DeepTree: Modeling Trees with Situated Latents

Xiaochen Zhou, Bosheng Li, Bedrich Benes, *IEEE Senior Member*
Songlin Fei, Sören Pirk

Abstract—In this paper, we propose *DeepTree*, a novel method for modeling trees based on learning developmental rules for branching structures instead of manually defining them. We call our deep neural model "situated latent" because its behavior is determined by the intrinsic state -encoded as a latent space of a deep neural model- and by the extrinsic (environmental) data that is "situated" as the location in the 3D space and on the tree structure. We use a neural network pipeline to train a situated latent space that allows us to locally predict branch growth only based on a single node in the branch graph of a tree model. We use this representation to progressively develop new branch nodes, thereby mimicking the growth process of trees. Starting from a root node, a tree is generated by iteratively querying the neural network on the newly added nodes resulting in the branching structure of the whole tree. Our method enables generating a wide variety of tree shapes without the need to define intricate parameters that control their growth and behavior. Furthermore, we show that the situated latents can also be used to encode the environmental response of tree models, e.g., when trees grow next to obstacles. We validate the effectiveness of our method by measuring the similarity of our tree models and by procedurally generated ones based on a number of established metrics for tree form.

Index Terms—Botanical Tree Models, Deep Learning, Shape Modeling, Generative Methods

✦

1 INTRODUCTION

VEGETATION is ubiquitous in almost all environments, ranging from vast outdoor landscapes to indoor spaces. Consequently, models of trees and plants serve as essential assets in applications such as games and movies, mixed reality, architecture and urban planning, agriculture, and forestry, or even the training of autonomous agents. For many applications, it is of utmost importance to define plant models with a high degree of geometric detail to produce high-quality renderings or model interactions with plant models, e.g., to plan a path around them. Procedural models have proven to effectively capture the wide variety of plant forms found in Nature and efficiently generate large sets of models.

Procedural and developmental modeling of vegetation in CG has been used in diverse ways, often combined with other approaches, such as modeling the environmental response of trees [1], the reconstruction of tree models from images or point sets [2], [3], [4], inverse procedural modeling [5], by focusing on interactive modeling with user-defined sketches [6], or by leveraging neural networks that compose trees of procedurally generated branching patterns [7]. The breadth of these approaches is a testament to the complexity of realistically modeling plants. However, despite the progress, generating trees with procedural models remains an open and challenging research problem. In particular, developmental models are difficult to control. They use a large set of mutually dependent non-linear parameters that affect the generated structures in unclear ways.

- *X. Zhou, B. Li, and B. Benes were with the Department of Computer Science, Purdue University, West Lafayette, IN, 47907.*
- *S. Fei was with the Department of Forestry, Purdue University, West Lafayette, IN, 47907.*
- *S. Pirk was with Adobe Research, San Jose, CA, 95110*

We seek to develop a generative model that would capture the resulting shape of the bud's behavior. While engineering such behavior is challenging because each tree node experiences varying conditions, deep learning is suitable for learning and encoding the bud's response to such conditions. We propose a novel deep neural model that we call "situated latent" because its behavior is determined by the intrinsic state -encoded as a latent space of a deep neural model- and by the extrinsic (environmental) data that is "situated" as the location in the 3D space and on the tree structure. We use a large dataset of 3D tree models to train a pipeline of neural networks on *branchlets* – atomic branching structures consisting of a node and its immediate children. We learn a situated latent space, a representation that makes predictions for branch growth based on where a branch node is located in the growth space of a tree model. Once trained, the neural networks can predict the topology (number of children) and geometry (position and the branch width of children) for a single input node. The deep neural model is a compact representation that encodes the tree response based on the environment (extrinsic) and the positional information within the tree structure (intrinsic). We encode the entire tree and its environmental response as a single compact deep neural representation. Our model is the first deep-learning model for tree branching, and it overcomes the difficult control of procedural models by learning the possible scenarios that a branchlet can encounter.

A new tree is generated iteratively by executing the network on individual nodes of the branching structure. Starting from the root node, the single network is executed for each node, and it predicts the number and positions of its children. The predicted child nodes are added to the branch graph, and in the next iteration, the network predicts their immediate children again. Similar to procedural modeling algorithms that are also implemented recursively or iteratively, this allows producing complex branching structures only with a few iterations. While *DeepTree* learns
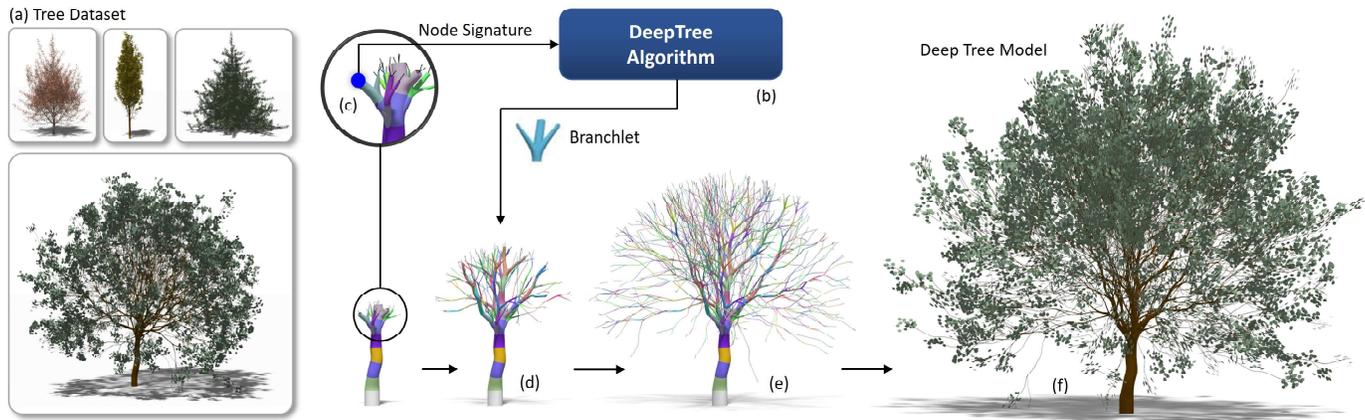
Fig. 1. Illustration of our *DeepTree* generation approach: we use a common procedural model to generate a dataset of different species (a). As part of our algorithm, we train a neural network pipeline (b) to predict the topological and geometric properties of branching patterns based on a situated latent space. This means that branch features are predicted locally and by only considering a single node (c). Given a node, our network pipeline is able to predict branchlets, the branching structures of the immediate successors of a node – thereby mimicking the developmental process of a tree model (d, e). Eventually, our algorithm will automatically terminate when a complex tree model has been generated (f). Branchlets are shown with colors in (c), (d), and (e).

the response of branchlets, it is *not* a reconstruction method as the generated trees are non-deterministic and only share low-level common properties of the entire training set.

Our results show that the situated latent space can encode and generate diverse branching patterns of multiple tree species by successfully predicting branch growth's primary and secondary attributes (e.g., branching angles, internode length, branch width, etc.). The network also predicts the termination of branches (no immediate children) for twigs toward the outer regions of the tree crown, which then terminates the development of a model. Moreover, we show that the situated latent space can also be used to encode the environmental response of trees, i.e., the adaptation of branching structures according to tropisms and obstacles in the model's environment.

We are not aware of any deep neural models capable of generating 3D biological tree shape. Therefore, we validate the effectiveness of our method in encoding tree form by comparing tree models generated with a state-of-the-art procedural model, and our qualitative results show that our method generates branching structures with almost identical geometric properties. We measure the similarity of branching structures based on histograms over the geometric properties at different levels of fully developed tree models. We also use a recently proposed perceptual metric to assess the quality of generated tree models further.

Examples of a procedurally generated tree and a *DeepTree* model are shown in Fig. 1. The *DeepTree* model is generated by iteratively querying a neural network on the current set of terminal nodes. The network predicts new branch nodes – thereby mimicking the growth process – that are then added to the branch graph. A complex tree model can be generated with a few iterations.

In summary, our contributions are: (1) we propose a novel method for learning to predict the topological and geometric properties of branchlets based on a novel neural network pipeline; (2) we show that the learned situated latent space can be used to iteratively generate tree models,

which provides a novel way to encode and generate tree form; (3) we validate our method through state-of-the-art qualitative and quantitative metrics.

2 RELATED WORK

Due to the importance of vegetation in several application domains, modeling trees and plants has received significant research attention over the past decades (see review [8]). The goal of the early approaches was to focus on faithfully modeling the morphology of branching structures. Approaches include fractals [9], repetitive patterns [10], environmentally sensitive automata [11], and particle-systems [12]. L-systems [13] – as one of the most fundamental procedural approaches – enable to define the tree growth as a set of parallel production rules and has been shown to allow for capturing a wide variety of plant forms, even when considering the environmental response of plants [14]. However, while L-systems are a powerful mechanism for generating plant form, defining production rules to generate complex trees is an elaborate task that requires a substantial amount of knowledge, which often renders their use for applications impractical. Early procedural models [15] aim to address these shortcomings by combining rule-based approaches with geometric modeling and user interaction [16]. However, they are more specialized and do not support modeling the same range of branching patterns. Finally, the goal of inverse procedural modeling [5], [17] is to automatically find the parameter values of procedural models of plants through optimization schemes or to find the production rules [18].

Recent procedural methods for plants emphasize more principled representations [19], [20] or the underlying biological processes of plant development. Šťava et al., [5], for example, add biological priors to their procedural model to provide more nuanced control for shape-defining parameters. Other methods focus on the self-organization of plants [1], their environmental response [14], [21], the competition for resources [11], [22], or modeling features of spe-
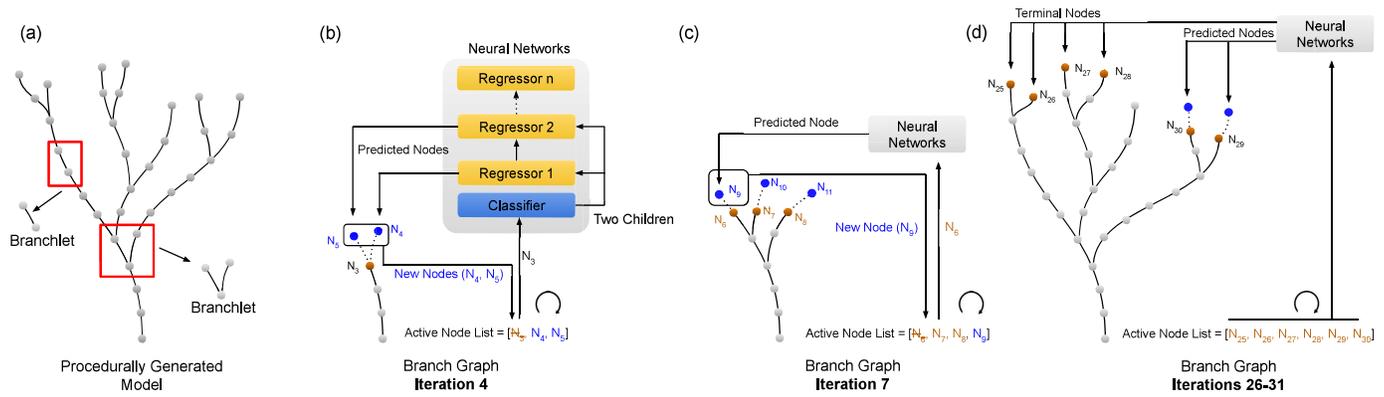
Fig. 2. Overview: We use branchlets of tree models generated with a common procedural model to train our neural networks (a). To generate a tree, we iteratively use our neural network pipeline to predict the topology and geometry of new nodes. A classification network predicts the number of new children for a node that triggers a cascade of regression networks predicting the signatures of each child by using the output of the previous regression network (e.g., Regressor 2 uses the parent node signature as well as the output signature of Regressor 1). Starting with the root node, we maintain a list of active nodes. We pop an active node from the list in each iteration and trigger the neural networks to predict its children. The children are then added to the branch graph and the list of active nodes (b, c). The generation of a tree model stops when the classification network predicts zero children for an active node, which happens individually for different branches (d).

cific types of plants [23]. The availability of more powerful graphics hardware-enabled approaches that use procedural models of plants to simulate the animation of growth [24], [25], their physical response to wind [26], [27] and fire [28], advanced material properties of wood [29], [30], or to find efficient representations of tree models [31]. More recently, module-based tree representations have gained popularity to enable simulation of complex vegetation-related phenomena, such as wildfires [32] or climatic gradients [33].

Sketch-based and data-driven approaches often use procedural models by guiding the generation according to the defined input data to generate convincing branching structures. User-defined sketches provide the intriguing advantage that plants can be modeled with direct control to meet artistic requirements [34], [35], [36], [37]. Longay et al., [6] propose an advanced framework to efficiently produce complex tree models that can even run on mobile hardware. Data-driven approaches, on the other hand, focus on solving a reconstruction task to generate plant models to faithfully match a captured tree that can either be represented as one or several images [38], [39], [40], [41], videos [42], or point clouds [3], [43]. More recently, it has also been recognized that neural networks are a powerful means to guide procedural modeling by learning bounding volumes to aid the reconstruction from single images [2] by decomposing point clouds [7] into semantically meaningful patterns, or by learning parameters for the placement of plants [44]. Moreover, synthesizing large data collections from a few examples appears to be an increasingly important pursuit [45]. Finally, some approaches also focus on defining trees based on partially defined branching structures – procedurally generated – to enable level of detail schemes [46] or the processing of large collections of plants [47].

The work of Estrada et al., [48] follows a similar objective to ours. Their approach also uses a parametric tree-growth model to regularize topology estimation for tree-like structures. However, despite these advances, none of the existing approaches for trees and plants uses neural networks to encode the rules and parameters for locally encoding branching patterns for botanical tree models.

3 OVERVIEW

Our goal is to learn all possible combinations of branching patterns as they occur from the perspective of a single node in a branch graph. Instead of defining these patterns manually based on parameters and rules, we employ a set of neural networks that learn to predict the child nodes based on the developmental (growth stage, trunk vs. twig, etc.) and environmental context (obstacles, light, gravity, etc.) of a parent node in the branch graph. We train the networks on *branchlets*, a tree node, and its immediate children (Fig.2, a). A node is defined by all its attributes that we refer to as *node signature*. The node signature serves as input to our network pipeline, and the output is the signatures of up to three child nodes that are then used for the generation of the next nodes.

The *DeepTree* neural network pipeline consists of a classification network that predicts the topology of a *branchlet*, i.e., how many children a node needs to have. We train individual regression networks for each child node that predict the attribute values of the child node's signature, i.e., their thickness, length, and growth direction. The input of the first regression network is the signature of the active parent node, and it predicts the signature of the first child. We then query the second regression network to predict the signature of the second child. This network receives the signature of the active parent node and the predicted signature of the first child from the previous step as input. This process is continued for the number of children the classifier predicted.

The key idea behind our method is that the networks learn to predict meaningful attributes of child nodes only based on where the parent node – along with all its signature attributes – is located within the tree model. Training the network with branchlets enables the network to learn a latent space that encodes which topological and geometric configurations are reasonable for certain locations in the tree's growth space. As an example, for the illustrated tree graph (Fig. 2, a), the networks first need to predict the trunk – each node only has one child – for the first few iterations. The further the generation of a tree model progresses, the
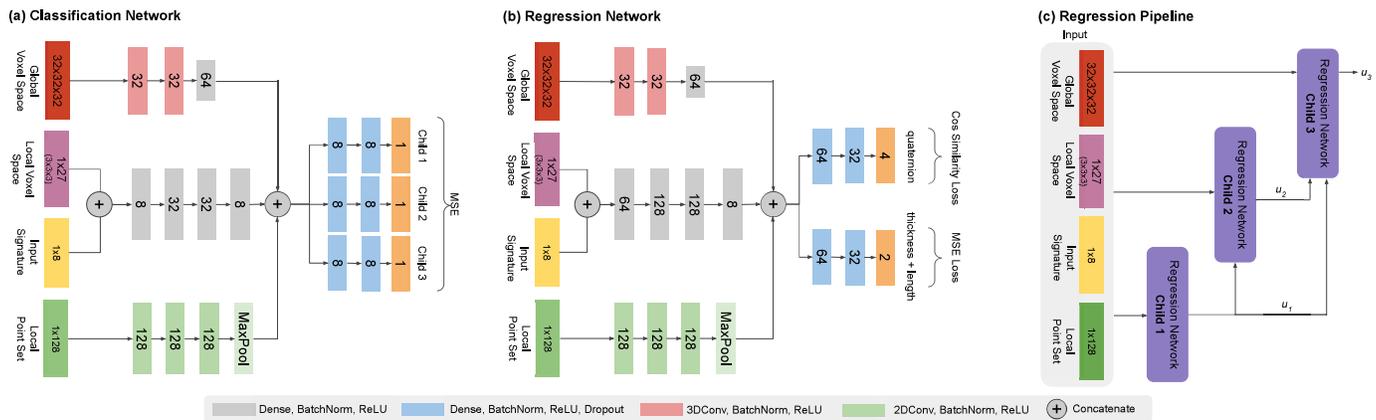
Fig. 3. Network architecture: our neural network pipeline consists of a classification network to predict 0-3 children for a given node (a). Depending on the number of children, we use a cascade of regression networks (c) to predict output signatures for each child. Each regression network is defined as an MLP (b). For each network, we use the current node's signature, the local voxel space for the node, and the global voxel space as inputs. We add a point encoder network to obtain a feature from local point sets and to enable predicting branchlets for reconstruction tasks.

more the networks need to predict more diverse and finer branching patterns up to the point where only small twigs are generated (Fig. 2, b-d). Eventually, the network also has to predict the termination of branches, which stops the generation of a tree model (Fig. 2, d).

The objective of our pipeline is not to reconstruct a given tree model. Instead, our goal is to propose an alternative method for defining the rules and parameters used for generating branch graphs with procedural models. A procedural model generates a distribution of tree models based on value ranges defined for the various parameters. A parameter value is obtained at each growth step by sampling from the range. For complex procedural models, the ranges of parameter values can also vary over time. For example, to capture the impact of gravitropism, branching angles may vary from a more narrow range for the trunk to a wider one for the thinner branches in the tree crown. Our method aims to learn these distributions of branching patterns and how they vary across a tree model. We use neural networks to imitate the developmental process locally.

4 DEEP MODELING OF BRANCHING STRUCTURES

Here we introduce our framework, including a formal definition of the used representation of trees, node signatures, the used neural networks, and the implementation of the environmental response of tree models.

4.1 Tree Representation

We represent tree models as a directed acyclic graph $G = \{N, E\}$, where N are the nodes and E the edges. Given an oriented edge $e_i = (n_s, n_t)$ with a starting node n_s and the end node n_t, a hierarchical relationship is generated. The node with no predecessor is the root node n_{root}. Branches are defined as chains of a varying number of edges $C = \{e_1, e_2, \ldots, e_m\}$; the length of a chain is denoted as m. We assign to each node the Hack [49] ordering number o (also called Gravellius ordering) that assigns $o = 0$ to the trunk and one order higher for each branch recursively (hierarchical ordering). Formally, a node is defined as a tuple of attributes

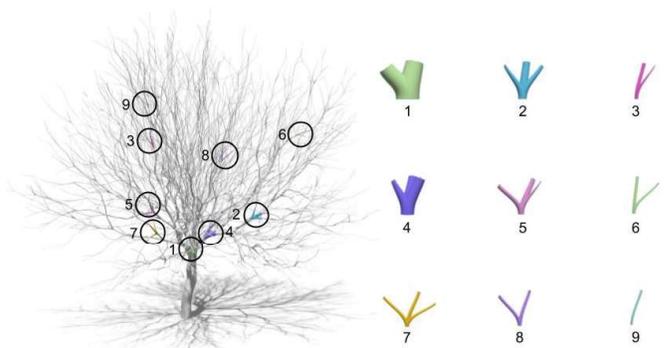$$n = (p, t, d_r, d_s, q_p, l_p, o, m_p, a, v, s), \qquad (1)$$

Fig. 4. Illustration of our tree representation: a branchlet is a node with its immediate successors (0-3 children). Branchlets can have different geometries, depending on where they are located within a tree model.

where p represents the 3D position of a node in Euclidean space, t is the branch thickness, d_r is the distance of a node to the root node in the node hierarchy, d_s is the distance to the start of the branch (chain), q_p is the normalized quaternion representing the orientation of the node w.r.t. the parent node, l_p is the length of the edge to the parent node, m_p is the number of children of the parent node. Additionally, we add the global attributes: age a, gravitropism v, and species s, where a and s are integers. We refer to the tuple n as the *input signature* of a node, which is used as the input of our neural networks.

A branchlet is defined as a set of edges of a node and its children $B = \{(n_i, n_{i+1}), \ldots, (n_i, n_{i+c})\}$. The number of children of a node is denoted $c = 1, \ldots, 3$. A branchlet can also have no children $B = \{(n_i, \emptyset)\}$, which is required for terminal nodes. In Fig. 4, we show renderings of branchlets with different topological and geometric properties and where they are located within a tree model.

4.2 Neural Network Pipeline

Most procedural modeling algorithms aim to produce a graph G to define the properties of a given tree species, which is accomplished by defining the parameters and rules for generating node topology and geometry. For example, one may define the angle at which a branch grows away

from its parent branch as a range of values that are then sampled randomly when the procedural model is evaluated to generate a tree model.

Learning Topology: Contrary to common procedural approaches, our goal is to learn the generation of *branchlets* – the branching pattern for a single node $n \in N$ of a branch graph G. To successfully generate local branching patterns, we need to simultaneously predict the number of children (topology) and their geometric attributes (e.g., position, thickness, and branching angle). Jointly learning the topological and geometrical properties of a branch graph is challenging as most deep network architectures do not support generating outputs of arbitrary lengths, and the regression of high-dimensional outputs is difficult. Therefore, we learn the number of children and their geometric features separately. To predict the topology of a branchlet, we train a classification network to predict the number of children a given parent node should have as probabilities. More specifically, we aim to learn the mapping:

$$f_{class}(n) : N \to \mathcal{X}, \tag{2}$$

where $x \in \mathcal{X}$ denotes a tuple of probabilities $x = (p_1, p_2, p_3)$ for each child node of $n \in N$. The probabilities determine whether or not to grow a child node. The goal of this network is only to predict the topology of a branchlet.

Learning Geometry: To learn the geometric properties of a child node, we define the tuple $u = (q, t, l)$, where q denotes a normalized quaternion representing the direction of a child node w.r.t. the parent node, t is the branch thickness of a node, and l the length of the edge between the parent node and the child node. We refer to the tuple u as the *output signature* of a node and use it as the label for training the regression networks.

To generate the children of a parent node, we need to predict the geometric properties of up to three tuples u_1, u_2, u_3 for each potential child node. We use a cascade of up to three regression networks to predict these tuples. The first network receives only the *input signature* of a parent node as the input and is defined as:

$$f_{reg}^1(n) : N \to \mathcal{U}_1, \tag{3}$$

where $u_1 \in \mathcal{U}_1$ is the output tuple of the first child for a given node n. For the subsequent children, the second and third network also receive the predicted output signatures of the previous steps. Consequently, the networks for the second and third child can be defined as:

$$f_{reg}^2(n, u_1) \quad : N \to \mathcal{U}_2, \tag{4}$$

$$f_{reg}^3(n, u_1, u_2) \quad : N \to \mathcal{U}_3, \tag{5}$$

where $u_2 \in \mathcal{U}_2$ denotes the output tuple of the second child, and $u_3 \in \mathcal{U}_3$ is the output tuple of the third child for node n respectively. Please note that while we did not experiment with predicting more than three child nodes for each parent node, the architecture can be extended as outlined here.

4.3 Iterative DPM Tree Generation

We iteratively apply our network for the nodes of the branch graph of a tree model to generate the 3D tree structure. Given the signature of a node n, the classification network predicts the number of children that need to be generated

---

ALGORITHM 1: Geometry Generation.

Input: Root node n_{root}
Output: Tree graph G.
1 **Procedure:**
2   Add n_{root} to G.
3   Add n_{root} to list of active nodes L.
4 **while** $|L| > 0$ **do**
5   — $n_a \leftarrow$ Pop node from L.
6   — $x \leftarrow$ Predict probability of each child $f_{class}(n_a)$.
7   — $u_1 \leftarrow$ Predict output signature of child 1
8   —   with $f_{reg}^1(n_a)$.
9   — $u_2 \leftarrow$ Predict output signature of child 2
10   —   with $f_{reg}^2(n_a, u_1)$.
11   — $u_3 \leftarrow$ Predict output signature of child 3
12   —   with $f_{reg}^3(n_a, u_1, u_2)$.
13   — **for** $p_i \in x$ **do**:
14   — $\alpha \leftarrow$ Generate a random number in the range $[0, 1]$.
15   — **if** $\alpha > p_i$:
16   — Make node n_i with u_i and add to G.
17   — Push n_i to L.
18   — **end**
19   — **end**
20 **end**

---

for this node. We then use the cascade of regression networks to predict up to three child nodes. The output of each of the regression networks is an *output signature* containing the quaternion q, the thickness t, and the length l, for each child node of n. Given the position p of a node, we can compute the position of each child node as

$$p_{child} = p + l \cdot d_{front}. \tag{6}$$

The growth direction d_{front} and the up direction d_{up} are:

$$
\begin{aligned}
d_{front} &= \boldsymbol{R} \times d_{init\_front}, \\
d_{up} &= \boldsymbol{R} \times d_{init\_up},
\end{aligned}
\tag{7}
$$

where $\boldsymbol{R}$ is the rotation matrix computed from the quaternion q. The vectors of the initial directions d_{init_front} and d_{init_up} are $[0, 0, -1]$ and $[0, 1, 0]$ respectively. The vectors d_{front} and d_{up} are used to generate the branch mesh. The thickness is stored with the child node and later also used to generate the branch mesh.

To generate a new tree model, a user defines the signature of the node n_{root}. The root node is added to a list of active nodes L. As long as the list holds nodes, an active node n_a is popped from the list. The classification network then predicts which of the three children should be generated; the result is returned as the tuple $x = (p_1, p_2, p_3)$. If a child node needs to be generated for n_a, the cascade of regression networks predicts the tuple u_i for each child. The child nodes are added to the branch graph, and the list of active nodes and the network pipeline can be queried for each node in the subsequent iterations. The generation of a branch terminates when the classification network predicts zero children for a node. Pseudocode for our algorithm is shown in Alg. 1 and an illustration of the iterative construction of a tree model is shown in Fig. 2.

Fig. 5. Eight different tree species modeled with the procedural model of Stava et al. [5] (left) and with our *DeepTree* approach (right). The learned situated latent spaces allow us to generate the topological and geometric details to model a wide variety of different tree species.

4.4 Environmental Response

We use a voxel space to learn how trees grow around obstacles in their environment [11]. We represent obstacles as bounding boxes and generate an occupancy grid Y that encodes where obstacles are located in the environment. We generate the occupancy grid by testing the bounding boxes against the voxels. We use the computed occupancy grids in two different ways. First, we obtain the 3^3 neighborhood of cells for each node as 27 binary occupancy values denoted as Z. We then add them to our node signature $n = n \cup \{Z\}$. As shown in Fig. 3 (a-b), we can use these values as additional input for our classification and regression networks (magenta boxes). Second, to obtain a global feature of the environment, we use an additional encoder consisting of three 3D convolutional layers. The global voxel space is a one-hot 3D vector where obstacle voxels are marked as one and empty voxels as zero; it has a size of 32^3 voxels. The tree's root is set in the middle of the voxel space. We extract the global feature through the additional encoder and added it to the local feature before the dense layers for the output heads of the networks. Adding the global feature of the environment allows us to generate environmentally sensitive tree models that can be placed arbitrarily into the environment. In contrast to more complex models of environmental sensitivity [1], [14], [21], this approach provides a lightweight way to generate branching structures for tree models placed in the vicinity of obstacles.

4.5 Point-based Modeling
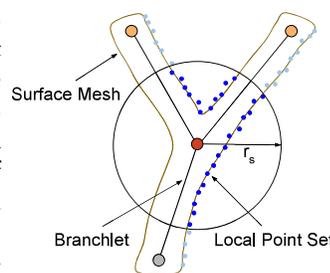
To provide additional control for the reconstruction and sketch-based generation of tree models, we have equipped our neural network architecture with an additional point-cloud encoder network (Fig. 3 a-b). We locally sample points from a point cloud of a procedurally generated tree model to train our architecture for tree reconstruction with point clouds. For each node of a branchlet, we define a sample radius r_s to compute a sphere around a node (inset figure). Given a point cloud of a tree, we select all points within the sphere; the obtained local point set is denoted as P. We then associate the local point set to our input signature $n = n \cup \{P\}$ and use it as input to our architecture to obtain a point feature. The point feature is then added to the signature embedding for the classification and regression networks. This training setup aims to condition the prediction of child node attributes based on the captured local point sets. The implementation of our point encoder follows PointNet [50]. For most of our experiments, a radius of $r_s = 2.0$ provides the best results for reconstructing the tree models.

Fig. 6. Modeling age and gravitropism: by adding priors for age and gravitropism to our node tuple n, our neural network pipeline is able to model realistic branching structures. Here we compare a procedurally generated Oak tree at different developmental stages (a-d) and the corresponding *DeepTree* generated ones (e-h). Our method can also capture the differences in branching patterns when considering different environmental factors, such as gravitropism. The procedurally generated tree models (i-l) show the same structural properties as those generated with *DeepTree* (m-p).

5 IMPLEMENTATION

Our framework is implemented as two components: the first is an interactive framework to efficiently generate large collections of tree models with the procedural model from [5]. Second, we use Pytorch for developing and training our classification and regression neural networks. Below we describe how we used these two components of our framework to generate training data and how we trained our neural network pipeline.

The deep learning model was trained on a single Nvidia RTX A5000, and it took 6 hours for the regressor and 4 hours for the classifier. Rendering was performed on a desktop computer equipped with an Intel(R) Core(TM) i9-9900K and Nvidia 3090 RTX.

5.1 Dataset Preparation

We used the 29-dimensional parameter space of the procedural model from Šťava et al. [5] to model eight distinct tree species: Beech, Corkscrew, Maple, Oak, Pine, Tulip, Walnut, and Willow (Fig. 5, left). An explanation of the parameters of the procedural model, along with the used parameter values, can be found in the Appx. Tab. 4 and Tab. 5. We then generate 500 unique tree models for each species and normalize their positions (attribute p) into a unit cube. We split these tree models into 400 for training and 100 for validation. The validation data is used for identifying optimal hyperparameters for our neural network architectures (e.g., learning rate, number layers, etc.). The procedural model and our *DeepTree* algorithm only generate branch graphs.

We generate a mesh out of generalized cylinders based on the branch graph and the stored thickness values to render images of trees with high visual quality. The voxel space that we use for representing the environment has 32^3 voxels. We compute the bounding box of each obstacle (e.g., walls) and check whether they occupy the corresponding voxels. This results in a 3D occupancy grid.

To train the point cloud network, we generate view-dependent (partial) point clouds for our procedurally generated tree meshes. To generate the point scans, we mimic a LiDAR scanner that we place on a 10m×10m ground plane outside the bounding box of a tree that is located at the origin of the plane. We then cast rays toward the tree by sampling points on a sphere located at the scanner position. This procedure generates partial point scans of our tree models with 8k-35k points. We then sample this point cloud for each branchlet to obtain a local point cloud with a maximum of 512 points as described in Sec. 4.5.

5.2 Training

We train the classification and regression neural networks of our pipeline based on *branchlets*. We generate the branchlets by decomposing the generated branch graph G. Any tree graph node (with 0-3 children) can be a branchlet. We traverse each graph in our training dataset and select each node along with its immediate children to generate a set of branchlets $\mathcal{B}$. A parent node of a branchlet with all its attributes becomes the *input signature*. We then compute the quaternion q and the length l based on how child nodes
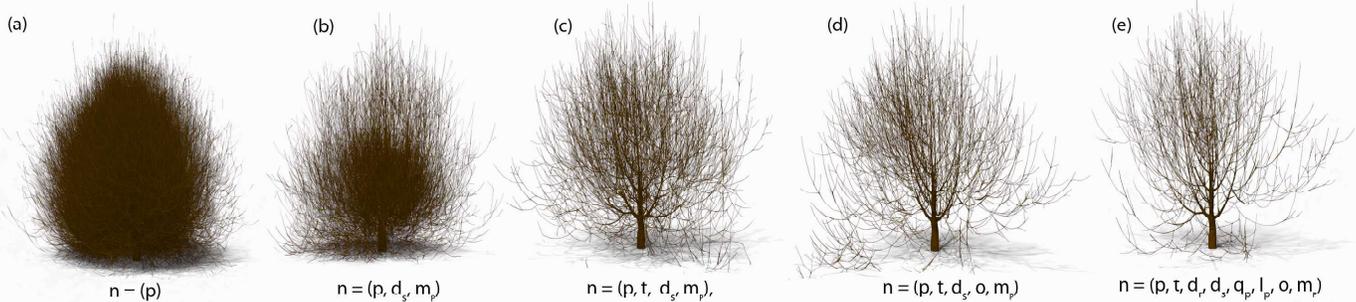
Fig. 7. Ablation study of the classification network: here, we show the impact of different attribute configurations of our node signature as an ablation study. Training only with the attribute position (p) generates a dense branching structure, and the network fails to terminate (a). Even if additional attributes such as the root distance (d_s) and the number of children of the parent node (m_p) are added, the model still fails to terminate (b). c) and d): only after adding additional attributes, such as the thickness (t) or branch order (o), the model generates more convincing branching structures, although some branches are still generated into the ground. Please note that we only use the per-node attributes for this experiment and not the set of global attributes such as species (s), age (a), and gravitropism (v). Then we used Beech tree species for this experiment (Fig.5).
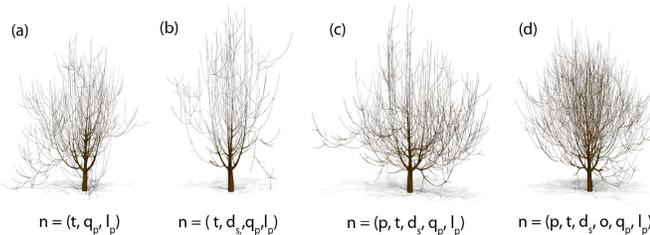

Fig. 8. Ablation study of regression networks: the selection of node attributes for training the regression networks profoundly impacts the generated branch graphs. Only using a few attributes leads to slim and sparse branching structures (a, b). Adding attributes to the input signature, such as the position or the order (c, d), leads to more realistic branching patterns. Here, we used a classification network with all parameters. The model with all node attributes is shown in Fig.7 (e).

are connected to the parent node and store these attributes along with the thickness value t as the *output signature* that is used as a label for training the regression networks. For training the classification network, we simply obtain the number of children attached to the parent node as the label.

Classification Network Training: we train our classification network for 150 epochs jointly on the 3,200 tree models of all species. As illustrated in Fig. 3, our classification network has three output heads. The last layer uses a sigmoid activation function. For each head, we aim to simultaneously predict the probability of whether a particular child needs to be generated (0=child is omitted, 1=child is added). We train for this objective with an MSE loss, where the label for adding a child is 1 and 0 to omit a child node. Depending on the species the network is trained on, we obtain a child classification accuracy of 83 - 93%.

To train this network more robustly, we use data augmentation. We add random noise to the position with a mean of zero and std of 0.0001. Additionally, we ensure a balanced sampling of nodes of the same properties. Considering that nodes with a different number of children appear with different frequencies (e.g., most nodes may have one child), we balance the number of nodes of each type during training to avoid network overfitting. For example, nodes at the trunk are rarely present in our dataset compared to the many twig nodes in the tree crown. Balancing the nodes helps to avoid overfitting. Training the classification network takes about four hours.

Regression Network Training: is performed individually for 200-300 epochs on the 3,200 tree models of all species. We trained the regression networks for up to 12 hours. Each regression network has two output heads: the first generates outputs for thickness and length. We train this head with an MSE loss. The second head predicts a quaternion and is trained with a cosine similarity loss. To avoid overfitting, we also add random noise to the position for training the regression networks with a mean of zero and std of 1e-4. Depending on the species the network is trained on, we obtain an MSE of 1.2e-4 - 3e-4 (branch length and thickness) and a cosine similarity of 0.04 - 0.20 (quaternion).

The networks are trained with the Adam optimizer, a learning rate of 1e-4, and a batch of 512. An illustration of our network architectures – also including the used number of units in each layer – is shown in Fig. 3. Please note that each of the used regression networks in the cascade of networks (Fig. 3, c) is trained individually.

5.3 Rendering

The generated branch graph from the procedural model and *DeepTree* are used to compute a surface mesh of branches from generalized cylinders. Leaves are represented as textured quads placed along with the outermost branches with additional procedural parameters for their placement (e.g., phyllotaxis) [51]. All models were rendered with a path tracer written with the Nvidia OptiX 7.4 API [52] tracing 512 rays per pixel.

6 RESULTS, EVALUATION, AND APPLICATIONS

Here, we discuss experiments on how our framework can be used to generate tree models of different species, ages, or tropisms. We also discuss applications for our method and show how we validate our framework based on geometric and perceptual metrics.

6.1 Results

Fig. 5 shows a qualitative comparison of eight procedurally generated tree species and the same eight species generated with *DeepTree*. Our method can capture the wide variety of distinct features across the eight tree species, including the fine nuances in shape and geometric detail. Generating these
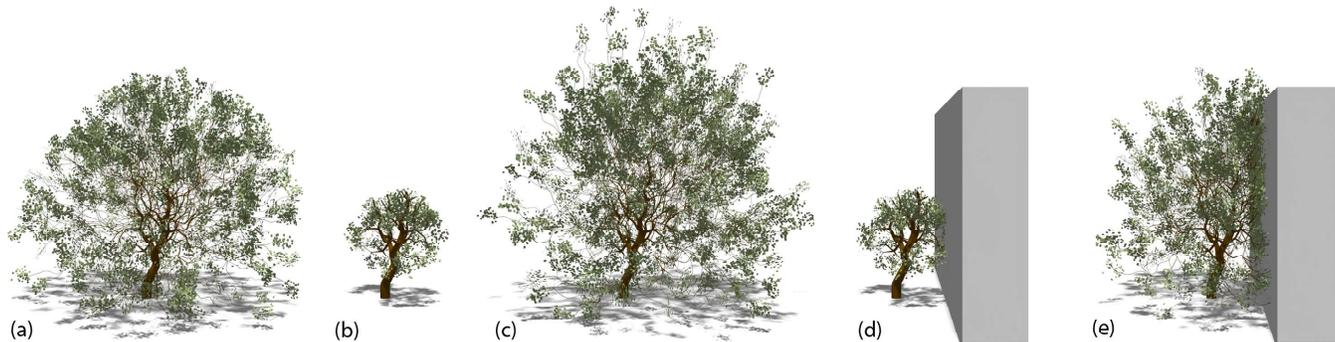
Fig. 9. Regrowth: our method can be used to regrow tree models to thereby adapt them to changing environmental conditions. A fully developed tree model (a) generated with our method is pruned and can be developed anew (b) to a fully grown tree model (c) by iteratively applying our *DeepTree* pipeline to the outermost nodes in the branch graph. If the tree model is placed in the vicinity of an obstacle (d) and then regrown (e), our method supports generating environmentally sensitive tree models.
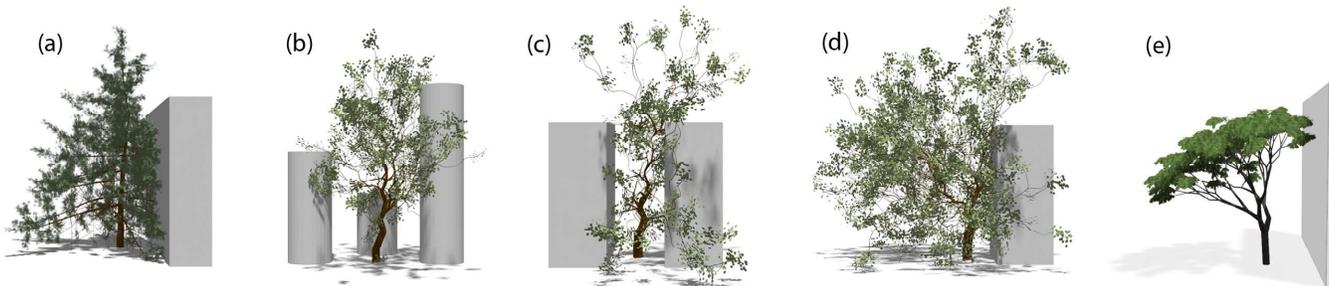

Fig. 10. Modeling with environmental sensitivity: when provided with the environment as occupancy grids, our situated latent space is able to generate convincing branching structures to mimic the growth response of trees to obstacles in their vicinity (a-d). Similar to existing approaches, such as Pirk et al. [21] (e), our method is also able to generate environmentally-aware branching structures.
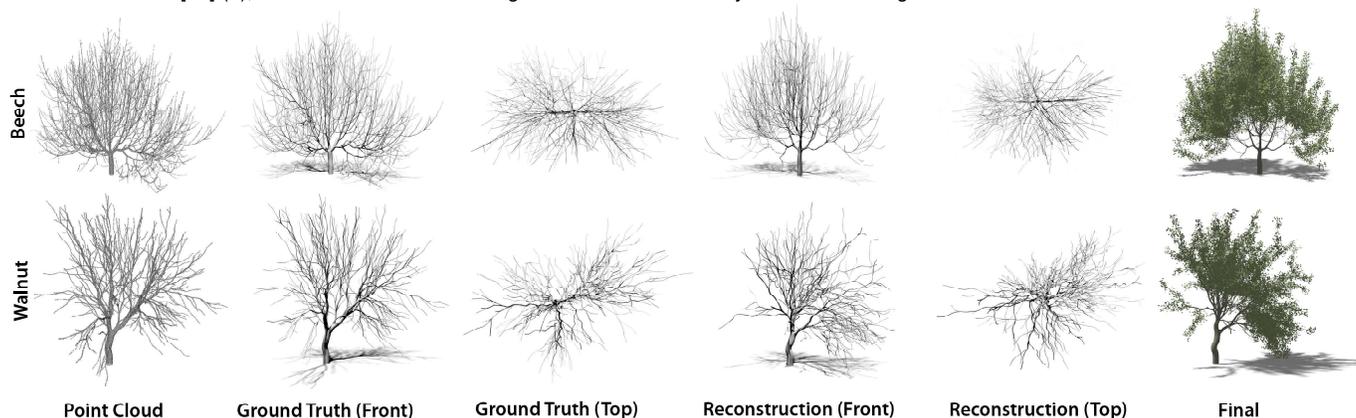

Fig. 11. Reconstruction of tree models from point cloud data: we sample our procedural trees to generate point clouds. We then train our neural networks with local point clouds and use them to reconstruct unseen tree models from the validation dataset.

species with a procedural model is a significant modeling effort that requires manually specifying the procedural model and carefully fine-tuning all the parameters for each species. In contrast, our method learns the branching patterns of all species from data by training our network pipeline.

The results in Fig. 6 show that our approach captures the growth response of tree models and the impact of gravitropism. By adding global priors for age a and gravitropism v to the node tuple n, our pipeline can faithfully generate the structural details for modeling different age stages (e-h) and varying degrees of gravitropism (m-p). Compared to the procedurally generated branching structures (a-h) and (j-l), our method can generate branching patterns with almost identical topological and geometric features.

We conducted ablation studies to analyze the impact of different attributes in our node tuple n during training of the classification and regression networks. Fig. 7 demonstrates the impact of using more or fewer parameters during the training of the classifier. For the result shown in Fig. 7 (a), we only used the position p attribute for our classification network. Training the network only on this attribute prevents the network from terminating, which results in very dense and unrealistic branching patterns. Adding the distance to the branch root d_s and the number of child nodes of the parent node m_p by itself also does not enable training of more realistic branching structures (Fig. 7 b) as the model still fails to predict the correct number of children. However, by adding the branch thickness t, the network starts to successfully predict the termination of branches, which leads to more realistic, still dense branching structures (Fig. 7 c). When also using the other attributes, the model starts predicting branchlet topologies and geometries that lead to realistic branching patterns (Fig. 7 d, e).
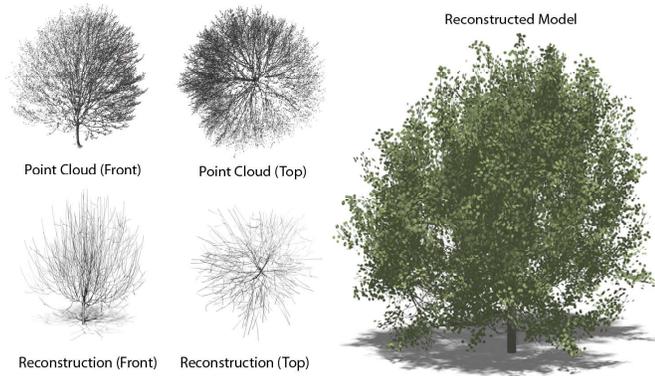
Fig. 12. Reconstruction of a real tree from captured point cloud data: *DeepTree* was trained on synthetic data and generated plausible tree models from real point clouds.
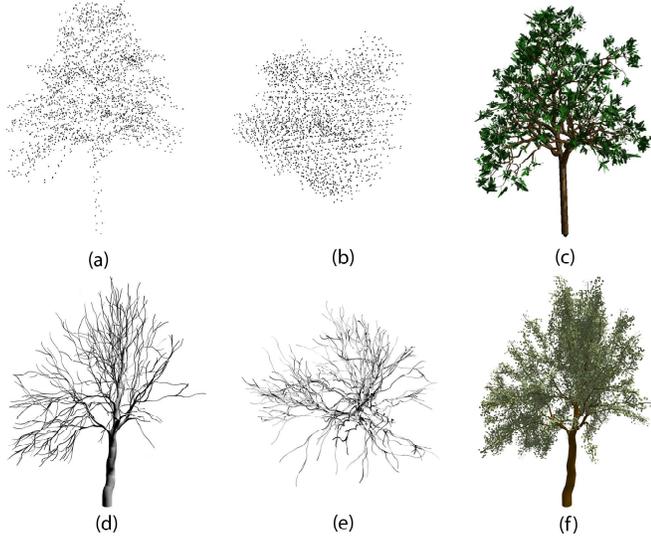

Fig. 13. Comparison to AdTree [4, Fig 11 e]: we use an existing point cloud from their paper (a-b) and show their reconstruction (c). *DeepTree* generates a similar tree model (d-f).

Please note that for (a) and (b), we manually stopped the iterative generation of the models as the neural networks would not automatically terminate. In Fig. 8, we show a similar ablation study for the regression networks. For this experiment, we used a classification network with all node attributes (Fig 7, e). Only using a few attributes for training the regression networks leads to slim and sparse branching structures (a, b). Adding more attributes generates more realistic branching patterns (c, d).

The result in Fig. 9 shows the usefulness of our method for content creation. After we generate a fully developed tree model with our method (a), we can prune and regenerate the branch graph (b) to another fully developed tree model (c). If the model is placed next to an obstacle and then regenerated, it adapts to its new environment (d) and develops an adapted and realistic branch graph (e). Generating a new tree model by pruning a branch graph can be repeated indefinitely. However, please note that each generated model is unique – our method cannot regenerate precisely the same model.

Fig. 10 shows that the situated latent spaces encode the environmental response of a tree model. By adding the local and global occupancy features, the neural network pipeline can be trained to model the growth response to obstacles.
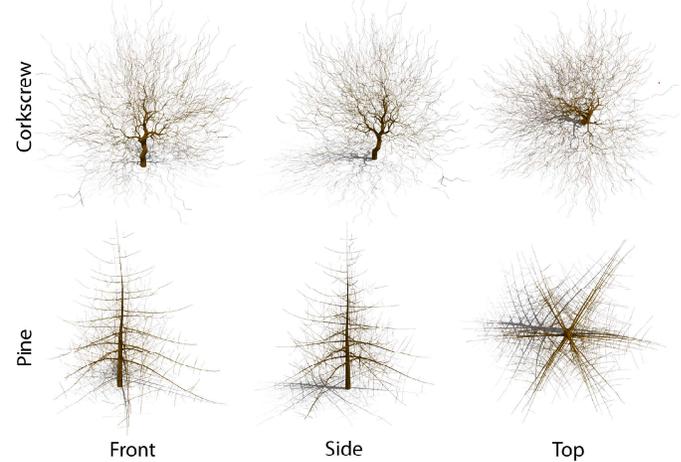

Fig. 14. Different viewpoints of tree models show that *DeepTree* generates tree models without dominant artifacts (e.g., half-reconstructed model) and captures tree phyllotaxis.

For this experiment, we also generated a dataset of 500 scenes (400 training, 100 validation) where obstacles were randomly placed in the vicinity of the procedurally generated trees. The procedural model accounts for the obstacles by computing the availability of light in its environment – branches then grow into regions where more light is available. By training with tree models and the occupancy grid from the scene, the neural network pipeline is able to mimic the modeling of the environmental response. Fig. 10 (a)-(d) compares our results to the method of Pirk et al. [21] (e), where environmental sensitivity is modeled based on an inverse algorithm. Our *DeepTree* learning-based approach can generate a similar response of adapted branching structures for trees grown in the vicinity of obstacles.

6.2 Applications

Fig. 11 shows a tree reconstruction result. We use our training data set to generate point clouds of procedurally generated tree models. We then train our neural network pipeline on local point clouds to provide the network with a point feature (Sec. 4.5). Training with the point feature allows us to generate branching structures that follow the scanned points, which also works for point clouds of real trees, as shown in Figs. 12 and 13.

We show multiple views of the same tree models in Fig. 14 to demonstrate that our method generates plausible tree models from all view directions. Once trained, our framework enables the generation of tree models of a specific tree species. Moreover, *DeepTree* generalizes the branchlet and not the entire tree. Thus, each generated model is different. This makes our approach suitable for generating large datasets of new models, as shown in the selection of randomly sampled tree models in Fig. 17. Fig. 15 shows that our method can replicate the growth of more constrained tree shapes. For this result, we generated a dataset of 500 tree models for each of the shown shapes, including cone (a), cube (b), and ring (c).

Finally, Tab. 3 (Appx.) shows that our method can also be used for compressing generated tree models. We compare the memory footprint of a tree model generated by our
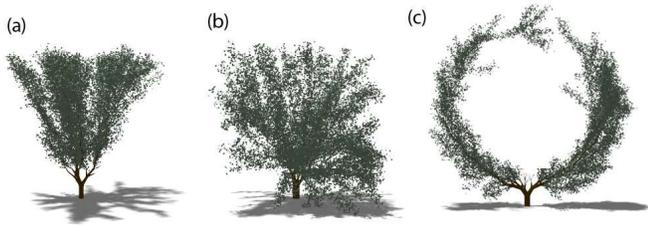
Fig. 15. Our method can replicate the growth process of form-controlled tree models. For this result, we trained on a dataset of tree models that were grown into meshes of cones (a), cubes (b), and rings (c).
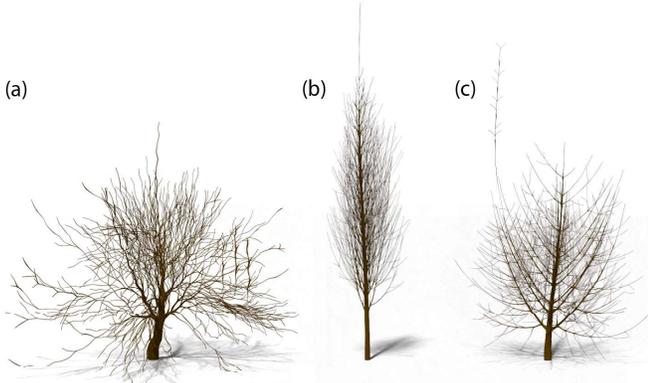

Fig. 16. Three failure cases: if the used neural networks are not properly trained our method tends to generate branching structures with too thin and too long branches. Here we show examples of Oak (a), Tulip (b), and Maple (c) trees.

method to common representations for tree models, such as skeletal graphs and surface meshes. *DeepTree* can deterministically re-generate the same tree model by using the same random seed and replacing the stochastic probability with a given number. As we only need to store the root node and the weights for our neural networks, our method allows us to compress tree models with a lightweight memory footprint that is even smaller than most skeletal graphs.

6.3 Evaluation

We are not aware of any deep neural generative model for 3D tree geometry. Therefore we compare our algorithm to the state-of-the-art procedural model of [2], [5]. We used 500 procedurally generated tree models (P) for each species, and we generated another 500 trees using *DeepTree* (DT). We then compare their geometry and perceived level of realism.

Geometry: We validate the geometric structure of the trees generated by our *DeepTree* approach by comparing their geometric properties to the ground truth, and the results are shown in Fig. 18 and Tab. 5 (Appx.). The overall branch length varies by 11% for all trees, the number of generated branches by 18%, branching distance by 12%, the number of generated nodes by 11%, and the angles (in order from the trunk) by 6%, 4%, and 6%. The variations are minor, and it is essential to note that there is a great difference in visual importance for the presented features, e.g., the angle of the branches coming from the trunk has a strong effect on the overall tree shape, the branching angle of the small branchlets is not so important. This observation is further supported by perceptual metrics.

TABLE 1
ICTree perceptual validation of the ground truth and generated trees (mean and std for the entire dataset).

Species	Ground Truth	*DeepTree*
Beech	[0.47, 0.10]	[0.49, 0.09]
Corkscrew	[0.48, 0.11]	[0.48, 0.12]
Maple	[0.49, 0.11]	[0.46, 0.12]
Oak	[0.50, 0.11]	[0.48, 0.12]
Pine	[0.50, 0.11]	[0.48, 0.11]
Tulip	[0.49, 0.11]	[0.45, 0.10]
Walnut	[0.49, 0.11]	[0.47, 0.10]
Willow	[0.49, 0.11]	[0.46, 0.10]
All trees	**[0.49, 0.11]**	**[0.47, 0.11]**

TABLE 2
Runtime performance comparison of procedurally generated trees (P) obtained with the method of [5]

and *DeepTree* generated trees (DT). The runtime is reported in seconds and only includes the time required to generate the branch graph.

Species	Method	Runtime	# Nodes	# Branches
Beech	P	23	3376	777
	DT	2.7	5672	1634
Corkscrew	P	26	2872	751
	DT	2.1	3041	566
Maple	P	25	2928	861
	DT	1.9	2951	817
Oak	P	27	4261	1084
	DT	2.1	6238	1738
Pine	P	26	3506	1000
	DT	1.9	3168	839
Tulip	P	26	3050	726
	DT	1.8	3360	773
Walnut	P	27	5638	589
	DT	2.0	4975	677
Willow	P	30	4796	1304
	DT	2.0	4975	1691

Perceptual Metrics: While the geometric comparison shows that our trees have geometric traits comparable to the ground truth, we also want to answer whether the generated trees are perceived as realistic by humans. While a standard and tedious approach to this validation is to run a user study, we validate the generated trees by recently introduced deep neural perceptual metrics ICTree [53] that estimate the visual realism of the 3D tree models. ICTree has been trained by a response of 4,000 human subjects, and it provides a value between zero to one that corresponds to the perceived visual realism of the input tree. The authors introduced view-independent ICTreeF and image-based ICTreeI metrics. We use the ICTreeF that uses geometric tree properties.

Tab. 1 shows the average and the standard deviation of the perceived realism of the ground truth and the generated trees. The overall perceived realism metrics of all generated trees is 0.47 ± 0.11 which is 4.3% different from the ground truth of 0.49 ± 0.11.

Runtime Performance: Tab. 2 shows the comparison of the runtime performance of procedurally generated tree models and *DeepTree*. The reported numbers represent the average measurements of each method for ten models for each species. we report the measured time that the two algorithms take to generate the branching structure of a tree model. As shown, our method outperforms the procedural algorithm for tree models of the same species and similar complexity. While the procedural model is implemented with a recursive algorithm to construct the branch graph,

our method queries the neural network pipeline of classification and a cascade of regression networks to generate the branch graph iteratively.

Network Comparison: To validate our cascaded neural network architecture (Fig. 3c), we compare it with a simpler architecture that jointly trains the classification and regression networks. The architecture for this experiment is the same as in Fig.3a, except the two additional heads on top of the joined embedding to generate thickness and length as well as the quaternion. We trained both architectures on all species with the same setup for loss functions and data processing. With the cascaded network architecture, we are able to obtain a classification result of 83 - 93%, an MSE of 1.2e-4 - 3e-4, and a cosine similarity of 0.04 - 0.20, whereas, for the joint model, we obtain a classification result of 89% - 90%, an MSE of 0.28 - 0.51, and a cosine similarity of 1.03 - 0.51. While the classification head is trained with similar accuracy, the regression results are inferior. The jointly trained network immediately predicted erroneous branching that caused the termination of the tree generation.

7 DISCUSSION AND LIMITATIONS

Our focus was on exploring the capabilities of neural networks to predict branching structures by only training them on local branching patterns. Training a neural network this way learns a situated latent space - a representation that can encode the necessary topological and geometric information to mimic branch growth. As we have shown, a trained situated latent space can serve as a powerful representation capable of generating a wide range of complex branching patterns and encoding the environmental response of trees to obstacles in their environment.

Using situated latent spaces, our approach is orthogonal to approaches that aim to learn graph structures end-to-end. In contrast to these approaches, our method is driven by the idea of only encoding local distributions, e.g., local variations of branching patterns. *DeepTree* intriguingly shows that locally learned representations can provide an interesting modeling alternative.

Our work is also similar to graph neural networks (GNNs) in that we also focus on predicting the topological and geometric properties of graphs. Research toward GNNs has recently gained a lot of momentum, and many approaches aim to predict graphs and their properties in different ways [54], [55], [56], [57]. GNNs leverage the connectivity of vertices to their neighbors in a graph by simultaneously predicting vertex properties and their connectivity information. Most of these methods focus on making predictions for local properties in graphs to solve classification tasks. Our work aims to estimate graph nodes' topology and geometric properties simultaneously. Regressing multiple geometric attributes remains a challenging problem we addressed by training a cascade of individually trained regression networks.

Our method is currently limited in three ways: first, we rely on using data generated by a procedural model. We decided to showcase our method on procedurally generated tree models as this is the only way to successfully generate a large dataset to train our neural network pipeline. To the best of our knowledge, large collections of reconstructed models from scans or images do not exist. *DeepTree* depends on data produced by biologically motivated models, which need to be tuned to generate training data. At present, the main advantage of the method appears to be the ability to provide a reduced representation of tree species, which allows for compression, improved runtimes, and constrained generation of trees (via point clouds or a sub-tree). The second limitation is that our method is currently unable to model tree-tree or branch-branch interactions. Procedural models can distinguish between the intrinsic plant environment (the plant itself) and the extrinsic (obstacles). However, as we rely on a coarse voxel space to encode the environment, we cannot solve intricate tree-tree or branch-branch collisions during tree generation. Finally, we observed that our method has difficulties correctly predicting the termination of branches for some node configurations, which generates artificial-looking branching structures (Fig. 16). While this leads to unrealistic branching structures for some tree models, we observed that this is highly correlated with the size of the dataset and the training duration. Related to the dataset is the problem of having a training set for each scenario (such as Fig. 15). While our method can replicate the form-controlled models, we treats them as special cases. Also, the density of the reconstructed canopy from point clouds may vary depending on the density of point clouds.

8 CONCLUSIONS AND FUTURE WORK

We have advanced tree modeling in computer graphics by introducing *DeepTree*, a deep-learning-based method for automatic and adaptive tree form generation. Instead of manually defining parameters and rules –commonly done in procedural modeling– *DeepTree* learns branching patterns locally as they can be observed for a single node in a branch graph. We have shown that a situated latent space that is evaluated for a single node in a tree graph, somewhere in the growth space of a tree model, can generate complex tree models with similar topological and geometric features as contemporary procedural and developmental modeling techniques.

We have shown that our method can generate a diverse set of branching structures from different species. Adding additional information to each node's signature (e.g., a species identifier) shows that our method can even be trained in a generalized manner, i.e., we can jointly train our networks on multiple species and successfully generate tree models. Furthermore, by encoding the environment, we have shown that our situated latent space can even support mimicking the environmental response of tree models when they grow next to walls or other trees.

A novel way of encoding tree form opens multiple avenues for future work. First, it would be interesting to explore how a situated latent space can be used to support the authoring of branching structures. In our current framework, we only automatically use our network pipeline. However, we could also control our network pipeline by user-defined sketches or adapt it to dynamically changing scenes. Second, our current architecture does not avoid inter-branch collisions. To this end, it would be interesting to explore additional global or semi-global encodings for the generated branches when querying the networks for
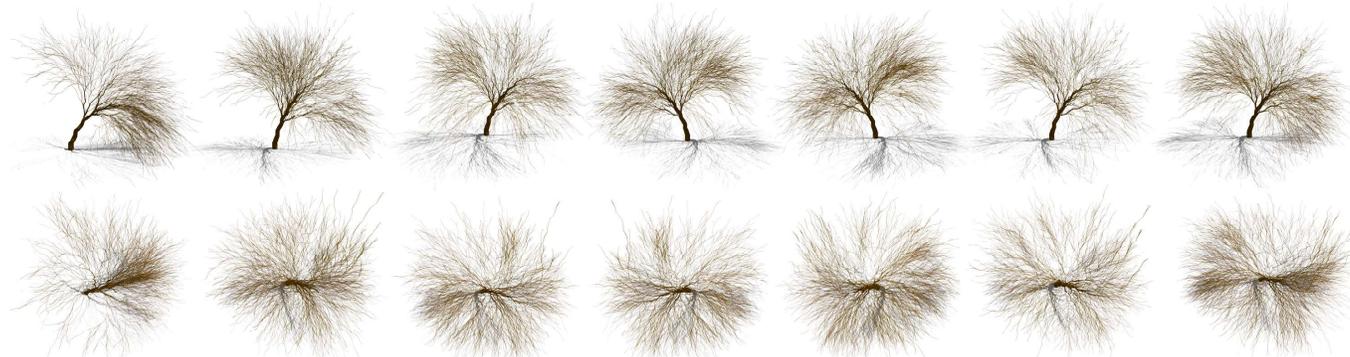
Fig. 17. Random sampling: a set of tree models of the same species generated by our neural network pipeline shown from the front (top row) and top (bottom).

producing new nodes. For example, encoding the neighboring nodes of a node could provide an additional feature for producing branching structures that do not collide. It also seems promising to use our proposed algorithm of iteratively querying a neural network pipeline to generate other objects, e.g., road networks or buildings.

ACKNOWLEDGMENTS

REFERENCES

[1] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz, "Self-organizing tree models for image synthesis," *ACM Trans. Graph.*, vol. 28, no. 3, pp. 58:1–58:10, 2009.

[2] B. Li, J. Kałużny, J. Klein, D. L. Michels, W. Pałubicki, B. Benes, and S. Pirk, "Learning to reconstruct botanical trees from single images," *ACM Transaction on Graphics*, vol. 40, no. 6, pp. 1–15, 2021.

[3] Y. Livny, S. Pirk, Z. Cheng, F. Yan, O. Deussen, D. Cohen-Or, and B. Chen, "Texture-lobes for tree modelling," *ACM TOG*, vol. 30, no. 4, pp. 53:1–53:10, 2011.

[4] S. Du, R. Lindenbergh, H. Ledoux, J. Stoter, and L. Nan, "Adtree: Accurate, detailed, and automatic modelling of laser-scanned trees," *Remote Sensing*, vol. 11, no. 18, p. 2074, 2019.

[5] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes, "Inverse procedural modelling of trees," *Comput. Graph. Forum*, vol. 33, no. 6, pp. 118–131, 2014.

[6] S. Longay, A. Runions, F. Boudon, and P. Prusinkiewicz, "Treesketch: interactive procedural modeling of trees on a tablet," in *Proc. of the Intl. Symp. on SBIM*, 2012, pp. 107–120.

[7] Y. Liu, J. Guo, B. Benes, O. Deussen, X. Zhang, and H. Huang, "Treepartnet: Neural decomposition of point clouds for 3d tree reconstruction," *ACM Transaction on Graphics*, vol. 40, no. 6, pp. 1–16, 2021.

[8] S. Pirk, B. Benes, T. Ijiri, Y. Li, O. Deussen, B. Chen, and R. Měch, "Modeling plant life in computer graphics," in *ACM SIGGRAPH 2016 Courses*, ser. SIGGRAPH '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:180.

[9] M. Aono and T. Kunii, "Botanical tree image generation," *IEEE Comput. Graph. Appl.*, vol. 4(5), pp. 10–34, 1984.

[10] P. E. Oppenheimer, "Real time design and animation of fractal plants and trees," *Proc. of SIGGRAPH*, vol. 20, no. 4, pp. 55–64, 1986.

[11] N. Greene, "Voxel space automata: Modeling with stochastic growth processes in voxel space," *SIGGRAPH Comp. Graph.*, vol. 23, no. 3, pp. 175–184, 1989.

[12] W. T. Reeves and R. Blau, "Approximate and probabilistic algorithms for shading and rendering structured particle systems," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 313–322, Jul. 1985.

[13] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., 1990.

[14] R. Měch and P. Prusinkiewicz, "Visual models of plants interacting with their environment," in *Proc. of SIGGRAPH*. ACM, 1996, pp. 397–410.

[15] J. Weber and J. Penn, "Creation and rendering of realistic trees," in *Proc. of SIGGRAPH '95*, New York, NY, USA, 1995, pp. 119–128.

[16] B. Lintermann and O. Deussen, "Interactive modeling of plants," *IEEE Comput. Graph. Appl.*, vol. 19, no. 1, pp. 56–65, 1999.

[17] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun, "Metropolis procedural modeling," *ACM TOG*, vol. 30, pp. 11:1–11:14, 2011.

[18] J. Guo, H. Jiang, B. Benes, O. Deussen, X. Zhang, D. Lischinski, and H. Huang, "Inverse procedural modeling of branching structures by inferring l-systems," *ACM Transactions on Graphics*, vol. 39, no. 5, pp. 1–13, sep 2020.

[19] L. Xu and D. Mould, "Procedural tree modeling with guiding vectors," *Computer Graphics Forum*, vol. 34, no. 7, pp. 47–56, 2015.

[20] v. Kohek and D. Strnad, "Interactive synthesis of self-organizing tree models on the gpu," *Computing*, vol. 97, no. 2, p. 145–169, feb 2015.

[21] S. Pirk, O. Stava, J. Kratt, M. A. M. Said, B. Neubert, R. Měch, B. Benes, and O. Deussen, "Plastic trees: Interactive self-adapting botanical tree models," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 50:1–50:10, Jul. 2012.

[22] B. Benes and E. U. Millán, "Virtual climbing plants competing for space," in *Proceedings of the Computer Animation*, ser. CA '02. USA: IEEE Computer Society, 2002, p. 33.

[23] S.-K. Wong and K.-C. Chen, "A procedural approach to modelling virtual climbing plants with tendrils," *Comput. Graph. Forum*, 2015.

[24] S. Pirk, T. Niese, O. Deussen, and B. Neubert, "Capturing and animating the morphogenesis of polygonal tree models," *ACM TOG*, vol. 31, no. 6, pp. 169:1–169:10, 2012.

[25] T. Hädrich, B. Benes, O. Deussen, and S. Pirk, "Interactive modeling and authoring of climbing plants," *Comput. Graph. Forum*, vol. 36, no. 2, pp. 49–61, 2017.

[26] S. Pirk, T. Niese, T. Hädrich, B. Benes, and O. Deussen, "Windy trees: Computing stress response for developmental tree models," *ACM TOG*, vol. 33, no. 6, pp. 204:1–204:11, 2014.

[27] R. Habel, A. Kusternig, and M. Wimmer, "Physically guided animation of trees," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 523–532, 2009.

[28] S. Pirk, M. Jarzabek, T. Hädrich, D. L. Michels, and W. Palubicki, "Interactive wood combustion for botanical tree models," *ACM TOG*, vol. 36, no. 6, pp. 197:1–197:12, Nov. 2017.

[29] B. Wang, Y. Zhao, and J. Barbič, "Botanical materials based on biomechanics," *ACM TOG*, vol. 36, no. 4, pp. 135:1–135:13, Jul. 2017.

[30] Y. Zhao and J. Barbič, "Interactive authoring of simulation-ready plants," *ACM TOG*, vol. 32, no. 4, pp. 84:1–84:12, 2013.

[31] B. Neubert, S. Pirk, O. Deussen, and C. Dachsbacher, "Improved model- and view-dependent pruning of large botanical scenes," *Computer Graphics Forum*, vol. 30, no. 6, pp. 1708–1718, 2011.

[32] T. Hädrich, D. T. Banuti, W. Pałubicki, S. Pirk, and D. L. Michels, "Fire in paradise: Mesoscale simulation of wildfires," *ACM Trans. Graph.*, vol. 40, no. 4, Jul. 2021.

[33] W. Pałubicki, M. Makowski, W. Gajda, T. Hädrich, D. L. Michels, and S. Pirk, "Ecoclimates: Climate-response modeling of vegetation," *ACM Trans. Graph.*, vol. 41, no. 4, 2022.

[34] T. Ijiri, S. Owada, and T. Igarashi, "Seamless integration of initial sketching and subsequent detail editing in flower modeling," *Comput. Graph. Forum*, vol. 25, no. 3, pp. 617–624, 2006.

[35] M. Okabe, S. Owada, and T. Igarashi, "Interactive design of botanical trees using freehand sketches and example-based editing," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07, 2007, p. 26–es.

[36] J. Wither, F. Boudon, M.-P. Cani, and C. Godin, "Structure from silhouettes: a new paradigm for fast sketch-based design of trees," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 541–550, 2009.

[37] X. Chen, B. Neubert, Y.-Q. Xu, O. Deussen, and S. B. Kang, "Sketch-based tree modeling using markov random field," *ACM TOG*, vol. 27, no. 5, Dec. 2008.

[38] D. Bradley, D. Nowrouzezahrai, and P. Beardsley, "Image-based reconstruction and synthesis of dense foliage," *ACM TOG*, vol. 32, no. 4, pp. 74:1–74:10, 2013.

[39] P. Tan, G. Zeng, J. Wang, S. B. Kang, and L. Quan, "Image-based tree modeling," *ACM TOG*, vol. 26, no. 3, 2007.

[40] P. Tan, T. Fang, J. Xiao, P. Zhao, and L. Quan, "Single image tree modeling," *ACM TOG*, vol. 27, no. 5, pp. 108:1–108:7, 2008.

[41] B. Neubert, T. Franken, and O. Deussen, "Approximate image-based tree-modeling using particle flows," *ACM TOG*, vol. 26, no. 3, 2007.

[42] C. Li, O. Deussen, Y.-Z. Song, P. Willis, and P. Hall, "Modeling and generating moving trees from video," *ACM TOG*, vol. 30, no. 6, pp. 127:1–127:12, 2011.

[43] H. Xu, N. Gossett, and B. Chen, "Knowledge and heuristic-based modeling of laser-scanned trees," *ACM TOG*, vol. 26, no. 4, pp. Article 19, 13 pages, 2007.

[44] T. Niese, S. Pirk, M. Albrecht, B. Benes, and O. Deussen, "Procedural urban forestry," *ACM Trans. Graph.*, vol. 41, no. 2, mar 2022.

[45] O. Argudo, C. Andújar, and A. Chica, "Image-based tree variations," *Computer Graphics Forum*, vol. 39, no. 1, pp. 174–184, 2020.

[46] X. Zhang, G. Bao, W. Meng, M. Jaeger, H. Li, O. Deussen, and B. Chen, "Tree branch level of detail models for forest navigation," *Computer Graphics Forum*, vol. 36, no. 8, pp. 402–417, 2017.

[47] M. Makowski, T. Hädrich, J. Scheffczyk, D. L. Michels, S. Pirk, and W. Pałubicki, "Synthetic silviculture: Multi-scale modeling of plant ecosystems," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019.

[48] R. Estrada, C. Tomasi, S. C. Schmidler, and S. Farsiu, "Tree topology estimation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 8, pp. 1688–1701, Aug. 2015.

[49] J. T. Hack, *Studies of longitudinal stream profiles in Virginia and Maryland*. US Government Printing Office, 1957, vol. 294.

[50] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *CVPR*, 2017, pp. 77–85.

[51] O. Deussen, A. Dowden-Williams, and B. Lintermann, *Digital Design of Nature: Computer Generated Plants and Organics*, ser. X.media.publishing. Springer Berlin Heidelberg, 2005.

[52] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, jul 2010.

[53] T. Polasek, D. Hrusa, B. Benes, and M. Cadik, "Ictree: Automatic perceptual metrics for tree models," *ACM Transaction on Graphics*, vol. 40, no. 6, p. 1–15, Dec. 2021.

[54] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[55] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[56] J. Xu, J. Chen, S. You, Z. Xiao, Y. Yang, and J. Lu, "Robustness of deep learning models on graphs: A survey," *AI Open*, vol. 2, pp. 69–78, 2021.

[57] H. Shao, T. Kugelstadt, T. Hädrich, W. Pałubicki, J. Bender, S. Pirk, and D. L. Michels, "Accurately solving rod dynamics with graph learning," in *NeurIPS*, 2021.

Xiaochen Zhou is a Ph.D. student in Computer Science at Purdue University. He researches deep learning and the simulation and reconstruction of vegetation.

Bosheng Li is a Ph.D. student in Computer Science at Purdue University. His research mainly focuses on the functional-structural modeling of vegetation.

Bedrich Benes is a professor of Computer Science at Purdue University. His area of research is in procedural and inverse procedural modeling and simulation of natural phenomena, and he has published over 190 research papers in the field. He received his Ph.D. and M.S in Computer Science from Czech Technical University in Prague. Bedrich is a senior member of IEEE and ACM and a Eurographics Fellow.

Songlin Fei is a professor and dean's chair of remote sensing at Purdue University. His research focuses on the ecology and management of invasive species, the understanding of forest responses to climate change, and the modernization of forestry into the digital age. He has published over 130 research papers in these fields. He received his Ph.D. degree in Ecology and a concurrent MS degree in Statistics from Penn State University.

Söeren Pirk is a Senior Research Scientist at Adobe Research. He received his Ph.D. from the University of Konstanz, Germany. His research interests are in Visual Computing and Artificial Intelligence with a focus on time-variant phenomena and multi-modal data streams. He develops efficient geometric representations to enable solving complex tasks like interactively simulating natural phenomena, encoding interactions and behavior, or for 4D computer vision.