a skeleton graph consisting of points and their connections. To produce smooth models of the bending branches, cubic splines that go through the data points are used. These splines then form the axes for the generalized cylinders that are used to create the surface of the tree. To produce the surface (mesh) from the splines, cross-sections of varying radii are taken along the curves. With the *axial resolution* describing the number of cross-sections taken along the curves, while the *circumferential resolution* describes the number of points on each cross-section see figure 2.6. To prevent unwanted twisting continuous Frenet Frames are calculated at each cross-section, consisting of three vectors, the *Tangent, Normal* & *Binormal*. The Frenet Frames (or more specifically their normal and binormal) form the orientational basis for the vertices generated at each cross-section. The vertices created this way are then connected to from the surface mesh and the ramiform, the places where branching occurs, are modelled using free form-surfaces.

Many of the techniques that followed in the area of tree modelling have since used it or very similar techniques for generating their meshes, e.g. Sun et al. [SJJ09] mention using it, while Rynkiewicz & Napieralski [RN16] simply describe their modelling approach as

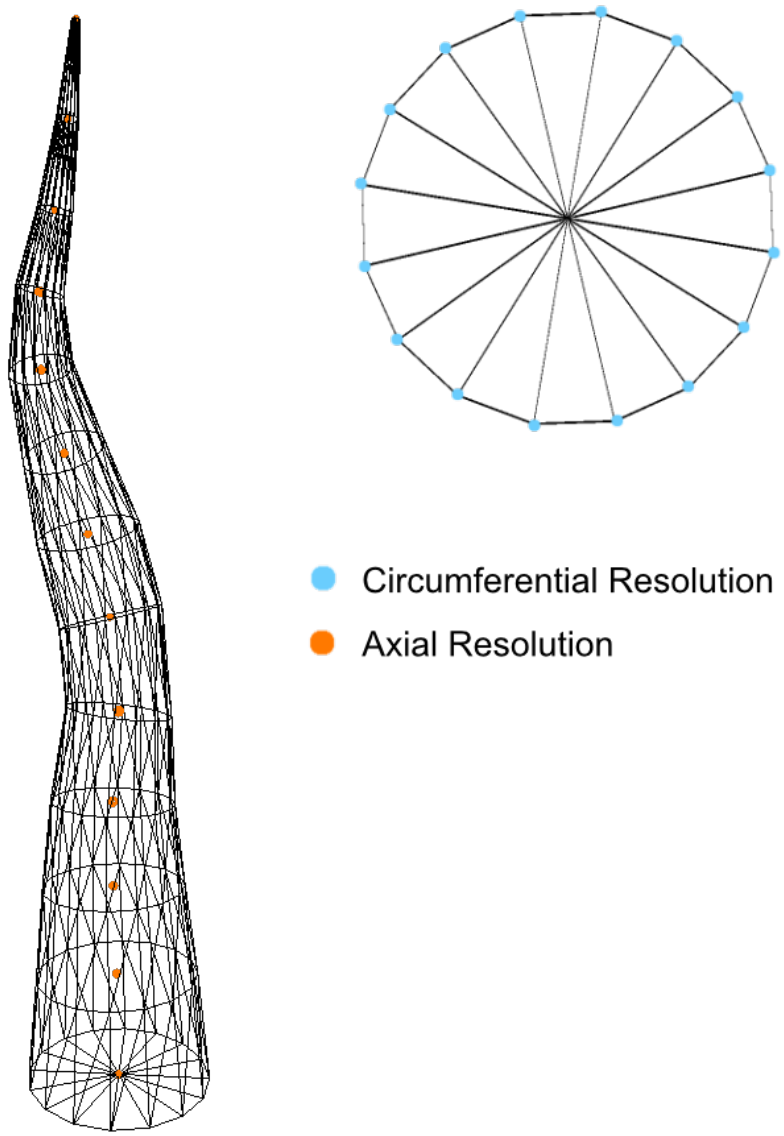


Figure 2.6: Circumferential resolution visualized as dots on the circle in cyan and axial resolution visualized as orange dots in the center of the branch.

points on Bézier curves being the centre of cylindrical meshes, using normal, binormal and tangent for calculating the bending.

Meanwhile, both Runions et al. [RLP] & Pirk [Pir13, Ch.6.1.2] use the approach while improving on the Frenet Frames by utilizing "*parallel transport frames*" (PTF) (see Hanson & Ma [HM95], which eliminate the issue of not being able to compute Frenet Frames in points where the acceleration is zero.

Further, Pirk et al. [Pir13, Ch.3] proposed a new approach for the structure in which to represent the model of a tree. Making use of the "Human Visual Systems" limitations, they divide the model into two distinct entities: the main branching structure as skeleton graph and the overall shape of the foliage as leaf clusters. This abstract representation reduces the memory footprint (especially for multiple trees of the same species) for both storage and transmission. The clusters are textured/filled in the modelling step with branch patches of a predefined library, thus allowing for the use of LOD as well as instancing as these smaller branches don't have to be unique models [Pir13, Ch.3].

To aid in the real time rendering, Pirk [Pir13, Ch. 6] also discusses multiple techniques to be applied for efficient processing and rendering of trees. First of, it is pointed out, that the handling of the surface mesh on the CPU is costly and thus proposed to instead use a skeletal graph on the CPU while handling the generation of the mesh on the GPU. To that end, they put forth the following process:

1. They create *Chains*, sequences of edges for each branch. A chain then directly relates to the space curves needed to generate the surface mesh over a whole branch. (These chains will be discussed in more detail late on in 3.5)
2. The PTF algorithm is applied onto each chain, producing smooth frames along the entire branches.
3. Refining the skeleton graph consisting of the *chains* on the GPU in the "control" & "evaluation" shader stages.
4. Generating a generalized cylinder along each refined branch (chain) in the "geometry" shader stage.

This approach allows for refining the mesh based on the distance of the camera on the fly being highly efficient on the GPU while also allowing for interaction with the tree e.g. through wind by applying these forces on the coarse skeleton graph.

2.5.2 Improving the Mesh

Previously, it was already noted, that many techniques create the geometry for the individual branches as generalized cylinders. However, they often do not utilize the ramiform free-form surface, e.g. in [RLP] and not minding the ramiform might produce visual artefacts or inconsistencies as noted by Lluch et al. [LVM04]. To that end, Lluch

et al. [LVM04] introduced a technique (mentioned in section 2.3.5) for creating a single polygonal mesh over the entirety of a tree. It works by first producing the contours at the start and end of each segment in a tree and its branches. These contours are then stored via nodes in a linked data structure akin to the cross-sections created by Bloomenthal's [Blo85] method. Based on this data structure (and the contours within) the geometry is generated. In contrast to the other methods, however, the geometry is not produced by linking the contours together directly. Instead, the junctions are refined to allow for the generation of a single mesh. This is achieved by

1. Subdividing the segments between the parent and its child contours into further contours.
2. Analysing these contours for intersections.
3. If there is an intersection between contours
   a) They are combined to a single contour
   b) Then saved into the data structure containing the original contours [LVM04, CH 3.3].
4. With the structure produced by this refinement process it is then possible to generate a continuous single geometry for the entire tree.

2.5.3 Foliage

Previously the generation of the surface for the branching structure of plants has been discussed however plants have another important part, that has as of yet only briefly been addressed, the matter of foliage. Many a method technique for displaying the leaves has been developed, Reeves & Blau [RB85] used particle systems to render both the branching structure and the foliage. Bloomenthal [Blo85] modelled the individual leaves, using three polygons, bending them at their creases. Others, such as Miao et al. [MZGL13] developed techniques for detailed modelling of leaves. Pirk [Pir13] create detailed representations using textured quads that can dynamically refined using Non-Uniform rational B-Splines (NURBS). The NURBS describe the surface of a leave, thus making it possible to create varying degrees of detail by increasing and decreasing the sampling used to generate the vertices based on the distance to the camera. Though commonly, leaves are generalized as groups of texture quads (Billboard Clouds) [Pir13, Ch. 6.1.5, p. 101].

Many more techniques as well as different implementations of the ones introduced here exist, but introducing them all is outside the scope of this thesis.

CHAPTER 3

Theory and Method

This chapter will explain base concepts and elaborate on the methods that were employed in this thesis.

3.1 Definitions

3.1.1 Tree Skeleton

Bloomthal [Blo85] describes the tree skeleton as "Any representation of the branching pattern". In this thesis, "tree skeleton" & "skeleton graph" will be used interchangeably to describe the abstract description of the branching structure as a graph consisting of linked nodes, each node representing a segment of a plant. Akin to what Pirk [Pir13, Ch. 3] term the "skeletal graph". This graph holds all the important information on the general branching structure of a tree, like branching angles and relation of branches to one another, but can be used to hold other important information like branch radius, woodiness as well.

3.1.2 Coordinate Frames

A coordinate frame is given by a set of three orthogonal axes attached to a body, in our case these axes are given by 3D Vectors, and are used to describe positions relative to the body. With the origin of the coordinate frame being the point where the three axes meet. In the case of this thesis, the three axes will be given by the **tangent, normal & binormal** of a point on a space curve. This is based on the work of Hanson & Ma [HM95], whose parallel transport frame algorithm, which will be described later on, is used for moving a coordinate frame along the space curve to generate a smoothly connected mesh.

3.2 Generalized Cylinder

Agni [Agi72] define generalized cylinders as a space curve that represents the axis and a number of cross-sections along said axis. These cross-sections can be of varying form (e.g. Bloomthal [Blo85] uses disks of varying radii).

3.3 Spatial Curves

Spatial curves are an important part of modern 3D graphics. They can be used for generating surfaces, e.g. Bloomthal's [Blo85] modelling of the branches, as well as being used for paths along which objects travel [Len12, Ch.11, p.317].

Some of the most commonly used curves are *cubic curves* due to being both flexible as relatively simple. Their parametric representations can conveniently be written in the form of a matrix product:

$$Q(t) = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

or

$$\boldsymbol{Q}(t) = \boldsymbol{C}\boldsymbol{T}(t)$$

where t is a point on the curve $\boldsymbol{C}$ is the matrix of coefficients and $\boldsymbol{T}(t)$ the vector holding the variable and the degrees of the (cubic)polynomial function [Len12, CH 11, p.317-318].

The coefficients of the classes of cubic curves discussed in this chapter can further be split into two important components, $\boldsymbol{G}$ the **geometrical constraints** ($\boldsymbol{g}_1, \boldsymbol{g}_2, \boldsymbol{g}_3, \boldsymbol{g}_4$) and their **blending functions** (basis matrix $\boldsymbol{M}$), both of which can be represented using separate matrices. This leaves us with

$$\boldsymbol{Q}(t) = \boldsymbol{G}\boldsymbol{M}\boldsymbol{T}(t)$$

where

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{g}_1 & \boldsymbol{g}_2 & \boldsymbol{g}_3 & \boldsymbol{g}_4 \end{bmatrix} = \begin{bmatrix} (\boldsymbol{g}_1)_x & (\boldsymbol{g}_2)_x & (\boldsymbol{g}_3)_x & (\boldsymbol{g}_4)_x \\ (\boldsymbol{g}_1)_y & (\boldsymbol{g}_2)_y & (\boldsymbol{g}_3)_y & (\boldsymbol{g}_4)_y \\ (\boldsymbol{g}_1)_z & (\boldsymbol{g}_2)_z & (\boldsymbol{g}_3)_z & (\boldsymbol{g}_4)_z \end{bmatrix}$$

and

$$\boldsymbol{M} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{bmatrix}$$

with a_i, b_i, c_i, d_i being the coefficients of the blending functions for the geometric constraints at the given point on the curve t [Len12] CH 11, p.318-319.

3.3.1 Hermite Curves

The first class of curves that will be introduced here are cubic **Hermit Curves**. They are defined by the start- & endpoint ($\boldsymbol{P}_1, \boldsymbol{P}_2$) as well as their corresponding tangents ($\boldsymbol{T}_1, \boldsymbol{T}_2$), providing the **geometric constraints** $\boldsymbol{G}_h$ for this type of curve. The blending function matrix is given by $\boldsymbol{M}_H$ [Len12, CH11.2, p.320].

$$\boldsymbol{G}_H = \begin{bmatrix} \boldsymbol{P}_1 & \boldsymbol{P}_2 & \boldsymbol{T}_1 & \boldsymbol{T}_2 \end{bmatrix}$$

$$\boldsymbol{M}_H = \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

3.3.2 Bézier Curves

The next class of curves are the Bézier curves, which can easily be translated to and from Hermit Curves. While cubic Bézier curves are also defined through their start & endpoint, instead of tangents they are further defined by two interior points. The four points are called control points and give the **geometric constraints** $\boldsymbol{G}_B$. While the curve always runs through the first and last control point, the interior control points are approached by the curve, but the curve does not necessarily run through them. The blending functions are given by the basis matrix $\boldsymbol{M}_H$ [Len12, CH11.3, p.322-325].

$$\boldsymbol{G}_B = \begin{bmatrix} \boldsymbol{P}_0 & \boldsymbol{P}_1 & \boldsymbol{P}_2 & \boldsymbol{P}_3 \end{bmatrix}$$

$$\boldsymbol{M}_H = \begin{bmatrix} -3 & 6 & -3 \\ 3 & -12 & 9 \\ 0 & 6 & -9 \\ 0 & 0 & 3 \end{bmatrix}$$

3.3.3 Splines

The concept of splines stems from the work of French draftsmen, bending, combining and anchoring thin strips of wood together to create smooth, curves between specified points. Similarly, the mathematical concept of splines is used to describe curves that are hard to model directly. Curve segments are stitched together to create more complex curves. Notably, the splines are permitted to have certain (dis-)continuities at the junction points [ANW67,][BBB87]. The (dis-)continuities are described as parametric continuity C^n given if the n-th derivative of the curve segments in the junction point, as well as geometric continuity G^n if the n-th derivative is non-zero and points in the same direction [Len12, Ch.11.1, p.318]. These levels of continuity are of various importance, depending on the application, e.g. in vector graphics sudden changes in direction (C^0 but not C^1) might be wanted while in others like e.g. smooth animation it might not.

In the creation of splines there are generally trade-offs that have to be made, for that some more concepts will be introduced:

- Interpolation vs. Approximation [Len12, Ch.11.3, p.322]
  - A control point is *interpolated* if the curve runs through the point
  - A control point is approximated if it approaches the point
- Local vs. Global control [Len12, Ch.11.5, p.331]
  - Local control is given if changing a geometric constraints effects only the corresponding curve segment and its immediate neighbours
  - Global control means that a change to the geometric constraint of one segment leads to a change in the whole curve

Hermite Spline

A hermit spline is a spline made up of segments of Hermite curves. As the first derivative gives the tangent of a function at a given point, it follows, that a Hermite Spline has C^1 in its junction points if the tangents are chosen to be equal at the junction points for subsequent curve segments. Similarly, C^0 is given if the positions of the end & start point of subsequent segments is the same. For C^2 that unfortunately isn't the case. Further constraints could be applied, but that would result in a lack of local control. A spline that does that is the natural cubic spline, however as stated before at the cost of local control [Len12, Ch.11.5, p.331-334].

A brief note one Bézier Splines: as noted in section 3.3.2, Bézier curves can easily be translated to Hermit curves and vice versa. Thus, it follows that Bézier Splines can exhibit the same continuities with the proper constraint.

B-Spline

The B-Spline (Basis-Spline) achieves C^2 continuity over all junction points by constraining the blending functions to possess said C^2 continuity, while also allowing for local control. It achieves this, however, at the cost of not interpolating any of the control points. This interpolation can be achieved with non-uniform B-Splines, however that once again comes at the cost of continuity at the points that are interpolated. Simply put, the interpolation can be acquired by subsequently repeating the control points. Though each repetition results in the loss of one degree of continuity at that point [Len12, Ch.1.6-1.6.3, p.334-345].

3.4 Algorithm(s)

3.4.1 Parallel Transport Frame

The coordinate frames defined in section 3.1.2 are the base of a cylinder at a given point along a space curve. To achieve a smoothly connected mesh without twisting between consecutive frames, it's necessary to compute continuous frames. A common approach to this problem is the Frenet Frame. However, the Frenet Frame, requires the curve to have a non-vanishing second derivative [HM95], [Pir13, Ch.6.1.2, p.97-98].

Hanson & Ma [HM95] introduce the concept of the *parallel transport frame* (PTF), based on the insight that it's possible to transport a frame bye parallel transporting each component of the frame.

While in its basic form the PTF does, in general, not return to its initial value in closed curves [HM95] this is a non issue for the application in this thesis as the tree graph doesn't have circles.

Hanson & Ma [HM95] provide an algorithm (see algorithm 3.1) for calculating the PTF for a given set of tangent vectors, which was adapted and implemented in this thesis.

3.5 Data Structure

The structure for holding the data of the plants needs to encompass all the information required for the accurate construction of their surfaces. To achieve this, an approach heavily based on Pirk's [Pir13] as well as the seminal work of Bloomenthal [Blo85] was chosen. As the basis for generating the mesh using space curves, the skeletal structure of the tree is represented as segments, with each segment holding the following information in table 3.1. It should be noted that the structure could easily be expanded to hold other information as well, such as e.g. the woodiness or water content, which can then be used for differences in rendering or in simulating e.g. bending from wind. This has however been left out for the sake of the scope of this thesis.

These segments are then processed to form *chains* [Pir13, Ch.6.1.3, p.99] for each branch. A chain is a sequence of segments linked together such that each segment has only a

Algorithm 3.1: The PTF algorithm introduced by Hanson & Ma [HM95]

Input: (1) A list of unit tangent Vectors $\{\hat{T}_i\}$, $i = 0, \ldots, N$;
(2) An initial normal Vector $\vec{V}_0$, $\vec{V}_0 \perp \hat{T}_0$
Output: A list of parallel-transported normal vectors $\vec{V}_i, i = 1, \ldots, N$, $\vec{V}_i \perp \hat{T}_i$

for $i \leftarrow 0$ **to** $N - 1$ **do**
  $\vec{B} \leftarrow \hat{T}_i \times \hat{T}_{i+1}$
  if $\|\vec{B}\| = 0$ **then**
    $\vec{V}_{i+1} \leftarrow \vec{V}_i$
  end
  else
    $\vec{B} \leftarrow \vec{B}/\|\vec{B}\|$;
    $\theta \leftarrow \arccos(\hat{T}_i \cdot \hat{T}_{i+1})$; // $0 \leq \theta \leq \pi$
    $\vec{V}_{i+1} \leftarrow R(\hat{B}, \theta) * \vec{V}_i$; // Rotate the Normal Vector by angle θ about $\hat{B}$
  end
end

Name	Description
Position	Holds the position of the segment
Direction	Holds the growth direction of the segment
Length	Length of the segment
Radius	Holds the initial radius of the segment
Children	Links to all child segments
Peripherals	List of the peripherals (foliage) attached to the segment

Table 3.1: Table explaining the information encoded in the data structure.

single ancestor. Pirk use the *Gravelius Order* [Pir13, Ch.5.2.2, p.78-79] to determine these chains representing the main trunk & its side branches based on differences in the angle of subsequent segments as well as expending on the idea and additionally taking into account the length & thickness [Pir13, Ch.6.1.3, p.100].

CHAPTER 4

Implementation

This section will first describe the different ways available to extend Godot's capabilities, as well as Godot's limitations in that process. Afterwards it will go into the whole pipeline from importing the Data from Agro Godot, to transforming the data and generating skinned and textured Meshes.

4.1 Godot

An introduction to Godot was given in 1.2 this section here describes how to extend Godot's functionality.

4.1.1 Extending Godot

There are multiple ways in which new functionalities can be added to Godot. These will be discussed here, moving from close to the engine's source code to farther away, and can be divided into two categories. Changes that require recompilation of the engine and changes that don't.

Recompiling the Engine

On the side of the engine we have, of course, *directly modifying the core* of the engine. This is the most powerful as theoretically everything can be changed, added or modified, it is however also the most intrusive and most prone to breaking when e.g. a new version is released and incompatible changes are introduced.

Modules The next level are *modules* written, like the engine, in C++. They allow for adding new functionality to the engine, as the name suggests, in a modular way and without the need to modify the core directly [Goda].

This allows for e.g. binding to external libraries or adding entirely new functionality. Being written in C++ it also allows for highly performant code and manual memory management.

No recompilation

GDNative Still written in C++ (or C) but no longer in need of recompiling the engine, Godot offers *GDNative* to extend its functionality [Gode]. Though on that note, Godot introduced the successor for GDNative available from Godot version 4: *GDExtentions* which replaces GDNative fulfilling the same purpose, but with fewer downsides and more tightly integrated into the engine [Engc]. However, as this thesis is implemented in Godot 3, GDExtentions won't be expanded upon here.

Along with the advantage of not requiring the recompilation of the entire engine, GDNative offers some others [Gode]:

- It can be used equally in both the editor and the final application, whereas C++ modules require you to recompile the export templates.
- Stemming from the fact that the engine doesn't need to be recompiled, GDNative is easier to distribute and share with others.

It does however also have some disadvantages compared to C++ modules [Gode]:

- The access of GDNative is limited to the access the scripting API exposes.
- It has limitations in regard to platform support as it's not available on Universal Windows Platform and has only limited support for HTML5.
- In case the code requires lots of communication through the scripting API, it can be slower than the modules which do not have that overhead.

An additional downside of GDNative in comparison to the class of extensions introduced next is, that it has strict version requirements, only working for the exact minor version it was compiled for.

Editor Plugins Editor Plugins provide a way to create tools that expand Godot's capabilities, using *GDScript* or C# as well as Godot's UI and Scenes. While less powerful than the previously introduced options, they still are highly useful and their tight integration with Godot's Scenes and consequently UI makes them great for creating tools inside the Godot editor that expand Godot's base functionality [Godc].

4.1.2 Graphics API Limitations

Godot 3.5, the version used for this thesis, offers two renderers: *OpenGL ES 3.0* (OpenGL 3.3 on desktop) which they recommend for desktop and *OpenGL ES 2.0* which they recommend for mobile [Godb]. However, while the *geometry shader* stage is supported as of OpenGL 3.2 [Geo] it is not available in Godot, and while initially planned for Godot 4 these plans were later discarded [Gita], [Gitb].

Due to that the implementation of the tessellation of the models was done entirely on the CPU, instead of the GPU using the geometry shader, like [Pir13, Ch.6.1.4, p.101] suggested. Similarly, the tessellation stage isn't supported, in this case because it's only in OpenGL core since version 4.0 [Tes].

4.1.3 Built in Functionality

As mentioned in 2.5.3 rendering large amounts of leaves can be quite expensive and while various techniques have been developed, this thesis uses Godot's built in MultiMesh to achieve instancing to reduce the amount of draw calls [Godd].

4.2 Plant Data

4.2.1 File Type

In its initial stages, the JSON file format was used for its readability. Each node represented a start and/or endpoint of a segment of a plant, with each node holding the following information:

- ID - Unique identifier of that node used for linking nodes together.
- Position - Vector 3 holding the position of the node
- Radius - Float holding the radius of the branch at that point
- Rotation - Vector 3 holding the orientation of the branch at that point
- Children - A list holding the IDs of all the child nodes of the current node, allowing for branching as well as a continuous branch.
- Branch - ID of the branch the current node belongs to in order to generate a continuous mesh over the whole branch.

To reduce the file size, the decision was felled to change the format from JSON to binary, as well as using the edges of the tree skeleton instead of the vertices. This results in a slight loss of information (as the radius is uniform along an entire segment), but is deemed acceptable for the reduction in size. It is also very similar to the way Pirk et al. [Pir13, Ch.5.3.1, p.80] defined their data structure.

In its final form, the plugin is able to read binary data from Agro Godot holding the following information:

- Version number of the file
- Number of plants in the file
- Plants consisting of:
  - Number of Organs in the plant
  - 3 Types of Organs (Bud, Stem and Leaf) with:
    - Common: ParentId, OrganType, OrganId, Energy Level, Water Level
    - Bud: Position
    - Stem: Length, Radius, Coordinate frame at centre of stem
    - Leaf: Coordinate at centre of leaf

The files holding this data need to end in the `.prim` file extension.

4.2.2 Conversion into local Data

Upon reading the data from the file, it is converted into an *RtPlant* a data structure holding the skeleton graph of a plant implemented as a class in `C#`. The graph is represented by a list of *branches* also implemented as `C#` class holding its child branches as well as the connected edges based on the concept of *chains* introduced in section 2.5.1. Each edge in the branch can additionally hold its corresponding leaves, while the buds, as well as the energy & water level are ignored for the purpose of this thesis. Further, the data structure of an edge once it's loaded differs from the one saved as a file. It consists of a start & end node holding the Coordinate Frame & radius at the position. This is needed to achieve the C_1 continuity discussed in section 3.3.3. Upon converting the edges from the file data, the start node of edge a is made the end of its predecessor b. This ensures that the position and tangent are equal at each junction point, granting C_0 and C_1 continuity.

Since the binary format no longer holds the information of which branch an edge belongs to, the *chains* have to be determined from the given data. As mentioned earlier, Pirk [Pir13, Ch.6.1.3, p.100] uses an approach based of *Gravelius Order* to determine the main trunk and its side arms. In this thesis a very simplified approach is taken, and the chains are determined solely on the length, aiming for the maximum length of any chain by iterating over all segments from outer to innermost, recursively maximising for the longest chain.

As a branch holds the segments in a list, it would be easy to convert it to an array holding all the data necessary to do the calculations for the mesh generation on the GPU.

Previous ideas such as implementing it as objects linked together by references to each other or a dictionary holding all the edges in a dictionary would not have that advantage and were therefore discarded.

To enable the loading of the files, the plugin written in this Thesis introduced a new Godot node called *PlantLoader*. This node provides a field called *Plant File Path* which users can click to select their desired file. Upon selection, the file is imported and the plant(s) in the file are loaded. Changing the selected file causes the new plants to be loaded automatically, removing the previously loaded ones.

The node also exposes other properties as well that influence the look of the plant:

- **Plant Scale** - float number - Used for scaling the plants loaded up and down.
- **Texture Repeat Factor** - float number - Enables scaling of the texture along the mesh of the branches, as the required scaling may differ from plant to plant. A higher number corresponds to repeating the texture more often.
- **Branch Material** - Godot Material - Defines the material used for the mesh of the branches.
- **Leaf Mesh** - Godot Mesh Instance - The Mesh to be used for the leaves that will be instanced (more on that later in section 4.3.3.
- **Subdivisions** - The subdivisions multiplier, influencing how detailed the plants can get.

4.3 Mesh Generation

As mentioned previously, Godot supports neither the *geometry* nor the *tessellation* shader stages, thus the generation of the mesh as well as the subdivision happens entirely on the CPU, using `C#`. This section explains how the mesh generation and the Level of Detail was implemented.

After loading and converting plants into instances of *RtPlant*, as explained in the previous sections, the mesh is generated for the editor view.

The way the mesh is generated is the same in the editor and at runtime, however the Level of Detail depends on the distance to the camera at runtime, and therefore the mesh isn't continually updated or regenerated in the editor. Instead, it's updated when any of the values exposed by `PlantLoader.cs` are changed.

The next paragraphs will describe the process of generating the surface mesh.

4.3.1 Process

The plugin generates continuous surfaces over entire branches, but not over entire plants. Meaning each branch is its own individual surfaces. The generation process is propagated from the bottom up (trunk to outer branches) and thus could be stopped at a certain branch level if the need arose.

The amount of subdivisions is calculated for each branch. In the current implementation, this is based on the length of the branch as well as the distance to the camera for LOD, but could be easily replaced by a different mechanism.

For each segment of a branch (given by the edge), a cubic Bézier curve is generated and the points along it are sampled x times, where x is the number of subdivisions. These points are saved as *GeometryNodes* in a list, with each node holding the ***CoordinateFrame*** (*orientation*, *position*) & *radius*.

However, while the list created this way holds the correct positions, tangents and radii, the Frames are not yet continuous. To achieve this, the PTF algorithm introduced in 3.4.1 is run over the list of geometry nodes, producing a list of continuous geometry nodes.

This new list is then the used to generate *CrossSections* at each geometry node, disks represented by x vertices where x is the number of radial subdivisions. The radial subdivisions are based on the overall subdivisions of the branch.

The *CrossSections* are linked together and used to create one continuous mesh over the whole branch, akin to the process by Bloomenthal [Blo85] described in 2.5.1.

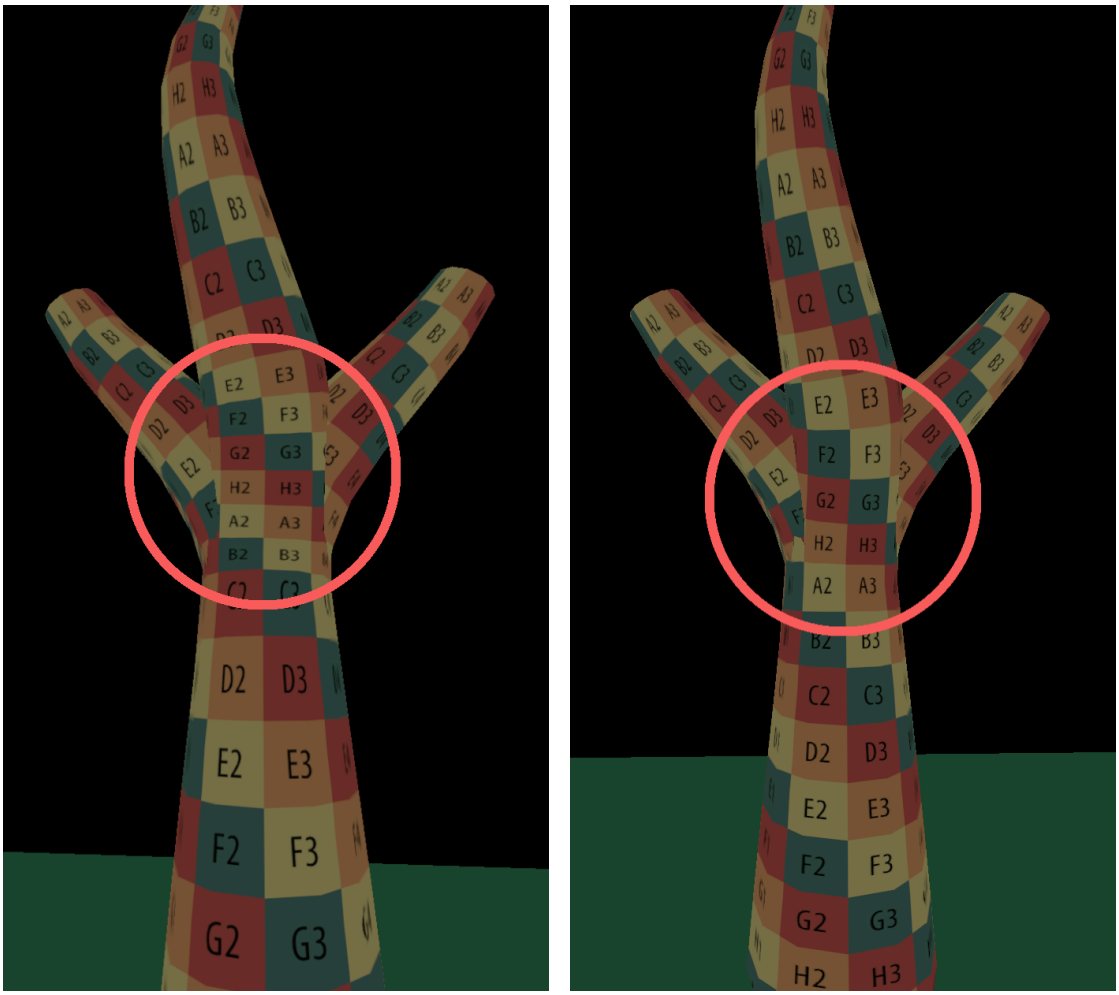
4.3.2 Texturing/UV Mapping process

To achieve a continuous texture, the V-Coordinate is set based on the position of the current node compared to the length of the whole branch. The difference of this versus uniform V-Coordinates can be seen in figure 4.1.

Further, the scale of the Texture is applied and the length of the branch factored in. One issue that arose in the texturing process, however, was that in longer branches with great difference in the start & end radii the textures ended up having highly stretched or compressed textures towards the ends, depending on what texture scale was chosen see figure 4.2. This stems from the fact that the same square image is projected onto surfaces of different sizes and is a current limitation of our approach.

4.3.3 Foliage Generation

For the foliage (the leaves) of the plants a simple instancing approach was chosen. Each *RtPlant* has a wrapper of a *MultiMesh* instance. Upon mesh generation, each branch that has leaves adds the position and scale of the leaves to a list in the wrapper of the *MultiMesh* which then renders leaves using Godot's *MultiMeshInstance*.



(a) Uniform UV (b) Relative UV

Figure 4.1: Difference between uniform distribution of the V-Coordinate irregardless of its position along the branch vs. relative to the position along the branch.

Figure 4.2: Difference in texture stretching based on the radius difference along the branch.

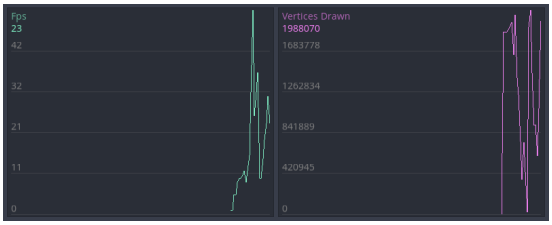
CHAPTER 5

Results

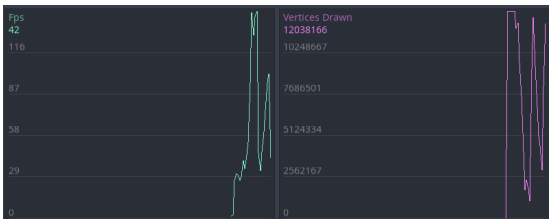
5.1 Evaluation

The performance was evaluated by moving a camera along a predefined track through a scene populated with plants. This was done once with the mesh being generated on the fly, adaptive to the distance to the camera with a *subdivision factor of 16* and *5 distinct LOD* steps. The second time, the meshes were generated and exported from the plugin, using the *minimal LOD step* (with the highest subdivisions) and the *subdivision factor of 16* as well to justify direct comparison. The plants used were trees with foliage and medium height plants, seen in figure 5.2. The results of these measurements can be seen in table 5.1. The stats measured were the frames per second (FPS) and the vertices drawn each frame using Godot's debugger monitor feature and Godot's debugger profiler was used for the frame time, specifically the idle time. The idle time in Godot's profiler is the time spent in the *Process* function in scripts (called every frame). The device used for the measurements was the *Lenovo Legion Y540-15IRH*, with an *Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz* CPU, *16GB* RAM and a *NVIDIA GeForce GTX 1660 Ti* GPU connected to an external monitor allowing for up to 144 FPS.

As can be clearly seen, the adaptive method has major drops in FPS, even going below 11 FPS at times, while the same scene with meshes at the subdivisions of the maximum LOD step stays above 30 FPS for almost the entire time, even reaching frame rates above 116 FPS at times. Further, the average FPS show the discrepancy even more clearly, with the adaptive method averaging out at about 23 FPS over the whole run time, while using the meshes resulted in an average of 42 FPS. In the profiler, we see that the worst frame for the adaptive method was frame #1 with an idle time of *234.66 ms* and an overall time of *234.78 ms* for the frame. In comparison, the worst frame using just the meshes was #1004 with an idle time of just *36.35 ms* and *36.41 ms* total for the frame. This clearly shows that the adaptive mesh method implemented in `C#` on the CPU for Godot is not usable for the purpose of improving the performance. However, as can be seen



(a) Monitor for the scene with adaptive subdivisions



(b) Monitor for the scene with static meshes

(c) Profiler for the scene with adaptive subdivisions

(d) Profiler for the scene with static meshes

Figure 5.1: Comparison of the performance between the adaptive subdivisions and the exported meshes for the scene. In the profiler the frame that took the longest was selected to show the difference in the worst case.

in the monitors, the adaptive meshing method does achieve a quite drastic reduction of vertices drawn. Again the average was taken from the monitors and the adaptive method came in at an average of 1988070, while the pure meshes came in at 12038166. About 6 times more vertices. Thus, it could arguably improve performance, if it would be possible to reduce the time it takes to change the mesh, e.g. by performing the calculations on the GPU and their corresponding shader stages.

5.2 Showcase

Texture from `https://www.texturecan.com/` were used for the ground & stem/trunk textures and the leaf texture are from `https://opengameart.org/` all under the CC0 licence.

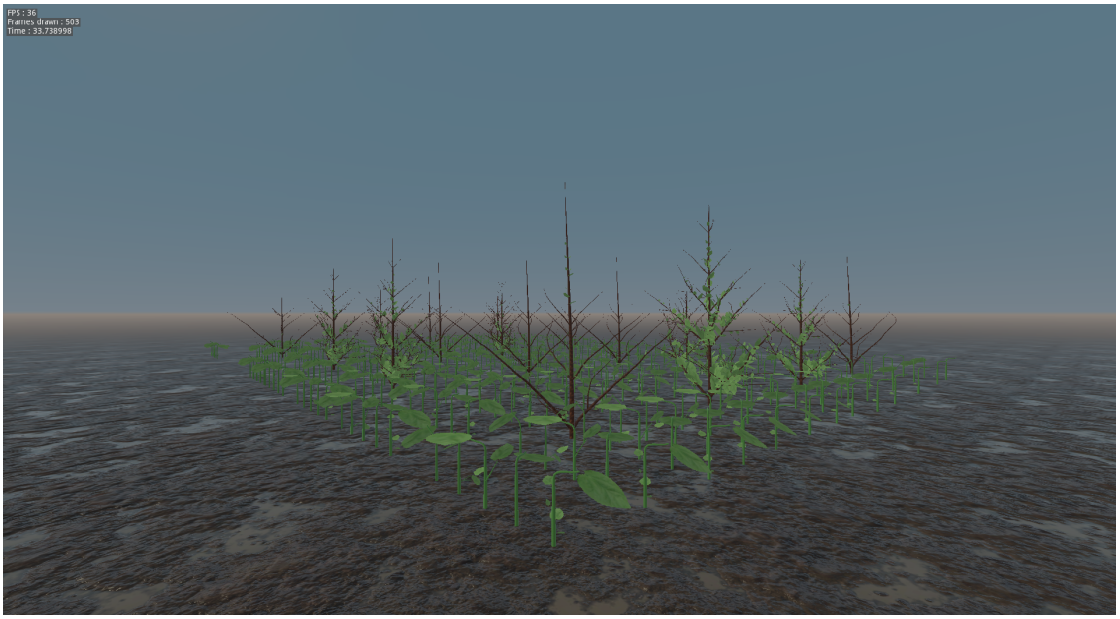


Figure 5.2: The scene used for testing the performance with a subdivision factor of 16.

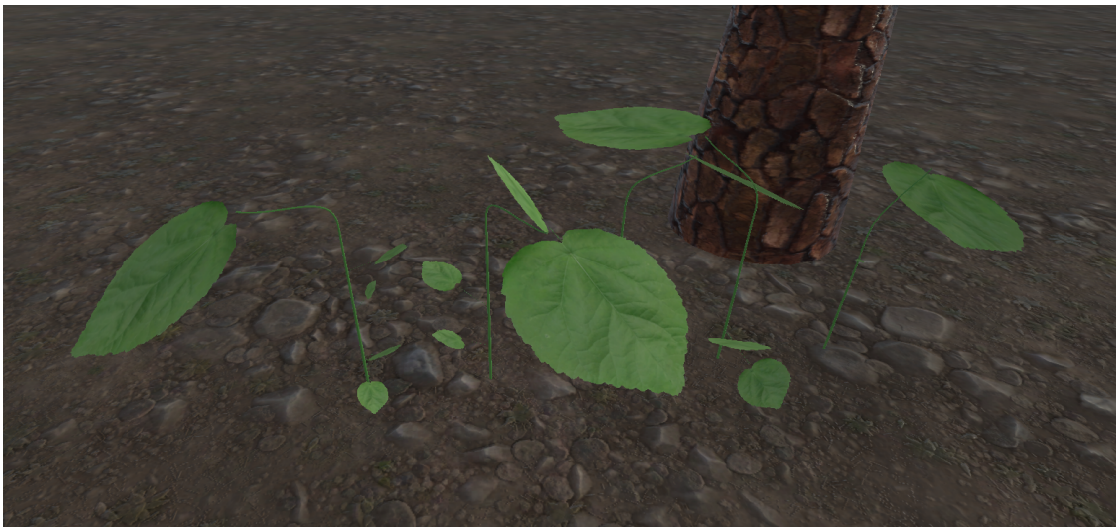


Figure 5.3: Some medium sized plants with a subdivision factor of 24.

Figure 5.4: Full view of a tree with lots of foliage from the Agroecosystem project modelled using the plugin.

Figure 5.5: Full view of a scrawny tree with little foliage from the Agroecosystem project modelled using the plugin.

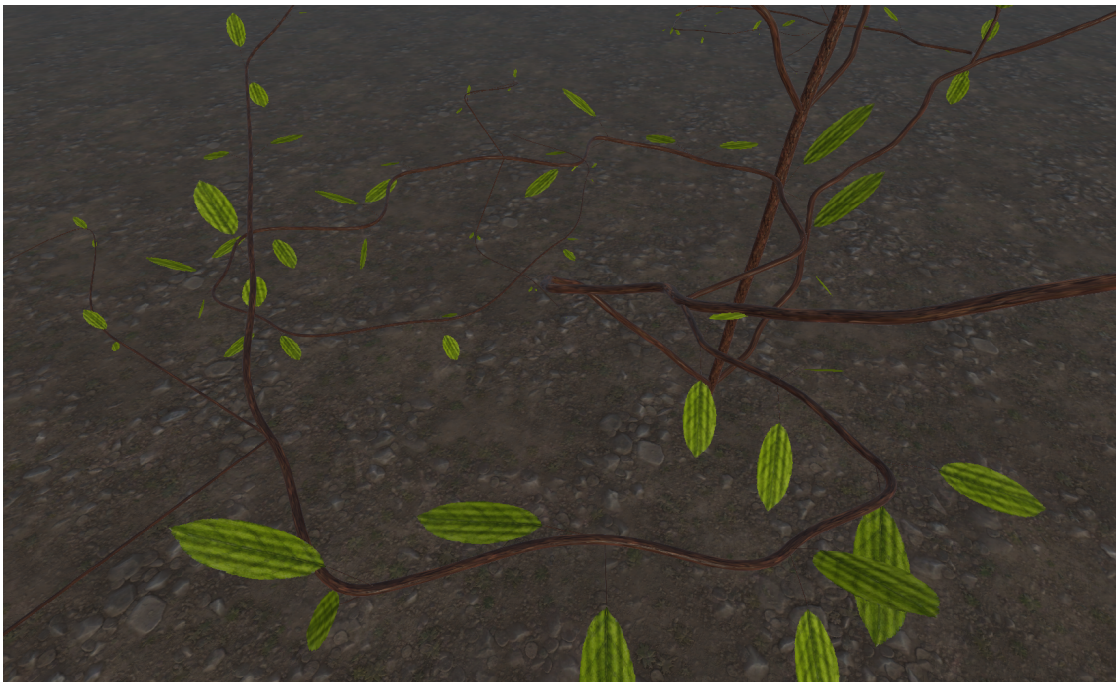


Figure 5.6: Close up of a scrawny tree with little foliage from the Agroecosystem project modelled using the plugin.

CHAPTER 6

Conclusion and Future Work

This work offers many opportunities for improvements in various areas, and this section will discuss some of those possible areas and provide a brief summary of the thesis at the end.

6.1 Conclusion

This thesis makes two main contributions. First, the plugin for Godot 3.5 which allows the visualization of data from the AgroSus project. Providing high fidelity geometry by converting the data to the chain's data-structure introduced by Pirk et al. [Pir13] to generate the geometry. Second, the evaluation of on the fly subdivision using Pirk et al.'s strategy in Godot 3.5, which is shown to not be usable as it is limited to the CPU and therefore slower than static geometry.

In its current form, the plugin created during this thesis is useful for creating meshes in the desired resolution from the imported data, providing an easy way to create pleasing visualizations. However, it does fail at providing efficient adaptive meshes during runtime. This stems from the fact that Godot doesn't expose certain stages of the render pipeline, which would be required in order to process the plants on the GPU. This results in the adaptive subdivision being done entirely on the CPU, which, as mentioned, fails at being efficient enough to be viable. Still, this thesis shows that it is possible to drastically reduce the amount of vertices that need to be drawn, which shows that this might be a viable approach in future Godot versions that do expose enough of the GPU capabilities to handle the adaptive generation on the GPU.

6.2 Performance

The likely most prominent area for improvement is that of performance. While it seems this is not possible in Godot 3 without changes to the core of the engine, it might be

possible to achieve major improvements in Godot 4. One possible avenue to explore would be the generation of the meshes using compute shaders and the *RenderingDevices* available in Godot 4. In the same vein, integrating the tree-skeleton plus leaf clusters approach introduced in 2.5.1 could result in further improvements to the performance, and it might be possible to implement them using compute shaders as well.

6.3 Texturing

Another possible area of improvement would be the area of texturing. For one, the stretching issues of textures issue mentioned in 4.3.2. Further, it might be interesting to generate the texture procedurally as well. This would make it possible to take into account further information on the state of branches such as woodiness or hydration to change the look of the individual branches.

6.4 Meshing

The area of meshing also leaves many possibilities for improvements, from integrating Lluch et al.'s' [LVM04] approach for creating a single continuous mesh, to enabling making the branches and trunk more gnarly by using shapes other than circles for the cross-sections.

6.5 Animation

Being able to efficiently generate the mesh from the skeleton graph on the fly would allow for efficient animation of vegetation by applying the forces to the tree graph, as described by [Pir13, CH 6, p.97] e.g. the forces of wind, or in case of a game it might be possible that the player characters traversal of a tree might lead to bending of the branches. Similarly, since SusAgro simulates the growth of vegetation, it might be interesting to skip the export and import step and instead directly visualize the data generated by it.

6.6 Interactability

Finally, enabling interaction with the tree-skeleton like e.g. pruning might be an interesting area for extension as well.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Agi72] Gerald J. Agin. Representation and Description of Curved Objects. Technical report, October 1972.

[ANW67] J.H. AHLBERG, E.N. NILSON, and J.L. WALSH. The Theory of Splines and Their Applications. In *Mathematics in Science and Engineering*, volume 38, pages 1–8. Elsevier, 1967.

[BBB87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to the Use of Splines in Computer Graphics*. 1987.

[Blo85] Jules Bloomenthal. Modeling the mighty maple. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 305–311, New York, NY, USA, July 1985. Association for Computing Machinery.

[BW20] Andrzej Barczak and Hubert Woźniak. Comparative Study on Game Engines. *Studia Informatica*, (23):5–24, December 2020.

[dA] Andrea diSessa and Harold Abelson. Turtle Geometry. https://mitpress.mit.edu/9780262510370/turtle-geometry/.

[Enga] Godot Engine. About Godot 4, Vulkan, GLES3 and GLES2. https://godotengine.org/article/about-godot4-vulkan-gles3-and-gles2/.

[Engb] Godot Engine. First public release! https://godotengine.org/article/first-public-release/.

[Engc] Godot Engine. Godot - Introducing GDNative's successor, GDExtension. https://godotengine.org/article/introducing-gd-extensions/.

[Geo] Geometry Shader - OpenGL Wiki. https://www.khronos.org/opengl/wiki/Geometry_Shader.

[Gita] GitHub. Issue - Add Support for Geometry Shaders · Issue #10817 · godotengine/godot. https://github.com/godotengine/godot/issues/10817.

[Gitb] GitHub. PR Geometry shader support by Chaosus · Pull Request #28237 · godotengine/godot. https://github.com/godotengine/godot/pull/28237.

[Goda] Godot - Custom modules in C++. https://docs.godotengine.org/en/3.6/development/cpp/dev

[Godb] Godot - List of features. https://docs.godotengine.org/en/3.6/about/about/list_of_features.h

[Godc] Godot - Making plugins. https://docs.godotengine.org/en/3.6/tutorials/plugins/editor/tutoria

[Godd] Godot - MultiMesh. https://docs.godotengine.org/en/3.6/classes/classes/class_multimesh.htr

[Gode] Godot - What is GDNative? https://docs.godotengine.org/en/3.6/tutorials/scripting/gdnative

[HLZ+17] Shaojun Hu, Zhengrong Li, Zhiyi Zhang, Dongjian He, and Michael Wimmer. Efficient Tree Modeling from Airborne LiDAR Point Clouds. 2017.

[HM95] A. Hanson and Hui Ma. Parallel Transport Approach to Curve Framing. 1995.

[Hon71] H. Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the sample of the tree-like body. *Journal of Theoretical Biology*, 31(2):331–338, May 1971.

[IOI06] Takashi Ijiri, Shigeru Owada, and Takeo Igarashi. The Sketch L-System: Global Control of Tree Modeling Using Free-Form Strokes. In *In Smart Graphics*, volume 4073, pages 138–146, July 2006.

[JSJC10] M. Jaeger, R. Sun, J. Jia, and V. Chevalier. EFFICIENT VIRTUAL PLANT DATA STRUCTURE FOR VISUALIZATION AND ANIMATION. *undefined*, 2010.

[LCV03] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. In *Proceedings of the 2nd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '03, pages 31–38, New York, NY, USA, February 2003. Association for Computing Machinery.

[LD99] Bernd Lintermann and Oliver Deussen. Interactive Modeling of Plants. *Computer Graphics and Applications, IEEE*, 19:56–65, February 1999.

[LDY10] Luis Lopez, Yuanyuan Ding, and Jingyi Yu. Modeling Complex Unfoliaged Trees from a Sparse Set of Images. *Comput. Graph. Forum*, 29:2075–2082, September 2010.

[Len12] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Course Technology, Cengage Learning, Boston, MA, 3rd ed edition, 2012.

[LVM04] J. Lluch, R. Vivó, and C. Monserrat. Modelling tree structures using a single polygonal mesh. *Graphical Models*, 66(2):89–101, March 2004.

[LWW] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel Generation of L-Systems.

[MZGL13] Teng Miao, ChunJiang Zhao, XinYu Guo, and ShengLian Lu. A framework for plant leaf modeling and shading. *Mathematical and Computer Modelling*, 58(3):710–718, August 2013.

[PHL+09] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics*, 28(3):58:1–58:10, July 2009.

[Pir13] S. Pirk. Efficient Processing of Plant Life in Computer Graphics. 2013.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer, New York, NY [u.a.], 1990.

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 313–322, New York, NY, USA, July 1985. Association for Computing Machinery.

[RFL+05] Adam Runions, Martin Fuhrer, Brendan Lane, Pavol Federl, Anne-Gaëlle Rolland-Lagan, and Przemyslaw Prusinkiewicz. Modeling and visualization of leaf venation patterns. *ACM Trans. Graph.*, 24:702–711, July 2005.

[RLP] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling Trees with a Space Colonization Algorithm.

[RN16] Filip Rynkiewicz and Piotr Napieralski. Procedural fractal plants generation. page 12, December 2016.

[SJJ09] R. Sun, J. Jia, and M. Jaeger. Intelligent tree modeling based on L-system. *2009 IEEE 10th International Conference on Computer-Aided Industrial Design & Conceptual Design*, 2009.

[Ste] SteamDB Technologies. https://steamdb.info/tech/.

[TE19] Marcus Toftedahl and H. Engström. A Taxonomy of Game Engines and the Tools that Drive the Industry. In *DiGRA Conference*, 2019.

[Tes] Tessellation - OpenGL Wiki. https://www.khronos.org/opengl/wiki/Tessellation.

[Ula62] Stanislaw Ulam. On some mathematical problems connected with patterns of growth in figures. In *Mathematical Problems in the Biological Sciences*, pages 215–224. 1962.

[Uni] Unity. Scriptable Render Pipeline Overview. https://blog.unity.com/technology/srp-overview.

[XGC07] Hui Xu, Nathan Gossett, and Baoquan Chen. Knowledge and heuristic-based modeling of laser-scanned trees. *ACM Transactions on Graphics*, 26(4):19, October 2007.

[YLG+15] Lei Yi, Hongjun Li, Jianwei Guo, Oliver Deussen, and Xiaopeng Zhang. Light-Guided Tree Modeling of Diverse Biomorphs. *Pacific Graphics Short Papers*, page 5 pages, 2015.

[YLG+18] Lei Yi, Hongjun Li, Jianwei Guo, Oliver Deussen, and Xiaopeng Zhang. Tree Growth Modelling Constrained by Growth Equations. *Computer Graphics Forum*, 37(1):239–253, 2018.