Tree Reconstruction from a Point Cloud using an L-System

Synthesis Project 2021

Students		Supervisors
D. Dobson	5152739	L. Nan - TU Delft
H. Dong	5302501	S. Du - TU Delft
N. van der Horst	4697952	D. Voets - Cobra
L. Langhorst	4299922	
J. van der Vaart	4450752	
Z. Wu	5360684	

TU Delft

Abstract

Storing accurate models of complex geometries in a compact way has become an increasingly challenging issue, especially when dealing with large datasets. One of such datasets is Cobra-Groeninzicht's database of all trees in the Netherlands. In the gaming industry, a new technique is being used to generate tree models: the L-system. An L-system stores a string representation of the structural model of a tree, with the added possibility for recursive modelling using growing rules. This format proves a promising alternative to more traditional methods of storing complex geometries. However, it remains unclear whether it can be an accurate enough representation for modelling and analysing real-life trees.

In this research project, the AdTree algorithm is used to reconstruct a skeleton from a point cloud of a single tree. This skeleton is then transformed to an L-System string format, as well as a CityJSON format (both in JSON structure). The L-system format comes with the advantage that it allows for several methods of increasing its compactness further (growing, generalisation). The overall size of these files also indicates fewer storage space is needed to store the tree geometry. The quality of the L-System skeleton is nearly equal to the input, the skeleton generated by. Assuming it can be read and drawn using a Turtle program, the L-system thus allows for storing the same geometric information more compactly than traditional storage formats, with sufficient accuracy, and the added possibilities of growing or generalising the model.

Delft, June 2021

Acknowledgements

We would like to dedicate this page to the people who made this project possible and guided it into success. We are grateful and stand on the shoulders of Liangliang Nan and Shenglan Du from Delft University of Technology, whose work we have built upon. It was an interesting adventure, and Liangliang Nan and Shenglan were there every step of the way with deep insights and extensive support. We are proud to be given the opportunity to work on such an interesting topic, and to be able to contribute to the scientific domain of Geomatics. We also very much appreciate the idea, the data and help given to us by Dirk Voets and Leonardo Mauri from Cobra-Groeninzicht. The whole project would not have existed without them. We hope this project will grant them insights that are valuable in the domain of forestry management.

Delft, June 2021

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem statement and objective . . . 1
1.3 Research questions . . . 2
1.4 Project objectives . . . 2
1.5 Organisation of the paper . . . 3

2 Related work 4
2.1 3D tree modeling from point clouds . . . 4
2.1.1 AdTree . . . 4
2.2 Compact representations of 3D tree structures . . . 6
2.2.1 L-system . . . 6
2.2.2 Turtle (graphics) . . . 6
2.2.3 Compactness . . . 7
2.2.4 File encoding . . . 8

3 Methodology 9
3.1 Pipeline . . . 9
3.1.1 Point cloud. . . . 9
3.1.2 Skeleton extraction (graph) with AdTree . . . 10
3.1.3 The L-system format . . . 10
3.1.4 Creating Mesh . . . 11
3.1.5 Generalization . . . 11
3.1.6 Tree growing . . . 11
3.2 L-System . . . 11
3.2.1 L-System JSON format . . . 11
3.2.2 L-system initialisation . . . 12
3.2.3 Obtain nesting . . . 13
3.2.4 Compute relation between nodes . . . 14
3.2.5 Write to L-string . . . 16
3.2.6 L-System to AdTree . . . 17
3.3 Extensions . . . 21
3.3.1 Generalisation . . . 21
3.3.2 Growing . . . 22
3.4 CityJSON semi-explicit storage format . . . 24
3.5 Classic skeleton explicit storage format . . . 26

4 Results and discussion 27
4.1 Assessment . . . 27
4.2 Effect of point cloud density . . . 28
4.3 Robustness to data sources . . . 30
4.4 Differences in encoded geometry: AdTree output and the L-system. . . . 32
4.5 Differences in encoded geometry: AdTree output and the CityJSON . . . 38
4.6 Branch tip generalisation . . . 40
4.7 Simulated growth function L-system . . . 44
4.8 Effect of file formats on storage efficiency. . . . 47

5 Conclusion 48

Bibliography 50

1

Introduction

1.1. Motivation

Trees are one of the most essential elements of the Earth's landscape, both in nature and urban areas. Trees provide us with many benefits including social, aesthetic, climatic, ecological and economic benefits [25]. However, trees can also cause harm by falling down on a person or home, get plagued by insects or diseases and therefore need to be monitored, maintained and controlled [7]. In support of monitoring trees and forests, up-to-date forest inventories are needed to measure the extent, quantity, and condition of forest resources [11]. The resulting information provides a base for making management decisions at operational and strategic level, such as harvest planning and forest protection [27]. Often, data is acquired, then stored in a digital spatial database, and processed in a Geographical Information System (GIS).

In this project, research is performed in conjunction with Cobra-Groeninzicht, a company that is responsible for monitoring around 100 million trees throughout the Netherlands, Flanders and North Rhine Westphalia. Their forest inventory, Treemonitor (*Dutch: Bomenmonitor*) [7], is largely built upon point clouds, mostly acquired from airborne remotely sensed (sparse) LiDAR data (AHN3) and some custom (high density) LiDAR data from Prorail. AHN(3) LiDAR data, that comes in the form of a point cloud, is collected and processed as an initiative of the Dutch government. It is (free) open data and is updated at least yearly [21]. Hence, it is part of the vision of this project to create a pipeline that could automatically extract tree representations from a dataset of this quality (sparse and incomplete point clouds). Although, considering the feasibility of this research, better quality data (dense and complete point clouds) is used primarily, from deciduous trees without foliage.

1.2. Problem statement and objective

Working with AHN3 LiDAR data for building a forest inventory comes with a few challenges, including but not limited to: classification, segmentation, visualisation and storage. The prior work "AdTree" [3] could be built upon, and in this research the focus lies mainly on the storage issue. In essence, AdTree reconstructs the 3D geometry of a tree from a point cloud input. It extracts the tree's skeleton in the form of a graph, and reconstructs the 3D geometry of that graph with a cylinder fitting process [3]. The skeleton (graph) is available as export to an ASCII format (e.g. .ply) and the corresponding 3D reconstruction as a mesh (e.g. .obj). Although they store an accurate, detailed and automatically modelled tree, both formats have low interoperability in the forestry monitoring domain, and point clouds and meshes are inefficient storage-wise for the inventorization and visualisation of a hundred million trees.

For forestry maintenance and control, data needs to be at least regularly updated. AHN3 data was acquired between 2014 and 2019, and its successor AHN4 is also flown in parts over the course of years [24]. Trees continue to grow yearly, and a data discrepancy of up to five years makes apt forestry management cumbersome. This is where the L-system approach comes in. The L-system is chosen as they are extensively studied in the botanical field, may be extended to create flexible geometric models of plants, and are fit to simulate growth and external forces (e.g. light, gravity, resources) [2]. In its most basic application, the L-system models the structure of a tree as nested set of relative movements between nodes. From this structure, the complex ge-

ometry of the model can be retrieved. It may be extended with the capacity to describe the growth of a plant in a recursive manner, by means of a fractal, therefore describing the geometric model even more compactly. The L-system is an established method to model trees in the gaming industry, pioneered as early as 1999 [16] and commercialised by IDV Inc. in early 2000's under the name SpeedTree [10]. SpeedTree is still the state-of-the-art today, and famous for creating the trees in James Cameron's movie *Avatar*[18]. L-systems are considered an emerging technology for tree simulation models in the forestry management domain. Micro-climate model maker ENVI-met is rolling out an L-system based module to realistically simulate trees in 2022 [4].

Hence, the objective of this project is to create a pipeline, built upon AdTree, that can incorporate the concept of an L-system to generate accurate tree models that are stored compactly to address storage efficiency, format interoperability, and data discrepancy.

1.3. Research questions

In order to study the possibility of addressing the aforementioned problems with an L-system, the following research question was formulated:

How can AdTree be extended to incorporate an L-system to obtain a compact representation of a tree geometry, to increase storage efficiency and interoperability, and simulate growth?

While this research question encompasses the scope of the project and addresses the stated problems, several sub questions were formulated to dissect the research question into different parts:

- How does AdTree reconstruct a 3D model from a point cloud and what are its (intermediate) outputs?
- How does point cloud quality and method of acquisition affect the reconstruction?
- What is an L-system, how can it help to store and reconstruct a representation of a tree from a point cloud, and how does it perform compared to current AdTree outputs?
- What other formats could be synthesized for storing a (intermediate) representations of a tree to increase storage inefficiency and interoperability, and how do these perform?
- How does an L-system, derived from a point cloud, simulate growth?

1.4. Project objectives

At the beginning of the project, using the MoSCoW method, the expectations of the project were defined. A number of "must have" points were defined that were deemed the highest priority goals set by the team. In the following table these expectations are shown, along with whether they were achieved or not. As can be seen, nearly all expectations were met. Solely number 4; "Estimate branch diameter per branch" is not achieved. During the process of this project, this was deemed irrelevant to the scope.

Must have

	Expectations	**Results**
1	Pre-processing to clean AHN3/4 point cloud into points that only belong to trees	Achieved
2	Find and fit the trunk from AHN3/4 data using custom code	Achieved
3	Store a graph representation of the tree skeleton	Achieved
4	Estimate branch diameter per branch	Not Achieved
5	Retrieve and store branch node location, thickness and angle of the branches	Achieved
6	Evaluate the proposed pipeline with multiple single tree point clouds provided by COBRA	Achieved
7	Evaluate the proposed pipeline with raw AHN3/4 data to obtain a model of a single tree	Achieved

Table 1.1: Must haves of project (expectations and results)

1.5. Organisation of the paper

In Section 2, *Related work*, an elaboration on the prior work this research is built upon (mainly AdTree) can be found, as well as an introduction of the concept of an L-system and compactness. In Section 3.1, *Pipeline*, the research approach is described in detail, guided by the development pipeline. This is followed by the corresponding results and discussion thereof, in Section 4, *Results and discussion*. A summary of the most important results and discussion to answer the research questions, complemented with recommendations for future work forms the conclusion of this research in Section 5, *Conclusion*.

2
Related work

2.1. 3D tree modeling from point clouds

In 3D modelling, a decision has to be made on whether the model should be data or model driven. The decision is based on the characteristics a model of choice is ought to have. If a model should represent the true 3D geometry as much as possible, the model will be designed to stay true to its input *data* and is therefore called data driven. A drawback of this approach is that it is dependent on (the quality of) its input data. If a model should represent a (reasonable) approximation of the 3D geometry, while ignoring some or all its input data it is called model driven.

The AdTree method to model trees was chosen as it proven to be an automatic and accurate approach for the reconstruction of 3D tree branches from point clouds, recovering both topology and geometry of the tree branches, outperforming other state-of-the-art methods [3] such as SimpleTree [8], PypeTree [1] or TreeQSM [20]. As the Cobra-Groeninzicht's Treemonitor has around one hundred million trees, the automatic aspect is important as well as the topology and geometry accuracy for apt forestry management. AdTree can be characterised as a data driven model, and shares the drawback of input data dependency. Moreover, the AdTree method does not incorporate natural growing rules of tree branches. Therefore, the L-system approach is chosen to extend AdTree to incorporate natural growing rules. This could address data dependency and data discrepancy (Section 1.2), while potentially making the representation of the tree more compact. As the L-system would make the model more implicit, a graphical interpreter is needed such as a Turtle. Hence, the AdTree method, compactness including the L-system concept, and the Turtle agent are elaborated upon in the following sections. The incorporation of the L-system would shift the AdTree method towards a more flexible *model* driven approach, and therefore free it from its data dependent limitation in some aspects.

2.1.1. AdTree

AdTree is an algorithm developed to accurately and automatically reconstruct detailed 3D mesh models from point clouds of individual trees, by Du et al. [3]. To achieve the best results with this method, the input data (point cloud) should be a dense, complete, isolated tree, not contain any foliage and cleaned from outliers. The selection and cleaning of input data can be done automatically, though done manually in this study for simplicity of the research. AdTree's main steps are: *skeleton initialization, skeleton simplification, branch/-cylinder fitting, and adding realism*, see Figure 2.1 for the overview. The focus of this research will be on the first three steps, as realism is outside of the scope of this research.

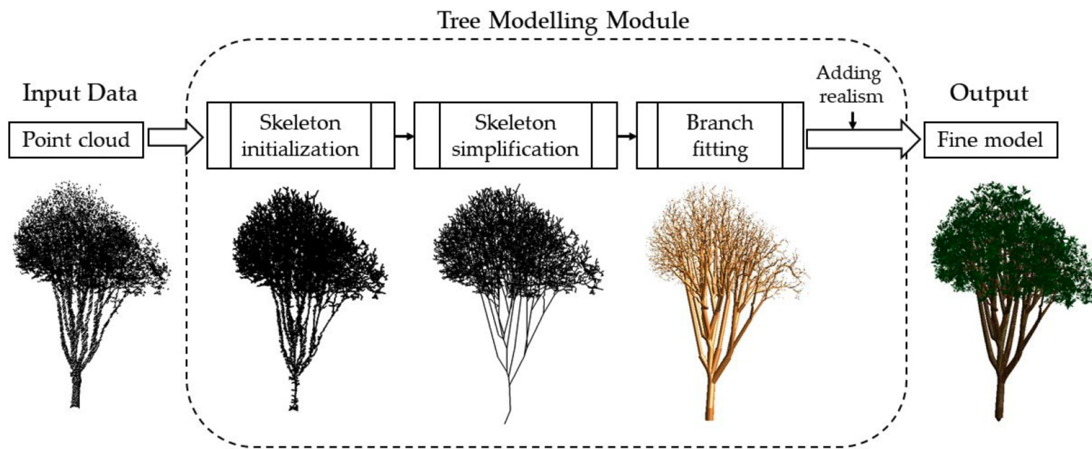


Figure 2.1: Overview of AdTree method, by Du et al. adopted from [3].

Skeleton initialization
In this first AdTree step, all input points of the point cloud are triangulated with a Delaunay triangulation, which yields an initial graph. Its edges are weighted using their lengths defined in the Euclidean space. Then, Dijkstra's shortest path algorithm is applied to compute the MST (Minimum Spanning Tree) to extract the initial tree graph. The idea is that points that are close to each other are likely to belong to the same branch. However, this intermediate MST result is noisy, thus not fit for a compact representation of a tree. A mean-shift algorithm is applied to centralize main-branch points to improve the quality of the skeleton.

Skeleton simplification
The initial tree skeleton resulting from the previous step contains a large number of redundant vertices and edges, most of which do not contribute to the overall shape of the structural tree skeleton. They can therefore be omitted. This is achieved by two main steps, iteratively. First, vertices and edges are assigned importance based on their length relative to their child vertices and edges. Second, adjacent vertices with one child that are within a certain threshold of distance from each other are merged by the Douglas-Peucker method. Vertices that have multiple children with similar (positioned and oriented) adjacent vertices and edges are merged as well. The results can be observed in Figure 2.2.

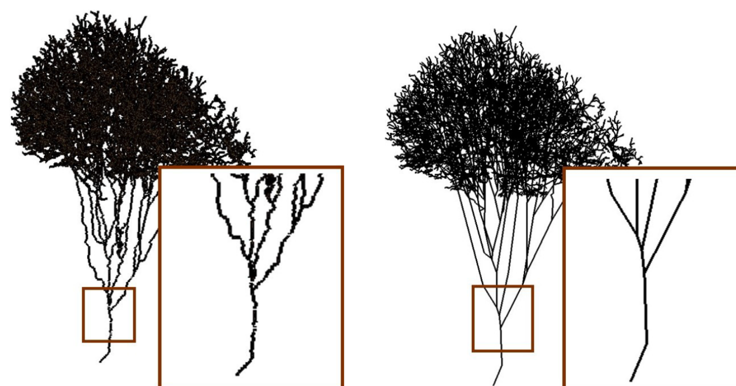


Figure 2.2: Simplification by removing noisy small branches (left) and by merging similar vertices and edges (right), by Du et al. adopted from [3].

The simplified skeleton is the most interesting intermediate result of AdTree for this study. It is a light-weight skeleton that can be used as a compact representation of a tree, it has a fitted trunk (a common problem in modelling tree models), and its quality and usability has been improved by means of simplification. For these reasons, the simplified skeleton outputted by AdTree at this step is the input used for the L-system

intervention (described further in Section 2.2.1). The result of the L-system intervention can be re-inserted into the pipeline, after which the branch fitting step (Section 2.1.1) can continue.

Branch fitting

In this step, the simplified skeleton undergoes a cylinder-fitting process. The geometry of the tree is approximated by fitting a sequence of cylinders onto the graph nodes. This is achieved by first segmenting and identifying the tree in different parts. An accurate radius of the tree trunk can be obtained by initially optimizing in the non-linear least squares sense. The optimization problem is then further solved using the Levenberg Marquardt algorithm. The radii from the subsequent branches are derived from the main trunk. It is important to note that if the point density of the trunk is below a certain threshold, a radius cannot be extracted for the main trunk. This means that the rest of the branch radii can also not be determined. Therefore, if a point cloud is too sparse in the trunk area or is missing it completely, the reconstruction will fail. Tree trunks can be artificially added by means of tree trunk estimation. This is however out of the scope of this research. This step is also altered, which is described in Section 3.2.6.

2.2. Compact representations of 3D tree structures

2.2.1. L-system

Late 1960's Lindenmayer system (L-system) by Aristid Lindenmayer is a mathematical, rule-based approach to represent the growth of vegetation [15]. It describes the development of branching structures with a parallel string-rewriting mechanism [9]. L-system rules possess a recursive nature, which leads to self similarity and are thus fit to be described as a fractal [22]. Trees can also be defined as a recursive structure, making the L-system a promising possibility for modelling them compactly. By increasing the number of recursions, the L-system structure can slowly be grown as well. An L-system can be parametric, allowing for one or more variables to determine the outcome of a rule. Parametric L-systems are defined as a tuple [19]: $G = (V, \omega, P)$, where V is a set of symbols of variables (changeable) and constants (unchangeable), ω (start, axiom) is a string of symbols that describe the initial state of the system , and P is a set of production rules. The parametric L-system variant is used in this research as input for drawing the L-system (the Turtle: see Section 2.2.2).

In essence, the L-system encompasses a set of characters describing drawing rules (in graphical applications) that for instance a Turtle drawing program could follow. The characters in V are the encoding describing all drawing operations possible, as well as (in this case) structural relationships between nodes. ω contains the starting point, the initial drawing instructions using the symbols in V. P then describes the L-system rules, the ways in which ω should be extended with each iteration. A parametric L-system allows some symbols in V to be assigned a value. These values, in this case, describe exactly in what way the turtle should draw: with which length it should draw an edge, which way to turn, and by how many degrees.

L-systems can also used to combine actual measurements with more advanced tree data, such as tree species data, botanical models, or environmental parameters. This data can be used to model trees more compactly, and make and process the models on a large scale and automatically[14]. These advanced models can be used for simulation purposes, focused on tree growth [14]. While complex tree growth models are out of the scope of this research, it is a promising prospect that L-systems could be used for large scale forestry management, as is the case for the one hundred million trees in the forest inventory of Cobra-Groeninzicht.

2.2.2. Turtle (graphics)

In computer graphics, a turtle is an agent that can follow a certain set of commands to move, thus "drawing" by tracing its movements [6]. The turtle has a starting location, a direction, and a "pen" [5].

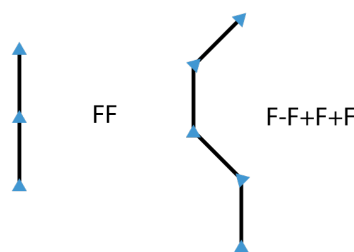


Figure 2.3: Here is demonstrated how a turtle could follow some basic drawing rules, where "F" means forward in a set length, -/+ gives a rotation in each direction.

As mentioned in Section 2.2.1, a turtle can be used to follow the recursive drawing rules of an L-system. Using the characters as defined in V of the L-system, one can give the Turtle instructions. In this research, the following characters were used: "F" for forward with a set length, "+/-" for rotation on the Y-axis in either direction (left or right), and "</>" for roll on the Z-axis (also in either direction). Figure 2.4 shows a 3D example of a recursive non-parametric L-system defined and drawn in this manner.

Figure 2.4: Demonstration of how a turtle could follow L-system drawing rules, using "F", "+/-", and "">/<" as characters, as well as "[/]" to indicate nesting.

2.2.3. Compactness

In mathematics, the Heine-Borel theorem states that a set S is compact if its closed and bounded, meaning if every open cover of S has a finite subcover (e.g. V), for S in $\mathbb{R}^n$ [26]. V is a finite subcover if V has finitely many elements. Intuitively, a subset of S can have many or an infinite amount of subcovers that intersect that cover S. So much so, that the subcovers have redundant coverage of S. It is also possible to have a smaller subset of S that covers S that only has a finite number of covers, and thus is compact.

Due to the size of the database maintained by Cobra-Groeninzicht, every tree model would in an ideal situation be stored as compactly as possible. Compactness as defined in this work does not just entail file size; it is defined as a compact representation of complex geometric data. A compact representation of complex geometry stores the same data (S) in a more efficient and indirect manner (with a smaller subset of S that covers the same information). For example, if one were to want to store a sphere, there are several possibilities. One could store all points of the surface of the sphere that one would want to display directly. Alternatively, one could store the sphere parametrically. The parametric representation of any point on a sphere (x, y, z) is noted in the following formula: $(x - x_0) + (y - y_0) + (z - z_0) = r^2$, where r is the radius of the sphere, and (x_0, y_0, z_0) is its center. This way, instead of storing a large set of points, every point on the sphere is described with the same formula: this is more compact. For complex geometry, in the case of this work a set of cylinders,

the usual representation is to store all vertices and all edges of the geometries that make up the cylinders. A more compact representation, from which all vertices and edges of the complex geometry could be inferred, but which are not necessarily stored directly, would be beneficial for both storing in a database and doing computations. The L-system could form such a representation.

2.2.4. File encoding

The file format in which data is stored relates directly to this notion of compactness. Aside from the L-system, the CityJSON format is also investigated in this research as a possible more compact alternative to traditional methods of storing complex tree geometry. It was developed as an alternative to CityGML by TU Delft's Hugo Ledoux, along with a team of 3D experts. The JSON encoding is emerging as a replacement of the XML standard for transferring data over the web. The same is true for the storing and transferring of 3D city models with CityJSON, for which CityGML used to be considered the standard. The developers of the CityJSON model state its compactness: "The aim of CityJSON is to offer a compact and developer-friendly format, so that files can be easily visualised, manipulated, and edited." By changing from GML encoding to JSON the file is smaller in size, can be parsed and edited by many existing programming languages, and is thus more fit for transference over the web [13].

Any JSON file consists of object properties that can be assigned string, boolean, or numerical values, and two data structures: objects and arrays. These elements can then be nested and combined to create a data structure to the liking of the user. CityJSON uses these elements in a predefined manner. The "Geometry" property of 3D objects, which describes their geometric elements, has several options for representing geometry. A geometry is built out of a list of boundaries being one dimension lower than the object itself, i.e. a line will have vertices as its boundary, and a surface will have arrays of vertices representing lines as its boundaries. Semantics can be defined as well, to specify characteristics of every entity in the object. For every object's geometry, a type will need to be defined ("Solid", "MultiSurface", "MultiLineString", etc.), as well as the level of detail. Additionally, parents and children for the object may be defined. Lastly, a CityJSON model will contain a list of vertices. The objects in the "geometry" property describing the boundary of the object will refer to these vertices using indices, which prevents redundancy in specifying the 3D locations of these vertices.

3
Methodology

3.1. Pipeline

When going from point clouds to L-system representation, the entire process can be divided into several parts. These consecutive parts are described in the method pipeline (Figure 3.1). In short, raw point cloud data of a tree is filtered and passed on to AdTree. After AdTree extracts the skeleton from the point cloud data, the L-system format can be written. This is also where the CityJSON writer can use the extracted skeleton. The L-system approach is integrated into the AdTree software as an extension. In this chapter, each step of the pipeline will be introduced: starting from the raw point cloud and performing data selection and cleaning, obtaining a graph with AdTree, writing the L-system (nesting, relation between nodes, writing to L-string), then reading the L-system in with the Turtle, re-entering the read skeleton into AdTree to convert it to a mesh geometry, and lastly the tree generalization and growing possibilities of the L-system. The method used to write the tree skeleton in CityJSON format is explained as well. After introducing all steps of the pipeline, the most important parts will be described extensively.

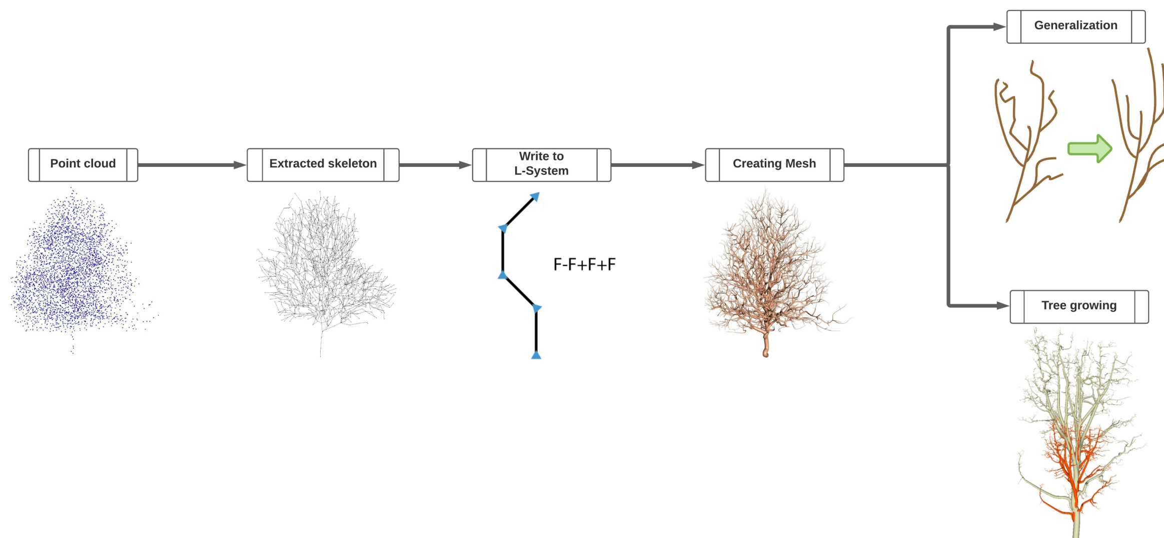


Figure 3.1: Pipeline of the entire L-system process.

3.1.1. Point cloud

As explained in Section 2.1.1, the input data for the native AdTree method should be of good quality (i.e. dense, complete, isolated etc.) for accurate and robust modelling. For the purpose of testing the performance of the L-system on AdTree modelling, it is important to satisfy the data quality criterion such that the native AdTree method is performing as expected. The original modelling performance of native AdTree is considered the “ground truth", in lack of a true ground truth (e.g. dense point cloud validated with pictures). Thus, all

input data used is manually selected, cleaned and validated to result in an accurate model. In this process, the quality of the point cloud in the trunk is most important.

Acquisition source	Accuracy	Foliage
Airborne LiDAR (AHN3)	~5cm-20cm	No
Airborne LiDAR (custom)	1-2cm	No
TLS LiDAR (static)	~2mm*	Yes
TLS LiDAR (mobile)	~5cm	Yes

Table 3.1: The different input sources used and their accuracy. Furthermore, winter data contains no foliage in the case of deciduous trees. Foliage can have a negative impact on the performance of AdTree to reconstruct the geometry of the skeleton. *in 1m-50m measuring distance.

To understand whether the AdTree method remains robust after the implementation of the L-system, the effect of different acquisition techniques and corresponding qualities are tested. An overview of the different acquisition techniques can be observed in table 3.1. Although the accuracy of the acquisition method used does not completely explain the quality of the inputs, it often gives some indication of the density of the point clouds. Generally, it was found the more accurate the input source, the denser the point clouds were. Foliage makes the result of AdTree less stable, and performs better without. However, density of the point cloud is more important as becomes clear in sections 4.2 and 4.3.

3.1.2. Skeleton extraction (graph) with AdTree

The AdTree algorithm, as described in Chapter 2.1.1, creates an output of a simplified skeleton in the form of a graph. The native AdTree software gives the output option of the skeleton in a 3D model format (.ply). This format is rather storage efficient, although it has low interoperability. In terms of storage efficiency, this original skeleton output format is the benchmark used for the CityJSON and L-system formats. The creation of the simplified skeleton marks the point in the pipeline where the intervention into the AdTree algorithm starts. The simplified skeleton contains a simplified MST graph spanning (most of) the input points from the point cloud model of the tree. This skeleton graph is read by both the CityJSON and L-system steps in the pipeline, including the L-system extensions (growing and generalisation).

3.1.3. The L-system format

The skeleton graph extracted in the previous step is used to create the L-system L-string, axiom and rules, which are read by a turtle to then draw the skeleton of the (botanical) tree. This skeleton can then be converted to a mesh geometry. The full pipeline of this process will be explained the following sections.

The first step is obtaining the nesting, meaning finding the branches that are connected to one another and more specifically what child branches originate from one parent branch. The entire tree can be seen as a nested structure, with branches containing child-branches, and these children also containing child-branches. The structure of the L-system and its notation in the L-string, is determined by traversing the entire tree recursively and finding this nesting. When traversing the tree branch-by-branch, the relative position between nodes is computed along with the nesting. The next step is thus defining the spatial relation between two nodes: how to move through a branch and to its children in Cartesian space.

The spatial relation between two nodes is described using 3 variables: a variable for rotation, roll, and forward movement. Starting from the bottom of the trunk (the root), the movement to reach every next node up the branches will be described relatively with these variables. To move from one node to the next, a specific change in roll angle, rotation angle, and distance is used. This method of describing the tree allows the tree to be entirely scalable, not only in absolute size, but when using patterns (L-system rules), the tree can also be "grown" (one of the main advantages of the L-system). A drawback however is that the accuracy of the relations between the nodes can have a large impact on the branches further away, since a small error at the start of a large branch will show more severely further down the branch.

The acquired roll angles, rotation angles, and forward movements are written in traversal order to the L-system's L-string. The syntax of this L-string will be explained in Section 3.2.5. The L-string thus describes the traversal of the entire tree by means of relative movements between two nodes. The L-string contains nesting as well. Branches may contain other branches, and every one branch can have multiple nodes. Any branch

node may be the origin of one or several new branches.

3.1.4. Creating Mesh

To read in a skeleton from the generated L-string, a program called the LsTurtle is used. It reads the L-string and converts it to the geometry required for creating a valid tree mesh. As explained in Section 2.2.2, a turtle can follow a set of rules to draw lines from a starting position and direction. This property of the turtle thus allows it to follow the L-string down the tree step-by-step, follow rules specified for this string, and recursively, starting at the trunk, draw it. The turtle system will be explained further in Section 3.2.6.

The final result of the AdTree algorithm is a 3D mesh of the tree, every branch being a cylinder with decreasing diameter towards the tips of the branches. To obtain such a mesh from the L-system, the skeleton drawn by the turtle is fed back into the AdTree algorithm. To obtain the mesh, the AdTree pipeline with the native AdTree functions is followed.

3.1.5. Generalization

With the obtained L-system string, the ability is gained to generalize to increase compactness and thus storage efficiency. To achieve this while minimizing the amount of alterations done on the originally created tree skeleton, the program starts with the branch tips. The relative position of the tree tip segment to its parent segment is obtained for every single branch tip, from which the average is computed. The original relation for every branch tip segment to its parent segment is then replaced by this average, saving the L-system string required to store all final branch tips. This same process can also be done for more segments in the branch tips, thus saving more space in the L-system string, but having a larger impact on the skeleton of the tree. A more in depth explanation is given in Section 3.3.1.

3.1.6. Tree growing

Another application of the L-system is the ability to “grow” trees either from scratch or from existing skeletons. To grow the trees, the nearest node to a branch tip is taken, from which new branches can sprout out of and grow. All other branches additionally grow in length and in thickness, with the outer most branches growing faster than the branches closer to the trunk. The parameters for the growing process can be altered to fit certain tree types. the tree is grown several times over, allowing it to greatly grow in size in several directions. This application can be useful in the filling of gaps in data, such as growing a tree in accordance with the time elapsed since the data was obtained. This process is described in more detail in Section 3.3.2.

3.2. L-System

This section will explain the L-system process in further detail. The creation of the L-System consists of several integrated parts. Each of these will now be explained thoroughly, starting with the format in which it will be stored.

3.2.1. L-System JSON format

The L-system is stored in a JSON file format. This file format not only allows for the storage of the axiom and the rules, but also of required additional data to successfully store the whole tree. It is able to do so while remaining easy to read for humans and (relatively) easy to parse for machines. XML would also have been a valid option, however the added compression of the JSON format compared to XML was seen as a desirable goal of the file and storage format. Generally, a JSON file requires less storage space while storing the same data as a XML file [23]. This is partially due to the verbose nature of XML files. Additionally, XML files are a harder to read for humans than JSON files are.

The L-system JSON, or LsJSON, does not only store the axiom and rules that encode the L-string and the skeleton, it also contains a range of other data. This data helps store the volumetric data of a tree in a set location in space. The complete LsJSON stores:

- the data needed to construct the L-string
- the trunk parameters:
  - the anchor point of the trunk
  - the radius of the trunk
- default values for step forward, rotate and roll

- the angle format (degrees or radians)

A tree is a volumetric shape that is located at a certain place. The LsJSON stores this required information as two attributes: the "anchor" and the "radius". The radius object stores the radius of the trunk at the base of the tree. This parameter is estimated by the AdTree program. With minimal changes to the native code this parameter can be written to the LsJson when exporting data as an L-system. The anchor is a 3D point at which the base of the trunk of the tree starts in 3D space. It is computed from the point cloud by AdTree. This parameter, like the radius data, can also be extracted and stored without interfering too much with the native AdTree code.

An example of the an LsJSON can be found in Figure 3.2. This is a simple example in 2D to clarify the attributes stored in the file. In real world scenarios the axiom and rules are longer and more complex.

```
{
    "recursions": 5,
    "axiom": "F",
    "rules":
    {
        "F": "F[+F]F[-F][F]"
    },
    "trunk": {
        "anchor" : [100,500,0],
        "radius" : 10
    },
    "dimensions":
    {
        "forward" : 3,
        "rotation" : 22.5,
        "roll" : 20
    },
    "degrees": true
}
```

Figure 3.2: Example of a 2D L-system stored in the LsJSON format. It can be divided in four general groups. The recursions, axiom and rules are the data needed to construct the full L-string. The trunk data is needed to place a volumetric tree at a location. The dimensions are the default turtle command values. Finally the boolean degrees encodes if the angle and roll values are stored as degrees or as radians.

As is covered in Section 3.2.5, every L-string character can have a value connected to it. For example, "F(10)" is a step forward with a distance of 10 meters, and "-(50)" is a left rotation of 50 degrees (or radians). To allow for more compression, default values can be stored under "dimensions". These values will be used when a character does not have a parametric value attached to it. For example, the "F" in the LsJSON in Figure 3.2 will be seen as "F(3)" instead of "F(0)", and the "-" will be seen as "-(22.5)".

Storing tree model data in this format is rather implicit, compared to a more classical approach where all the nodes, edges and meshes are explicitly stored. The only explicit data that is stored in the LsJSON is the trunk anchor point and the radius. It is assumed the rest of the values needed to read this file and construct a mesh of the tree can be calculated from this data. This means that an LsJSON file will occupy less storage space, at the cost of needing more computations to reconstruct the mesh. This adds the potential for deviations and errors that can propagate trough the entire model, due to the lack of correction points.

3.2.2. L-system initialisation

To generate the L-system, the first step is the initialisation of the L-system class. The L-system stores as attributes:

- the graph
- whether it is in degrees or radians

- the root node
- the anchor point
- the radius of the trunk
- the L-string
- the axiom
- the rules
- movement parameters
- growth parameters
- generalisation parameters

The skeleton that was generated by AdTree is copied and duplicated. This is done to be able to edit the attributes of the skeleton's nodes without editing the original skeleton that is in memory. An extra attribute was added to the nodes of the skeleton: a map that stores the L-system representation of each individual node. This representation is a description of the relative movement from its parent towards this node, stored as a string. It contains the forward motion, rotation, roll and nesting. A detailed description of this movement will be given further in Section 3.2.4.

Aside from the starting skeleton itself, several attributes are stored to describe it. These consist of the root node, trunk radius, and root position (anchor), which were previously computed by AdTree. This information is needed to convert the L-system into geometry.

The L-system has several options. The default method reads the AdTree skeleton and converts it to an L-system. This can be either in radians or degrees. After this, two further operations are possible. One can generalise the L-system, aggregating all branch tips into an average and storing it as a single operation, and one can grow the L-system, describing its development step-by-step. Both these options will be explained further in this chapter.

The L-system itself consists of the L-string, axiom, and zero or more rules. The L-string is the complete description of the graph in L-system notation. It is one long string, describing the relative position of all nodes, as well as the length of the branch segments between them, and the nested structure of the branches. The axiom is initially assigned as equal to the full L-string. It can be converted into a shortened version with rules using the generalisation option.

3.2.3. Obtain nesting

The pipeline described in Chapter 3 shows that first the nesting is computed, then the relative position between nodes, and lastly this information is written to the L-string. However, in reality this is a recursive process. The sections below will go into further detail about the relative movement between nodes and the L-string representation. This section will describe the recursive process itself, as well as the nesting.

A (botanical) tree consists of branches, which can then again consists of (sub)branches. This implementation of the L-system needs this information when converting the L-system to geometry. The markers "`[`" and "`]`" are used to note respectively the beginning and end of a branch. When the start of a branch is detected, the current position is stored in order to return to this position when the end of the branch is reached. The rest of the tree can then be traversed from the start of the branch that was found.

The nesting, as well as the relative movement between all nodes, is found by traversing the skeleton recursively. The recursive traversal method (Algorithm 3.1) takes as input two nodes, as well as a pointer to the skeleton. The current node to traverse to is the `startV`, which is a `SGraphVertexDescriptor` from the Boost package. Its parent, `prevV`, is passed as well. The movement of `startV` that will be noted in the L-string is the relative movement from node `prevV` to node `startV`. The root node, which does not have a parent, is thus not included in the L-string directly, but as the starting point (`prevV`) of the movement to the first node after it. The absolute position of the root node in 3D coordinates is stored in the "anchor" property of the LsJSON. The rest of the tree can be generated relative to this point.

Algorithm 3.1: Tree traversal

```
1 input: starting Boost node startV,
2     parent Boost node prevV,
3     skeleton Boost graph skel
4 output: L–string (attribute of L–system class)
5
```

```
6  children = None
7
8  write movement prevV to startV
9
10 if startV is leaf then
11     return startV
12 else do
13     foreach (outward edge eout of startV) do
14         if (eout destination != prevV) then
15             add eout destination to children
16
17 if (startV has 1 child) do
18     return traversal(prevV = startV, startV = child)
19 else do
20     Boost node leaf = None
21     foreach (child in children) do
22         add "[" to L-string
23         add "[" to child.lstring["nesting"]
24         leaf = traversal(prevV = startV, startV = child)
25         add "]" to leaf.lstring["nesting"]
26         add "]" to L-string
27
28     return leaf
```

The traversal method is first called with the root node, as this is the natural staring point. The root node has itself as its parent, and thus for the initial traversal step it is passed as both the `prevV` and the `startV`. When the traversal method is called for all following nodes, the movement towards the current node `startV`, relative to the position of its parent node `prevV`, is written to the L-string. After this, all children of the node are found and traversed recursively. This is where the nesting is detected and written to the L-string as well. The number of children found determines the nesting structure. If one or zero children are found, the branch respectively continues or has reached a leaf. No new nesting structure is initiated. Detecting two or more children means a branching point has been found. In this case, a new branch start marker ("[") is inserted into the L-system L-string for all children. The node-specific L-string nesting description property is also updated. This property is used for the generalisation extension (Section 3.3.1). Each child thus notes the beginning of a new (sub)branch, both in the L-string and as a property of the node itself.

Noting the ending nodes of branches is slightly more difficult. A branch should be ended at the tip, meaning when a leaf node is reached from the last branching point. However, at the point in the code where the beginning of a branch is noted, the end node is still unknown. The ending marker can simply be inserted into the L-string after the recursive call of the traversal method. Adding the ending marker as a property of the correct node however requires the program to return a pointer to the leaf node when it is found. The traversal method thus returns the respective end of a branch to the point where it was started, allowing the ending marker "]" to be added as a property of the correct leaf node.

3.2.4. Compute relation between nodes

Algorithm 3.2 describes the way the relative movement between two nodes is computed. This movement consists of a rotation angle (angle around the Y-axis), roll angle (angle around the Z-axis), and the distance between the two nodes. The relative angle between nodes was split up into two axes because of the way the LsJSON file is read. The part of the code that performs this translation, the Turtle, reads the rotation one axis at a time. It first rotates, then rolls, and lastly moves forward. This means that in order to describe movement in 3D, rotation had to be described per axis. The third axis of rotation, the X-axis, pitch, or forward direction, is in this case irrelevant. This rotation would correspond to a movement similar to if one were to "roll" a branch between two fingers. Any rotation in this direction would not be noticeable in 3D (unless textures are used for the geometry of the branches). The rotation in 3D between two nodes could thus in this case be described using just 2 of the axes.

Algorithm 3.2: Relative movement between nodes

```
1 input: starting Boost node startV,
2     next Boost node nextV,
3     skeleton Boost graph skel,
4     int accuracy
5 output: tuple movement <rotation, roll, forward>
```

```
6
7   movement = {0, 0, 0}
8   Boost node prevV = parent startV
9
10  if (nextV != root) do
11      find the the coordinates of the 3 nodes
12      branch length = distance startV to nextV
13
14      vector to_nextV = coords_next − coords_start
15      vector to_startV = coords_startV − coords_prevV
16
17      define vectors of axis system (X, Y, Z, righthanded)
18
19      project the 2 vectors between nodes to the XY plane:
20          projected vector Z = 0
21
22      find the roll angle of both vectors:
23      for (both vectors in the XY plane) do
24          double roll angle = None
25          if (Y < 0) do
26              roll angle = − acos(dot product(vector,
27                              X axis) / (length(vector)
28                              * length(X axis)))
29          else do
30              roll angle = acos(dot product(vector,
31                              X axis) / (length(vector)
32                              * length(X axis)))
33          if roll angle = NaN, do roll angle = 0
34
35      roll the 2 vectors to the XZ plane:
36          new vector = roll old vector with − roll angle
37
38      find the rotation angle of both vectors:
39      for (both vectors in the XZ plane) do
40          double rotation angle = None
41          if (Z < 0) do
42              rotation angle = acos(dot product(vector,
43                              X axis) / (length(vector)
44                              * length(X axis)))
45          else do
46              rotation angle = 2pi − acos(dot product
47                             (vector, X axis) / (length(vector)
48                              * length(X axis)))
49           if rotation angle = NaN, do rotation angle = 0
50
51      relative roll angle = roll angle to_nextV − roll angle
52      to_startV
53      relative rotation angle = rotation angle to_nextV −
54      rotation angle to_startV
55
56      round angles close to 360 or 0 degrees to 0
57
58      return <relative rotation angle,
59              relative roll angle,
60              distance>
```

The method for splitting up the relative rotation between two nodes into two axes was based on the concept of spherical coordinates. Figure 3.3 shows how the rotation and roll angles were found. First, the vector between two nodes is projected onto the XY plane. This can simply by done by making the Z-coordinates of the vertices equal to 0. The roll angle, around the Z-axis, is then the angle between this projected vector and the X-axis. The angle between two vectors (θ) can be found with the following formula:

$$\begin{aligned} \cos\theta &= \frac{\vec{u} \bullet \vec{v}}{\|\vec{u}\| \bullet \|\vec{v}\|} \\ \theta &= \arccos \frac{\vec{u} \bullet \vec{v}}{\|\vec{u}\| \bullet \|\vec{v}\|} \end{aligned} \tag{3.1}$$

Using this formula as-is would not distinguish between vectors on different sides of the X-axis, meaning when

a vector is mirrored in the X-axis, it will have the same angle to the X-axis. Because this angle is used to roll around the Z-axis, vectors on the "other side" of the X-axis need to be accounted for. This is done by rotating with a negative angle if the projected vector has a negative Y-coordinate. Vectors on the negative Y-side of the X-axis will be rolled in the opposite direction of those on the positive Y-side. The angle to the X-axis in this case can be calculated using $\theta = -\arccos \frac{\vec{u} \bullet \vec{v}}{\|\vec{u}\| \bullet \|\vec{v}\|}$. Mirrored vectors in the Z-direction were not taken into account, since all vectors used here were projected to the XY plane.

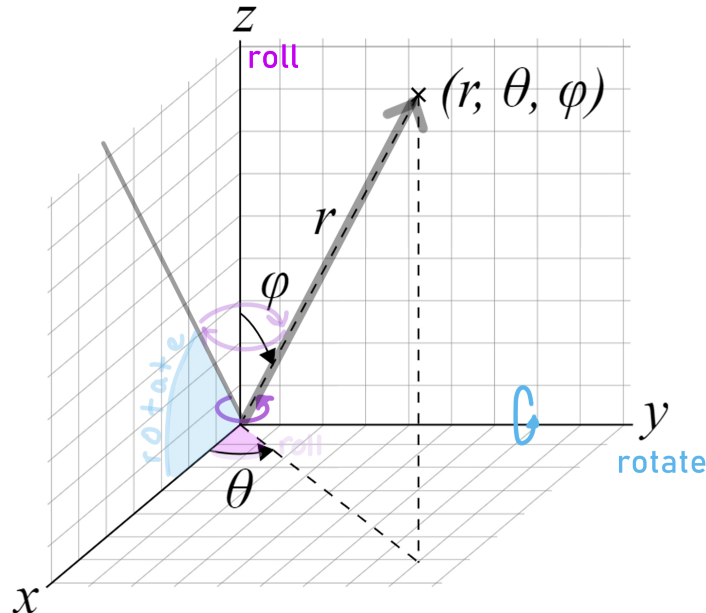


Figure 3.3: Method of finding relative angles, for both Y and Z axis.

After the roll angle around the Z-axis is found, the rotation angle around the Y-axis is calculated. Projecting the original vector to the XZ plane by setting the Y-coordinate to 0 here would result in a vector on the XZ plane with a different inclination than the original, meaning it no longer points in the right direction. To project the original vector to the XZ plane, it is rolled around the Z-axis instead. The angle needed to rotate exactly onto the XZ plane has already been calculated: it is the opposite of the roll angle. For finding the rotation angle, the same formula as above is used to find the angle between the vector and the X-axis. In this case, mirrored vectors will have a negative Z-coordinate. The angle to the X-axis in this plane is opposite to the positive rotation direction using the right-hand coordinate system, meaning the real angle to rotate with will be a full circle (2π) minus the found angle. For mirrored vectors, the original found rotation angle will be the correct one. Found rotation angles of vectors with a negative Z-coordinate are therefor not subtracted from 2π. Since vectors are rolled onto the XZ plane, accounting for vectors mirrored in the Y direction is not necessary.

The final result of finding the relative movement from one vertex to the next will be the roll angle, the rotation angle, and the distance between the two vertices. To get the relative angles of a vertex `nextV`, the global angles of both the vector towards it, as well as the global angles of the edge that preceded it need to be calculated. The current edge, here noted as $\vec{b}$ will be from `startV` to `nextV`, the previous edge, here noted as $\vec{a}$ will be from `prevV` to `startV`. The relative roll and rotation angles towards node `nextV` are then the global angles of $\vec{b}$ minus the global angles of $\vec{a}$.

3.2.5. Write to L-string

Algorithm 3.3 displays the way the L-string of the L-system gets filled, as well as how the relative movement of each node is stored as an attribute of the node. Each movement to a new node is marked by one occurrence of "F" in the L-string. Occurrence of rotation ("+", "-") and roll (">", "<") are optional, and are only written if they exist. Thus, the angles only get written if they are not 0, preventing redundancy.

Algorithm 3.3: Tree traversal

```
1  input: starting Boost node nextV,
2      next Boost node nextV,
3      skeleton Boost graph skel,
4      int accuracy
5  output: L-string (attribute of L-system class)
6
7  compute movement from startV to nextV
8  movement = roll angle(rad),
9      rotation angle(rad),
10     forward distance(m)
11
12 if (L-system: degrees) do
13     convert rotation and roll angle to degrees
14
15 if (rotation angle > 0) do
16     round angle to accuracy
17     write "+(angle)" to global L-string
18     write "+(angle)" to rotation of nextV
19 if (rotation angle < 0) do
20     round angle to accuracy
21     write "-(angle)" to global L-string
22     write "-(angle)" to rotation of nextV
23
24 if (roll angle < 0) do
25     round angle to accuracy
26     write ">(angle)" to global L-string
27     write ">(angle)" to roll of nextV
28 if (roll angle > 0) do
29     round angle to accuracy
30     write "<(angle)" to global L-string
31     write "<(angle)" to roll of nextV
32
33 if (forward distance > 0) do
34     round distance to accuracy
35     write "F(distance)" to global L-string
36     write "F(distance)" to forward of nextV
```

Corresponding to the order in which the Turtle will read in the L-string movement, the rotation angle gets written first. The roll angle follows, with the forward marker after that. As discussed previously, the nesting markers are not written here, but inside the recursive traversal method. Because the Turtle can roll and rotate in both directions, a distinction is made between positive and negative angles. Which direction is positive, and which is negative, is determined using the right-hand 3D axis system.

3.2.6. L-System to AdTree

As described in Section 2.2.2, an approach based on the python library turtle has been used to translate the LsJSON file back to 3D geometry. The class with this functionality is called the LsTurtle, for ease of reading the python turtle will be called just turtle. Unlike a normal turtle, the LsTurtle is not limited to only drawing edges by following simple commands. The lsTurtle additionally generates all the data that is necessary to allow AdTree to recognize the LsJSON as the components of a tree. This allows the program to take advantage of the existing functions of AdTree to transform these components into a volumetric tree model. In the native version of AdTree these components are computed from a point cloud, however, this original data source is not available when opening an L-system.

The processes the LsTurtle executes can be summarized in the following steps:

- Access and collect the attributes stored in LsJSON
- Translate the axiom and rules into an L-string (if needed)
- Read the L-string
- Compute the height and bounding distance of the tree

The rest of this subsection will describe how the LsTurtle functions based on these four steps.

Access and collect the attributes stored in LsJSON

The augmented AdTree allows the user to select an "Open L-system" option from the GUI. This will prompt the user with a file dialog that allows for easy file selection. The chosen file path is fed into the LsTurtle

instance. The LsTurtle accesses the data that is stored in the LsJson with the help of the nlohmann JSON library [17]. The data that is stored in the JSON is handled in different ways. Data like the default dimensions are stored privately and are only accessible by the LsTurtle instance itself. Data like the trunk anchor and radius are also stored privately but accessible with getters that enable parts of the substituted AdTree code to have easy access to the values. Some data, like the axiom and the rules are accessed, used for computation, and immediately afterwards discarded.

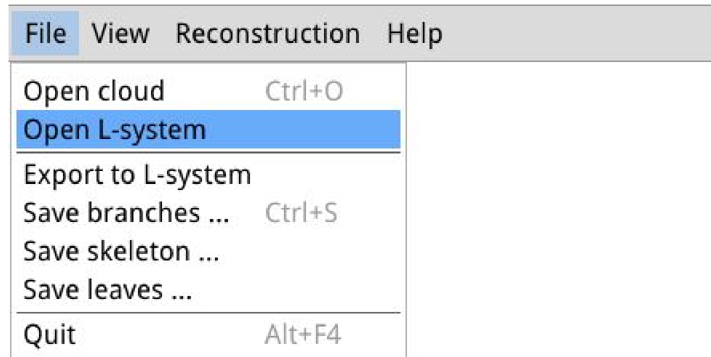


Figure 3.4: The added GUI element that allows the user to easily select and open an LsJSON file

Translate the axiom and rules into an L-string

As mentioned before, the LsTurtle does not store the rules or axiom. This data is only used to create the L-string from them, and is discarded afterwards. The creation of the L-string is called the translation. The axiom is translated with the help of the supplied rules and the recursion variable. If no rules are supplied, the translation process is bypassed. In this case the axiom is seen as the L-string.

If rules are supplied by the JsJSON the translation process is executed. This is done by first constructing a dictionary. This is a map of the rules, created with the key set as the predecessor part of a rule and the mapped value as the matching successor part. The actual translation is done by iterating over ever character of the axiom. Every time a character matches with a key in the dictionary, the character is replaced with the matching mapped value. With this process the axiom-like string, generally, grows longer. This iteration process is repeated until it has looped over the complete axiom-like string a specified amount of times (this value is supplied by the LsJSON).

The "recursions" object in the LsJSON dictates the number of times the axiom-like string is iterated through. The term "iterations" was not used here, due to the possibility of the L-system to grow both iteratively and recursively. The diagram in Figure 3.5 gives a simple example of how this translation creates a recursive growth. The first section of the diagram shows a section of an LsJSON file, the second section shows the resulting L-string per recursion number. In this example the characters used as rules do not directly translate to LsTurtle movement, this is however also a possibility.

```
{
    "recursions": 5,
    "axiom": "A",
    "rules":
    {
        "A": "AB"
        "B": "A"
    },
    ...
}
```

```
recursion = 0 : A
recursion = 1 : AB
recursion = 2 : ABA
recursion = 3 : ABAAB
recursion = 4 : ABAABABA
recursion = 5 : ABAABABAABAAB
```

Figure 3.5: Top: fragment of a recursive LsJson file. Bottom: the resulting L-string per iteration. Note that when recursion = 0, the axiom is the same as the L-string and thus no translation is needed.

Due to the way the L-string is read in a later step, the occurrence of nesting without any "F" characters can cause errors. Examples are cases where a subsection of the L-string is "...[X]...", "...[+]..." or "...[>]...". To avoid the errors that these and similar cases bring, these situations are removed from the L-string in a cleaning process.

During the cleaning process, the L-string is iterated over backwards and nesting is marked. Whenever a nesting is found that does not include any sub-nesting, nor an "F" character, the nesting is removed from the L-string. After the complete reverse iteration the cleaned L-string is finished and can be used for the reading process.

Read the L-string

The cleaned L-string is transformed into geometry in the first process, which resembles a turtle. The L-string encodes the steps the turtle takes in 3D space. The turtle iterates through the L-string and executes the command the current character encodes. The location of the turtle while walking is stored into a skeleton graph, after reading this will resemble the complete structural skeleton of the stored tree.

The "F" character is the only "storing character". The "F" encodes a step forwards. When this character is encountered, the turtle does not only step forward but it also stores the point it walked to as a vertex. This point is stored in an adjacency list. To connect the vertices, the edge between the point it moved from and the point it moved to is also stored.

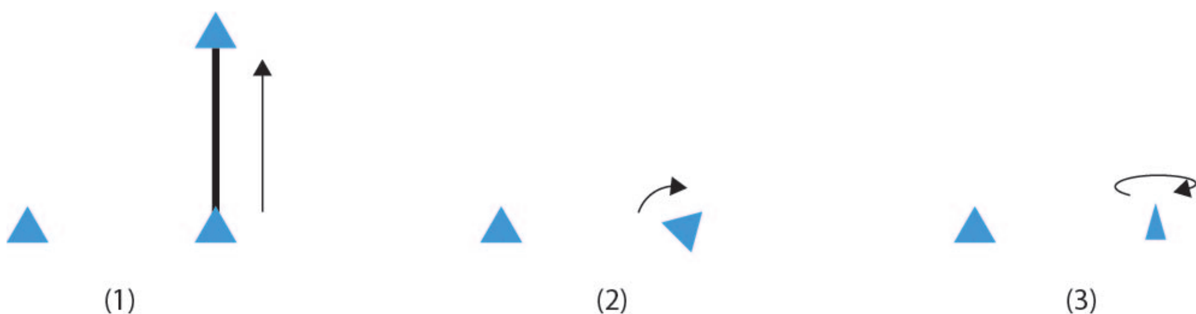


Figure 3.6: Visual example why the "F" char is the only "storing character". Case 1: the "F" char is encountered, a new point and a new edge is created. Case 2: a rotate char is encountered ("+" in this case), the turtle rotates but no new point nor a new edge is created. Case 3: a roll char is encountered (">" in this case), the turtle rolls but no new point nor a new edge is created. So in case 2 and 3 there is no new information related to the graph to store.

Figure 3.7: The simplification of straight line segments during the reading process. When passing over a "F" char no new point is stored nor a new edge if the next char is also an "F". The location of the turtle does get updated however. If the next "F" char is not followed by another "F" char, the point and edge are stored. (1) the location updates of the turtle, (2) the stored edge and points.

During the reading process the LsTurtle also applies the standard dimensions when necessary, and simplifies the straight parts of the skeleton. An example of the simplification can be seen in Figure 3.7.

An LsTurtle instance has no direct functionality to create branching chains. However, a tree has a skeleton that branches heavily. In the L-string branching, or nesting, it is declared with the square brackets "[]". When during the reading process the the "[" char is encountered, a new LsTurtle instance is called. The main lsTurtle will iterate further over the line until the ending of the nesting, "]", at the right level is found. The new LsTurtle will read the nested line and return the found edges and vertices. This data is stored and connected in the main LsTurtle's graph. This creates a branching L-system by recursively calling linearly working turtles.

Figure 3.8: Example of the recursion of the LsTurtle reading. This example is the 2D skeleton of the L-string "F[+F-F[+F]F]-F". The main LsTurtle is fed the line "F[+F-F[+F]F]-F" and executes "F-F" . The second child LsTurtle is fed the line "+F-F[+F]F" and executes "+F-FF". The second child LsTurtle is fed the line "+F" and executes this completely.

The constructed tree graph is stored in a Boost adjacency list. This is a list that collects the vertices, indices, and their respective properties. The adjacency list is also used in the native AdTree to store the skeleton information, enabling AdTree to use the LsTurtle skeleton graph directly without the need of any conversion of type.

Compute the height and bounding distance of the tree

After the LsTurtle has reconstructed the skeleton graph, the final computations related to the created skeleton can be made. These are the height of the skeleton and the bounding distance of the skeleton. These two variables are needed, just like the trunk variables, to allow AdTree to be able to function with the supplied data.

The height is computed by looping through the collected vertices and computing the distance between the anchor and the point at that iteration. If the height delta (point z - anchor z) is larger than the stored height, the stored height gets updated to this height delta. In that same loop the bounding distance is computed. This is done by computing the distance from the anchor and the point at that iteration. If this distance is bigger than the stored bounding distance the bounding distance is updated.

From LsTurtle to AdTree data

The LsTurtle collects and computes the information that AdTree computes from a point cloud. Thus, the LsTurtle bypasses a major part of the AdTree process. The output of the LsTurtle needs to be fitted to the correct variables in the native AdTree in order to use AdTree to create 3D volumetric geometry from the skeleton.

The vertices that are created by the turtle are used as points in the 3D viewer, allowing the user to visualize the vertices stored in the skeleton. These points are not of the same nature as they are when opening a tree point cloud directly. The points outputted by LsTurtle do not reflect the original point cloud from which the skeleton is extracted. This data is not stored in the LsJSON used to create it. The simplified skeleton, now internalized in AdTree, is used to draw the edges in the viewer and connect the earlier copied vertices to create a visualized graph.

The following creation of a volumetric tree is completely done with native AdTree functions. After the clone skeleton function, native AdTree functions are used to reconstruct the mesh. This is all still happening in one connected process after the "open L-system" option is selected by the user. This means that when the user imports an LsJSON into AdTree it creates a cloud, skeleton and mesh in one go. It does not constrain itself, like when importing a cloud, to only one of these steps until the user requests the next one. This is done because the LsJSON encodes the data of both the skeleton and the mesh. This is unlike the point clouds, where all these parameters need to be computed/approximated. The option to create leaves is however still kept as a separate option that needs to be selected by the user. The LsJSON and LsTurle do not directly encode or compute the leaf data. However, due to the close interaction between the LsTurtle and the augmented AdTree the native AdTree option to add leaves functions very well on the L-system trees.

3.3. Extensions

The base L-system functionality reads an AdTree skeleton into an L-system L-string. Two extensions are possible as well: one can grow the tree step-by-step, and one can generalise the tips of branches. Both possibilities will be discussed further in the following sections.

3.3.1. Generalisation

The generalisation functionality replaces the tips of all branches with a computed average. It starts with a previously computed L-system. Both the generalisation and the growing work with a custom branch structure. This branch structure computes and stores additional structural parameters of the L-system, such as the leaf nodes. For the generalisation functionality, the leaf nodes stored in this structure are used to initialize the algorithm. Starting from each leaf node, the generalisation takes a step back, towards the parent of the leaf. It then compares the relative movement from all parents, to all leaves. The relative motion towards a node was previously stored as an attribute of all nodes when the L-system was generated. This information is now used to compute the average roll angle, the average rotation angle, and the average branch length towards all leaves of the tree. The motion towards these leaf nodes is then changed to be this average. Instead of working with the full L-string as the axiom, the L-system will now work with an altered axiom and rules. For each leaf node, the averaged movement will be marked as a rule. The full movement previously noted in the L-string/axiom will now be replaced with a rule marker. This can be any currently unused string character, in this case "X" is used. A corresponding rule is added to the L-system. It will map "X" to the average roll, rotation and forward motion of the branch tips.

Depending on user input, one or multiple steps can be averaged in this manner. If the number of steps to average is larger than one, the algorithm will average out multiple nodes. Starting from the leaves, an average is computed for each step back towards the root of the tree. The rule noting the generalised movement will be extended with an average of each step. A generalisation with 3 steps for example will have the following structure:

```
{rule:  averaged rotation to all parents of parents of leaves
        averaged roll towards all parents of parents of leaves
        average distance to all parents of parents of leaves

        averaged rotation to all parents of leaves
        averaged roll towards all parents of leaves
        average distance to all parents of leaves
```

```
        averaged rotation to all leaves
        averaged roll to all leaves
        averaged distance to all leaves
}
```

In the rule component of the L-system, this will for example look like the following (with the angles in degrees):

```
{"X": "+(20)>(30)F(1.2)-(33)>(170)F(0.4)-(163)<(11)F(2.3)"}
```

One can see three sets of rotation, roll, and forward motion, corresponding to the three steps that were averaged. The order of the steps will be the order of the eventual traversal, starting from the last node that was averaged, and ending with the leaf. In the axiom, only the first node of each rule (the parent of the parent in the example) will be marked. The other two nodes, the leaves and their other parents, will not be marked. The three consecutive nodes are thus replaced with a single marker "X" at the start of where the rule occurs. All branch tips are replaced with the same averaged structure. This saves in complexity of the L-system, as all branch tips will now be described with a marker at the start of each tip in the axiom, and a single rule noting the same relative movement(s) towards all leaves.

The rewriting of the axiom is done using the relative movement property that was previously stored per node. When averaging nodes, for each node that is used, the "forward" property gets changed to contain the rule marker, instead of the original forward movement. For each step except the last, an asterisk is added to the rule marker to note this node will not be noted in the axiom. After all averages are computed, the generalisation algorithm traverses the skeleton one last time, starting from the root. The axiom is emptied, and passing over all the nodes, gets re-written incorporating the generalisation rules. If the forward property of the node is equal to the rule marker ("X"), an "X" gets added to the axiom instead of the stored relative movement to the node. If the forward property is equal to the rule marker plus an asterisk, nothing happens. If no rule marker is present in the forward property, the relative movement to the node gets written to the axiom. This way, all generalised nodes are replaced with a single rule marker, at the start of each sub-branch that was generalised.

As for the nesting, this was stored as an attribute per node as well. In principle, the nesting is not changed when inserting averaged rules. In practice, the algorithm also filters out redundant nesting markers. This is done by seeing if the nesting attribute of a node contains both opening and closing markers ("[" and "]"). If this is the case, the markers get counted, and the opening or closing markers that are redundant are removed. The amount of opposite markers is then decreased accordingly. For example, a node with 3 opening markers and 5 closing markers ("[[[]]]]]") will be written as 0 opening markers and 2 closing markers ("]]"). All other markers were redundant.

3.3.2. Growing

Similar to the generalisation, the growing function can be called during the process of writing to L-string.

Several hypotheses were formed about how a tree will sprout and grow its branches:

1. The position of a bud for a new branch should be specific on a branch rather than random.
2. The sections of branches near the tips will grow faster than those near a trunk.
3. The thickness of branches grows at a different speed compared with the growth speed of length.

In practice, the values of the position to sprout, the speed of branch growth in length and thickness, and the faster speed for sections near tips are provided by the user based on the species information. In this project, a default set of these values was defined, where users can change them in the GUI as well.

The specific position for a node mentioned in hypothesis 1 is a relative position, which is equal to how many nodes are between a node and the tip (leaf) of the branch it belongs to. For example, when the sprout position is 1, the new branches will be added to the nearest node of leaves on every branch. Moreover, if the node is already a branch point, which means two or more branches are attached to it, a new branch will not be added. Figure 3.9 shows an example of how a tree will grow in this method when the sprout position is 1.

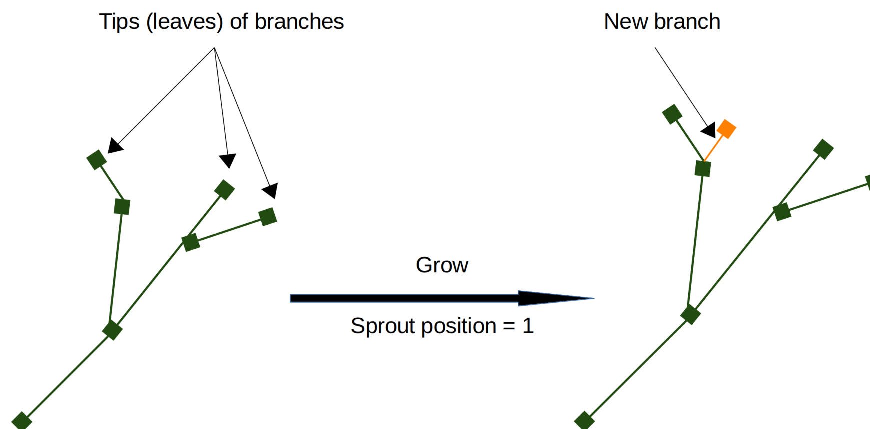


Figure 3.9: An example of growth process.

To implement this, first it is needed to get the relative position of each node. Thus, a depth-first search method is used to split branches of a tree and then compute the relative position for each node.

Algorithm 3.4: Get relative position

```
1  input:
2      skeleton Boost gragh skel,
3      Boost node rootV
4  output:
5      map relative_position <Boost node, int position>
6
7  wait_list = []
8  back push rootV into wait_list
9  while (wait_list is not empty) do
10     root_ = wait_list.back
11     branch_list = []
12     back push root_ into branch
13     next_list = []
14     find all next Boost nodes for root_ and store into next_list
15     next_ = nexts[root_.visit_time]
16     root_.visit_time += 1
17     if (root_.degree <= root_.visit_time) do
18         back pop wait_list
19     while (next_ is not a leaf) do
20         back push next_ into branch_list
21         if (next_.degree-1 > next_.visit_time) do
22             back push next_ into wait_list
23         next_.visit_time += 1
24         next_list_ = []
25         find all next Boost nodes for next_ and store into next_list_
26         update next_ with next_list_[next_.visit_time-1]
27
28     back push next_ into branch_list
29     next_.visit_time += 1
30     for (node in branch_list) do
31         if node in map map relative_position and value position > node index in branch_list do
32             continue
33         else do
34             insert {node, node index in branch_list} into map relative_position
35     back push branch_list into branches
```

After finding all relative positions for nodes in the tree, the next step is to generate L-strings for the new branch. The process is very similar to the generalization function. During writing to L-string, rule markers are put into the axiom if the relative position of nodes equals the position of a bud that was set up by the prorgam or from the user's setting. There are 4 rules responding to the 4 cases of the tips of branches: positive rotation and positive roll, negative rotation and positive roll, positive rotation and negative roll, and negative rotation

and negative roll. A new branch near a tip with positive rotation and positive roll will have the following structure:

```
{rule:  averaged rotation to all leaves that have positive rotation
        averaged roll to all leaves that have positive roll
        averaged distance to all leaves
}
```

In the rule component of the L-system, there will be 4 rules as well, for example like the following:

```
{"A": "[+(159.210)>(93.597)F(0.360)]"},
{"B": "[-(173.812)>(93.597)F(0.360)]"},
{"C": "[+(159.210)<(113.746)F(0.360)]"},
{"D": "[-(173.812)<(113.746)F(0.360)]"}
```

Notice that there are brackets "[]" around the new branches. This means the rule markers are put in front of the original movement, so when reading this, the turtle will first draw a new branch, then go back to continue drawing the original branch.

After this, the distance of movement will be updated with the product of the original distance and the basic growth speed. This way, the tree does not only gain more branches while growing, but existing branches also grow longer. In this project, if the relative position of the start node of a movement is smaller than 4, the distance of the movement will be updated with the product of the original distance and the faster growth speed. Last but not least, the radius in the L-system will be updated with the product of the original radius and the growth speed of thickness. This way, branches also grow thicker as they grow in length.

3.4. CityJSON semi-explicit storage format

For using CityJSON to store tree models, the CityJSON standard was followed as closely as possible. However, to reduce the file sizes some minor elements were changed. The first one is that the "geometry" set does encode geometry, but not in the way that a regular CityJSON would do. The skeleton is explicitly stored with all the nodes and their 3D coordinates as the "boundary". In the "semantics" map there are "types", which store the radii present in the volumetric tree model, and "values", which store the "types" of radius belonging to an edge. This means that the "boundary" set does not directly encode geometry, but rather constructs the rails across whcih the "types" of radii are swept to create geometry. An analysis of the resulting file sizes can be found in Section 4.8.

The different "types" of branches have unique radii. These classes of radii differ per tree, since the radius of a class for a smaller branch is dependent on the radius of that of the largest radius, being the trunk. There are two classes that additionally indicate quite an important characteristic of each branch, which is whether the branch is the trunk of the tree (maximum radius), a branch tip (minimum radius) or somewhere in between.

Not storing the geometry directly into the "boundary" map, but rather in this alternative manner, means that normal CityJSON readers are unable to read the files and that according to validation service cjio [12] the CityJSON file is invalid. This could be resolved by setting the "Cityobjects""type" to "road" or "railway". Doing this will not eliminate but only move the issue: the software packages used to open these files will open it as if it is either a road or a railway object. Thus, not only classifying it incorrectly but also being unable to execute the correct mesh creation process as described previously. Only the skeleton will be drawn, instead of the volumetric tree. The "Cityobjects""type" is thus kept as "SolitaryVegetationObject". This means the custom CityJSON format is not valid, not standardized and can not be open natively by CityJSON readers. However, it utilizes most of the valid CityJSON formats and the creation of geometry is a simple process that could be incorporated into the CityJSON standard and readers.

```
{
    "type": "CityJSON",
    "version": "1.0",
    "CityObjects": {
        "oneTree": {
            "type": "SolitaryVegetationObject",
            "geometry": [
                {
                    "type": "MultiLineString",
                    "lod": 2,
                    "boundaries": [
                        ...
                    ],
                    "semantics": {
                        "types": [
                            {
                                "class": 1,
                                "radius": 0.00937500037252903
                            },
                            ...
                             {
                                "class": 10,
                                "radius": 0.09375
                            }
                        ],
                        "values": [
                        ...
                        ]
                    }
                }
            ]
        }
    },
    "vertices": [
    ...
            ]
    ]
}
```

Figure 3.10: The structure of the CityJSON tree format. Note that this is not a standardized storage method and although we call it a CityJSON tree it is not according to the current standard. This is an alternative approach that would allow more compression. For a more in depth description of the format see Section 3.4.

Figure 3.10 shows an example of the custom CityJSON format that was created. In order to start the process of writing the entire tree structure to a new format, it is first needed to get all necessary information from the tree model created and stored by AdTree. This starts with the collection of skeleton data, meaning the nodes and edges and the radius data that is stored per edge. To correctly collect this, the point cloud needs to be filtered. In cases where a whole point cloud is loaded in to create a tree model, following the native ADTree modelling approach a lot of points are present in the in memory skeleton graph that are not used in the actual skeleton. The unconnected points are discarded and only the connected points/nodes of the skeleton are stored.

The collected edges do not need filtering, however they do need correction. The edges are generated by ADTree by connecting points via their indices. If points are discarded when filtering the point cloud, like in the case when points in the cloud are not part of the skeleton, these indices are not pointing to the correct points anymore. Thus, when collecting the edges for the CityJSON export these indices are corrected so they

connect the right points.

Every edge in the skeleton created by ADTree contains information about the radius the mesh pipe around it should have. This data is collected and stored in a list that is in the same order as the edges. From this list the maximum and minimum radius is computed. This created domain is split in steps of 10, allowing extra data compression by reducing the amount of similar data. For every radius in the radii edge list, a step is chosen and these index values are stored in a list.

The JSON file is written with the help of the nlohmann JSON C++ library [17]. This allowed for easy JSON access with no need for complex code.

3.5. Classic skeleton explicit storage format

Aside from the functions required to allow the user to export, generalise, grow and open L-systems, the augmented AdTree also has a reworked export skeleton function. This function will export a skeleton into a more classical format. The points are stored as 3D coordinates, the edges contain two indices referencing two points and thus connecting them. This information is encoded in the .PLY file format.

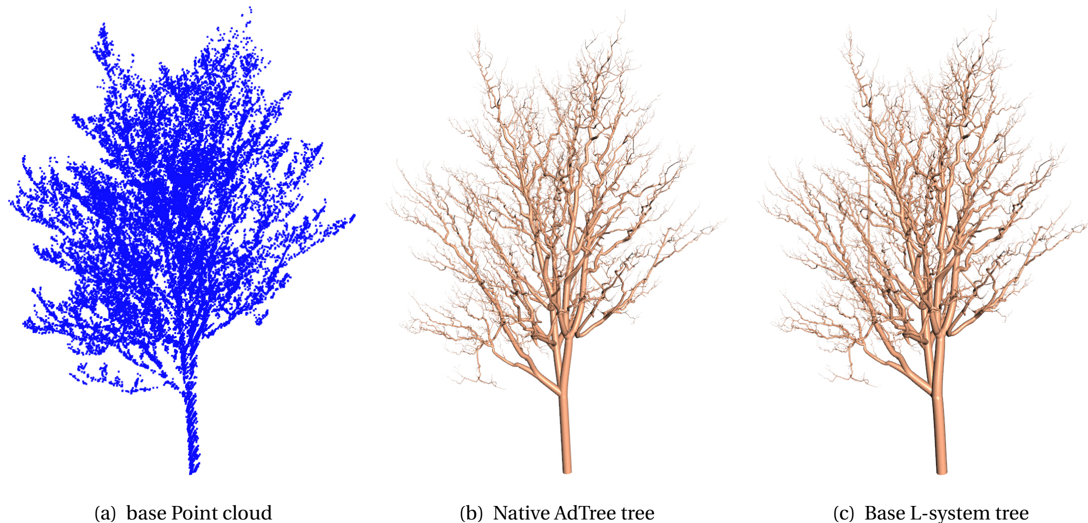
Although the functionality to write .PLY files did exist, the native AdTree was unable to write valid files. The output was not readable by any of the 3D viewers and modellers that were tested. To add functionality to AdTree, but also to add an easy way to check the skeletons for issues while programming, this method was fixed. The augmented AdTree now writes the correct header, edges and vertices to be opened in other software. This writer is similar to the CityJSON writer, where the point cloud data and edge data had to be either corrected or removed from memory before the writing process could start. For an example of a .PLY file as outputted by the augmented AdTree program, see Figure 3.11.

```
ply
format ascii 1.0
element vertex 4468
property float x
property float y
property float z
element face 0
property list uchar int vertex_index
element edge 4467
property int vertex1
property int vertex2
end_header

127395.5156 401149.75 13.08600044
127395.5625 401149.8438 12.68599987
...
127395.7031 401149.875 12.59599972
127397.0078 401151.7812 11.9119997

457 506
705 3027
...
2011 170
1530 1531
```

Figure 3.11: The structure of the classical skeleton in the PLY file. This is not a completely valid structure due to the absence face objects, making it challenging to be opened fully by programs that are depended on the existence of these objects, like Blender and Rhino3D.

4
Results and discussion

This section demonstrates how the pipeline as described in Section 3.1 performs in practice. The pipeline was, as mentioned before, developed and integrated into the existing software of AdTree, allowing the utilization of the already implemented structure, GUI and functions.



(a) base Point cloud (b) Native AdTree tree (c) Base L-system tree

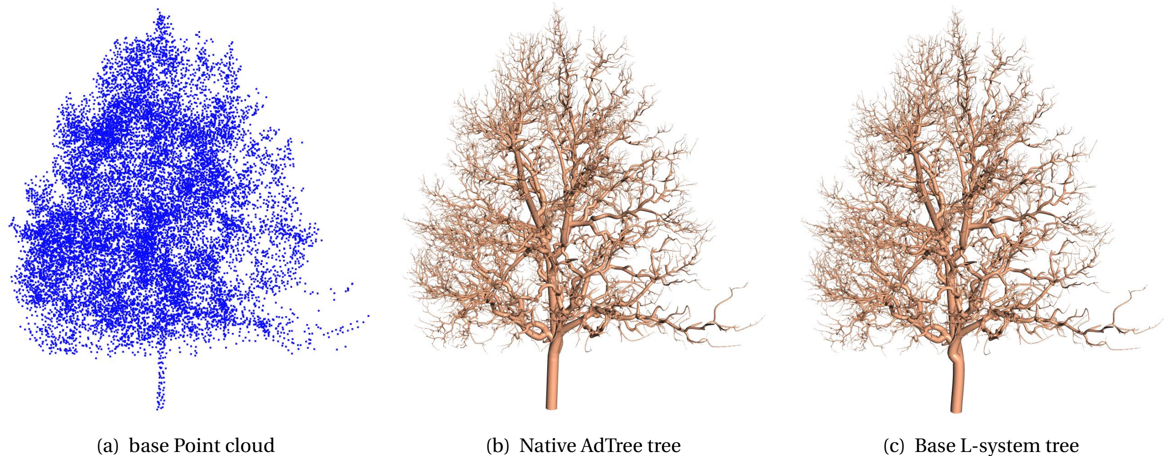
Figure 4.1: The resulting AdTree model and L-system model of the Paris_Luxembourg tree point cloud

4.1. Assessment

To judge whether the L-system format is a good alternative to more traditional methods of storing complex geometries of trees, several aspects were tested. In order to assess the quality of the geometries created from the L-system, different kinds of trees were loaded into AdTree, exported to L-system, transformed into mesh geometries, and inspected visually. Differences in acquisition method (Section 4.3) and input point cloud density (Section 4.2) were also investigated in this manner. The L-system geometry is compared with the geometry as outputted by AdTree as well (as this output is what would result from the input if no L-system intervention was done), both visually and quantitatively. The same is done for the two skeletons. Aside from the capability to be transformed into accurate complex mesh geometry, another measure of the L-system's quality and compactness is file size in comparison with more traditional formats (Section 4.8). It is important to note that file size is but an indication of compactness, file size is also determined by other factors not directly related to the compactness of the format itself. Lastly, some inefficiencies are discussed, as well as the functionality and effect on compactness of the generalisation and grow extensions.

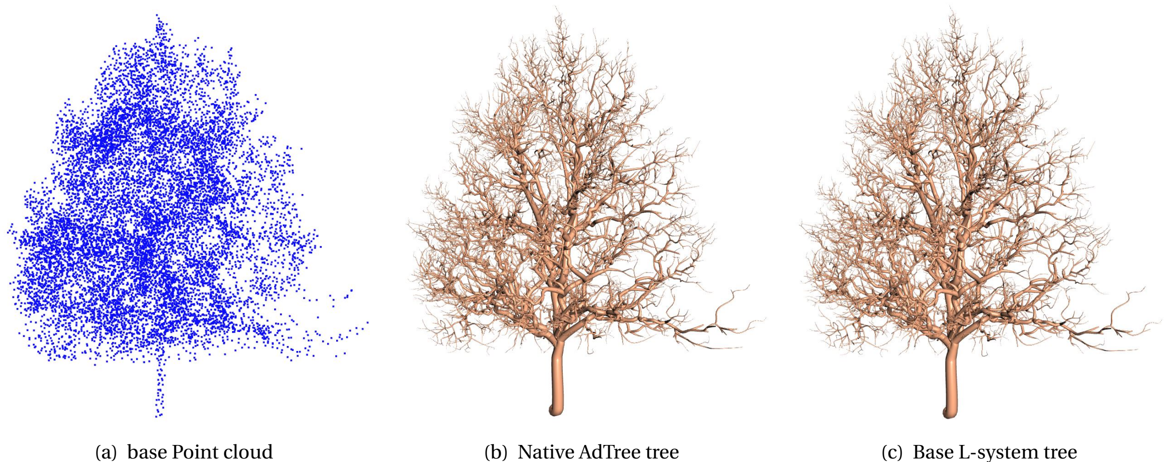
4.2. Effect of point cloud density

The following figures show the effect of point cloud density on the quality of the reconstructed tree skeleton. Points were removed by means of a minimum distance threshold from a previously dense point cloud to achieve different densities. It can be seen that as density decreases, model accuracy decreases as well. In the images shown, model accuracy remains relatively good, even at the lowest density. In reality however, sparse point clouds will not have points distributed as evenly as in the examples. Sparse point clouds often have regions where no points were detected, and may be more inaccurate in general as well. Additionally, the points missing are often crucial trunk points.



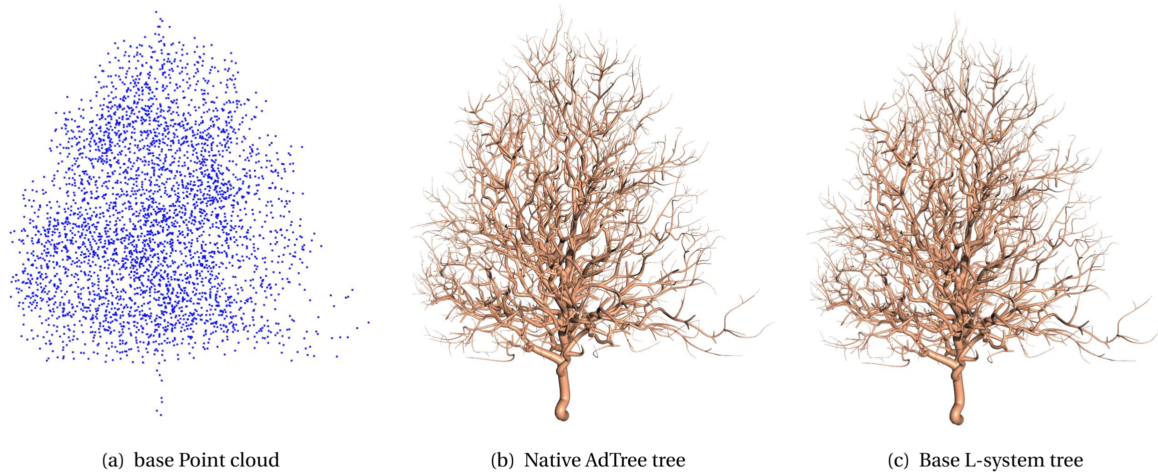
(a) base Point cloud (b) Native AdTree tree (c) Base L-system tree

Figure 4.2: In this figure one can observe the point cloud at its original density. Figure 4.2(b) shows the corresponding AdTree surface reconstruction, Figure 4.2(c) shows that of the L-system extended AdTree.



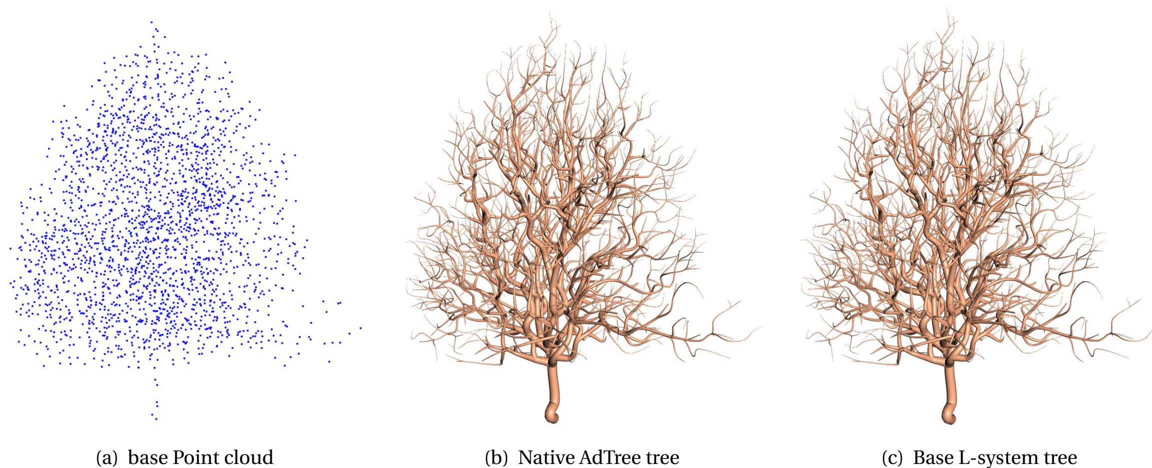
(a) base Point cloud (b) Native AdTree tree (c) Base L-system tree

Figure 4.3: In this figure one can observe the effect of a lower point cloud density (distance threshold = 0.1) on the surface reconstruction of both AdTree and the L-system.



(a) base Point cloud (b) Native AdTree tree (c) Base L-system tree

Figure 4.4: This figure shows an even sparser point cloud (distance threshold = 0.2), and the corresponding AdTree and L-system reconstructions. The main effect seems to be fewer branches, for both reconstructions.



(a) base Point cloud (b) Native AdTree tree (c) Base L-system tree

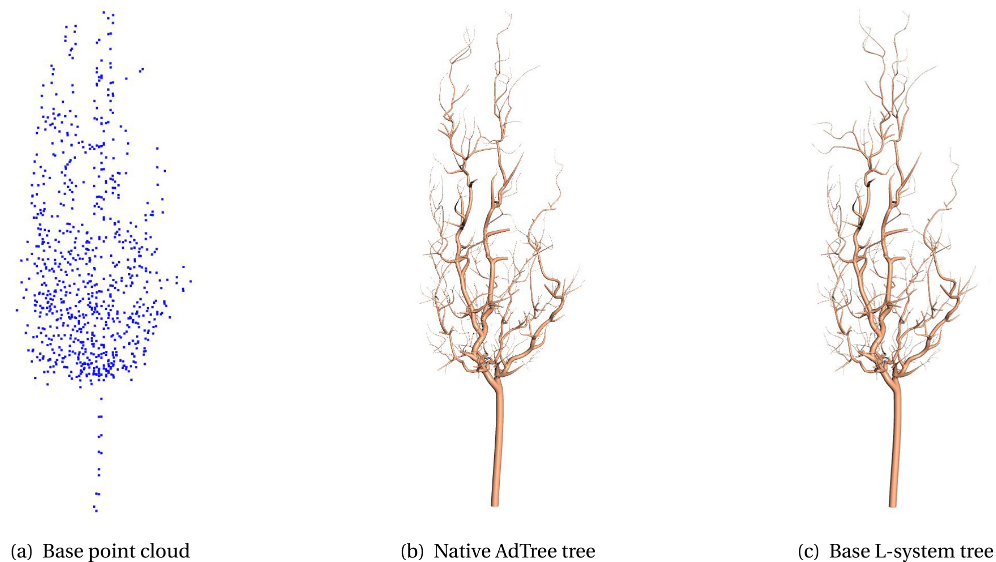
Figure 4.5: This figure shows the sparsest point cloud that was tested(distance threshold = 0.25), and the corresponding AdTree and L-system reconstructions. The main skeleton/structure of the tree is still clearly present.

From the point cloud sparsity comparison (Images 4.2, 4.3, 4.4, and 4.5) it can be seen that the lower the density is, the less accurate the branch tips become. At lower densities, there are fewer branch tips detected overall, tips are more inaccurate, and larger parts of the tree belong to these diverging tips. Despite this, even at the lowest density tested the main structure of the tree remains present. However, as mentioned above, this test does not represent the reality of very sparse point clouds accurately. Even in this test one can already see inaccuracies in the trunk emerge when the density of the point cloud decreases (the "twist" near the bottom, visible both in AdTree and the L-system). The trunk is probably the part of the tree most vulnerable to the effects of sparse input clouds, as it is both often difficult to detect and critically important to the quality of the result. It is therefor expected that in principle low-density point clouds will not cause significant inaccuracies in modelling the main structure of the tree (both with AdTree, and the L-system) - provided sufficient trunk points are present in the data.

Comparing the AdTree and the L-system models, one can see no obvious discrepancies between the two representations. Overall, they are very similar. This makes sense, as the AdTree skeleton is used as input for the L-system. Possible structural inaccuracies between the two representations can only be the result of inaccuracies in the L-system translation process.

4.3. Robustness to data sources

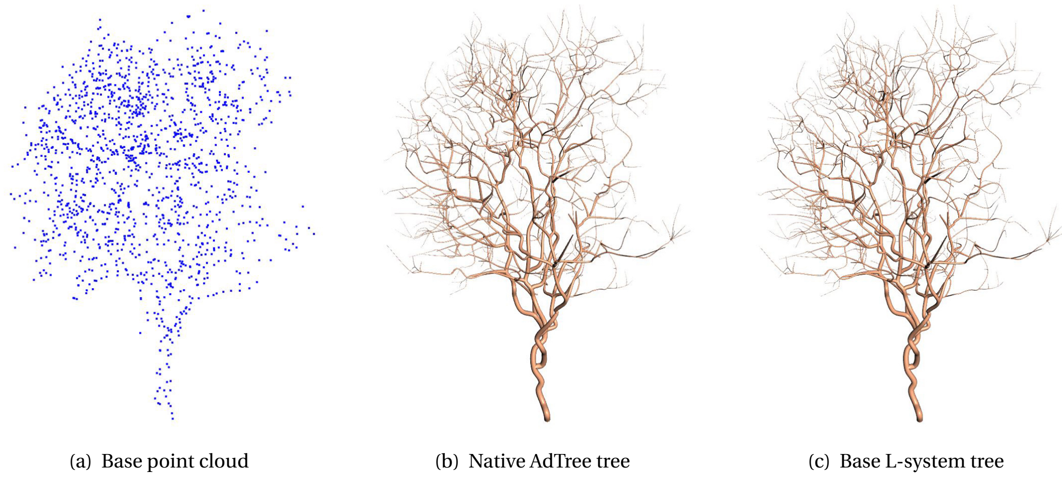
In order to study the effect of input quality on the performance of skeletonization, variations of point cloud input sources were explored as well. The following figures (Figure 4.6, 4.7, 4.8) show the performance of AdTree and the L-system when dealing with data from different acquisition methods. For AHN3 and custom flown LiDAR data, the L-system result was initially faulty. This was due to an inconsistency in the way the cylinder meshes were rendered. The error was fixed by ensuring the direction of the mesh edges was correct. Interestingly, the flipped edges error did not occur for the TLS data, as well as some other custom flown datasets. Once again, it can be seen that the actual structure of the trees is very similar in the AdTree and the L-system models.



(a) Base point cloud (b) Native AdTree tree (c) Base L-system tree

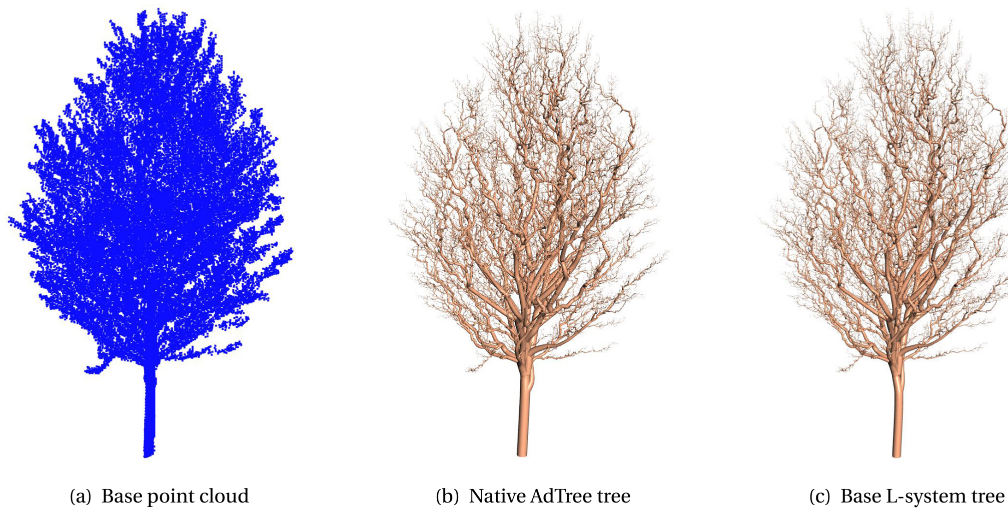
Figure 4.6: The AdTree and L-system surface reconstructions based on a AHN3 airborne LiDAR cloud.

Different methods of acquisition proved to have an impact on the quality of the AdTree and L-system results. The areal LiDAR methods (AHN, custom flown) resulted in much sparser point clouds. TLS LiDAR detected many more trunk points as well, resulting in more defined, more realistic trunks, and a smaller change of the reconstruction failing altogether. This makes sense, as terrestrial LiDAR would have much better access to the lower parts of the tree, be less obstructed by the leaves of the tree in general, and will scan trees from a much closer distance. Both the static and dynamic TLS result in models that contain many dense small branches. As mentioned below, dense point clouds may cause faulty models if they are also inaccurate. This is not the case for TLS, the small branch tips detected seem to follow a realistic image of a tree. How true these small branches are to the structure of the real tree remains however difficult to judge without inspecting the tree in reality. Another interesting point can be seen in the AHN3 model, where an accurate trunk was extracted despite the limited number of trunk points detected. This did not hold true for all AHN3 data, which often has too few trunk points to make an accurate reconstruction of it. However, it is thus shown that reconstruction of trees from AHN3 data using an L-system is possible.



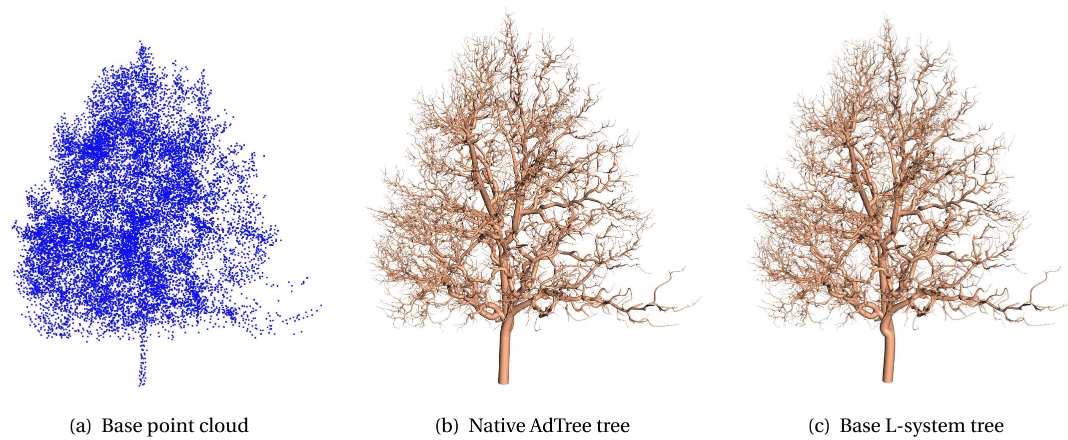
(a) Base point cloud (b) Native AdTree tree (c) Base L-system tree

Figure 4.7: The AdTree and L-system surface reconstructions based on custom flown airborne LiDAR.



(a) Base point cloud (b) Native AdTree tree (c) Base L-system tree

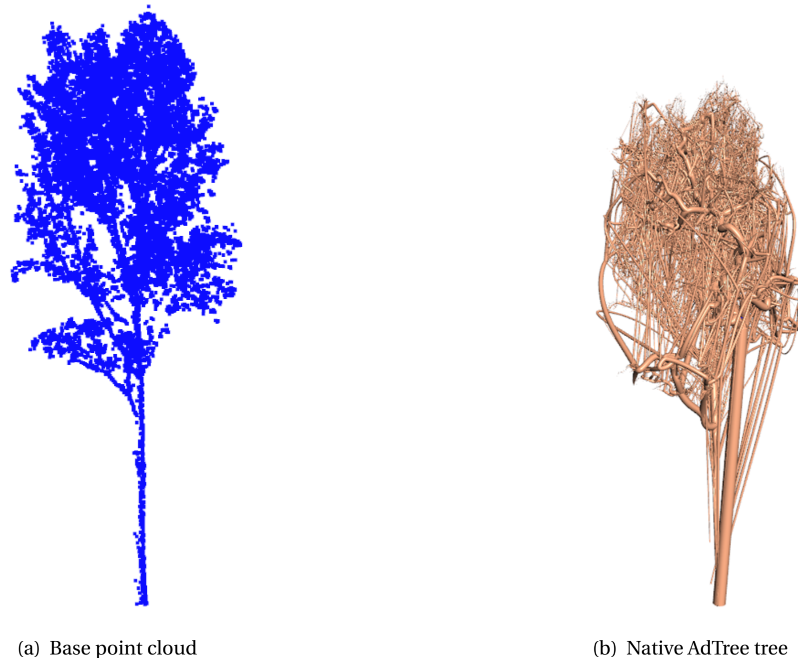
Figure 4.8: The AdTree and L-system surface reconstructions based on static TLS LiDAR.



(a) Base point cloud (b) Native AdTree tree (c) Base L-system tree

Figure 4.9: The AdTree and L-system surface reconstructions based on mobile TLS LiDAR.

Figure 4.10 shows a common error. This error already occurs in the AdTree skeleton (Figure 4.10(b)), and has to do with the density of the point cloud. In this case, the point cloud is too dense, resulting in AdTree not being able to construct clear branches. The main structure is present, but the smaller branches get connected to seemingly random vertices all throughout the model. This issue was common for LiDAR data that was not collected in winter. This data would contain leaves, confusing the program and introducing many unnecessary, non-structural points. A maximum distance filter between two consecutive branch points may help reduce the severity of this error.



(a) Base point cloud

(b) Native AdTree tree

Figure 4.10: Reconstruction error in a tree with many inaccurate or unnecessary branch points.

4.4. Differences in encoded geometry: AdTree output and the L-system

Due to the difference between the storage approaches of the native AdTree (mesh and classical skeleton[1]) and the L-system storage approach, there are chances of introduced inaccuracies and deviations. To examine the presence of inaccuracies and their magnitude, the native AdTree skeleton outputs are compared to their compressed L-system representations. Figure 4.11 shows the pipeline that was used to obtain and examine these representations. It is possible to do this comparison in third-party 3D software due to the fact that AdTree was altered to not only export, but also read/import L-system files. This means the point cloud data can be converted to an AdTree representation, exported to an L-system, imported again into AdTree, and exported to a classical skeleton or mesh file. These re-exported files can be visualized by common 3D viewers. Thus, the limits that the L-system compression presents can be investigated, and the L-system data can be compared to the data before it was converted and stored as an L-system.

[1]A structure is considered a classical skeleton when the nodes are explicitly stored with three coordinates and the edges as the lines between those nodes.

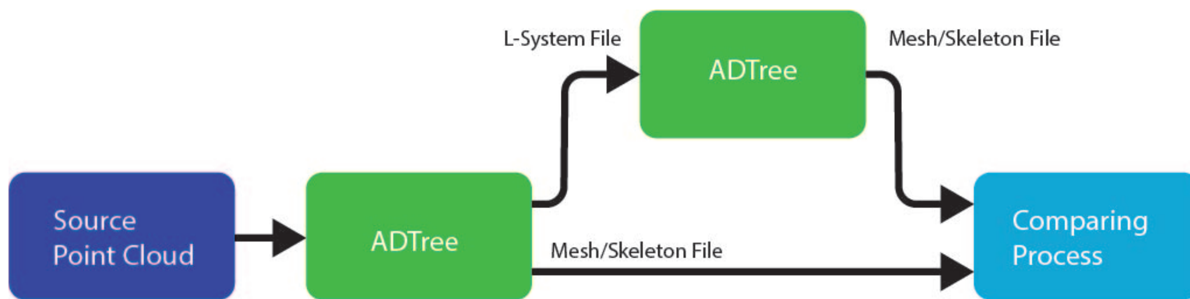


Figure 4.11: Visual representation of the file pre-processing that happens prior to the evaluation of the performance of the L-System files. The pointcloud is loaded into AdTree and exported as an L-system and as a skeleton or mesh. The L-system is again loaded into AdTree and subsequently exported as a skeleton or a mesh. This enables us to compare the L-System files to the "normal" skeleton and mesh files in 3D viewers/modellers.

In order to inspect whether the skeleton generated from the L-system is accurate, first a visual inspection is performed, an example of which can be seen in Figure 4.12. Due to a minor bug in the augmented AdTree code the L-system geometry is translated slightly, resulting in the L-system geometry being placed at a slightly incorrect location[2]. This was manually corrected before the models could be compared. The models were aligned on their anchor point, which corresponds to the location of the root of the tree. This is the only point that is explicitly stored in the L-system, making it the only valid point possible to align both models with.

The visual inspection shows in every comparison set two fairly similar skeletons. In general, no significant anomalies were detected. However, the L-system skeleton's deviation form the baseline geometry becomes larger towards the the tips of the branches, see Figure 4.12. This is due to the fact that the baseline skeleton geometry has been explicitly stored, while the L-system skeleton geometry only has an explicitly stored anchor point. The rest of the geometry is stored relative to the anchor. This means that if the string of steps from the root point towards the tip of a branch is longer, and thus more complex, a deviation from the actual location of the tip is more likely. This is most presumably caused by either small inaccuracies in the creation of the steps and/or by the rounding of values to a specified number of decimals (3 in this case), which is the floating point length in the L-system file. The more steps that are taken, the more effect the introduced error of an earlier step has, and the more rounded steps are stacked upon each other, possibly reinforcing the errors.

One should note that the mentioned loss of accuracy is not persistent in every export to L-system. Only when exporting from the original skeleton file to L-system will there be a loss. An opened L-system can be stored to L-system without an extra loss of accuracy. Thus, an L-system can be opened, edited and stored without the need of a classical skeleton file as a source to keep the accuracy above a certain threshold. Opening an L-system will draw edges and nodes with step and angle sizes adhering to the originally stored rounding accuracy. Thus, when exporting this to L-System again, the computed L-string will already fit this rounding, which means no extra loss of accuracy occurs.

[2]This translation was usually around 0 cm to 10 cm depending on the shape of the crown. A tree with an asymmetrical crown will be more susceptible to this translation bug.

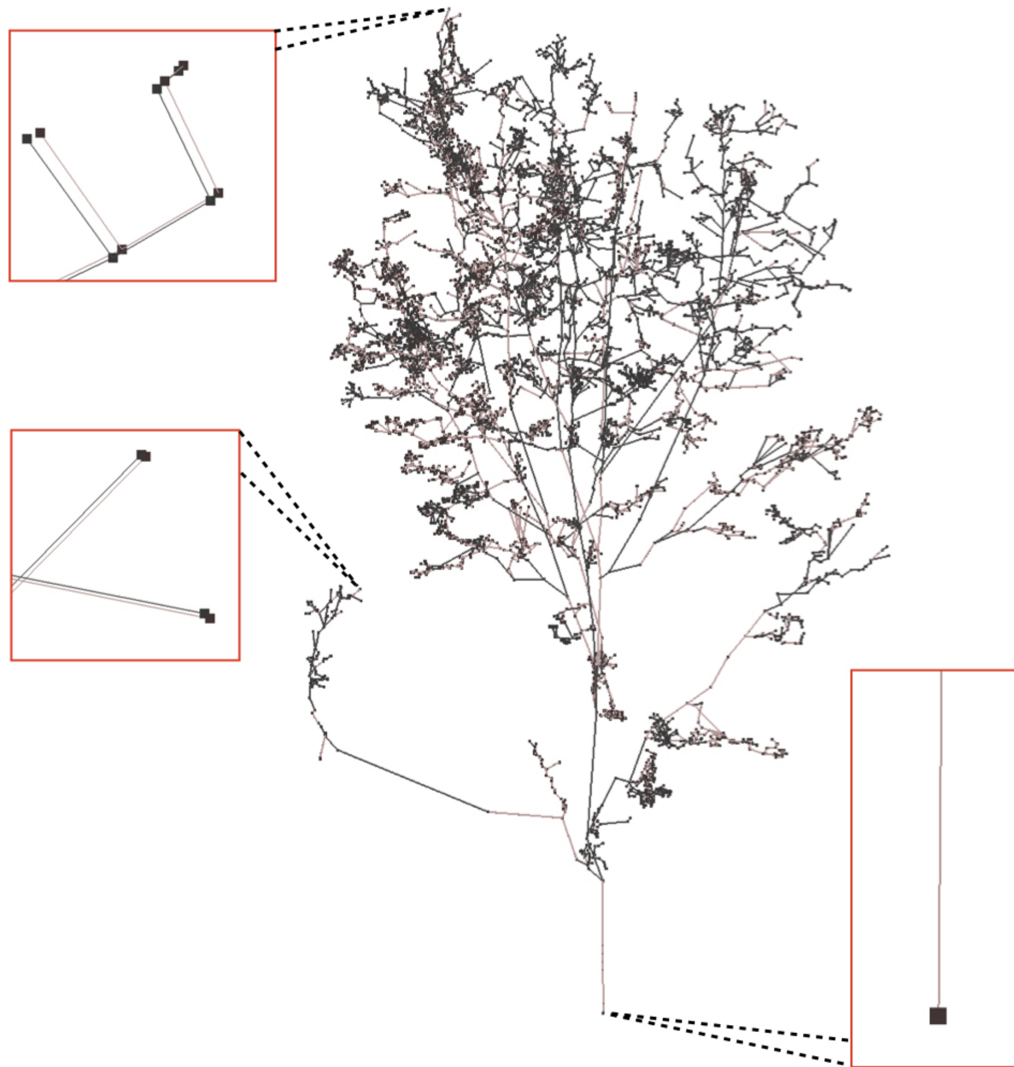


Figure 4.12: The visual inspection of the skeletonization of AdTree versus the L-system approach. The original AdTree skeleton has been colored in black while the L-system skeleton is colored in brown. Both do clearly resemble the same tree, however when taking a closer look the discrepancies at the branch tips become noticeable.

With an altered test version of the augmented AdTree code and a simple Rhino Grasshopper script it is possible to quantify the variations that were seen in the visual inspection. This is done by putting the nodes of both skeleton files in the same order, resulting in two files with matching pairs of points at the same index. With these files the distance of the L-system skeleton nodes to their ground truth location can be computed. The results of this analysis can be found in Table 4.1. The skeletons based on the L-systems do indeed show some deviation from the original input points. This distance is fairly small and does not render the storage method unusable. There are however some large outlier values that can be spotted, see for example the maximum distances of tree1 and tree7 in Table 4.1. The same script was also used to visually display the distance deviations. This analysis confirms the conclusions drawn in the visual inspections: the more steps a branch contains, larger the inaccuracy at its tip.

	Min distance*	Max distance	Median distance	stdev distance
Mobile_tree_1	0,00 ...	0,39	0,11	0,07
tree1	0,05	46,00	0,11	2,43
tree7**	0,00...	299,93/32.11**	1,11	53,74/1,78**
Lille_2	0,00...	5,04	0,13	0,07
Lille_11	0,04	0,05	0,04	0,07
LAS_009***	0,00...	631,99	343,89	99,59

Table 4.1: Statistics of the distances between point pairs from the L-system and non L-system skeletons. Note that these differences are from the by AdTree created skeletons, and not the differences from the real life tree. *: The minimal distance has the anchor/root point excluded, due to that always being 0. **: these threes show extreme outliers, being more than ten times larger than the nearest other distance discrepancy. These values do also not reflect the nature of the tree. The double values in this row reflect the values with the outliers in the set on the left of the slash, and with these extreme outliers excluded from the set on the right. ***: Las_009 is an example where the flip bug occurs fairly near the base of the model, resulting in a major part being mirrored and thus being extremely inaccurate.

The occurrence of some bugs that were overlooked in the visual evaluation could now also be detected. Figure 4.13 displays one of the visual outputs of the evaluation script. It can be seen that in general the distance between a point from the L-System skeleton and the matching normal skeleton is extremely small. The median value is often considerably smaller than 1 cm. However, some major outliers may occur. These outliers are often a bug that occurs in either the translation to or from the L-system format by the augmented AdTree. This bug causes the rotation of a node to be flipped around an axis, resulting in the rest of the branch being the exact mirrored projection of the original one. This is a persistent bug that occurs when during the rotation the length of the Z-axis becomes very close to 0. The number of occurrences of this bug has been reduced considerably since the first encounter, however it has proven to be too challenging to completely remove so far. A close-up of an example of this bug and its effect can be found in Image 4.14. This example is an occurrence close to the tip of the branch and thus not that severe. It can however occur anywhere in the tree, resulting in the possibility of more extreme deviations. Tree LAS_009 is such an extreme case.

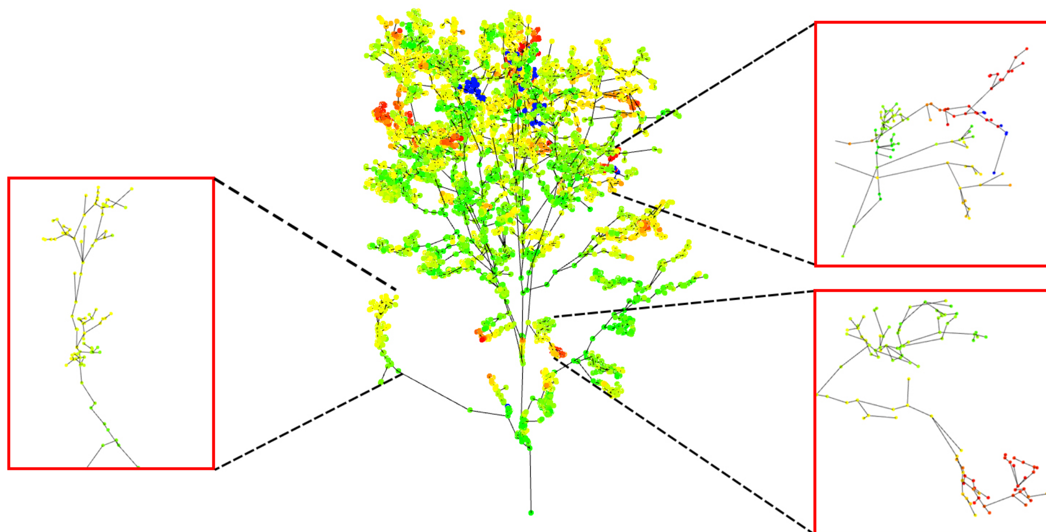


Figure 4.13: The visual output of the Rhino/Grasshopper script. This output displays in color the distance a l-system point is deviated from its location of ground truth point. The gradient goes, in this case, from no deviation (green) to 0,26 cm deviation (red). The blue points are outliers which are have a much larger deviation than the other points.

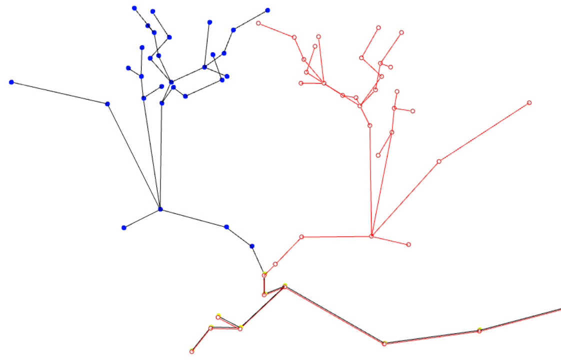
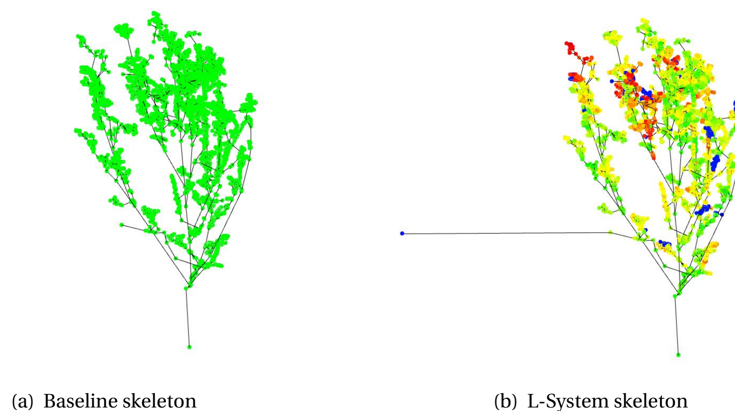


Figure 4.14: A isolated fragment of Figure 4.13 where one of the persistent bugs in the augmented AdTree code is clearly visual. If during the rotation of the turtle the Z-axis becomes extremely close to 0 the turtle executes a flipped rotation. This results in the rest of the subsequent branch becoming an almost perfect mirrored copy from the original. Note how close the original (red) and L-System skeleton (in black/blue) resembled each other before the faulty rotation.

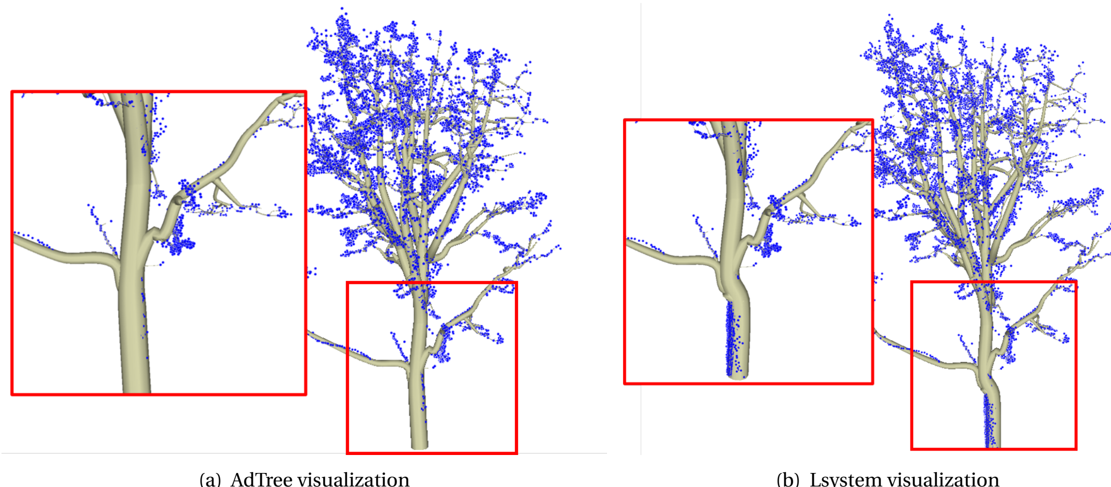
Aside from this issue, another bug has been spotted in the L-System skeletons. This bug causes the node of the skeleton to be translated exactly 299,93 cm from the original location along the vector direction of the edge that precedes it. So far it only occurs in one specific tree model (tree7), where it occurs twice. The results can be seen in Table 4.1 and a visual result can be seen in Figure 4.15. It is unclear what triggers this bug, or how it can be prevented. Currently it only occurs in one test case and can not be recreated in any subsection of this same model, or in any other tested tree model.



(a) Baseline skeleton (b) L-System skeleton

Figure 4.15: The clear isolated outlier (image (b) on the left side of the image) that is created by an L-system translation bug causing the node of one branch tip to be translated 299,93 cm compared to the baseline location. It seems to only occur in this one tree model.

The mentioned similarities and differences between the skeleton of the L-system and the baseline model can also be recognized when visually inspecting the volumetric mesh trees, see Figure 4.16. However, the visual inspections show an anomaly here that was not seen in the skeletons: the bending of main trunk. The section of the trunk starting from the anchor of the tree to the first branching is bent in a manner that does not reflect the mesh of the ground truth. When comparing the skeletons of the L-system and the ground truth, no abnormal deviations that could have caused this are visible, see Figure 4.12 and 4.11. The same bent in the skeleton, which in the L-system's case gets translated into the L-system mesh as well, is present in both the baseline skeleton and the L-system skeleton. Currently it is unclear why this is only translated into a mesh bend in the mesh of the L-system tree.



(a) AdTree visualization (b) Lsystem visualization

Figure 4.16: Comparison between AdTree and Lsystem

Aside from the bending of the trunk, the meshes from the L-system and the baseline model seem to resemble each other very closely. However, the earlier mentioned edge extending bug can also be spotted here. The extending of the branch in skeleton of tree7 thus translates into the resulting volumetric model. Note that this is an issue with the skeleton of the L-system tree: the mesh itself has been created correctly around the skeleton.

With a variation on the earlier used Rhino/Grasshopper script it is possible to quantify the variations between tree meshes, enabling more in-depth analyses. The script projects the centroids of the triangular surfaces of the L-system mesh on the baseline mesh and computes the distances from the original point to the projected one. The results of this script can be found in Table 4.2. This table tells a similar story as Table 4.1. This is because the rules related to the skeleton are true here as well. Depending on the complexity and length of the branch, the mesh will become less accurate the further away the edge or node is located from the root of the tree.

	Min distance*	Max distance	Median distance	stdev distance
Mobile_tree_1	0,00...	16,1	0,37	0,33
tree1	0,00...	13,66	0,36	0,59
tree7**	0,00...	272,40/xxx	0,18	13,00/xxx
Lille_2	0,00...	28,38	2,27	1,37
Lille_11	0,00...	8,86	0,20	0,24
LAS_009***	0,00...	44,07	5,00	5,05

Table 4.2: Statistics of the distances between point pairs from the L-system and non L-system mesh. *: The minimal distance has the anchor/root point excluded, due to that always being 0. **: these trees show extreme outliers, being more than ten times larger than the nearest other distance discrepancy. These values do also not reflect the nature of the tree. The double values in this row reflect the values with the outliers in the set on the left of the slash, and with these extreme outliers excluded from the set on the right. ***: Las_009 is an example where the flip bug occurs fairly near the base of the model, resulting in a major part being mirrored.

The Rhino/grasshopper script also allows one to visually display the values to see where the largest discrepancies are located. These visuals, see Figure 4.17, show new issues that were not spotted when visually comparing the models.

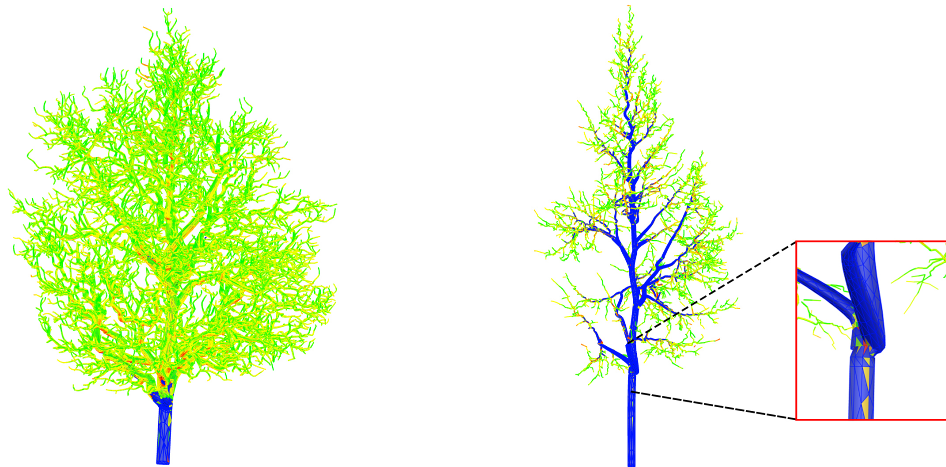


Figure 4.17: Two visual outputs of the Rhino/Grasshopper script. The scale goes from no deviation in green to large deviation in red. The blue locations are outliers which generally have a deviation that is 10 times or larger than the rest of the areas. The left tree shows that although the main trunk is not bend, it still is inaccurate. The tree on the right shows the result of a bend in the trunk (in this case the trunk even folds in upon itself), which results in the rest of the branches being too narrow.

The earlier spotted skeleton bugs are again recognizable in the mesh: the extension bug, and the mirroring bug as seen in Figure 4.14. Additionally, there are two new issues. First, the trunk seems to always be inaccurate. Even when it visually resembles the model very closely, the trunk is often the least accurate area of the tree. This is remarkable due to the close proximity of this part of the tree to the only explicitly stored vertex location. However, these deviations are not so severe that the model should be deemed faulty. The second issue can be considered more critical. In some cases the whole tree, aside from the end areas of the branches, is inaccurate. This occurs only in tree meshes that display fairly major trunk bending. A potential explanation for this issue is that due to a relation between the trunk bending and the fashion in which AdTree calculates branch radii, the mesh branches are created too thin. This averages out towards the end of a branch, resulting in the script registering more accurate tips than the rest of the tree model. This is however not a new bug, but an artifact from the earlier mentioned trunk bending bug, see Figure 4.17.

4.5. Differences in encoded geometry: AdTree output and the CityJSON

The CityJSON file encoding of the trees is more closely related to the classic skeleton and mesh storage methods, but it is still different. Thus, there are chances of introduced inaccuracies and deviations. Similar to Section 4.4, the native AdTree outputs (as ground truth) are compared with the CityJSON representation to examine the presence of inaccuracies, and their magnitude. Figure 4.18 shows the pipeline of this comparison. It is possible to do this in Rhino3D with the help of a Grasshopper script that opens and transforms the special CityJSON trees to mesh and skeleton objects.

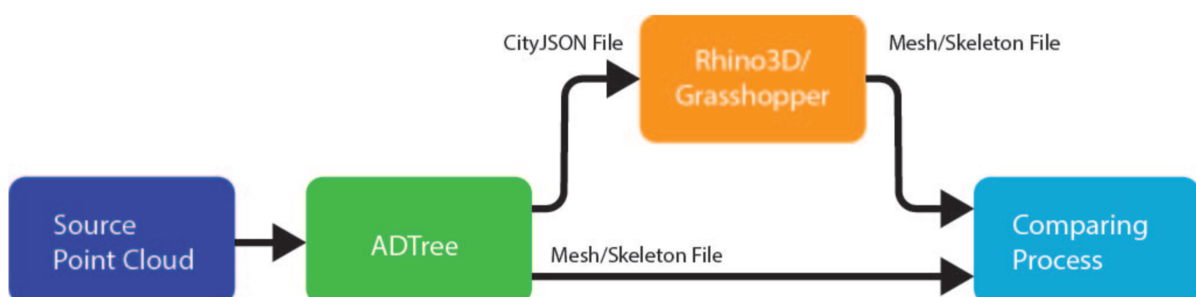


Figure 4.18: Visual representation of the file pre-processing that happens prior to the evaluation of the performance of the L-System files. The pointcloud is loaded into AdTree and exported as a cityJSON and as a skeleton or mesh. The cityJSON is loaded Rhino via a Grasshopper JSON reader transforming it back to a skeleton or a mesh. This enables us to compare the L-System files to the "normal" skeleton and mesh files in 3D viewers/modellers.

Like traditional skeleton formats, the CityJSON explicitly stores the coordinates of every node of the skeleton. This results in a skeleton file stored with pinpoint precision. The visual inspection shows no deviations, anomalies, or bugs. With the help of the same grasshopper script used in Section 4.4 it is also possible to quantify any deviations between the baseline and the CityJSON skeleton that may occur. The outputs show that there is no deviation between the two. They are, aside from the order of the edges and nodes, carbon copies of each other.

However, when visually comparing the mesh of the CityJSON and the native AdTree output, a distinct difference can be seen. The CityJSON based mesh is both unsmoothed and seems to be generally a lot thicker than the native AdTree output mesh, see Figure 4.19. However, both meshes follow the same structure, and no extreme outliers or anomalies can be spotted. Both meshes clearly resemble the same tree, where the CityJSON based model seems a more unsophisticated representation of it.

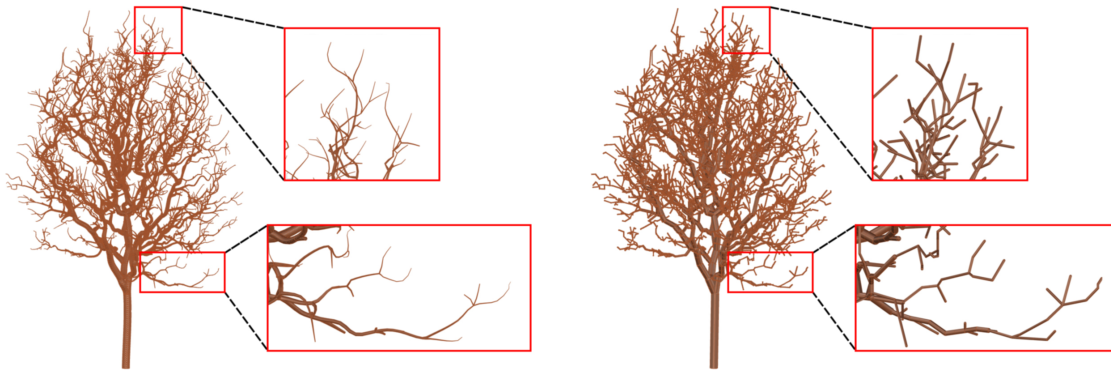


Figure 4.19: An example of a native AdTree mesh output (left) and a CityJSON output (right). It can clearly be seen that the AdTree mesh is more sophisticated than the CityJSON generated mesh. Note that the differences between both the meshes is mostly related to the way the CityJSON is read and not due to the actually encoded data.

It is presumed that this is not directly related to the way the data is stored in the CityJSON, but more to the way it is read by Rhino/Grasshopper. With the current state of the custom CityJSON tree writer, meshes are directly created by sweeping pipe shapes over the encoded skeleton. This skeleton is not smoothed prior to the sweeping process, resulting in jagged branches. The sweeping of the pipe-like shapes is done with a single radius per line section, resulting in clear steps in width, with no tapering. This lack of tapering also results in the tips of the branches being a lot wider than the AdTree native mesh.

With the same Rhino/Grasshopper script that was used in Section 4.4 to compare meshes, one can again quantify the variation between the tree meshes. This allows for a more in-depth analysis and a comparison with the performance of the L-system storage approach. The results can be seen in Table 4.3. This table lays bare some interesting properties of the CityJSON mesh. The CityJSON mesh, with its current reader, is more accurate than the L-System mesh. Due to the explicit encoding of the CityJSON the skeleton is stored very accurately, resulting in the mesh being less susceptible to bugs and deviations. An example of this is the mesh tree7 and tree LAS_009, where the L-system showed a significant error, the CityJSON does not.

	Min distance*	Max distance	Median distance	stdev distance
Mobile_tree_1	0,00...	14,96	1,00	1,13
tree1	0,00...	9,49	1,82	1,00
tree7	0,00...	15,86	0,84	0,52
Lille_2	0,00...	29,40	3,17	2,11
Lille_11	0,00...	7,72	0,91	0,70
LAS_009	0,00...	6,37	0,65	0,08

Table 4.3: Statistics of the distances between point pairs from the CityJSON and the native AdTree mesh. *: The minimal distance has the anchor/root point excluded, due to that always being 0.

The visual output of the testing script confirms this deduction. For example, the model of Lille_11 shows some deviation from the ground truth, see Figure 4.20 (right image). This deviation however does not propagate to

the following branches, due to the partially explicit storing of the mesh. This is unlike the L-system deviations of the same tree, see Figure 4.17 (right image). Due to the almost completely implicit storing of the mesh in the L-system approach, an error in the trunk can propagate through major parts of the tree.

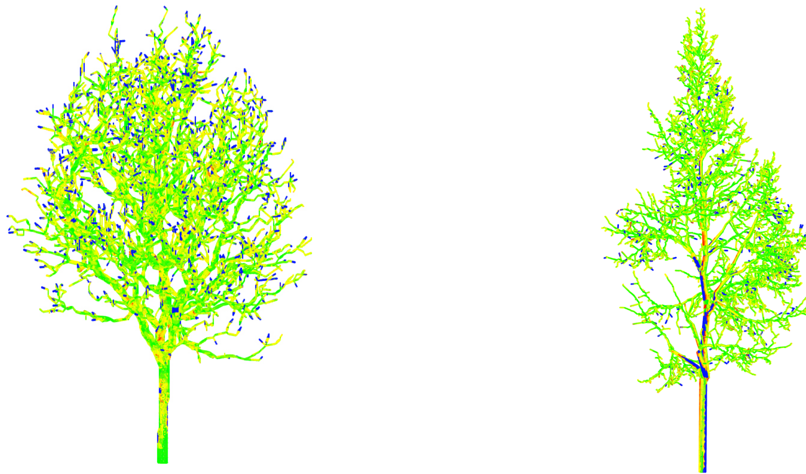


Figure 4.20: Two visual outputs of the Rhino/Grasshopper script. The scale goes from no deviation in green to large deviation in red. The blue locations are outliers which generally have a deviation that is 10 times or larger than the rest of the areas. The trucks and some main branches have noticeable difference from those in AdTree-created meshes. And the tips also have high deviations.

Unfortunately, it is challenging to compare the ground truth, CityJSON and L-system performance in more depth due to the limited CityJSON reader. Presumably, the CityJSON model would perform better than the L-System in terms of accuracy, at the cost of using more storage space, see Section 4.8. This is due to the more explicit storage method that is used in the CityJSON. This way of storing data will however also limit the possible further computations that would be possible with an L-system, see Section 4.6 and 4.7. However, the CityJSON would be more easily readable by a selection of software packages, due to already relying for a large part on an existing infrastructure. The L-System tree model infrastructure that this L-system implementation requires is still in the early stages of development.

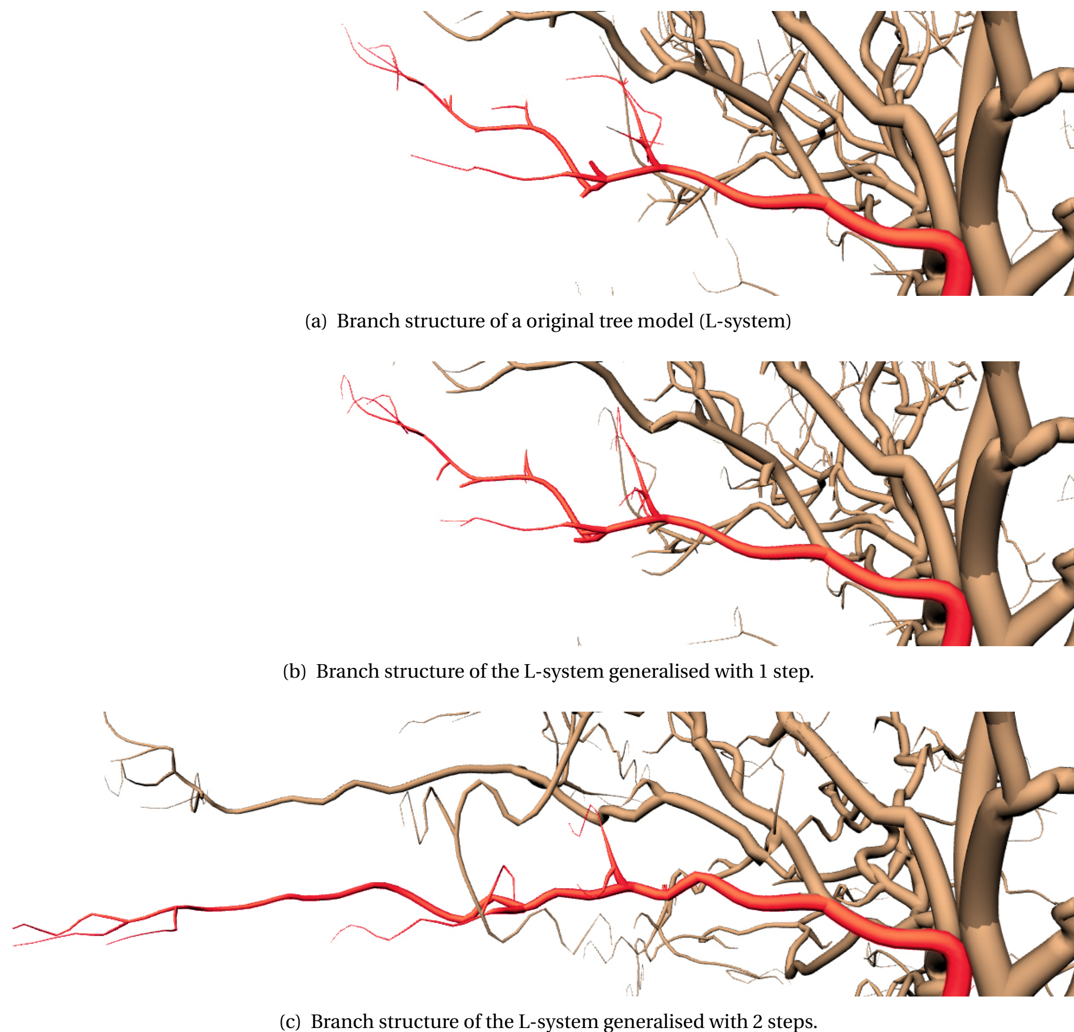
4.6. Branch tip generalisation

Figure 4.21 shows a closer look at the effect of generalising the branch tips of a tree model. The figure shows the original tree model and the effect of generalising with both 1 and 2 steps. it can be seen that generalising with 1 step barely has any influence on the quality of the model. However, the L-system contains far fewer characters, while describing the same tree model. It is therefor more compact. Generalising 2 steps of the branch trees does lead to some more inaccuracies, such as the zig-zagging pattern visible at some tips. This pattern is likely caused by the particular nesting structure at these tips, where several sub-branches with only 1 edge until the leaf node sprout from a consecutive structure. Generalising with 2 steps will average these tips in an overlapping way, as some branch endings are only 1 step long, but get averaged over 2.

This effect is not very problematic at 2 steps of generalisation, as it mostly affects the shortest branch endings and not the main structure of the tree. It has proved more of an issue at further generalisation steps (Figure 4.22 and 4.23), although at 2 steps it likely caused the missing lowest branch of the tree in Image 4.21. Larger branches suddenly pointing in a wrong direction, which is what happened in this case, is likely the result of generalising branch tips that are shorter than the amount of steps to be generalised. If a branch tip of length 1 sprouts from the lower part of a large branch, an edge in the main structure of this branch will be incorporated into the generalisation as steps past the actual branch tips are generalised. The generalisation thus "moves" down the main structure of a branch, and can cause a bigger branch to change, and with it the rest of the branch changes direction as well.

The overall structure of the tree, however, is still well defined at 2 steps of generalisation. It is very similar to the original and the 1-step generalisation models. Accepting some inaccuracies around some branch tips,

generalising 2 steps could also be an acceptable method of compressing the model further.



(a) Branch structure of a original tree model (L-system)

(b) Branch structure of the L-system generalised with 1 step.

(c) Branch structure of the L-system generalised with 2 steps.

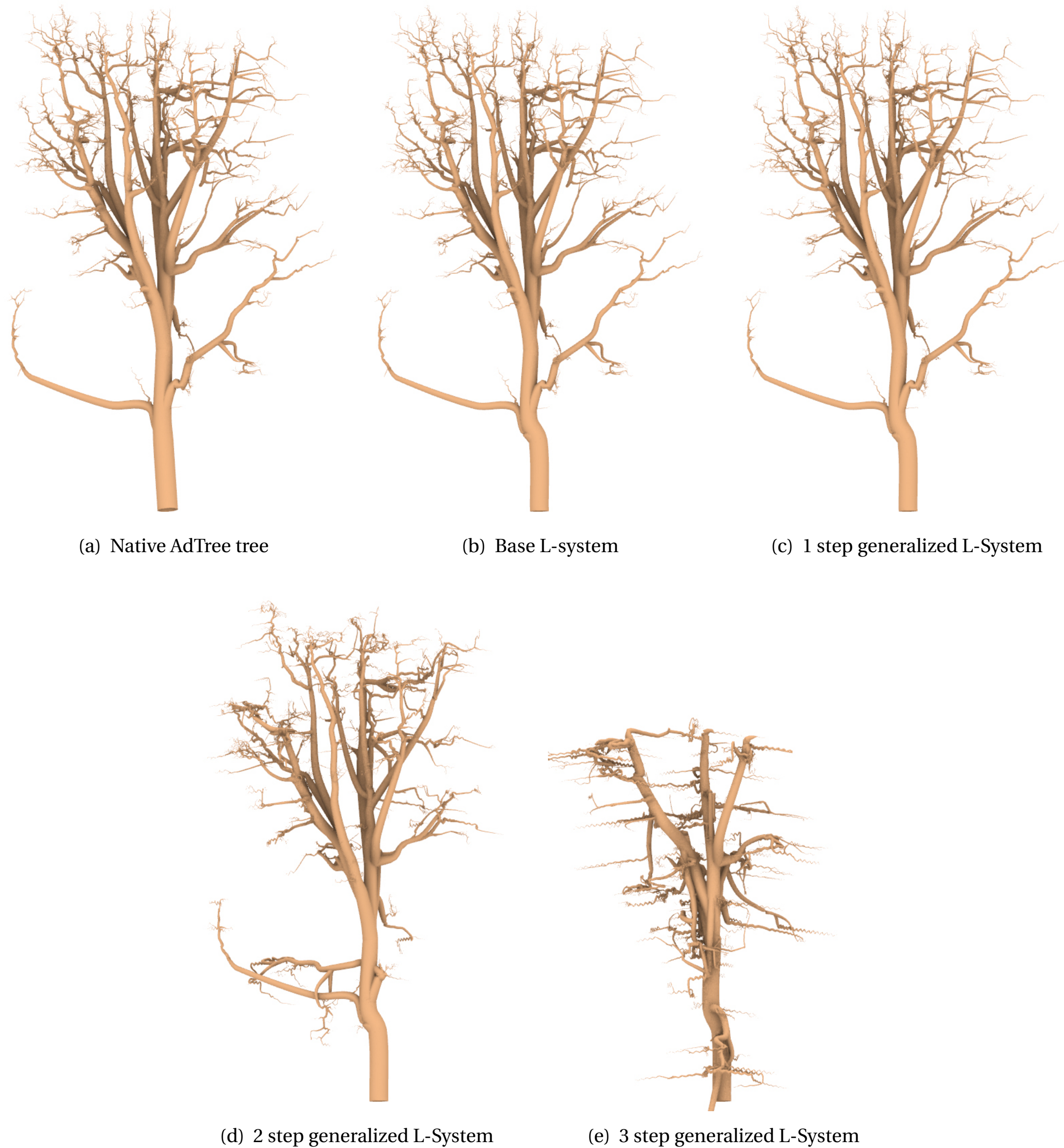
Figure 4.21: Effects of different amounts of steps of generalisation, one branch has been highlighted for easy evaluation. The main structure remains clear, while the branch tips are averaged and represent the actual tree model less accurately, but more compactly. The model does however lose accuracy noticeably at 2 steps, although the size of this effect varies widely from tree model to tree mode.

	Min distance mesh	Max distance mesh	Median distance mesh	stdev distance mesh	L-system axiom length
L-System	0,00	13,66	0,36	0,59	199 170
L-System step 1	0,00...	16,42	1,11	0,96	136 423
L-System step 2	0,00...	90,70	12,42	10,68	77 123
L-System step 3	0,00...	140,89	12,35	24,17	42 627

Table 4.4: Comparison of the (generalized) L-system meshes compared to a native AdTree mesh as ground truth. An added variable is the length of the axiom of the L-system.

In Figure 4.22 and 4.23, a visual comparison has been made for the generalisation option of the entire tree model, for two different input trees. The most important parameter to consider is the number of steps to generalise, which is a user parameter. Generalisations have been visualised for 0 (original tree), 1, 2 and 3 steps. It can be clearly seen that this parameter should not be set too large, or an unrealistic tree model will be the result. For both trees, generalising with 1 or 2 steps results in a model similar to the original tree, but generalising with 3 steps results in odd angles and clearly wrong branch directions. This can be explained by the fact that the more steps are generalised, the more relevant branch segments are included

in the generalisation. Which branch segments are relevant will depend on the overall size of the tree: larger trees will have more and larger branches, and thus their tips will relatively be less important. In smaller trees, branch tips are more relevant, and should be generalised less. The accuracy of the generalised model will also depend on the overall similarity of the branch tips. The more similar they are, the better the averaged representation will approximate them. A tree with very similar branch tips could perhaps be generalised further than a tree with very dissimilar tips, as similar steps carry less importance.



(a) Native AdTree tree (b) Base L-system (c) 1 step generalized L-System

(d) 2 step generalized L-System (e) 3 step generalized L-System

Figure 4.22: Comparison of generalisation with different amounts of steps to generalise, tree 7.

Efficiency wise, generating a generalised model with 1 step for both trees took about 30 seconds. Generating a model with 3 steps took about 2 minutes. In principle, this does not seem like a long processing time, but considering the huge amount of trees in the client's database, this becomes an important factor to consider. Lastly, one should consider minimum branch length. As of now, no check is performed to ensure branches are not generalised more steps than they are long. Generalising branches further than realistically possible will result in invalid geometries, and invalid L-system files. This should be prevented, either by user inspection, or ideally as a possible future improvement in the code itself. In conclusion, considering also the observations regarding compactness in the beginning of this chapter, in general a tree model could be generalised at least one or two steps. Provided processing time, minimum branch length, and branch importance are taken into account, generalising the L-system model will result in a much more compact storage option, while preserving enough accuracy to present a realistic model of the tree.

(a) Native AdTree tree (b) Base L-system (c) 1 step generalized L-System

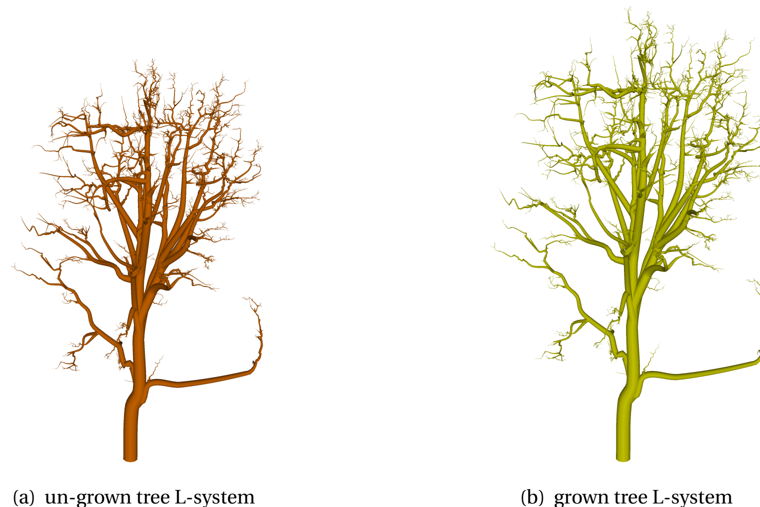
(d) 2 step generalized L-System (e) 3 step generalized L-System

Figure 4.23: Comparison of generalisation with different amounts of steps to generalise, Ahn tree 9.

4.7. Simulated growth function L-system

To study the way the growing function actually grows the tree model, the non grown L-systems are compared to grown L-systems. The native AdTree output could also have been chosen to be compared to the grown L-system, however this would have made the analysis more challenging. There are differences between the L-system and the native AdTree output which could be mistaken for differences caused by the growing process.

The normal L-system is compared with the default setup growing L-system in Figure 4.24. Its challenging to compare the two due to the growing L-system changing multiple elements of the tree, like branching, trunk radius and size. However it is clear that both the L-systems closely resemble each other and aside from the scaling it is believable that both are the same tree with a temporal difference. The scaling seems to be too large for the minor changes to the rest of the tree, it has a 3D modelling software uniform scale command feeling. Aside from this unnatural scaling there are no artifacts of the growing process that change the tree in a drastic manner that seems incorrect on the first glance.

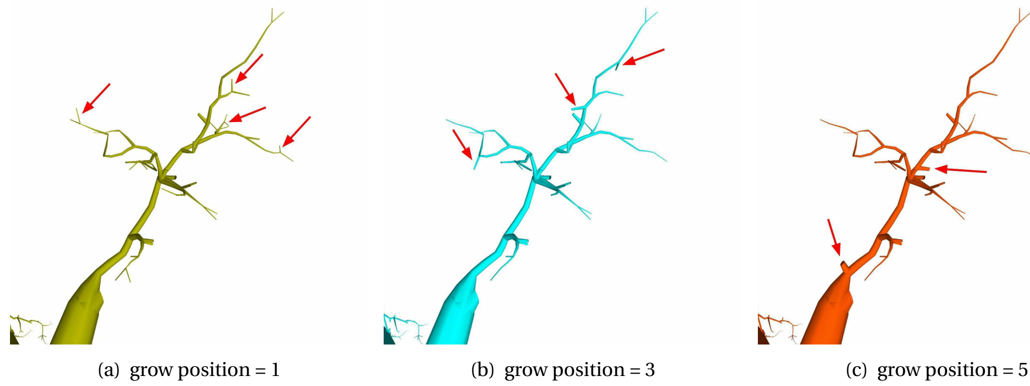


(a) un-grown tree L-system (b) grown tree L-system

Figure 4.24: The difference between the un-grown tree based on the normal L-system and the grown tree based on a grow L-system with the default settings

To allow a more in depth analysis of the growing process the effect of the different growing parameters is studied in a isolated way. This isolation is done by manually changing one parameter while setting the others to default (value = 0). This way any effects of the parameter can be identified and studied. For every variable three outputs have been made, one with the minimum possible value, one with the mean value, and one with the highest possible value. The order of the growth parameters is not based on importance but based on the order at which they are mentioned on the GUI. Thus, sprout position, (basic) branch grow speed, growth ratio and grow coefficient.

The sprout position can be set to a value from 1 to 5 (0 = the default value), and the effects it has can be seen in Figure 4.25. The sprout position parameter dictates how close to the tip of a branch a new branch can sprout. With a high value, 5 for example, branching occurs 5 nodes from the tip of the branch. In 4.25(a) this can be seen by the distinct V like shape this causes at the tip of the branches. These do not occur in cases where the growth position value is higher.



(a) grow position = 1 (b) grow position = 3 (c) grow position = 5

Figure 4.25: A fragment of a tree model where the different results are based on the grow position parameter can be seen. The difference in branching have been highlighted with red arrows

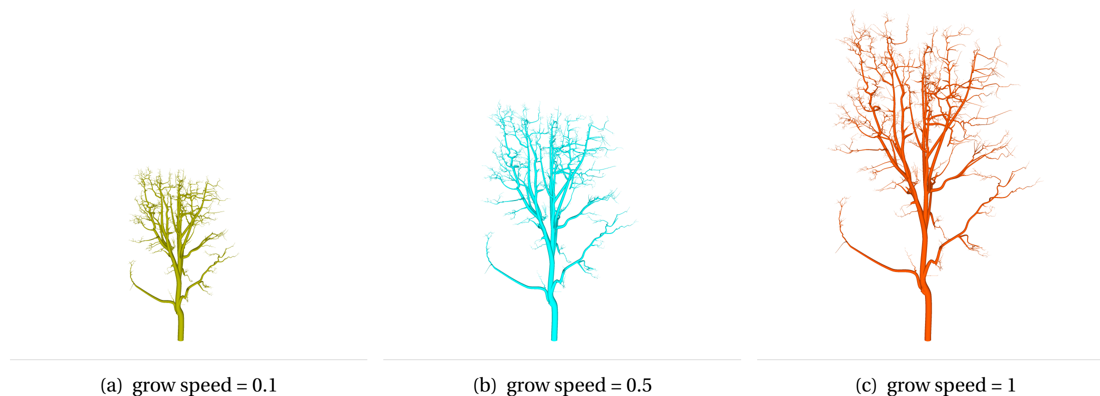
An added rule to the sprout position code causes the sprouting not to happen in cases where the node it would sprout from already had a fork. No nodes can thus be forking in threes due to the created growing process. This has the added effect that the sprouting process feels more natural, even though the sprouting position is set to 3, not every third node from the tip is spit. This means that the tree will not get a mechanical feeling, split at every branch at the same location. This mechanical feeling is also avoided due to the model of the tree itself, edges between nodes have different lengths avoiding a homogeneous forking distance from the tip. The only mechanical feeling sprouting is with the sprout position 1, which creates distinct v like shapes at the tips.

sprout position	1	2	3	4	5
New vertices count	958	611	367	234	157

Table 4.5: The amount of new vertices that are added to the skeleton per sprout position value on the tested tree model

Aside from the more natural feeling sprouting positions with a higher parameter value this parameter also causes there to be less sprouting in total. The trees this has been tested on had, in general, less naturally present forking in the first and second position than in the third to fifth one. Less sprouting in total thus occurred in the situations where a larger setting was chosen. This phenomena can also been seen in table 4.5.

The (basic) branch growing speed can be set to a value from 0.1 to 1. This dictates the length that every edge can grow. The results can be seen in Figure 4.26.



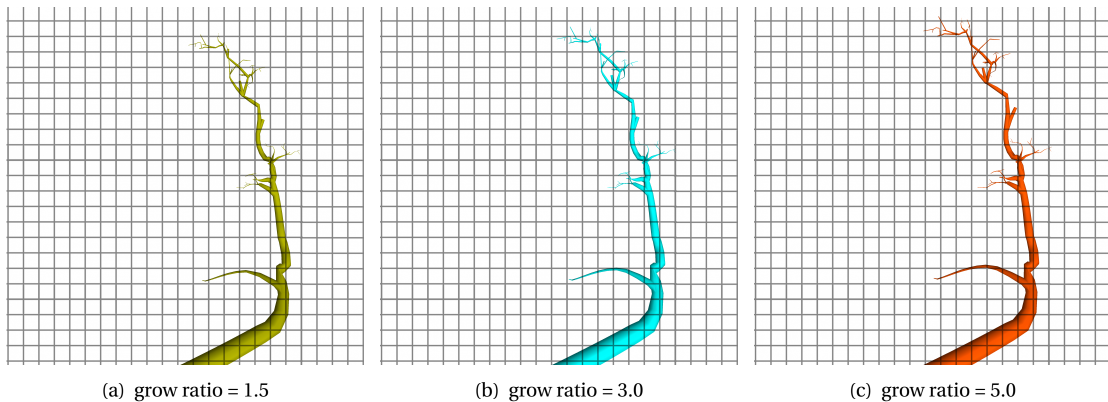
(a) grow speed = 0.1 (b) grow speed = 0.5 (c) grow speed = 1

Figure 4.26: Three grown tree models based on the same tree starting model. The different results based on the grow speed parameter can be clearly seen.

It becomes clear that the variable does not only have an effect on the branches, but also on the main trunk, making the whole tree grow. This is something a real life tree might not do in a similar manner. The effect

of this parameter seems to closely resemble a uniform 3D scaling operation in 3D modelling software. The growing speed variable can be seen as the scaling value ranging from 1.1 to 2 times as large as the original. Only using this variable to grow a tree would yield vastly unrealistic results, as can be seen in Figure 4.26.

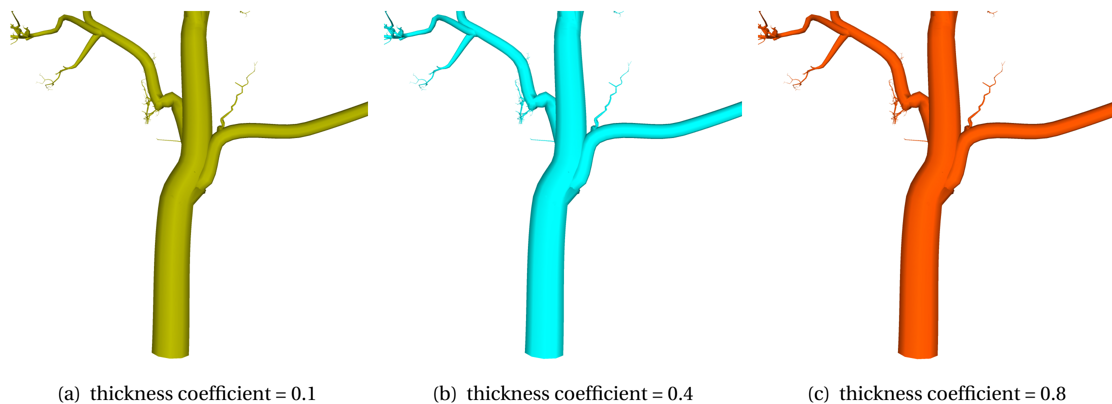
The growth ratio can be set to a value of 1.5 to 5. It is a more nuanced variant of the (basic) branch growing speed where the growing will have more effect the closer to the tip of a branch it is. The results can be seen in Figure 4.27.



(a) grow ratio = 1.5 (b) grow ratio = 3.0 (c) grow ratio = 5.0

Figure 4.27: A fragment of a tree model where the different results are based on the grow ratio parameter. The images have been overlaid with a grid to clarify the differences.

How nuanced this effect can be becomes clear when comparing the different situations created by the different set values. To clarify the figure a grid has been added. With the help of this grid it can be seen that close to the base of the tree the growing is very limited, while the closer the tip is approached the bigger the growth is. This growth feels natural and removes the uniform scale like behavior the (basic) branch growing speed introduces in the process.



(a) thickness coefficient = 0.1 (b) thickness coefficient = 0.4 (c) thickness coefficient = 0.8

Figure 4.28: A fragment of a tree model where the different results are based on thet thickness coefficient parameter

The grow coefficient can be set to a value from 0.1 to 0.8. This dictates the width the trunk gains when growing. The results of this can be seen in Figure 4.28.

The differences in trunk width are challenging to see in the figure, however they are present. The increase in thickness is also propagated into the branches. This means that not only the main trunk gains girth but also the branches. The girth drop off is more severe when a bigger grow coefficient is picked, so the tree becomes more bottom heavy. this is due to the fact that a bigger radius is spread out over the length of the skeleton. Following from the root of the tree, through the trunk to one of the branches the wider radius has to be divided (over different branches) and lessened. The length of the branches is not enlarged by using this variable, so this process needs to divide and lessen the enlarged girth over a non enlarged distance.

4.8. Effect of file formats on storage efficiency

The efficiency of the different storage approaches is crucial. The aim is to be able to compress as much information in small file sizes without losing a consequential amount of accuracy. This would allow easy and accurate storage of the forest inventory of Cobra-Groeninzicht, which contains around 100 million trees. In table 4.6 the file sizes of four storage mediums are compared: The lsJSON, the generalized LsJSON, the PLY skeleton, and the OBJ tree mesh. The file size of the source point cloud .xyz file is also mentioned. It is important to note that file size is not a direct measure of format compactness, but rather an indication. File size also depends on other factors such as encoding and the number of decimals. All files used were encoded in ASCII, and all numbers (except the input point cloud files) have been rounded to three decimals.

	Source Point Cloud (.XYZ)	L-System (LsJSON/.JSON)	Generalized L-System (LsJSON/.JSON)	CityJSON (.JSON)	Classical Skeleton (.PLY)	Tree mesh (.OBJ)
Mobile_tree_1	893 kb	392 kb	260kb	3848 kb	530 kb	30 528 kb
tree1	245 kb	200 kb	137 kb	1922 kb	225 kb	11 674 kb
tree7	316 kb	195 kb	132 kb	1871 kb	221 kb	11 264 kb
Lille_2	1572 kb	537 kb	376 kb	5295 kb	734 kb	46 680 kb
Lille_11	1037 kb	353 kb	257 kb	3448 kb	475 kb	22 132 kb

Table 4.6: The sizes of the files created from the by AdTree created data. The source point cloud is the cloud that AdTree uses to generate the tree data, the following columns are different storing approaches to store this data. The gray marked cells represent non volumetric data, while the white cells encode volumetric data.

The smallest way to store a tree model is as a generalized L-system. This is not only the smallest method tested, but also considerably smaller than the source point cloud itself. However, as mentioned before, the generalization is an extra compression method that may reduce detail in the model. If this is not desirable, a normal L-System could be a better storage medium. The resulting files are still smaller than the source point cloud size, yet encode all the data that AdTree generated in a fairly lossless manner. While the amount of compression is negligible in small point clouds, once the point cloud file's size become larger the compression that the L-System provides becomes more noticeable. Another factor in the amount of compression the LsJSON format is able to provide is the complexity of the tree. An example of how tree complexity influences the resulting LsJSON file size can be seen in table 4.6: mobile_tree_1 is a cloud that occupies less storage space than the cloud of Lille_11, but when the data is stored as an L-system it is the other way around.

The L-system files occupy less space than the classic skeleton files, while encoding more information. The skeleton files only store the edges and nodes of the structural tree model, without any added volumetric data. The skeletons also do not support the additional operations that are possible with L-systems. The kind of data the L-systems contain is most comparable to a tree mesh. Although a mesh will also not support the L-system operations, it still is a volumetric representation of a tree. The L-system files are considerably smaller than the mesh files. Often the difference is in the magnitude of 80 times or bigger. However, a difference to keep in mind is that mesh files are supported by a large set of software packages, while the L-system files have to be converted to a more commonly used mesh file (obj) to be used in those packages.

When comparing the storage approaches mentioned in 4.6, a distinction needs to be made in the amount of information they store. For example, the skeleton files do not encode any volumetric data like the branch diameter. So, although the skeleton file is considerably smaller than the source point cloud and the CityJSON, it does not contain the same wealth of information, making a direct comparison invalid.

The CityJSON format falls in between the L-System and the tree mesh in terms of compression. It encodes the same volumetric data as the L-System and the mesh format do, but it does not allow for the L-System specific operations. Although both approaches utilize the .JSON file format, they store data in a distinctly different manner. The L-system only stores a couple of attributes explicitly, while the majority of the data is stored implicitly in a string. In contrast, the CityJSON format stores the geometry and every single attribute explicitly. It may seem like a CityJSON file would thus be easier to transfer between machines, but all recipients are required to install specialist software to visualize and use the data stored. However, the same can be said about the custom L-System format. A major difference is that CityJSON has packages and plugins available in software like QGIS and FME. However, these are currently not able to read the attributes of CityJSON trees.

5 Conclusion

The objective of this project was to create a pipeline that can take open data and the data collected and owned by Cobra-Groeninzicht and generate accurate tree models that are stored in a compact manner, possibly as an L-system. This would not be a completely newly created pipeline, but would augment already available tools, like AdTree and Cobra-Groeninzicht's pre-processing algorithms.

The JSON format was selected to store the results, since it could contain additional information about the tree geometry, and is designed for ease of use. The CityJSON format was considered as an alternative to the custom L-system JSON format, which would allow for a complete 3D geometry of the tree to be stored, and provides an interoperable structure as well. The L-system JSON, however, proved to be more compact. It is of much smaller size, and stores geometry information indirectly instead of explicitly. The L-system format does not store more than the relative path between consecutive nodes and a few parameters for the turtle to use in order to read the tree and construct the mesh model from it. In order to store tree models in L-system format, a graph structure (the skeleton) from the existing proprietary software AdTree was first used as input. From this skeleton the nesting, relative rotation angle, roll angle, and forward distance between all nodes of the graph were computed and translated to the L-system format. The L-sting contains all these motions, and could be read by the LsTurtle and transformed into both a structural skeleton graph and a geometric tree mesh model. The geometry could then be visualized with the built-in 3D viewer of AdTree, as well as exported to .ply and .obj format to be visualised using more standard 3D modelling software.

To assess the quality of the results, the AdTree skeleton, which was used as input, was compared with the skeleton as outputted by the Turtle reader. The mesh models resulting from both skeletons were also compared. The structural model of the L-system output and the original AdTree model were very similar, both for all tested input point cloud densities and acquisition methods. The inaccuracies found were larger at the tips of the tree branches. This can be explained by the fact that the L-system notation uses relative movement. An inaccuracy at the start of the graph will accumulate as the process is repeated and further movement is built upon it. Barring a few detected bugs, differences were still relatively small. In the context of modelling botanical trees, inaccuracies in the centimeter range, as they most often were, are not problematic. The structural layout of trees can vary due to a variety of factors, for example the accuracy of the used acquisition machinery, whether the data was collected in winter or when foliage is present, or whether the wind was blowing the tree out of position at the moment it was scanned. A difference in centimeters will not affect the quality of the L-system representation, nor of the possible analyses that Cobra-Groeninzicht could carry out on the model. These analyses would mainly be environmental predictions (sunlight, temperature, wind, moisture) and risk analysis (overhanging branches, danger in case of storms). They are on the scale of whole threes, meaning as long as the main structure of a tree is modelled accurately, the exact position of its branches may vary in the cm range.

Limitations of the method described mainly consist of managing inaccuracies and missing elements in the input data. In order to construct a valid model of a tree, estimating the trunk position and diameter is essential. Currently, the method is completely dependent on the input data, which often does not contain enough points to accurately model the trunk, and the quality of the skeleton and its variables as generated and out-

putted by AdTree. Aside from this, two known internal issues of the method described exist: the deviating trunk geometry as described in the results section, and the translation issue. For more accurate results, these issues need to be resolved, although they do not pose an integral threat to the quality of the current results. In order to improve the method, either input data with more trunk points or better trunk estimation is needed, as well as the testing and analysis of larger datasets.

Considering this, the L-system implementation described here can be regarded as an accurate, more compact manner of storing a structural model of a botanical tree. Aside from this, the L-system format has several other advantages. Firstly, it can be used for further analysis. For example, by means of the software EnviMet, which carries out environmental analysis on L-system tree data and has interested Cobra-Groeninzicht greatly. Another useful property of an L-system is that it can be grown. This report has described some initial exploration of this topic. The ability to grow a tree model provides insights in the development of a (certain species of) tree, as well as predictions of its further development. This is useful when the most current dataset available is several years old, as is usually the case with the AHN data, as well as when one wants to anticipate tree development further into the future.

Since it is a model originating from the botanical field, several existing botanical models may be used to improve the model's correspondence to actual, real-life trees. For example: using botanical knowledge as a supplement, initial graphs could be improved and validated to approach a realistic structure of a tree, even if the input data was initially not accurate enough to determine this. This method also provides a direction for further improvement: this system currently does not work well on AHN3 data, since it is too sparse. Having a method to appropriate realistic trees, without being as tightly bound to (inaccurate and/or incomplete) input data may prove promising. Analysis of the L-system model of a large dataset of trees, meaning aggregating data and generating statistics, may also be a method of achieving this. Lastly, one could determine a species profile, based on a database of L-system trees. This profile could include parameters such as average trunk diameter, height, growing behaviour, or branch density. This information may be used to further improve the methods described above.

Bibliography

[1] Sylvain Delagrange, Christian Jauvin, and Pascal Rochon. Pypetree: a tool for reconstructing tree perennial tissues from point clouds. *Sensors*, 14(3):4271–4289, 2014.

[2] Ben Discoe, 2005. URL `http://vterrain.org/Plants/Modelling/`.

[3] Shenglan Du, Roderik Lindenbergh, Hugo Ledoux, Jantien Stoter, and Liangliang Nan. Adtree: Accurate, detailed, and automatic modelling of laser-scanned trees. *Remote Sensing*, 11(18), 2019. ISSN 2072-4292. doi: 10.3390/rs11182074. URL `https://www.mdpi.com/2072-4292/11/18/2074`.

[4] ENVI-met. Trees and vegetation - envi-met, May 2021. URL `https://www.envi-met.com/trees-and-vegetation/`.

[5] Python Software Foundation. turtle - turtle graphics¶, Jun 2021. URL `https://docs.python.org/3/library/turtle.html`.

[6] Ron Goldman, Scott Schaefer, and Tao Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36(14):1471–1482, 2004.

[7] Cobra Groeninzicht. Home, 2015. URL `https://www.bomenmonitor.nl/`.

[8] Jan Hackenberg, Heinrich Spiecker, Kim Calders, Mathias Disney, and Pasi Raumonen. Simpletree—an efficient open source tool to build tree models from tls clouds. *Forests*, 6(11):4245–4294, 2015.

[9] James Hanan. *Parametric L-systems and their application to the modelling and visualization of plants.* Citeseer, 1992.

[10] IDV Inc. Speedtree - about us, 2017. URL `https://store.speedtree.com/about-us/`.

[11] Annika Kangas, H Gove Jeffrey, and Charles T Scott. Introduction (chapter 1). *In: Kangas, Annika; Maltamo, Matti, eds. Forest inventory, methodology and applications, Vol. 10 [in series: Managing forest ecosystems]. Dordrecht, Netherlands: Springer: 3-11.*, 2006.

[12] Hugo Ledoux. cjio. URL `https://github.com/cityjson/cjio`.

[13] Hugo Ledoux, Ken Arroyo Ohori, Kavisha Kumar, Balázs Dukai, Anna Labetski, and Stelios Vitalis. Cityjson: a compact and easy-to-use encoding of the citygml data model. *Open Geospatial Data, Software and Standards*, 4(4), 2019.

[14] Chi Wan Lim and Yi Su. Tree species modelling for digital twin cities. *Transactions on Computational Science XXXVIII*, 12620:17, 2021.

[15] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968. ISSN 0022-5193. doi: https://doi.org/10.1016/0022-5193(68)90080-5. URL `https://www.sciencedirect.com/science/article/pii/0022519368900805`.

[16] Bernd Lintermann and Oliver Deussen. Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999.

[17] Niels Lohmann. nlohmann/json, 2013. URL `https://github.com/nlohmann/json`.

[18] Emilie Lorditch. Buzz blog, 2014. URL `https://www.physicscentral.com/buzz/blog/index.cfm?postid=4512988575527688739`.

[19] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer Science & Business Media, 2012.

[20] Pasi Raumonen, Mikko Kaasalainen, Markku Åkerblom, Sanna Kaasalainen, Harri Kaartinen, Mikko Vastaranta, Markus Holopainen, Mathias Disney, and Philip Lewis. Fast automatic precision tree models from terrestrial laser scanner data. *Remote Sensing*, 5(2):491–520, 2013.

[21] Rijkswaterstaat. Actueel hoogtebestand nederland 3 (ahn3), Dec 2018. URL `https://data.overheid.nl/dataset/11513-actueel-hoogtebestand-nederland-3--ahn3-`.

[22] Grzegorz Rozenberg and Arto Salomaa. *Lindenmayer systems: impacts on theoretical computer science, computer graphics, and developmental biology*. Springer Science & Business Media, 2012.

[23] Muzafer H Saračević, Munir Šabanović, and Emruš Azizović. Comparative analysis of amf, json and xml technologies for data transfer between the server and the client. *Periodicals of Engineering and Natural Sciences*, 4(2), 2016.

[24] Dorien ter Veld, Feb 2020. URL `https://simcms.ahn.nl/_flysystem/media/artikel-ahn-geo-info-april-2020-.pdf`.

[25] Liisa Tyrväinen, Stephan Pauleit, Klaus Seeland, and Sjerp de Vries. Benefits and uses of urban forests and trees. In *Urban forests and trees*, pages 81–114. Springer, 2005.

[26] Oswald Veblen. The heine-borel theorem. *Bulletin of the American Mathematical Society*, 10(9):436–439, 1904.

[27] Michael A Wulder, Christopher W Bater, Nicholas C Coops, Thomas Hilker, and Joanne C White. The role of lidar in sustainable forest management. *The Forestry Chronicle*, 84(6):807–826, 2008. doi: 10.5558/tfc84807-6. URL `https://doi.org/10.5558/tfc84807-6`.