Controllable Modular Trees

Suzuran Takikawa
University of British Columbia
Vancouver, Canada
stakikaw@cs.ubc.ca

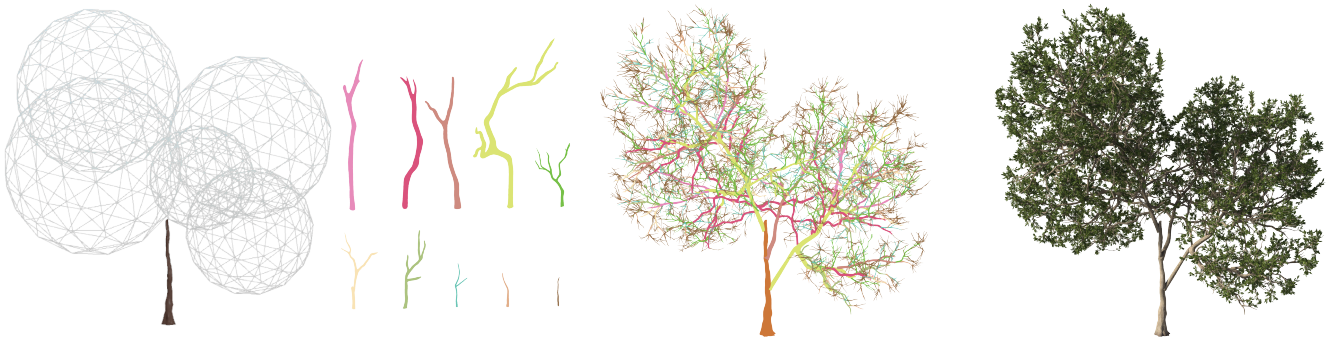
Riccardo Tinchelli
Netflix Animation Studios
Vancouver, Canada

Mike J Davison
Netflix Animation Studios
Vancouver, Canada

Curtis Andrus
Netflix Animation Studios
Vancouver, Canada

James John Drown
Netflix Animation Studios
Sydney, Australia

Alla Sheffer
University of British Columbia
Vancouver, Canada



(a) Input Trunk, Crown Shape, and Branch Modules

(b) Modular Tree Branch Structure

(c) Modular Tree Render with Leaves Added

Figure 1: Given as inputs a crown shape, trunk, and branch modules (a) we automatically generate a natural looking modular tree (b) that aligns with the input crown shape. Storing our tree using the modular structure without the leaves (c) requires an uncompressed file size of 1.8 MB, while storing it as a mesh would require 180 MB. (If the leaves are included, the sizes are 6.6 MB and 1.2 GB, respectively)

Abstract

Trees are a ubiquitous part of virtual environments, but faithfully modeling trees is challenging because of their incredible diversity. Tree modeling methods face two key challenges, as users want to create trees that look realistic but retain control over key elements of their appearance, such as the shape of the tree crowns. Traditional tree modeling methods are geared to produce tree models that are both detailed and unique and are typically stored as high-triangle count meshes or skeletons plus radii. Storing and manipulating such traditional tree models comes with a significant memory cost. Recent approaches have proposed to use modular trees consisting of a finite set of branch modules, which are transformed (scaled, translated, and rotated) to jointly form realistic looking trees. Consequently, modular trees require orders of magnitude less memory than their traditional counterparts. However, existing methods for modeling modular trees focus on visual realism and provide only minimal mechanisms for artists to control the look and shape of the resulting trees. We propose a controllable method for modeling modular trees, enabling artists to define the tree's overall crown shape while maintaining a plausible appearance using a small set of replicated branch modules. We formulate the computation of realistic modular trees that conform to a user-specified crown shape as a constrained mixed-variable optimization problem. We then compute the trees that satisfy these constraints using a method that grows trees one layer of branches at a time, maintaining realism throughout and promoting accurate approximation of the target crown shape. We extensively test our method on diverse crown shapes and compare against baselines, demonstrating its effectiveness.

CCS Concepts

• **Computing methodologies** → **Shape modeling**.

Keywords

Tree modeling, modular trees, tree silhouette control

ACM Reference Format:
Suzuran Takikawa, Riccardo Tinchelli, Mike J Davison, Curtis Andrus, James John Drown, and Alla Sheffer. 2025. Controllable Modular Trees. In *Graphics*

GI'25, Kelowna, BC, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Interface (GI'25), May 26-29, 2025, Kelowna, BC, Canada. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Trees are ubiquitous in digital media applications, including video games, movies, landscape visualization, forestry simulation, and many more. Tree modeling methods often need to balance two potentially conflicting sets of requirements as users want to create trees that are realistic, but retain control over their appearance. Advances in tree modeling methodologies have enabled efficient modeling of realistic trees through many approaches, including recursive or hierarchical modeling [16, 31, 35], sketch-based modeling [23, 25], data-driven modeling [5, 15, 17], and many more, some methods offering more control than others. However, storing highly detailed realistic trees requires a lot of memory. Traditional tree modeling methods typically produce trees that are stored as high-triangle-count meshes or skeletons plus radii. While the latter representation can be more compact for storage, for texturing and rasterization, it typically requires converting it into a dense mesh. As an example, storing the tree in Fig. 1 as a mesh requires 180 MB (not including the leaves). The large memory footprint of realistic looking trees poses a significant challenge, especially for large-scale scenes with hundreds or thousands of trees.

This challenge motivates the work to reduce the memory cost of representing realistic trees. Replicating entire trees across a scene dramatically reduces realism and is thus unappealing. A more practical and recent alternative focuses on *modular trees* [10, 24, 32]. Modular trees are modeled by assembling replicated branch shapes to represent a tree (see Fig. 1b). Replicated branch shapes are called branch modules, which in our case are a fixed set of precomputed branch geometries. The tree can then be represented with a set of branch module types and their spatial transformations. For storage purposes, rather than storing the mesh of every branch, we store one mesh for each branch module type and the spatial transformations for each copy used in the modeled tree. This approach results in a significantly smaller memory footprint than that of traditional trees through GPU geometry instancing [6], while allowing significant diversity via generation of different branch module combinations. Existing work on modular trees allows only minimal artist control of the resulting tree shape (Sec. 2).

We know that controlling the crown shape is one of the most popular mechanisms for artist control [4, 31, 41, 42, 49]; this preference is consistent with our observations of artists' preferred workflows for tree modeling in game and video production. Although imposing crown shape constraints for traditionally modeled trees is a well-understood problem, e.g. [31, 42], these methods do not extend to modular trees. Our experiments (Sec. 5) suggest that using naive extensions of these methods to the modular tree settings results in trees that either appear unnatural or do not adhere to the artist-specified crown shape. Consequently, we introduce a new method for controlling the shape of the tree crown while operating with modular trees (Fig. 1).

We first analyze the properties we need our output trees to satisfy in order to achieve realism while retaining control. Our analysis suggests a modeling method that should adhere to the following properties: crown shape adherence, scaling consistency, and natural connectivity. We formalize our definitions of these properties in Sec. 3. These properties guide the development of our approach and facilitate the creation of natural-looking trees (Fig. 1). Given the above requirements, we formulate the problem of controllable modular tree modeling as a constrained mixed-variable optimization problem that solves for the selection of branch modules, branching positions, branching directions, and scales. We solve this optimization problem using an iterative approach to build the tree structure by layers of branches in each iteration. We satisfy our objectives via a combination of a global step that identifies potential branching directions and branch geometries, and a local step that ensures that the output branches satisfy the properties above, ensuring the realism of each branch layer.

We demonstrate the effectiveness of our method across diverse shapes and compare with baselines (Sec 5). We show that compared to different baselines, our method produces significantly better results visually and fulfills our goals more effectively. Our controllable modular trees require two orders of magnitude less memory compared to their traditional counterparts: our experiments found that the average compression ratio of our trees was 1:110.

2 Related Works

We build on the extensive body of work on plant modeling, focusing on modular tree creation and artistic control. We review the most pertinent works below and refer the readers to a survey of procedural generation of virtual worlds [45] and a course on plant modeling [33] for a more comprehensive background.

Non-instanced Tree Modeling Methods. The vast majority of tree modeling methods target non-instanced, or non-modular trees. Early works model tree structures in many different forms, including fractals [30], discrete grammars [1], L-systems [21, 35], and particle systems and cellular automata [2, 8, 40]. L-systems in particular were highly influential, fostering a line of extensions, including capturing continuous development [36] and environmental interaction [27]. A well-known problem with L-systems is the requirement of significant skill and time to create the desired plant models. Some recent work has alleviated this issue, such as utilizing Markov chain Monte Carlo to control stochastic grammars [46], or machine learning to learn L-systems [9, 16], but it remains a difficult task.

Many works have recognized that modeling the underlying biological processes for plant growth is important for realistic branching structures. These biological processes include competition for resources [31, 41, 42], growth simulation [23], environmental interaction [27, 34], root systems [18], and complex forms such as holes or twists [19].

Due to the challenge of modeling realistic branching structures, multiple methods focus on reconstruction of tree models from sensor data, namely images and point clouds. Approaches to reconstruct trees from multi-view images have used L-systems [43], volumetric representations [39], structure from motion [38], particle flow [28], and statistical modeling [5]. For single image reconstruction, user annotation [47] and more recently machine learning [15, 17, 20] has been used. Many approaches using point clouds focus on reconstructing the tree skeleton [12, 50]. Xie et al. [49]

uses point cloud scans to assemble a tree-cut dataset to combine and generate new trees. While modeling methods allow for varying degrees of artistic control, as discussed below, reconstruction-based methods have little to no control of output tree shape with these shapes almost entirely dependent on input images or laser scans.

All above methods create non-instanced, unique, tree geometries that are stored as either dense triangle meshes or skeleton with associated radii. While the latter representation is more compact to store than a mesh, it still requires a lot more space than a modular one. Skeleton-based representations must be converted into a dense mesh for rendering, dramatically increasing their memory footprint. Our work builds on modular tree representations, described next, that require two orders of magnitude less memory (Fig. 1, Sec. 5).

Modular Trees. Modular trees leverage the self-similar nature of tree structures and are modeled using replicated branch connectivity [24] and/or geometry [22] (Fig 1). This approach enables the use of GPU geometry instancing [6] to store and render trees, and has garnered significant industry attention due to its ability to scale when rendering large scenes or simulating large sets of trees. Modular trees have been successfully applied to large-scale simulation, including modeling wildfires [10] and responding to climate changes [32]. Existing work in this space supports minimal artistic control over the crown shape of the tree. Livny et al. [22] generates a modular tree that approximates a laser scan of a tree, and Makowski et al. [24] uses biological processes to model modular trees and has some collision avoidance. Other related work includes work that uses preconfigured branches to fabricate structures that fit a specified 3D surface [14].

Control. Artists require control when modeling trees, and various methods have been developed to provide this flexibility for non-instanced trees. Sketch-based tree modeling methods allow artists to directly draw branch skeleton strokes, lobes, or silhouettes that the tree should adhere to [7, 13, 23, 25, 29, 48, 49]. While these methods can effectively model trees that adhere to the artist-drawn controls, the produced result is a non-modular tree that must be converted to a mesh, and there is no clear way of extending them to modular trees.

Crown shape, also known as silhouette, volume, or overall shape (Fig 1a, grey) is one of the most common properties that artists want to control. Crown shape constraints allow for modeling of diverse trees with the same look, and the shape of the crowns can be defined procedurally. This control mechanism can therefore be used to generate diverse tree collections. Prusinkiewicz et al. [37] aims to produce trees that look like a topiary rather than a tree that happened to naturally grow to fit the specified shape. Boudon et al. [4] constrained tree shapes to a hierarchy of envelopes that defined their overall structure. More recently, works involving competition for space [31, 41, 42] integrated this control directly within the competition for space, allowing the creation of more natural looking trees. The Space Colonization Algorithm (SCA), proposed by Runions et al. [42], constructs a tree skeleton to fit a point cloud sampled from the desired crown shape. This point cloud, called attraction points, guides the growth of the tree skeleton. In each iteration, the attraction points are paired with nearby tree nodes on the skeleton, and the tree nodes grow in the direction

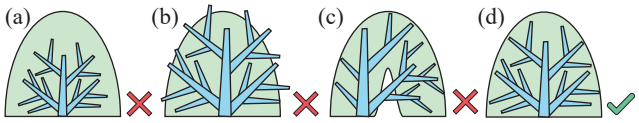


Figure 2: Control requirements: the distance from the crown surface to the tree should not be large (a), tree branches should not protrude outside the crown (b) nor cross areas outside it (c).

of their associated attraction points. Our work is inspired by the line of work for competition for space, but while these methods are designed for non-instanced and non-modular trees, we target modular trees. Our experiments (Sec. 5) demonstrate that naively applying competition for space approaches in our setting results in unnatural looking trees or do not adhere to the specified crown shape. We propose the first, to our knowledge, method for modeling artist controllable modular trees (Fig 1bc). Our method is capable of generating modular trees that combine a natural look with strict adherence to a crown shape constraint.

3 Problem Formulation

We take as inputs the trunk geometry, the desired crown, and available branch modules, where our branch modules are both fixed in connectivity and geometry. We aim to output a modular tree that adheres to an artist-specified crown shape while fulfilling our desired properties to ensure realism. In order to achieve this, we must determine four key attributes for each branch in the tree: branch module, branching position, branching direction, and scale. To ensure that the generated modular tree adheres to the desired properties for realism, we formalize the constraints and optimization objectives from earlier in the introduction. These are defined explicitly as follows:

Crown Shape Adherence. The tree must conform to the artist-specified crown shape. Specifically, the tree should both fill the crown shape as much as possible and stay within the shape simultaneously, as seen in Fig 2.

Scaling Consistency. Scaling for any branch module must remain uniform to prevent texture distortion. For any branch module M, the scaling factor s must satisfy $s_x = s_y = s_z = s$. Furthermore, the scaling factor s must be restricted by the lower and upper bounds defined by the user: $s_\ell \leq s \leq s_u$. For all examples, we use $s_\ell = 0.5, s_u = 2.0$ except for Fig. 1, where $s_\ell = 0.8, s_u = 0.9$. Finally, deviations from the identity scaling ($s = 1$) are penalized to preserve the original proportions of the branch module.

Natural Connectivity. To ensure realistic branch connections, the following constraints are imposed. The radius constraint ensures that the radius r_c of a child branch must not exceed the radius r_p of its parent branch at the connection point: $r_c \leq r_p$. The branching direction constraints ensure that branching directions must conform to realistic angles observed in nature. For a parent branch with tangent direction t_p, the angle θ between the parent tangent and the child branch direction v_c should satisfy:

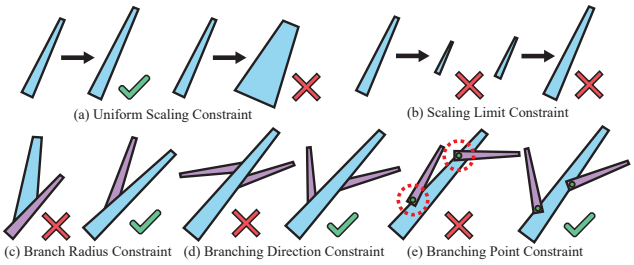


Figure 3: Scaling consistency and natural connectivity constraints: branches should not scale non-uniformly (a) or excessively (b); child branch radii should not exceed the parents' radii (c); branching direction should be consistent with tree species properties (d); child branches should grow away from their parents (e).

$\theta_\ell \leq \cos^{-1}\left(\frac{t_p \cdot v_c}{\|t_p\|\|v_c\|}\right) \leq \theta_u$. As this angle can vary by species, the specific angle bounds are set by the artist for each tree. For most of our examples we use $\theta_\ell = 15°, \theta_u = 90°$. Finally, the branching point constraint ensures that the placements of child branches are natural. A child branch should start on the surface of the parent, and grow outwards with respect to the parent branch. Some intersection between the parent and child branches at the root of the child branch is inevitable and acceptable. The branching point constraint exists to minimize the intersection between the parent and child branches and ensure the root is entirely inside the parent branch. (see Fig. 3e).

These constraints and objectives are integrated into the optimization framework, guiding both the global and local steps of our iterative algorithm.

4 Method

Our method is inspired by SCA [42], and we share the idea of guiding growth using attraction points, point cloud samples that represent the shape of the crown. Unlike SCA [42], which builds a tree skeleton with nodes, our method focuses on the placement of branch modules with the appropriate properties, including branch module type, position, orientation, and scaling in each iteration. The attraction points help us determine the appropriate properties in order to ensure that the resulting tree aligns well with the user-specified crown shape.

4.1 Overview

Our method begins with the trunk and iteratively adds branches layer by layer. Each iteration consists of a global step that identifies potential branch placements and a local step that ensures that our placements are consistent with our desired properties. These steps work in tandem to ensure that the resulting tree adheres to the artist-specified crown shape while maintaining realism. The algorithm terminates when either all attraction points are pruned or no more branches are placed. We illustrate an overview of our method in Fig. 4.

4.2 Initialization

All branch modules are preprocessed to generate an internal skeleton curve used to transfer radius and curve tangent information to our potential branching points. See Appendix A for details on the method.

We then sample the attraction points from the artist-specified crown shape. We first compute a signed distance field from the crown shape. Then, we sample points in an axis-align bounding box surrounding the crown shape. Any points outside the shape, determined with the signed distance field, are then culled. We continue until the target number of points is reached. This set of points is combined with another sampling of points on the surface of the crown shape. For all of our examples, we use a fixed set of 5000 internal and 2000 surface samples for a total of 7000 samples. Using a signed distance field enables the sampling to be biased to sample less or more near the surface of the shape. Since this sampling is randomized, it can be used to trivially create variations of trees with the same crown shape (Fig. 5).

4.3 Global Step

At the start of each iteration, we apply a global step to produce a set of potential branch placements and their associated properties. We first start by sampling potential branching points from the branches generated in the previous iteration, starting with the trunk for the first iteration. Any branching points that lie outside the crown shape are pruned at this point. For special trees where the trunk starts outside the crown shape, we do not prune these points to let the generated tree reach the crown shape first. An example of such a tree can be seen in Fig. 11. Furthermore, the sampled branching points are combined with the unused branching points from the previous iterations. The sampling uses the internal skeleton curve by casting a ray from the skeleton curve to the surface geometry at specific intervals of distance Δx and junction angles α, as seen in Fig. 6. This leads to a more plausible choice of branching points compared to a random sample on the surface geometry. To account for species variation, sampling can be controlled by parameters such as density, number of junction points, and angle of junction. Furthermore, points are culled with a random variable to introduce variance. These points represent the potential branching points where a branch can be placed, and the tangent and radius of the curve of the branch are transferred to each of these points from the internal skeleton curve.

Then, for each attraction point, we must choose its best branching point based on some criteria. A naive approach would be to choose the closest branching point as in SCA [42], but this leads to early termination due to the radius constraint as seen in Fig. 7. Instead, we compute a branching point cost σ for each possible pairing of attraction and branching points. If any attraction and branching point pair crosses over the crown shape surface, the pair is invalidated to ensure crown shape adherence. This is evaluated using Sphere Tracing [11] along the path between the two points. For efficiency, we only compute the branching point cost σ on the 25 closest branching points for each attraction point.

4.3.1 Branch Module Selection. In order to assign a cost to every pair of attraction points and branching points, the type of branch module and its scaling must be chosen for each pair. The goal

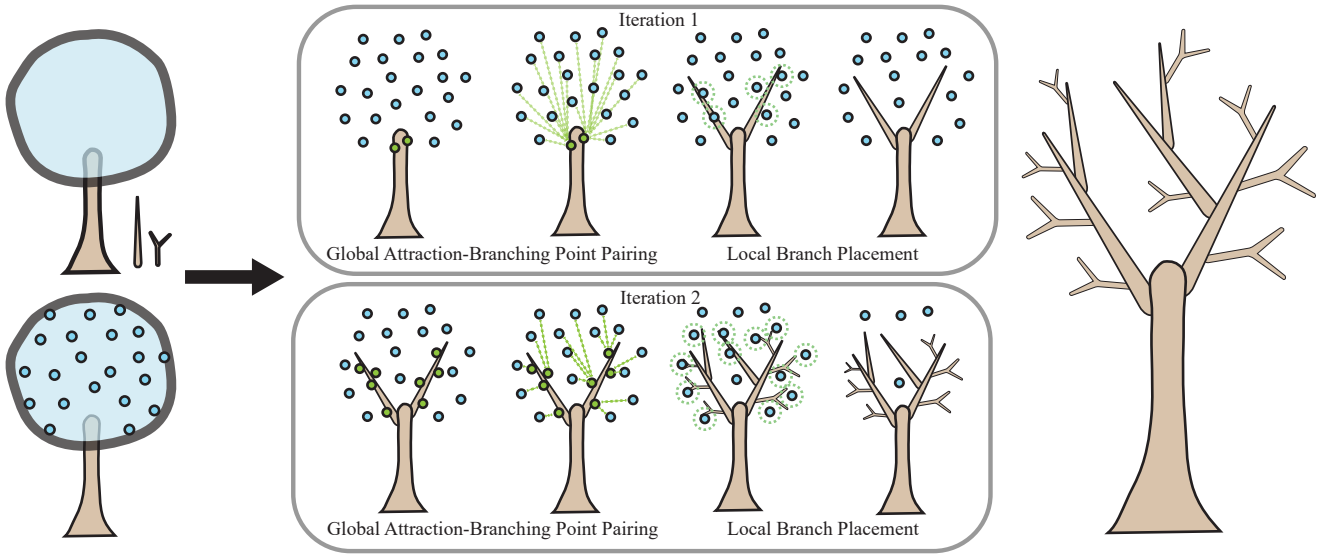


Figure 4: From the input crown shape, trunk, and branch modules we initialize the system by sampling the attraction points (a). We then iteratively add layers of branch modules using a combination of a global step that pairs branching and attraction points, and a local step that finalizes branch choice and placement (b). The final result is produced when attraction points are all pruned or branches can no longer be placed (c).

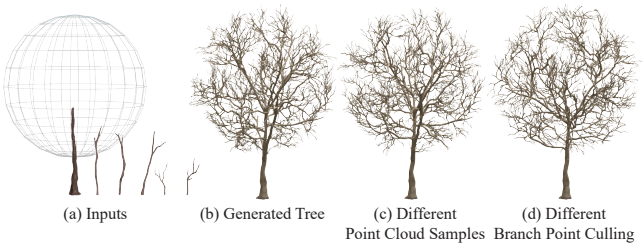


Figure 5: Different tree variations starting from a given input (a) using different randomization seeds. Default result is shown in (b). Changing the random seed for attraction point sampling lead to the same large branch placements while modifying the smaller branch placements (c), and changing the seed for the branching point culling leads to visibly different branch placement (d). For all of the variations, our modular structures would require 1 MB of storage, while storing them as mesh would require 110 MB.

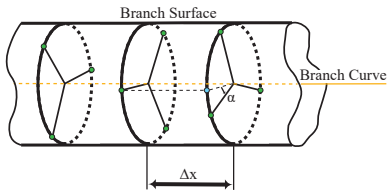


Figure 6: Branching point generation is done by ray-casting from the internal skeleton curve to the surface. The directions are determined by the number of junction points n and the junction angle α, and the interval is specified by Δx.

of the selection of the branch module and the scaling is to find the branch module M and the scaling $s(M)$ such that the scaled branch length $(\text{size}(M) \cdot s(M))$ is as close as possible to the target distance d_t, the distance between the attraction and branching points, without breaking the scaling or radius constraints. Thus, without considering constraints, the ideal scalar is $\frac{d_t}{\text{size}(M)}$. This scalar will be bound by the lower and upper limits of the scaling $[s_\ell, s_u]$ and the ratio of the radii of the parent and child branches $\frac{r_p}{r_c}$. The best branch module M_b and its scaling $s(M_b)$ for each pair are calculated as the branch module that provides the scalar closest to identity after the constraints are applied:

$$M_b = \arg\min_{M \in \mathcal{M}} |s(M) - 1|,$$
$$\text{where } s(M) = \max\left(s_\ell, \min\left(\frac{d_t}{\text{size}(M)}, s_u, \frac{r_p}{r_c}\right)\right) \quad (1)$$

4.3.2 Branching Point Selection. Once a branch module and scaling have been chosen for every pair of attraction points and branching points, we must select a single branching point for each of the attraction points. For each attraction point, the branching point with the lowest cost is selected. We first calculate d_c for each attraction point, which is the smallest distance between the attraction point and all of its paired branching points. By construction, d_c is strictly less than or equal to d_t.

We would like our cost to be structured so that when d_c is small, which means that the attraction point has at least one branching

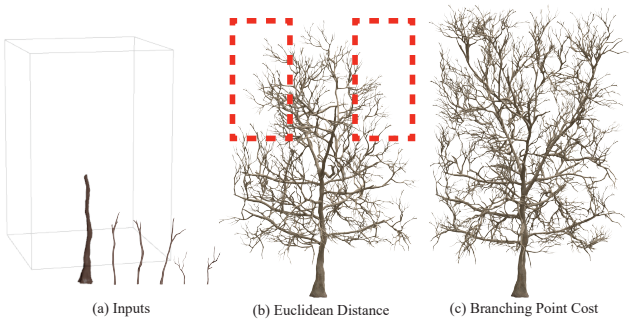


Figure 7: Naively using Euclidean distance for the attraction-branching point pairing leads to early termination (b), failing to conform to the input crown shape (a). Using our branching point cost promotes the growth of branches in areas with higher branch radius rather than proximity, so that all of the attraction points can be reached to conform closely to the crown shape (c). Storing (c) as modular structures requires 1 MB, while storing them as a mesh requires 150 MB.

point close in proximity, the branching point should be selected based on the distance to its attraction point, that is, d_t. On the other hand, when d_c is large, which means that all branching points are far away from the attraction point, the branching point should be selected based on whether a branch placed at that point can reach the attraction point or not. This means that when d_c is large, we often prioritize branching points where the radius of the parent branch is high, so a thick branch can be used, even if it is not the closest branching point to the attraction point. This helps prevent the problem of early termination when only using Euclidean distance proximity (see Fig. 7), as the selected branches will be thicker and able to reach the farther attraction points. We measure this as the ratio between the length of the scaled branch ($\text{size}(M) \cdot s(M)$) and the distance to the target d_t. In the ideal case, the ratio will be equal to 1.

To smoothly blend between these two cases, we use two positive weights λ_1, λ_2 that sum to 1 and set it to the saturation function $\lambda_1 = \frac{d_c}{d_c+C}$ with a constant C that controls the saturation rate. When d_c is small, $\lambda_1 \approx 0, \lambda_2 \approx 1$, so the second term will be dominant. When d_c is large, the first term will be dominant. To account for the scale difference between the two terms, we also use the saturation function for the target distance d_t with the same constant C. This prevents d_t from disproportionately affecting the cost by normalizing it to be between 0 and 1. We set this constant C to be approximately $1/5$ of the largest axis of the bounding box containing both the trunk and crown shape.

Finally, to compute the branching point cost σ, we take a weighted sum of the two terms:

$$\sigma = \lambda_1 \cdot \left| \frac{\text{size}(M_b) \cdot s(M_b)}{d_t} - 1 \right| + \lambda_2 \cdot \frac{d_t}{d_t + C}, \quad \text{where } \lambda_1 = \frac{d_c}{d_c + C}, \quad \lambda_2 = (1 - \lambda_1) \tag{2}$$

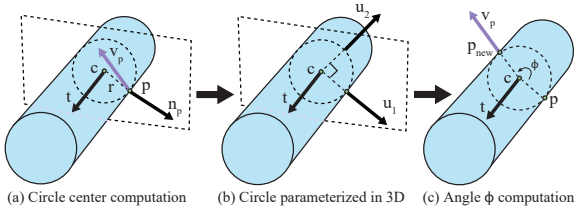


Figure 8: Branching points are adjusted when its position p and branching direction v are misaligned with the parent branch. A circular cross section is computed (a), parameterized (b), and a rotation angle ϕ for the branching point is found such that the branching direction will now face outward from the parent branch (c).

4.4 Local Step

In our local step, we operate on the set of potential branch placements produced from the global step and finalize the choice of branch placements and their associated properties. Only branching points that are associated with at least one attraction point continue to the local step. We compute its branching direction v as the average direction towards all of its associated attraction points.

4.4.1 Branching Point Adjustment. In some cases, the choice of branching point and branching direction can be misaligned, violating the branching point constraint. In most cases, the misalignment is due to the branching point being on the wrong side of the parent branch, thus causing severe intersections between the parent and child branches (see Fig. 3e). To resolve this, we want to rotate the branching point along the axis of the parent branch (skeleton curve tangent) so that the child branch is not blocked in its branching direction (Fig. 8). To accomplish this, for each branching point p, we can retrieve a radius r and curve tangent t from the internal skeleton curve. We also have the surface normal of the parent branch at the point p, which we denote as n. We now assume that the cross section, formed by the intersection of the parent branch with the plane defined by point p and normal direction t, can be closely approximated by a circle. With this assumption, we can project n to the plane to get n_p, and get a circular cross section of the parent branch with radius r and center $c = p - (r \cdot n_p)$ (Fig. 8a). Then, the circle can now be parameterized in 3D with basis vectors $u_1 = n_p$, $u_2 = t \times u_1$ (Fig. 8b). We then find the angle ϕ that aligns v_p so that it faces outside the circle, shown in Eq. 3, where v_p is the branching direction v projected to the cross-sectional plane (Fig. 8c). We can then compute the new rotated position of the branching point p_{new}.

$$\begin{aligned} \phi &= atan2((v_p \cdot u_2), (v_p \cdot u_1)) \\ p_{\text{new}} &= c + r \cdot (cos(\phi) u_1 + sin(\phi) u_2) \end{aligned} \tag{3}$$

4.4.2 Branching Direction Adjustment. The branching direction constraint is often violated when the branching directions stray far from the desired branching angles, leading to unrealistic branching structures (see Fig. 3d). To achieve better realism, we optionally apply a bias to align the branching direction closer to the desired

angle. For our examples, we add the curve tangent t to our branching direction with a weight (between 0.15-0.30) and renormalize the vector, aligning our branching direction closer to the curve tangent direction. Afterward, any branches with an angle that does not fit within the bounds $[\theta_\ell, \theta_u]$ are removed, where $\theta_\ell = 15°, \theta_u = 90°$ for most of our examples.

4.4.3 Final Branch Module Selection. In the global step, we computed the selection of the branch module and the scaling between each pair of attraction and branching points. However, the selection changes in the local step since the branching point and branching direction could have changed in the branching point adjustment or branching direction adjustment steps. Furthermore, branching points can be associated with multiple attraction points, which requires a common solution. We use Eq. 1 again to select the final branch module and scale, where the target distance is set as a weighted distance to all associated attraction points. The weighted distance incorporates both the average distance to the attraction points and the distance to the surface of the crown shape. We incorporate distance to the surface in early iterations because the average distance can be dominated by attraction points that are very close to the trunk rather than close to the surface. Finally, we cap the weighted distance by the distance to the surface in the branching direction so that we do not intersect outside the crown shape.

We skip the branch point if there is no branch whose scaling is between the minimum and maximum scaling. We prune all the attraction points that are within kill distance d_k of any of the potential branching points at this point before moving on to the next iteration. For our examples, we use a d_k between 15 and 25, which is roughly between 1/20 to 1/10 of the largest axis of the bounding box containing both the trunk and crown shape.

5 Results

We evaluate our generated modular trees qualitatively, using diverse input crown shapes, different sets of branch modules, and other tree parameters (Figs. 1, 5, 7, 10, 11). In all cases the output trees look natural and adhere to the input crown shapes, including highly non-standard crown shapes, such as a torus (Fig. 9, top), a tetrahedron (Fig. 11, bottom) or a collection of cubes (Fig. 11, top). The memory footprint of the trees we generate ranges from 1 to 2 megabytes, while storing these or comparable complexity trees (branches only) as meshes requires two orders of magnitude more space.

We further ablate our method against different baselines inspired by prior skeleton-based methods, and perform different direct ablations of the components of our method, as discussed next.

5.1 Ablations

Baselines. While no artist controllable methods for modular tree modeling exist, space colonization approaches that generate traditional trees that conform to a given crown shape, provide a potential starting point for addressing this problem and serve as inspiration for our work. We therefore compare our method to a series of baselines that use these approaches and specifically SCA [42] as a starting point. Our most straightforward baseline (Fig. 9b) modifies SCA [42] to be able to generate modular trees. Specifically, we use the initialization step shared by both SCA [42] and our method, generating random attraction points inside and on the crown surface at the start of the process and generating random branching points on the branches added in each iteration. We then follow SCA [42] and select the branch module based on the average attraction and branching point distance as the target distance. We then (Fig. 9cd) augment this baseline by enforcing the constraints used by our method to ensure a more natural tree look; as discussed below and as the figures demonstrate, such naive augmentation is not sufficient to produce satisfactory results.

In Fig. 9b we show the generated modular trees from the baseline. The baseline produces multiple visible artifacts compared to our method. The most blatant artifact is the lack of adherence to the desired crown shape. The baseline tree struggles to stay within the desired crown shape when the crown shape contains concavities, as there is no explicit restriction that prevents branches from crossing outside of the shape. Having to impose a convex requirement to avoid this problem would limit artists significantly since concave crown shapes are quite common. Furthermore, there are many visible branch radius inversions as well as unrealistic and inconsistent branching angles that reduce the realism of the tree.

In Fig. 9c we show the generated modular trees from the same baseline, but with our constraints for crown shape adherence enforced. Even with these constraints, we see that the baseline carries the same issues with regards to branch radius inversions and branching angles.

Finally, in Fig. 9d we show the generated modular trees from the baseline but with both constraints for crown shape adherence and branch radius enforced. We can see immediately that the generated branches reduce in scale drastically as layers are added. The branches reduce to sizes that are not consistent with artist expectations of realism. Furthermore, the generated trees do not sufficiently fill the desired crown shape, and the method essentially terminates prematurely due to its inability to grow more branches without violating the radius constraints.

Method Components. In Fig. 5 we showcase our ability to generate diverse trees using the same crown shape and set of branch modules. By varying the random seeds for attraction point sampling and branching point culling, variations can be generated trivially. In Fig. 10 we show our generated modular trees using a decreasing number of available branch modules. We show that even at an extremely low number of branch modules, our method produces a plausible tree adhering to the crown shape. In Fig. 11 we show results of our modular trees with more unique crown shapes and different sets of branch modules, as well as the placement of modular leaves.

5.2 Implementation Details

We implement our framework with Houdini [44], using both C++ and VEX. Much of our operations are parallel and heavily benefit from parallel execution via Shape Operators. Specifically, all of the operations per-attraction point and per-branching point can be executed in parallel. For rendering, we represent our branch modules as instanced geometry to save on memory. The computation time (time it takes to model the tree) of our algorithm on our simplest tree, Fig. 9 (bottom), is 500ms with 120 branch modules placed and

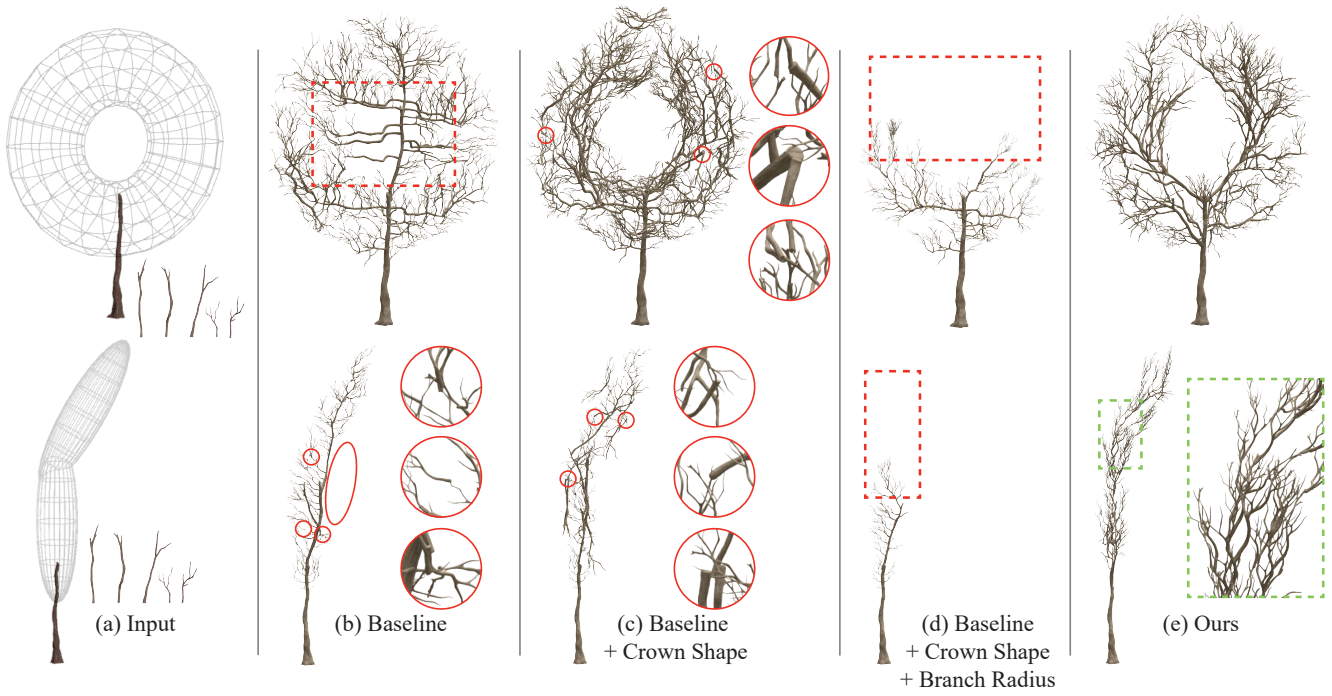


Figure 9: Comparisons of our modular trees (e) against baselines enforcing a subset of our constraints (b-d). (b) Baseline inspired by SCA (Runions et al.) [42], places attraction points inside the crown shape, but does not enforce any other constraints. (c) Enforces strict crown shape adherence on top of the baseline, and (d) enforces both crown shape adherence and the branch radius constraint. Noted by the ellipsoid next to the tree on the bottom of (b), the large branch can only generate child branches on one side of the tree due to being too close to the crown shape boundary. Storing the trees (top, bottom) as modular structures require 1 MB and 0.9 MB respectively, while storing as a mesh requires 80 MB and 30 MB respectively.

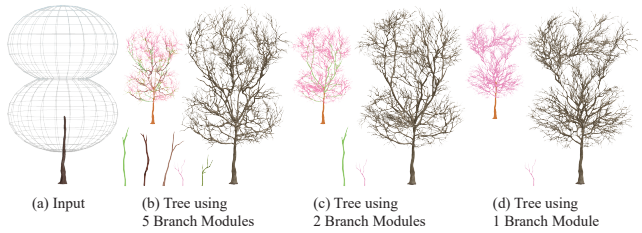


Figure 10: From the input (a), we produce multiple modular trees using a different number of branch modules (b-d). While the diversity of branches goes down, even with 1 or 2 branches modules we achieve natural looking trees that adhere to the crown shape. Storing these trees as modular structures require sizes ranging between 0.6 MB to 1 MB, compared to 140 MB to 170 MB for storing them as a mesh.

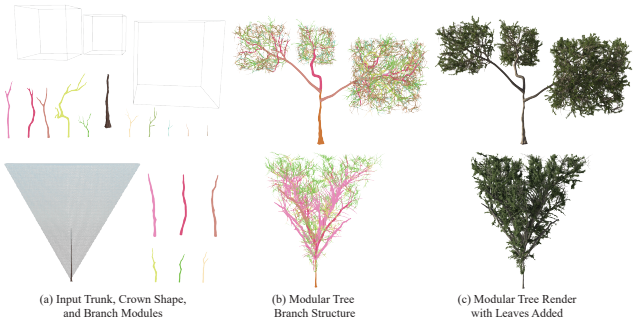


Figure 11: Unique tree shapes and different types of branch modules. The tree on the top features a slight variation to our method to allow growth from a trunk that sits outside of the crown shape(s). The bottom tree features branch modules created for pine-like trees. Storing the trees (top, bottom) as modular structures require 1.8 MB and 0.8 MB respectively, while storing as a mesh requires 180 MB and 120 MB respectively.

the computation time for our most complex tree, Fig. 11 (top), is 2400ms with 4600 branch modules placed. To generate the trees in our results we used a Intel®Xeon Gold 6226R, 16 × 2.90GHz with 96GB RAM, and an NVIDIA Quadro RTX 5000 GPU (16GB RAM).

6 Conclusions

We have demonstrated that our method effectively enables the generation of modular trees that adhere to artist-specified crown shapes, overcoming the limitations of previous approaches. By integrating constraints on shape, scaling, and natural appearance into a constrained mixed-variable optimization framework, we achieve the balance between realism and control. Our controllable modular tree modeling method represents a significant step forward in balancing artistic control, modular efficiency, and realism.

There is an inherent conflict between control and realism: adhering strictly to artist restrictions while satisfying strict botanical tree properties may not always be possible. This challenge is shared by all methods that support strong artist control. In particular, our iterative approach, which builds the tree layer by layer, does not inherently support the simulation of complex tropism behaviors, such as large branches drooping or bending significantly under the influence of gravity. Additionally, because our branches do not grow after placement, this can lead to creating dead ends where no additional growth can occur from the tip of the branch. These global tropism effects often require a holistic approach to tree generation, where the structural stability, deformation, and growth of the entire tree are considered simultaneously rather than in a localized and iterative manner. An interesting direction would be to solve this kind of challenge in a post-processing step, where the structure of the tree is already decided. Because our method is greedy and forward-only, another promising direction would be to extend our method to allow for backtracking, possibly by trying different sets of branching point samples.

Another limitation of our method is self-intersections. Although our algorithm inherently discourages self-intersections since attraction points are exclusive to a single branching point in any iteration, self-intersections are not explicitly prevented. When trees are viewed from very close, self-intersections between branches can be visible. Future work could focus on the intersection-free construction of trees, or deintersecting any branches that currently exist.

Acknowledgments

This work was supported by Mitacs through the Mitacs Accelerate program.

References

[1] Masaki Aono and Tosiyasu L. Kunii. 1984. Botanical Tree Image Generation. *IEEE Computer Graphics and Applications* 4, 5 (1984), 10–34.
[2] James Arvo and David B. Kirk. 1988. Modeling plants with environment-sensitive automata. *Proceedings of Ausgraph'88* (1988), 27–33.
[3] James Bartolozzi and Matt Kuruc. 2017. A hybrid approach to procedural tree skeletonization. In *ACM SIGGRAPH 2017 Talks (SIGGRAPH '17)*. Article 53, 2 pages.
[4] Frédéric Boudon, Przemyslaw Prusinkiewicz, Pavol Federl, Christophe Godin, and Radoslaw Karwowski. 2003. Interactive Design of Bonsai Tree Models. *Computer Graphics Forum* 22, 3 (2003), 591–599.
[5] Derek Bradley, Derek Nowrouzezahrai, and Paul Beardsley. 2013. Image-based reconstruction and synthesis of dense foliage. *ACM Trans. Graph.* 32, 4, Article 74 (2013), 10 pages.
[6] Francesco Carucci. 2005. Inside Geometry Instancing. In *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*, Matt Pharr and Randima Fernando (Eds.). Addison-Wesley Professional, Chapter 3. https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-3-inside-geometry-instancing
[7] Xuejin Chen, Boris Neubert, Ying-Qing Xu, Oliver Deussen, and Sing Bing Kang. 2008. Sketch-based tree modeling using Markov random field. *ACM Trans. Graph.* 27, 5, Article 109 (2008), 9 pages.
[8] Ned Greene. 1989. Voxel space automata: modeling with stochastic growth processes in voxel space. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*. 175–184.
[9] Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. 2020. Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. *ACM Trans. Graph.* 39, 5, Article 155 (2020), 13 pages.
[10] Torsten Hädrich, Daniel T. Banuti, Wojtek Pałubicki, Sören Pirk, and Dominik L. Michels. 2021. Fire in paradise: mesoscale simulation of wildfires. *ACM Trans. Graph.* 40, 4, Article 163 (2021), 15 pages.
[11] John C. Hart. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12 (1996), 527–545.
[12] Shaojun Hu, Zhengrong Li, Zhiyi Zhang, Dongjian He, and Michael Wimmer. 2017. Efficient tree modeling from airborne LiDAR point clouds. *Computers Graphics* 67 (2017), 1–13.
[13] T. Ijiri, S. Owada, and T. Igarashi. 2006. Seamless Integration of Initial Sketching and Subsequent Detail Editing in Flower Modeling. *Computer Graphics Forum* 25, 3 (2006), 617–624.
[14] Maria Larsson, Hironori Yoshida, and Takeo Igarashi. 2019. Human-in-the-loop fabrication of 3D surfaces with natural tree branches. In *Proceedings of the 3rd Annual ACM Symposium on Computational Fabrication (SCF '19)*. Article 1, 12 pages.
[15] Jae Joong Lee, Bosheng Li, Sara Beery, Jonathan Huang, Songlin Fei, Raymond A. Yeh, and Bedrich Benes. 2024. Tree-D Fusion: Simulation-Ready Tree Dataset from Single Images with Diffusion Priors. arXiv:2407.10330 [cs.CV]
[16] Jae Joong Lee, Bosheng Li, and Bedrich Benes. 2023. Latent L-systems: Transformer-based Tree Generator. *ACM Trans. Graph.* 43, 1, Article 7 (2023), 16 pages.
[17] Bosheng Li, Jacek Kałużny, Jonathan Klein, Dominik L. Michels, Wojtek Pałubicki, Bedrich Benes, and Sören Pirk. 2021. Learning to reconstruct botanical trees from single images. *ACM Trans. Graph.* 40, 6, Article 231 (2021).
[18] Bosheng Li, Jonathan Klein, Dominik L. Michels, Bedrich Benes, Sören Pirk, and Wojtek Pałubicki. 2023. Rhizomorph: The Coordinated Function of Shoots and Roots. *ACM Trans. Graph.* 42, 4, Article 59 (2023), 16 pages.
[19] Bosheng Li, Nikolas Alexander Schwarz, Wojtek Pałubicki, Sören Pirk, and Bedrich Benes. 2024. Interactive Invigoration: Volumetric Modeling of Trees with Strands. *ACM Trans. Graph.* 43, 4, Article 146 (2024), 13 pages.
[20] Yuan Li, Zhihao Liu, Bedrich Benes, Xiaopeng Zhang, and Jianwei Guo. 2024. SVDTree: Semantic Voxel Diffusion for Single Image Tree Reconstruction. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4692–4702.
[21] Aristid Lindenmayer. 1968. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology* 18, 3 (1968), 280–299.
[22] Yotam Livny, Soeren Pirk, Zhanglin Cheng, Feilong Yan, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chen. 2011. Texture-lobes for tree modelling. In *ACM SIGGRAPH 2011 Papers (SIGGRAPH '11)*. Article 53, 10 pages.
[23] Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. 2012. TreeSketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling (SBIM '12)*. 107–120.
[24] Miłosz Makowski, Torsten Hädrich, Jan Scheffczyk, Dominik L. Michels, Sören Pirk, and Wojtek Pałubicki. 2019. Synthetic silviculture: multi-scale modeling of plant ecosystems. *ACM Trans. Graph.* 38, 4, Article 131 (2019), 14 pages.
[25] Gilda Manfredi, Nicola Capece, Ugo Erra, and Monica Gruosso. 2023. TreeSketchNet: From Sketch to 3D Tree Parameters Generation. *ACM Trans. Intell. Syst. Technol.* 14, 3, Article 41 (2023), 29 pages.
[26] Chris Michael and Arunachalam Somasundaram. 2020. Termite: DreamWorks Procedural Environment Rigging Tool. In *ACM SIGGRAPH 2020 Talks (SIGGRAPH '20)*. Article 9, 2 pages.
[27] Radomír Měch and Przemyslaw Prusinkiewicz. 1996. Visual models of plants interacting with their environment. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. 397–410.
[28] Boris Neubert, Thomas Franken, and Oliver Deussen. 2007. Approximate image-based tree-modeling using particle flows. *ACM Trans. Graph.* 26, 3 (2007), 88–es.
[29] Makoto Okabe, Shigeru Owada, and Takeo Igarashi. 2007. Interactive design of botanical trees using freehand sketches and example-based editing. In *ACM SIGGRAPH 2007 Courses (SIGGRAPH '07)*. 26–es.
[30] Peter E. Oppenheimer. 1986. Real time design and animation of fractal plants and trees. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. 55–64.
[31] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. 2009. Self-organizing tree models for image synthesis. *ACM Trans. Graph.* 28, 3, Article 58 (2009).

[32] Wojtek Pałubicki, Miłosz Makowski, Weronika Gajda, Torsten Hädrich, Dominik L. Michels, and Sören Pirk. 2022. Ecoclimates: climate-response modeling of vegetation. *ACM Trans. Graph.* 41, 4, Article 155 (2022), 19 pages.
[33] Sören Pirk, Bedrich Benes, Takashi Ijiri, Yangyan Li, Oliver Deussen, Baoquan Chen, and Radomir Měch. 2016. Modeling plant life in computer graphics. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. Article 18, 180 pages.
[34] Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Radomír Měch, Bedrich Benes, and Oliver Deussen. 2012. Plastic trees: interactive self-adapting botanical tree models. *ACM Trans. Graph.* 31, 4, Article 50 (2012), 10 pages.
[35] Przemyslaw Prusinkiewicz. 1986. Graphical Applications of L-Systems. In *Proceedings of Graphics Interface and Vision Interface '86 (GI '86)*. 247–253.
[36] Przemyslaw Prusinkiewicz, Mark S. Hammel, and Eric Mjolsness. 1993. Animation of plant development. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*. 351–360.
[37] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. 1994. Synthetic topiary. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. 351–358.
[38] Long Quan, Ping Tan, Gang Zeng, Lu Yuan, Jingdong Wang, and Sing Bing Kang. 2006. Image-based plant modeling. *ACM Trans. Graph.* 25, 3 (2006), 599–604.
[39] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. 2004. Volumetric reconstruction and interactive rendering of trees from photographs. *ACM Trans. Graph.* 23, 3 (2004), 720–727.
[40] William T. Reeves and Ricki Blau. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*. 313–322.
[41] Yodthong Rodkaew, Prabhas Chongstitvatana, Suchada Siripant, and Chidchanok Lursinsap. 2003. Particle Systems for Plant Modeling. *Plant Growth Modeling and Applications. Proceedings of PMA03* (2003).
[42] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. 2007. Modeling trees with a space colonization algorithm. In *Proceedings of the Third Eurographics Conference on Natural Phenomena (NPH'07)*. 63–70.
[43] Ilya Shlyakhter, Max Rozenoer, Julie Dorsey, and Seth Teller. 2001. Reconstructing 3D Tree Models from Instrumented Photographs. *IEEE Comput. Graph. Appl.* 21, 3 (2001), 53–61.
[44] SideFX. 2024. *Houdini*. https://www.sidefx.com
[45] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. 2014. A Survey on Procedural Modelling for Virtual Worlds. *Computer Graphics Forum* 33, 6 (2014), 31–50.
[46] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. 2011. Metropolis procedural modeling. *ACM Trans. Graph.* 30, 2, Article 11 (2011), 14 pages.
[47] Ping Tan, Tian Fang, Jianxiong Xiao, Peng Zhao, and Long Quan. 2008. Single image tree modeling. In *ACM SIGGRAPH Asia 2008 Papers (SIGGRAPH Asia '08)*. Article 108, 7 pages.
[48] J. Wither, F. Boudon, M.-P. Cani, and C. Godin. 2009. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *Computer Graphics Forum* 28, 2 (2009), 541–550.
[49] Ke Xie, Feilong Yan, Andrei Sharf, Oliver Deussen, Hui Huang, and Baoquan Chen. 2016. Tree Modeling with Real Tree-Parts Examples. *IEEE Transactions on Visualization and Computer Graphics* 22, 12 (2016), 2608–2618.
[50] Hui Xu, Nathan Gossett, and Baoquan Chen. 2007. Knowledge and heuristic-based modeling of laser-scanned trees. *ACM Trans. Graph.* 26, 4 (2007), 19–es.

Appendix A Branch Module Preprocessing

Our method relies on the existence of an internal skeleton curve to retrieve radius and curve tangent information for each branching point. Many methods already exist to solve this particular task, such as [3, 14, 26] to name a couple. For our method, we have implemented a custom algorithm for this internal skeleton curve generation to provide us with flexibility with our method. Unlike some other approaches [14, 26], our method can compute skeleton curves on meshes with forks, and we utilize a 2D geodesic field rather than a 3D distance field [3].

To represent the skeleton curve, we utilize a hierarchy of splines so that we can handle forks in the branch module geometry. Our method begins by computing a geodesic distance field and accumulating the shortest distances along the mesh edges from every vertex to the ground plane. We find the maximum distance on the mesh and divide the distance according to how many segments we would like to have on the skeleton curve. This gives us the geodesic distance values to compute each isoline in the mesh.

To compute each isoline, we iterate through every triangle face on the mesh and compute the minimum and maximum geodesic distances of the vertices. If the geodesic distance value is between the minimum and maximum of the vertices, it must be located on two of the three edges of the face. We compute the two points on the two edges with interpolation to match the desired geodesic distance, and insert the segment formed by the two points into a disjoint set. We repeat this process for each face. After all of the faces have been processed, we can obtain the isolines from the disjoint set.

In the case where there is a fork in the mesh, the isolines will diverge into multiple islands. We treat the largest isoline as the parent and the other isolines as the child. To create splines of the skeleton curves, we compute the centroids of each isoline and connect them.