Vision, Modeling, and Visualization (2022)
J. Bender, M. Botsch, and D. Keim (Eds.)

Clasping Trees - A Pipeline for Interactive Procedural Tree Generation

Simon J. Lieb[1], Nicolas Klee[2] and Kai Lawonn[1]

[1]Friedrich Schiller University Jena, Germany
[2]Ubisoft Blue Byte Düsseldorf, Germany

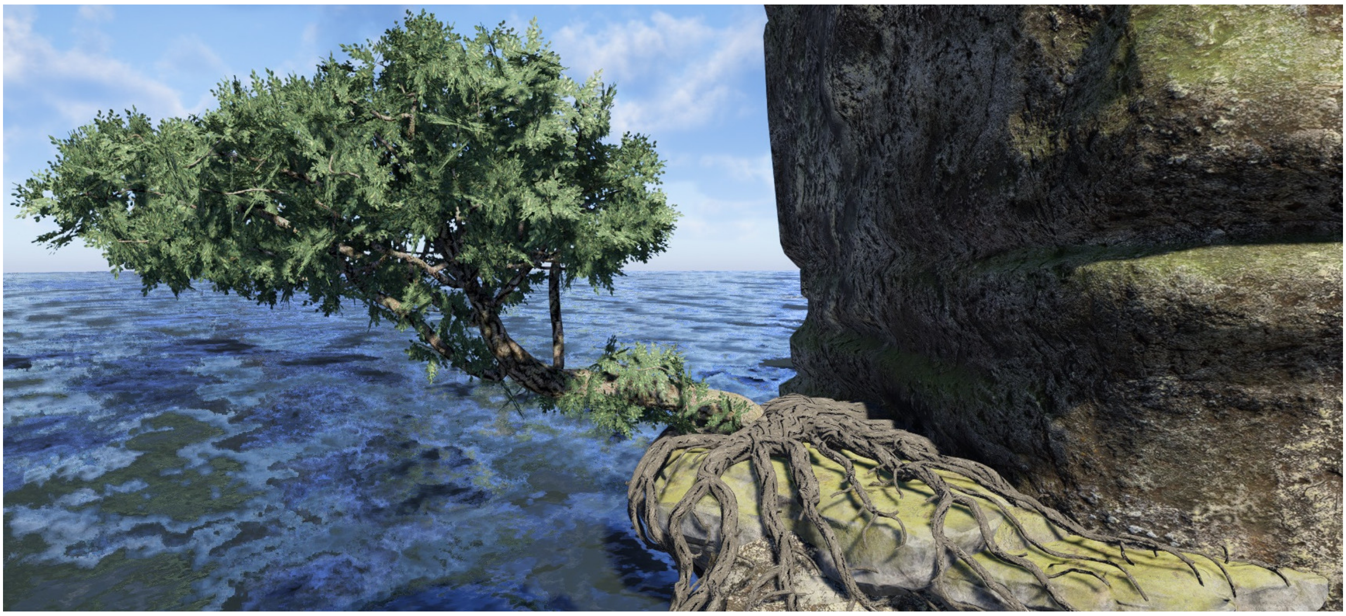

Figure 1: *A tree growing on a cliff, procedurally generated with our pipeline.*

Abstract

Trees in computer games are important components of an immersive game world. Realistic trees adapt to the environment in terms of shape and growth. Manually adapting each tree to its immediate environment is time-consuming. Hence, we present a pipeline to procedurally generate trees. This pipeline's input consists of tree-parameters and mesh sets. Tree-parameters have a direct influence on the final appearance of the tree. Meshes are used to indicate the space of the tree crown and surface for roots. We provide an overview of the necessary methods for procedural tree generation. Our method allows game developers to integrate the pipeline directly into their game engine, skipping the process of importing and maintaining external 3D-models. We used the Space Colonization Algorithm to generate roots of trees on the surface of a set of meshes. For the crown generation, we use an extended Space Colonization Algorithm called Self Organizing Trees. To receive the combined surface and volume of a set of meshes, we voxelize the individual mesh and compose it into a single voxel grid. We introduce two novel optimization methods to further increase the usability of the generated trees. These optimization methods decrease the necessary triangle count of the final mesh. The resulting trees can be used for real-life applications, such as games.

CCS Concepts

• ***Computing methodologies*** → ***Mesh models;*** *Mesh geometry models;*

1. Introduction

Automated game content creation is firmly anchored in modern game development. By employing procedural generation, entire game worlds with mountains, rivers, and forests can be created automatically.

The procedural generation of vegetation is still of interest in current research. The growth of a plant is highly dependent on its environment. Surrounding objects can be obstacles and influence the shape of a tree. Roots can wrap around objects, and objects that cast shadows affect the growth of plants by the availability of light. The realization of these properties in computer games leads to an immersive design of scenes. Therefore, special attention is paid to the adaptation of the local environment.

We created a complete pipeline for vegetation generation. The pipeline includes methods for the generation of abstract tree structures, the automatic generation of the corresponding mesh and its UV map, which is necessary for texturing objects. For the abstract tree structure generation, several methods from literature are combined and slightly modified to be usable in the pipeline.

To achieve better performance, we introduce two optimization methods to reduce the number of triangles created. Thus, we give users the ability to set the level of detail of the resulting tree. Adjusting the level of detail is important for game development since, besides visual conviction, performance is necessary for real-time applications.

In summary, we make the following contributions:

- We use voxelization to calculate the surface of combined meshes, to generate roots on it.
- We introduce two optimization methods during the generation to reduce the polygon count of the 3D object.
- We provide tree-parameter settings for an implementation of the pipeline.

2. Related Work

Computational plant generation originates in recursive relations. Ulam et al. [Ula62] hinted tree-like structures in recursive patterns. Honda [Hon71] described the effect of different branching angles and branch ratios using recursive branching patterns. Later, Lindenmayer and Prusinkiewicz [PL90] gave a large set of tools for generating plants. They used a formal grammar to dynamically implement recursive formulas. These tools of grammar-based systems are later called L-systems.

In contrast to rule based recursive approaches, simulation-based approaches exist. Particle flow, introduced by Rodkaew et al. [RCSL03], simulates an intrinsic energy transportation system within a plant, using particles. Kohek and Strnad [KS18] used particle flow to generate large forests on a GPU directly.

Another simulation-based and intuitive way to generate tree skeletons is using the *Space Colonization Algorithm* (SCA), introduced by Runions et al. [RLP07]. The main difference to the particle flow method is a top-down approach. Instead of particles flowing from the root to the leaves, space indicators, called *attraction points*, form the final shape of the tree. Attraction points represent free space and light, indicating the direction of the tree growth. At the beginning of SCA, a set of attraction points is generated. The iterative algorithm is divided into two phases: first, tree nodes are added to the tree skeleton corresponding to the remaining attraction points. Secondly, attraction points are deleted if they are reached. A detailed description can be found in the work by Runions et al. [RLP07], Palubicki et al. [PHL*09], and Nuić and Mihajlović [NM19].

Palubicki et al. [PHL*09] extended the idea of SCA by adding biological components like phyllotaxis, the concept of buds and shadow propagation. Their approach simulates internal signaling mechanisms to realistically generate vegetation and is used for image synthesis.

All the works mentioned use abstract tree graphs as a base representation. On these graphs, further operations and computations can be performed well, because these graphs form simple mathematical representations of the actual trees. Pirk et al. [PNH*14] used pre-generated tree graphs to perform wind operations and interactive modeling [PSK*12]. Strnad et al. [SKNŽ19] developed an efficient way to store tree-graphs. Their work can be used if there is the necessity to store the tree-graphs itself, rather than the final 3D object.

For generating tree-like meshes, two main approaches exist. Zhu et al. [ZJY15] used convolution surfaces to generate smooth ramifications. Convolution surfaces are generated by creating isosurfaces on a field of scalar values, based on the distance to tree nodes and internodes. Another approach is to use a combination of primitive meshes, such as cylinders. Lluch et al. [LVM04] designed a model to handle ramifications geometrically. Zhang et al. [ZBM*17] presented a method to generate LOD (*Level Of Detail*) models specifically for tree branches. Nuić and Mihajlović [NM19] presented a geometric method to smooth branches using Bézier curves. Within their work, Nuić and Mihajlović also show that SCA has a good balance between customizability and performance, and is therefore used as a base concept in our work.

For voxelization, we consider two approaches with different ease of implementation. Thon et al. [TGR04] developed a basic solution to voxelize the volume of meshes. Although the method is not the fastest, it can be used very well for quick implementation. However, Schwarz and Seidel [SS10] presented a fast method to voxelize both the surface and volume of meshes.

3. Tree Graph Generation

To generate a 3D-tree object, we first need to construct an abstract graph representing the tree as a tree skeleton. Palubicki et al. [PHL*09] introduced self-organizing trees that extend SCA with more complex buds. These buds adapt the biological concept of actual buds, which trees form at each branch end. We use the terminology as in [PHL*09]. Also, we use *occupancy radius* synonymous to *kill distance*, as well as *perception radius* synonymous to *influence distance*.

The root growth is terminated by endogenous influences, while shoot growth is influenced by external factors such as light [Bec10]. Due to the differences in root and shoot growth, we use different algorithms to generate each. For roots, we use SCA, since we do not include other influences than the available space and ignore the bud development of roots. For the shoots and tree crown, we use the self-organizing tree method with some adjustments, as we do not include the *bud fate*, described by Palubicki et al. [PHL*09], to save computation time.

The main idea of SCA and self-organizing trees is to generate a tree skeleton by reaching previously scattered *attraction points*. Attraction points indicate free space and light. During the iterative process, branches are added to the existing graph in the direction of nearby attraction points.

Each tree skeleton node represents a bud. A bud B is characterized by the following properties. B has a position and an initial growing direction. It has a *perception volume* that is defined as a spherical cone having its tip at the bud's position and its central axis aligned with the bud's growing direction. The length of the central axis of the perception volume is given by the attribute *perception radius*. We call the apex angle of the spherical cone the *perception angle* ω. With a wider perception angle, the bud is able to perceive more attraction points that are averted from the bud's growing direction, resulting in a less uniform and more unstructured growth. Again, the adjustment is crucial since setting ω too low leads to a low probability that attraction points are perceived at all. A too-large value subverts the effect of lateral bud positioning and growth control. Palubicki et al. [PHL*09] suggest a value of 90°, which we use as a preset. Yet, other values are also possible and can be changed by the user, as discussed later (see Figure 8).

Attraction points that lie within the *perception volume* of B are taken into account for the calculation of the final growing direction. If an attraction point lies within the perception volumes of two different buds, it is assigned to the bud with the smaller distance.

Besides the inner construction of a bud, self-organizing trees differ between buds. We distinguish two types of buds: *terminal buds* and *lateral buds*. The initial growing direction of *terminal buds* maintain the parent bud's final growing direction. *Lateral buds* sit in the same position as the terminal bud, but their direction is determined by attributes *branching angle* β and *phyllotaxis* φ. The branching angle β is the angle between the initial growing direction of the lateral bud and its corresponding terminal bud. Phyllotaxis φ is the rotation angle of the lateral bud direction around the terminal bud direction. While growing, the lateral bud rotation φ is calculated by adding a global phyllotaxis value to the previous lateral bud. The phyllotaxis defines the arrangement of leaves. Angles of 90° and 180° appear very frequently in nature. However, spiral angles that are not dividers of 360 also occur in nature. The angle of spiral forms is often 137.5°, corresponding to the golden ratio. The specific reason for the formation of different phyllotaxis patterns is unknown; see Okabe et al. [OIY19].

Two successive terminal buds form an internode that is later used to generate the mesh. The distance between terminal buds is the *internode length* which is set globally.

Child buds lie in the direction of the final growing direction of a bud multiplied by the internode length. The final growing directions of lateral and terminal buds are the normalized weighted average of the initial growing direction and the direction to the center point of the associated attraction points. The weight $\mu \in [0,1]$ represents an inertia of the growing direction, where a high weight in favor of the initial growing direction results in a high inertia.

We combine terminal buds, lateral buds, and internodes into *metamers*. A *metamer* is an internode, having a *terminal* and one or more *lateral* buds at its end. Both, lateral and terminal buds, can produce a metamer (see Figure 2).

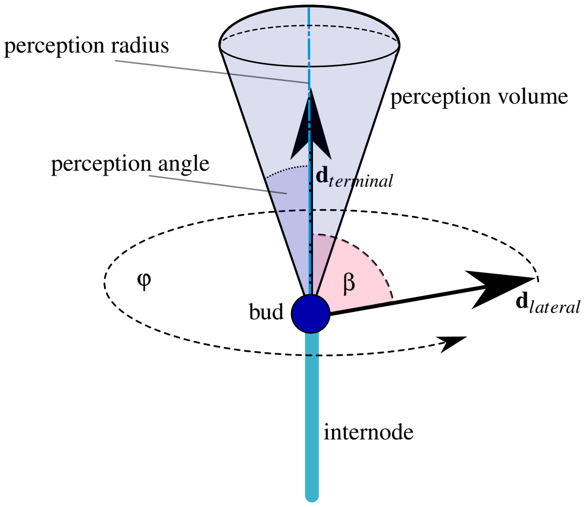

Figure 2: *Visualization of a metamer.* $\mathbf{d}_{terminal}$ *and* $\mathbf{d}_{lateral}$ *show the initial growing direction of the terminal and lateral bud. The position of the buds is indicated as a blue dot.* β *is the branching angle and* φ *the phyllotaxis. The perception volume is shown in light blue. The perception volume of the lateral bud was omitted for clarity. Perception radius and perception angle are shown as parameters of the perception volume. The internode is indicated as a connection between two successive terminal buds.*

Buds are deactivated after their processing. Therefore, a bud has a maximum of one internode. In contrast, the amount of generated internodes at a tree node of SCA is unlimited. Branching is made possible by the fact that each metamer contains at least two buds.

After the growth process within an iteration step, attraction points are sorted out. We delete an attraction point, if there is a bud whose distance is less than the *occupancy radius*.

The diameter is calculated for each bud using a post-order traversal through the tree skeleton. As suggested by Minamino and Tateno [MT14], the diameter b_{dia} of each bud b is then calculated using the set $B_{children}$, containing all child-buds of b using

$$b_{dia} = \sqrt[x]{\sum_{c \in B_{children}} c_{dia}^{x}}, \tag{1}$$

while the exponent $x \in \mathbb{R}^{+}$ defines the thickness of the tree. The exact value depends on the wished outcome or targeted plant species. However, values between 1.8 and 2.8 seem reasonable. Diameters of buds with no child buds are set to 1. The branch diameter is linearly interpolated between two successive buds.

A result using the presented method together with the root generation can be seen in Figure 1. Here, the corresponding 3D model has also been generated by using the tree skeleton.

4. 3D Model Generation

To produce a 3D model of a tree, we visualize the previously generated tree skeleton. The tree skeleton does not necessarily need to be generated by this pipeline. A tree object is constructed by generating mesh tubes for each internode. Besides that, we use quads for the foliage. The presented method is valid for the upper part of the tree and the roots.

4.1. Tube Generation

Branches are generated by connecting mesh tubes. Therefore, care must be taken to ensure that these tubes have aligned transitions. In addition, attention must be paid to maintaining a curved appearance with a minimum number of triangles concurrently. Therefore, we first define a single cylinder. Then, we describe the calculation of the transitions between cylinders and finally the method for smoothing branches.

4.1.1. Distorted Cylinders

The smallest section of a generated tree mesh is a distorted cylinder. We define it by two circles C_i with $i \in \{1,2\}$ that lie in space arbitrarily. Each is defined by the parameters $\mathbf{c}_i$ as its center point, r_i as is radius, and $\mathbf{n}_i$ as its directional vector. Directional vectors $\mathbf{n}_i$ are orthogonal to the circle area and define the alignment in space. We discretize C_i with polygons V_i, containing k_i vertices.

To align the assignment of points between the two circles, we propagate a supporting vector $\mathbf{t}_i$ for each circle. The order of assignment of points between the two polygons V_1 and V_2 is crucial, as can be seen in Figure 3. The transition from $\mathbf{t}_i$ to $\mathbf{t}_{i+1}$ is given by the cylinder quaternion discussed below. We distribute the vertices within the circles equidistantly by setting the first point to $\mathbf{c}_i + \mathbf{t}_i$. The following vertices are set by successively rotating $\mathbf{t}_i$ around $\mathbf{n}_i$ by the angle $\frac{2\pi}{k_i}$.

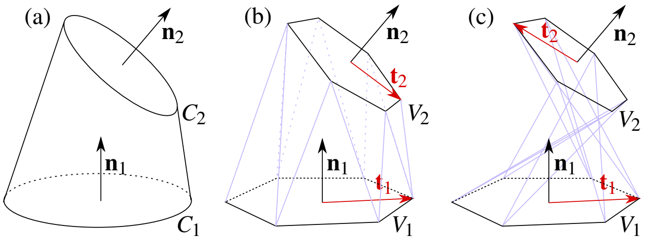

Figure 3: *Example of a distorted cylinder and its mesh representation. a) The schematic cylinder with circles C_1, C_2 and normals $\mathbf{n}_1$, $\mathbf{n}_2$. b) Polygons V_1, V_2 represent C_1, C_2. Vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ are rotated according to $\mathbf{n}_1$ and $\mathbf{n}_2$. The resulting mesh forms a sufficient representation. c) $\mathbf{t}_1$ and $\mathbf{t}_2$ are not aligned with the rotation of $\mathbf{n}_1$ and $\mathbf{n}_2$. The resulting mesh is twisted.*

For optimization, we decrease the count of vertices for small branches. Therefore, in some cases, the two polygons V_1 and V_2 have different numbers of vertices. Generally, we assume that $k_1 \geq k_2$, and use the surjective mapping: $\mathbf{v}_{2_j} \in V_2$ is assigned to $\mathbf{v}_{1_l} \in V_1$, if:

$$j = \left[\frac{l \cdot k_2}{k_1} \right] \tag{2}$$

where $[\cdot]$ represents a rounding to the nearest integer. We now use two successive tree nodes as center points and their growth direction as normals.

4.1.2. Successive Cylinders

Two successive cylinders must share the same supporting vector $\mathbf{t}$ at their connecting side. Otherwise, holes would be generated between those two cylinders. It is therefore enough to calculate one supportive vector for each cylinder, as the supporting vector of the bottom polygon is already given by the previous cylinder. To align two successive cylinders, a rotation is propagated through the generation process. This rotation is represented by a quaternion q_i for the i-th cylinder, starting with $q_0 = (0,0,0,1)$. In each propagation step i, a new quaternion q' is created, representing the rotation from the direction of cylinder Z_i to Z_{i+1}. Then, rotation quaternion q' is applied to the propagated rotation q_{i-1}, yielding the rotation q_i of the current cylinder:

$$q_i = q' \cdot q_{i-1}. \tag{3}$$

The alignment vector $\mathbf{t}_i$ of the cylinder is then calculated by applying q_i to a globally defined orthonormal vector $\mathbf{v}_0$ of root node direction $\mathbf{d}_0$. Each bud holds its rotation quaternion to establish a simple local rotation relation for subtrees of each bud.

4.1.3. Smoothed Tubes

For each internode of a tree graph, a mesh section has to be generated. Using just one distorted cylinder for each internode would lead to edgy and unevenly formed branches. To tackle this problem, we use third order Bézier curves as suggested by Nuić and Mihajlović [NM19].

Let B_n be a node of the tree graph, B_{n-1} the parent node of B_n, $\mathbf{p}_n$ the position of B_n, $\mathbf{d}_n$ the direction of B_n, $\mathbf{p}_{n-1}$ the position of B_{n-1}, $\mathbf{d}_{n-1}$ the direction of B_{n-1}, and $c \in \mathbb{R}^+$ a curvature strength factor. The cubic Bézier curve $\mathbf{b}$ is then defined by the set of four control points $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ with:

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{p}_{n-1}, \\ \mathbf{b}_1 &= \mathbf{p}_{n-1} + \mathbf{d}_{n-1} \cdot c, \\ \mathbf{b}_2 &= \mathbf{p}_n - \mathbf{d}_n \cdot c, \text{ and} \\ \mathbf{b}_3 &= \mathbf{p}_n. \end{aligned} \tag{4}$$

Specifically, in our implementation, the calculation of $\mathbf{b}(t)$ is split in two parts, $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$. This setup ensures that we also get the tangent of point $\mathbf{b}(t)$. We can then use the tangent to optimize the necessary polygon count for the section between B_{n-1} and B_n. Therefore, $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$ are computed by

$$\begin{aligned} \mathbf{b}_0^2(t) =& (1-t) \cdot ((1-t) \cdot \mathbf{b}_0 + t \cdot \mathbf{b}_1) \\ &+ t \cdot ((1-t) \cdot \mathbf{b}_1 + t \cdot \mathbf{b}_2), \text{ and} \end{aligned} \tag{5}$$

$$\begin{aligned}\mathbf{b}_1^2(t) =&(1-t)\cdot((1-t)\cdot\mathbf{b}_1+t\cdot\mathbf{b}_2)\\ &+t\cdot((1-t)\cdot\mathbf{b}_2+t\cdot\mathbf{b}_3)\end{aligned} \tag{6}$$

Given $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$, $\mathbf{b}(t)$ is computed by

$$\mathbf{b}(t) = (1-t)\cdot\mathbf{b}_0^2(t)+t\cdot\mathbf{b}_1^2(t). \tag{7}$$

Furthermore, the normalized tangent θ is computed by

$$\theta = \frac{\mathbf{b}_1^2(t)-\mathbf{b}_0^2(t)}{||\mathbf{b}_1^2(t)-\mathbf{b}_0^2(t)||}. \tag{8}$$

For more details on Bézier curves, we refer to the book by Gerald Farin [FF02]. Note that we normalize the tangent, and therefore cancel any multiplication factor. We do this to reduce computation time during the optimization discussed below. Generating a distorted cylinder for each point $\mathbf{b}(t)$ results in a high amount of triangles in straight areas with no visible benefit. The smoothness of the distorted cylinder depends on the step size Δt. We introduce an optimization to increase the triangle count only for areas with high curvature, while maintaining a low triangle count for straight branches and a strong smoothing with a small Δt.

For that, we introduce the straw factor δ, which represents the accuracy of the Bézier line fitting. We calculate $\mathbf{b}(t_i)$ successively with $t_0 = 0$ and $t_i = t_{i-1} + \Delta t$. During iteration, we keep track of the tangent θ', which represents the tangent of the last generated cylinder. For the first iteration, we set $\theta' = \theta_0$. At each iteration step i, we calculate the tangent θ_i. If $\theta' \cdot \theta_i \leq \delta$, a distorted cylinder is generated. Also, we set $\theta' = \theta_i$, to keep the latest relevant tangent. This results in a cylinder being generated just at small angles (see Figure 4). Due to its cocktail straw-like shape, we call this method *straw optimization* from now on.

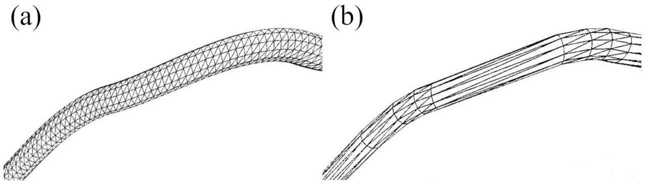

Figure 4: *The same generated branch, a) without straw optimization and b) with straw optimization.*

If $\delta = 1$, no straw optimization is used. Smaller values result in less triangles but more visible edges. We have found that values for δ between 0.98 and 1 give a convincing result. However, exact values depend on the application, specifically on the step size Δt. For example, low values below 0.98 could be used for low resolution LOD objects. Furthermore, the number of triangles of the unoptimized model depends strongly on the density of the Bézier points.

Another optimization we propose is *radius optimization*, which decreases the number of vertices representing the circles of the tubes depending on the radii (see Figure 5). Let D_{max} be the maximum diameter and $D_{min} = 1$ the minimum diameter. The number of vertices N_b for each tree node b with diameter b_{dia} can then gradually be decreased by using the formula

$$N_b = \frac{b_{dia}-D_{min}}{D_{max}-D_{min}}\cdot(N_{max}-3)+3 \tag{9}$$

with $N_{max} \geq 3$ being the maximum number of vertices for the largest diameter. Note that we add 3 as a baseline, so that the smallest diameters still result in three vertices, to maintain a 3D-shape.

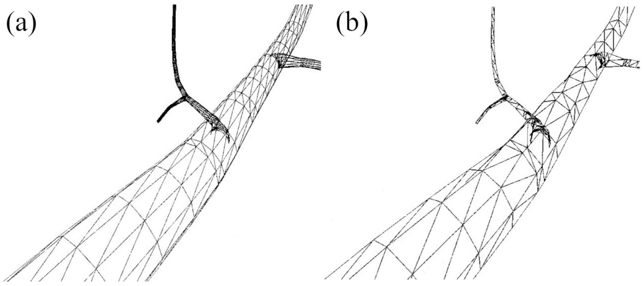

Figure 5: *The same generated branch structure, a) without radius optimization and b) with radius optimization.*

4.2. Leaves and Foliage

For every bud which has not produced a metamer and where the diameter is within 10% of the smallest diameters, a twig or leave cluster is added. This cluster contains a plane defined by four vertices with a twig texture, in which the branches converge in the center. The alignment of the quad is based on the rotation quaternion of the bud.

4.3. UV Mapping

The UV mapping is constructed so that the texture runs along the branches, see Figure 6. For each distorted cylinder, radii r_1 and r_2, as well as the orthonormals $\mathbf{t}_1$ and $\mathbf{t}_2$ to the terminal growing directions are given. Thus, the UV mapping is done by wrapping a rectangular coordinate field around the cylinder. Hereby, each circle is viewed independently. The UV coordinate (u,v) for the first

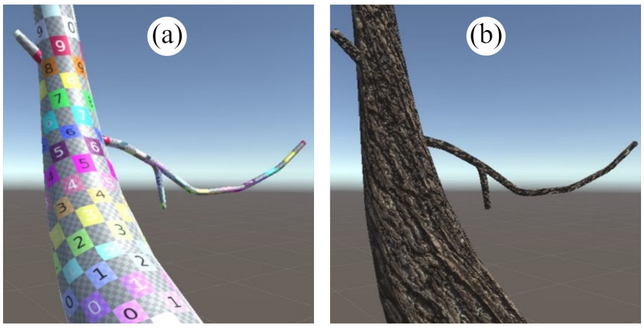

Figure 6: *UV-mapping on an example branch. a) Test-texture to visualize the UV mapping. b) A texture mapped accordingly.*

vertex is $(0,l)$, where l is the total distance to the root along the tree graph. Distance l is computed by propagating the distance during the tree graph generation. The UV coordinate $(u,v)_i$ of vertex v_i is then computed by

$$(u,v)_i = \left(\frac{i \cdot r}{k}, l\right), \tag{10}$$

with r being the radius of the current circle. The same applies to the upper circle of the cylinder. The texture will not be compressed if the radii differ since the mapping depends on r_1 and r_2. Although cut edges are present in this case and at branching points, they are barely visible if irregular textures are used.

5. Attraction Point Generation

We want to be able to define the shape of the tree. Therefore, we take arbitrary meshes and generate the attraction points inside them. For the crown, we use primitive meshes that represent the area of growth. For the roots, we use meshes from the environmental scene. Evenly distributed attraction points within a mesh approximate the mesh volume. The shape of the tree adapts to the attraction point distribution. Therefore, the tree adopts the shape of the input meshes.

To achieve a uniform distribution of points, a mesh is divided into voxels. Voxelizing has the following advantages: First, distributing points inside or on the surface of a mesh does not depend on the complexity of the mesh, as long as it is voxelizable. Second, multiple meshes can be combined into one voxel object. Here, the inner volume consists of the union of mesh volumes.

We divide the selection of meshes into four layers. The first layer is using primitive meshes to indicate the tree crown area. The second layer is a selection of meshes for the root growth. Layers three and four are selections for obstacles for the crown and root growth. Voxels extracted from layers three and four are subtracted from the voxel sets of layers one and two. A setup to use the different layers can be seen in Figure 7.

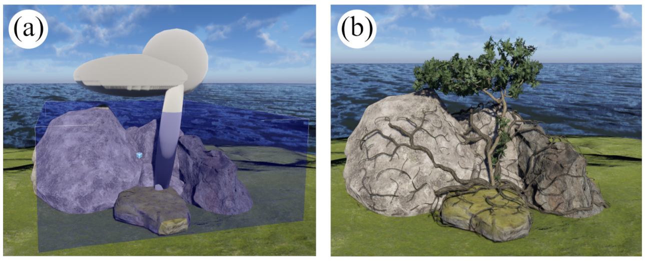

Figure 7: *Example of the generation process. a) The area for the roots is selected with the blue box. b) Combined spheres are used to set the attraction point volume for the tree trunk and crown. The rearmost boulder is selected as a subtractive layer for the roots.*

First, we voxelize all selected meshes in each layer. Then, we successively combine the voxelized meshes with a logical 'or' in each layer. We get the final layers by subtracting the obstacle layers from the crown and root layer by setting every voxel to zero if the corresponding obstacle layer voxel has a value of 1.

The surface voxelization step for the roots is done after the general voxelization. We compute an approximately uniform surface distribution by using the calculated voxels. Here, the surface voxels are those with at least one non-occupied voxel as a neighbor and are computet in two passes. In the first pass, each voxel that lies within any of the input meshes is occupied. In The second pass, only voxels that have at least one non-occupied neighbor stay occupied. All other voxels become non-occupied. Meshes are combined volumetrically during voxelization and, therefore, surfaces are also combined as desired.

Finally, we scatter uniformly distributed attraction points within the voxels. For each attraction point that is scattered, a random voxel is selected and its position is then calculated by three random numbers in the range of the voxel dimensions.

6. Experimental Results And Evaluation

We first provide values for parameters that have worked well for us. With these values, users may implement the described methods. After that, we present the required computation time and memory usage of the generation methods. Thus, use cases can be derived for users.

6.1. Parameter Determination

We suggest baseline parameters to generate high quality results. Note that there is a significant dependency between parameters.

All length parameters must have the same reference scale size. This scale can be chosen arbitrarily to obtain the desired tree size as long as the relations remain invariant. In the following, we assume an indefinite scaling size λ if no factor is given.

A fundamental parameter is the attraction point count. It defines the density of a tree crown, but also the connectivity between sparsely connected volumes. However, it depends on the mesh volume size and perception radius. We define the attraction point density AP_{dense} by

$$AP_{dense} = \frac{AP_{count}}{volumeSize}. \tag{11}$$

A density between 20 and 80 per λ^3 seems to be sufficient, with a given perception radius of 0.7. If the density falls below 20, the attraction point distribution is too sparse, i.e., the generation ends before all attraction points have been reached. A density of more than 80 can result in more detail, but this is disproportionate to the higher computing time required.

Palubicki et al. [PHL*09] suggest that a perception radius of 4 to 6 times the internode length results in moderate growth. However, a larger perception radius also works for the tree crown and requires fewer attraction points. They also suggest the occupancy radius to be two times the internode length, which give promising results. Having a smaller occupancy radius leads to an increased number of small side-shoots. Given an internode length of 0.1, an occupancy radius of 0.15, and a perception radius of 0.7 yield realistic results for the upper part of the tree, see 8a.

Since roots are generated on mesh surfaces, they need to be able to adapt to more detailed structures. Thus, a smaller internode

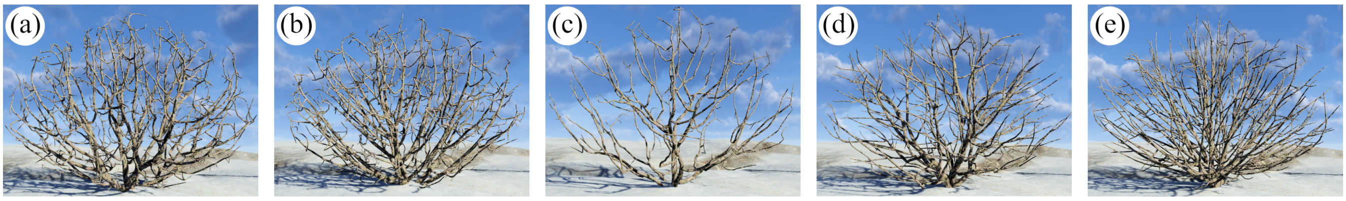

Figure 8: *A bush generated with five different parameter sets. First image serves as a baseline. The following images differ in one parameter to the baseline. a) attraction points: 2000; internode length: 0.1; perception angle: 3.14; perception radius: 0.7; occupancy radius: 0.15; phyllotaxis: 1.57; branching angle: 1.3; diameter exponent: 2.6; voxel grid resolution: 128. b) perception angle: 0.79. c) occupancy radius: 0.4. d) perception radius 2.0. e) branching angle 0.2.*

length must be chosen and, therefore, more attraction points must be scattered. Also, a smaller occupancy radius and a smaller perception radius are necessary. A much higher attraction point density can be chosen for the roots because points are only generated on the surface. With a voxel grid resolution of 256^3, an attraction point density of 300 to 600 per λ^3 is sufficient, given an internode length of 0.09, perception radius of 0.35, and occupancy radius of 0.2. However, the tree-parameters depend highly on the detail level of the used meshes and voxel grid resolution. We provide different suggestions for tree-parameter configuration in Figure 8.

6.2. Performance and Memory

We measured performance for the crown and root generation separately. Performance measurements for the skeleton and mesh generation, as well as its corresponding internode count, triangle count, and vertex count for the crown generation can be found in Table 1 (top). The generation method for roots is SCA. Since the branches of roots must adapt to more detailed structures, the internode length for roots is fixed to 0.09. Also, the Perception Radius is set to 0.35. Measurements for the root generation can be found in Table 1 (bottom). All tests were done on a workstation with the following properties:

Processor	Intel Xeon W-2255 @ 3.70GHz
RAM	64 GB
Graphics Card	NVidia GTX 1070

When comparing the generation from the upper part of the tree with the roots, it is noticeable that the root generation takes much more time and generates more triangles. Obviously, a higher amount of attraction points results in longer computation times. However, a higher amount of attraction points is required for the roots. At the same time, the number of internodes does not increase significantly.

The branches on roots can grow at any node in almost any direction. Thus, more angular branches are generated. Branches are meshed with more triangles to receive a smooth curvature. Therefore, the number of triangles of roots compared to crown growth is higher.

The memory consumption differs between generation steps. During the tree generation, a voxel grid is used. Since a single voxel holds less than 8 booleans, its memory consumption is approximately 1 byte. The overhead, i.e., the voxel grid's position and size, is negligible. Therefore, a voxel grid with a resolution of 256^3 results in a 16,8 megabytes data set. Nevertheless, optimizing the memory consumption of the used voxel grid is part of future work.

Table 1: *Internode count (Interns.), triangle count (Tris.) and vertex count (Verts.) depending on the attraction point count (APs) for the crown (top) and root (bottom) generation. Entries in columns Skeleton and Mesh represent the generation process time in milliseconds. Values represent the average of each set of generated samples. Note that pure SCA is used to generate the roots.*

APs	Interns.	Tris.	Verts.	Skeleton	Mesh
Internode length: 0.2, crown generation					
500	678.3	49922	28633	13.5 ms	12.7 ms
1000	1161.0	86728	49837	35.3 ms	13.8 ms
1500	1503.7	111030	63839	52.8 ms	28.2 ms
2000	1817.7	129842	74793	86.8 ms	24.0 ms
2500	2068.7	144776	83462	76.7 ms	32.2 ms
Internode length: 0.09, root generation					
2000	842.0	138621	75599	67.6 ms	18.8 ms
3000	1403	436928	236666	239.9 ms	89.8 ms
4000	616.3	351552	190463	328.3 ms	64.3 ms
5000	627.3	383616	207831	467.7 ms	60.6 ms
6000	660	408761	221438	526.0 ms	90.7 ms

The permanent memory consumption consists of two parts. The first part is the tree skeleton, containing all the information about the tree structure. The second part is the 3D model, containing vertices, normals, triangles, and UV coordinates. Keeping one of the two is sufficient to store the results since the 3D model is constructed from the tree skeleton. However, additional calculations can be performed on the tree skeleton, i.e., wind [PNH*14], or interactivity [PSK*12]. Hence, we analyze the corresponding memory consumption.

6.2.1. Tree Skeleton Generation

The memory size can be determined by using the size of a single node. The attributes *isTerminal* (bool), *diameter* (float), *parentNode* (pointer), *position* (float[3]), *rotation* (float[4]) and *childNodes* (on average one pointer) sum up to 49 bytes. Using C++, the size expands to 52 bytes due to structural padding. Table 1 indicates an internode count between 600 and 2100 for the upper part and 600 to 1500 nodes for the roots, highly depending on the real generation situation. With an overall node count of 1200 to 5000 nodes, the memory consumption sums up to roughly 62 to 260 kilobytes. Additionally, a constant overhead of tree parameters is stored.

6.2.2. 3D-Model Generation

The tree model can be stored in a simple mesh. This mesh contains a list of vertices, normals, triangle indices, and UV coordinates. The additional overhead caused by the specific implementation in the 3D environment is ignored.

Vertices and normals are stored in 12-byte float[3] each. Triangle indices must be stored as 4-byte integers since there are more than 2^{16} vertices to be addressed. UV coordinates are stored as 8-byte float[2]. Vertices, normals, and UV coordinates have the same number of elements and can be combined into 32 bytes per element. For each triangle, three indices are stored, adding up to 12 bytes per triangle.

Since vertices are reused while indexing the triangles, the average ratio of vertices to triangles is approximately between $\frac{27}{50}$ to $\frac{29}{50}$. The overall vertex and triangle count of the 3D model of a tree based on the performance measurements ranges from roughly 100 thousand vertices and 190 thousand triangles up to 320 thousand vertices and 580 thousand triangles. The memory size ranges from 5480 kilobytes to 17200 kilobytes accordingly. However, this describes only the raw size of the data.

7. Conclusions And Future Work

We introduced a complete pipeline to generate trees by setting parameters and input meshes. Two types of input meshes lead to the adaption to the tree's direct environment. The first type indicates free space for the tree crown. The second type indicates obstacles around which the roots must grow. This allows trees and bushes to better integrate with the rest of the scene. Suggestions for parameters for implementing our results were given and the results were evaluated by performance and memory consumption.

To take the surface of surrounding meshes into account, we used a basic voxelization method. The approximation of the mesh surface by voxelization and the subsequent generation of roots shows promising results. A faster method, like presented by Schwarz and Seidel [SS10], would provide a higher level of detail.

The resulting tree skeletons can be visualized with the presented meshing method. *Straw optimization* and *radius optimization* reduce the number of necessary triangles. This optimization can be used to create different LOD-levels during the generation process.

Because the generated trees look convincing, while the vertex count is kept low, we believe the generation pipeline can be implemented and used in game engines. The ability to customize the appearance by adjusting the tree-parameters also improves the usability as an interactive tool.

References

[Bec10] Beck, Charles B. *An introduction to plant structure and development: plant anatomy for the twenty-first century*. Cambridge University Press, 2010. Chap. 16, 279–321 2.

[FF02] Farin, Gerald E and Farin, Gerald. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann, 2002. Chap. 5,6 5.

[Hon71] Honda, Hisao. "Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body". *Journal of theoretical biology* 31.2 (1971), 331–338 2.

[KS18] Kohek, Štefan and Strnad, Damjan. "Interactive Large-Scale Procedural Forest Construction and Visualization Based on Particle Flow Simulation". *Computer Graphics Forum*. Vol. 37. 1. Wiley Online Library. 2018, 389–402 2.

[LVM04] Lluch, Javier, Vivó, Roberto, and Monserrat, Carlos. "Modelling tree structures using a single polygonal mesh". *Graphical Models* 66.2 (2004), 89–101 2.

[MT14] Minamino, Ryoko and Tateno, Masaki. "Tree branching: Leonardo da Vinci's rule versus biomechanical models". *PloS one* 9.4 (2014), e93535 3.

[NM19] Nuić, H and Mihajlović, Ž. "Algorithms for procedural generation and display of trees". *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, 230–235 2, 4.

[OIY19] Okabe, Takuya, Ishida, Atsushi, and Yoshimura, Jin. "The unified rule of phyllotaxis explaining both spiral and non-spiral arrangements". *Journal of the Royal Society Interface* 16.151 (2019), 20180850 3.

[PHL*09] Palubicki, Wojciech, Horel, Kipp, Longay, Steven, et al. "Self-organizing tree models for image synthesis". *ACM Transactions on Graphics (TOG)* 28.3 (2009), 1–10 2, 3, 6.

[PL90] Prusinkiewicz, Przemyslaw and Lindenmayer, Aristid. *The algorithmic beauty of plants*. Springer, New York, 1990 2.

[PNH*14] Pirk, Sören, Niese, Till, Hädrich, Torsten, et al. "Windy trees: computing stress response for developmental tree models". *ACM Transactions on Graphics (TOG)* 33.6 (2014), 1–11 2, 7.

[PSK*12] Pirk, Sören, Stava, Ondrej, Kratt, Julian, et al. "Plastic trees: interactive self-adapting botanical tree models". *ACM Transactions on Graphics (TOG)* 31.4 (2012), 1–10 2, 7.

[RCSL03] Rodkaew, Yodthong, Chongstitvatana, Prabhas, Siripant, Suchada, and Lursinsap, Chidchanok. "Particle systems for plant modeling". *Plant growth modeling and applications. Proceedings of PMA03, Hu B.-G., Jaeger M.,(Eds.). Tsinghua University Press and Springer, Beijing* (2003), 210–217 2.

[RLP07] Runions, Adam, Lane, Brendan, and Prusinkiewicz, Przemyslaw. "Modeling Trees with a Space Colonization Algorithm." *NPH* 7 (2007), 63–70 2.

[SKNŽ19] Strnad, Damjan, Kohek, Štefan, Nerat, Andrej, and Žalik, Borut. "Efficient Representation of Geometric Tree Models with Level-of-Detail Using Compressed 3D Chain Code". *IEEE transactions on visualization and computer graphics* 26.11 (2019), 3177–3188 2.

[SS10] Schwarz, Michael and Seidel, Hans-Peter. "Fast parallel surface and solid voxelization on GPUs". *ACM transactions on graphics (TOG)* 29.6 (2010), 1–10 2, 8.

[TGR04] Thon, Sébastien, Gesquière, Gilles, and Raffin, Romain. "A low cost antialiased space filled voxelization of polygonal objects". *GraphiCon 2004* (2004), 71–78 2.

[Ula62] Ulam, Stanislaw. "On some mathematical problems connected with patterns of growth of figures". *Proceedings of Symposia in Applied Mathematics*. Vol. 14. Am. Math. Soc. Vol. 14, Providence. 1962, 215–224 2.

[ZBM*17] Zhang, Xiaopeng, Bao, Guanbo, Meng, Weiliang, et al. "Tree branch level of detail models for forest navigation". *Computer Graphics Forum*. Vol. 36. 8. Wiley Online Library. 2017, 402–417 2.

[ZJY15] Zhu, Xiaoqiang, Jin, Xiaogang, and You, Lihua. "High-quality tree structures modelling using local convolution surface approximation". *The Visual Computer* 31.1 (2015), 69–82 2.