Controlling Procedural Modelling Interactively with Guiding Curves

Dave Pagurek van Mossel[*] University of Waterloo
Abhishek Madan[†] University of Waterloo
Tai Meng Liu[‡] University of Waterloo
Paul Bardea[§] University of Waterloo
Andrew McBurney[¶] University of Waterloo

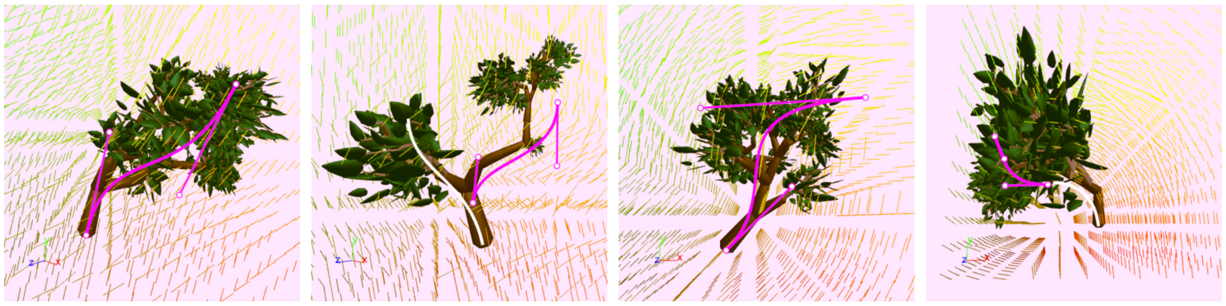


Figure 1: Examples of various tree models produced by the same generating grammar, selected by different guiding curves.

Abstract

Grammar-based procedural modelling on its own produces a larger space of generated models than is artistically desirable. Probabilistic sampling techniques can help search this result space for models that best fit a set of constraints. We aim to provide a useful probabilistic search function that can be run at interactive rates to enable the short feedback loops artists require for incremental, exploratory design. We present a constraint for use in Sequential Monte Carlo optimization where artists draw curves to guide the generation of models. The high-level structure of models can be intuitively specified by our constraint framework, allowing for variation in low-level details to be automatically filled in. We present a real-time model editor to demonstrate the artistic utility of our method.

Keywords: Procedural modelling; interactive design.

Index Terms: Computing methodologies—Computer graphics—Shape modeling; Theory of computation—Randomness, geometry and discrete structures—Computational geometry

1 Introduction

Procedural modelling has been an essential tool in the 3D artist's toolbox for many years, used in applications such as games and movies. It provides a way to add a level of detail and richness that would be impractical to create by hand. It is used, for example, to apply patterns at large scales to create whole landscapes, or on a smaller scale to create varied individual plants and creatures.

Procedural generation from context-free grammars [2] is a convenient way to encode the structure of such patterns. While it can produce great results, the challenge with grammar-based procedural modelling is efficiently sampling the model space to produce a broad range of interesting results while also excluding unfavourable results. One possible solution is to modify the grammar itself to restrict the space of possible outputs; unfortunately, this is an extremely difficult task, since small changes in the grammar can result in large and unpredictable variations in produced models.

Rather than using opaque "bottom-up" constraints like context-free grammars to specify recurrent details in the model, another alternative is to provide loose "top-down" constraints on the model's general shape. These constraints, when formulated as a cost function, reframe the problem as a matter of searching the model space for models with the lowest cost. Existing work uses probabilistic inference and sampling techniques such as Stochastically Ordered Sequential Monte Carlo (SOSMC) to find models that best match such a cost function [13]. The constraints given in prior work intend to give the artist control over the final form of a generated model by specifying target volumes, target silhouettes, volumes to avoid, or other similar controls. SOSMC performs these tasks well, but it can take tens or hundreds of seconds to run, depending on the cost function used, for it to produce good results. It can take several seconds to collect one's thoughts and resume a task after an interruption of this duration [10]; for an artist, this takes enough time that one cannot experiment with tweaking values and observing results incrementally. Instead, the artist is effectively encouraged to spend upfront time imagining and thinking through potential results *before* running SOSMC so that less time is spent waiting. By not assisting in the imagining of results, the potential utility of SOSMC as an artistic tool is restricted.

We believe immense artistic potential can be unlocked by shortening the amount of time it takes to run a search of model space and produce a result. In doing so, an artist can work experimentally and incrementally. The results of one round of search provide insight to the artist, which they may use to make changes iteratively, creating a feedback loop between the tool and the artist. Our work primarily aims to do as much useful work as possible in under a second of computation time to enable such a feedback loop.

In defining useful work, we look at what a useful cost function should be. Creating a target volume that is specific enough to produce interesting results but not so specific to be prohibitively time-consuming to model is a non-trivial problem. We believe that artists get the most value out of procedural generation when they have ideas of what the high-level structure should look like, but want to offload the generation of the finer details to the computer. Therefore, we aim to create constraints for SOSMC optimization that allow the high-level structure to be specified in such a way that

[*]email: dpagurek@uwaterloo.ca
[†]email: a7madan@uwaterloo.ca
[‡]email: taimeng.liu@uwaterloo.ca
[§]email: paul.bardea@uwaterloo.ca
[¶]email: armcburn@uwaterloo.ca

models can be generated at interactive rates.

This paper describes a cost function for SOSMC based on user-specified guiding curves that are followed by generated models of individual, hierarchical objects. In Section 3, we formulate the cost function, describe its parameters, and discuss how they can be used. Using our cost function, models generated from guiding curves such as the ones in Figure 1 are generated in under 200ms. This speed is attained by accounting for general alignment with the curves instead of the complete geometry of the model, which also simplifies the heuristic. Section 4 describes the architecture and design tradeoffs required to achieve this runtime and how we integrated it into a user-facing tool. Finally, in Section 5, we examine models created under this constraint and verify that guiding curves are at least as effective at constraining generated models as a silhouette matching cost function, which takes 45s to produce similar results.

2 Related Work

2.1 Generating Functions

Perhaps the best-known framework for specifying procedurally generated shapes is the L-system [7]. An L-system is a type of formal grammar where the output is a string, which starts from an axiom and is then expanded and modified with rewrite rules. The resulting string can then be read and interpreted as instructions for how to construct a shape. L-system grammars have been used extensively to generate detailed shapes such as plants [12] that have specifiable high-level patterns, but that would be tedious to model in their entirety by hand. Our work uses the versatility of generating grammars as a starting point, upon which we aim to build useful artistic tools.

2.2 Probabilistic Inference

A program that generates procedural models can be seen as picking a sample from a probability distribution, where the distribution is the space of every model the program can generate. Framed in this way, the prior of a model is found from its probability of being produced by a generating grammar, and the likelihood function is determined by a user-provided cost function, where the likelihood function is higher for samples with lower cost [15]. Sampling methods can then be used instead of more traditional optimization methods, which struggle in the high-dimensional spaces of procedural models.

Past work has used generalizations of Markov Chain Monte Carlo (MCMC) to demonstrate the capability of probabilistic inference to successfully optimize the cost of generated models [15]. MCMC receives feedback after generating full models; Sequential Monte Carlo (SMC) sampling improves this by incrementally gathering feedback on model quality as new evidence is collected over time. However, SMC operates over “flat” execution traces; Stochastically Ordered Sequential Monte Carlo (SOSMC) applies it to hierarchically-defined procedural modelling by collecting evidence in a randomized, breadth-first order. It has been proven that SOSMC correctly samples the same posterior distribution regardless of execution order, provided that the likelihood function is also independent of execution order [13]. Our work uses the SOSMC method to search a grammar’s model space.

2.3 Guiding Curves

Methods to draw vectors that guide procedural generation have proven useful in prior work. For the domain-specific case of tree generation, branch alignment with a vector field has been used as a cost function [17]. Our method aims to generalize this flexibility by making vector alignment the core framework through which artists can search the space of generated models, and by providing an intuitive way to specify a vector field.

We believe sketching is a convenient way of achieving this goal. In the past, free-form sketches have been used to specify the curvature parameters of L-systems [5]. Sketching provides a concrete, visual way to constrain the abstract generating functions and get interactive feedback from the output. We aim to bring these benefits to generating grammars in general, not just ones with dominating curvature parameters, so we use a sketch as an indirect way of specifying a vector field.

3 Approach

3.1 Overview

Our method of controlling procedural generation begins with a user-specified grammar defining how to grow models. We then apply SOSMC to search the space of models produced by the grammar for a model that minimizes a cost function. Like in past work using MCMC sampling [15], we search for an optimal derivation tree rather than an optimal string, as strings can often be formed by multiple valid derivations. For a given model x that SOSMC samples, we define the likelihood term $\exp\left(-(\textsc{Cost}(x)+\textsc{HeuristicScale}\cdot\textsc{HeuristicCost}(x))\right)$. The function $\textsc{Cost}(x)$, defined in Section 3.3, compares guiding curves drawn by the user along with user-provided parameters to the *skeleton* of x. The $\textsc{HeuristicScale}$ and $\textsc{HeuristicCost}(x)$ terms are defined in Section 3.4. The skeleton, which we define in Section 3.2, is based only on the spatial hierarchy in the grammar, ignoring concrete geometry. Our method operates under the assumption that geometry is aligned with the skeleton, which allows optimization to only look at the skeleton and still be useful.

3.2 Grammar Specification

Our method operates on a user-specified grammar that generates models. This grammar is a probabilistic context-free grammar [4,14] $G=(N,T,R,S,P)$, where N is the set of non-terminal symbols, T is the set of terminal symbols, R is the set of production rules, S is the starting symbol, and P is the set of probabilities associated with the production rules. When multiple production rules are available for a given symbol, one is randomly selected using the probabilities in P.

In the interest of simplicity and fast generation, we defer expensive geometry processing as much as possible. Many procedural generation techniques initially generate a skeleton for a model and then later grow the skeleton into a full model [3,8]. In similar spirit, we divide our procedural generation process into three phases: the *skeleton* phase, the *wrap-up* phase, and the *geometry* phase. The intention is to introduce a separation between the generation of the underlying structure of a model (what we refer to as its “skeleton”) and the concrete geometry that sits atop it. The distinction is made by partitioning N into disjoint sets N_s, the skeleton symbols, and N_p, the post-skeleton symbols. Rather than explicitly creating skeletal geometry, we instead create an implicit skeleton out of the affine transformations produced by symbols in N_s.

Literature on FL-systems describes the uses of generic objects as symbols [9]. We make use of this to attach data to terminals in addition to their types, such as the entries in a matrix. A terminal symbol in T is one of: [,], an affine matrix A, or a piece of geometry G. Each of these is interpreted as a geometric command when read in order. The rendering system maintains an affine transformation matrix M as it reads symbols in T one after the other. When an *affine transformation terminal* A is encountered, M is multiplied on the right by the affine matrix A. G is a *geometry terminal*, which contains a set of material parameters and points in $\mathbb{R}^3$ specifying a piece of geometry. When encountered, each point in G is multiplied by the current value of M and is added to the scene. [and] are the *push and pop terminals*. When [is encountered, the current value of M is copied and pushed to a stack of past transformations; when] is encountered, the last transformation is popped off of the stack and replaces M. Symbols in N_p are allowed to produce any kind of terminal. Symbols in N_s can produce any terminal or non-terminal except geometry terminals; it can only indirectly produce geometry by first producing a symbol in N_p. We refer to a *bone* b as a substring of the grammar’s output that takes the form $[A.^*]$, where .* refers to any number of other terminals. We only operate on well-formed strings, where all [symbols are matched with a corresponding] in the .* region inside the bone. This construct applies the transformation A to everything between the push and pop terminals. In the discussion of our cost function, we refer to the bone as being the *parent* of any other bones or geometry contained within its push and pop commands. With these parent-child relationships among bones, we construct a tree structure for generated models, which we call *skeletons*, based on the string of terminals from the grammar.

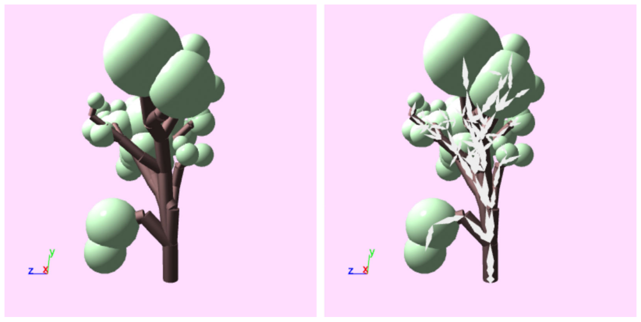


Figure 2: A model, shown with its skeleton overlaid above its geometry.

The initial skeleton phase is where SOSMC optimization takes place. A defining characteristic of SOSMC is that it randomly selects symbols in N to rewrite using a production rule. The skeleton phase restricts the choice to symbols in N_s, deferring creation of concrete geometry and freeing the optimization from looking at anything other than general shape. The middle wrap-up phase exists because an iteration limit is typically placed on the optimization process for performance reasons. With such a limit imposed, there may be non-terminal symbols in N_s left over at the end of the skeleton phase. The wrap-up phase allows these rules to be rewritten into N_p rules before moving on to the final phase. This is specified with a special set of production rules, R_w, used only during the wrap-up phase. In the final phase, the geometry phase, all the remaining non-terminal symbols should be in N_p, where they are rewritten to completion without any optimization.

3.3 Cost Function

The skeleton phase enables control of procedurally generated models by having the user draw guiding curves. The curves serve two functions: curve tangents are used to specify the growth direction of generated shapes at any point in space, and curve positions help constrain grown shapes to target regions. These two factors give the ability to specify what grown shapes should look like without over-constraining the results or limiting the scope of a generator's utility. The guiding curve control is implemented as a cost function designed to be used with the SOSMC method.

Users provide multiple guiding Bézier curves, Γ, to shape the generation of a model. For each guide $g \in \Gamma$ provided, we define $\textsc{Closest}(g, p)$ as the closest point on the Bézier curve to the provided point p, and $\textsc{Dir}(g, p)$ as the normalized tangent of the curve at point p.

For each bone b in the structure of a model, we define $\textsc{Parent}(b)$ as the bone that b is attached to, if it exists, $\textsc{Base}(b)$ as the world-space position of the point on b that was attached to its parent, or the origin if there is no parent, and $\textsc{Tip}(b)$ as the world-space position of the end of b (which we define by convention to be the world-space position of the bone coordinate space's origin). Using these definitions, the *direction* of a bone is $\textsc{Tip}(b) - \textsc{Base}(b)$.

With this, we define the cost function per skeleton bone b. First, as shown in Figure 3a, a single guiding curve γ is chosen to affect the cost of adding b by finding the closest curve to the base of b:

$$\gamma := \underset{g \in \Gamma}{\operatorname{argmin}} |\textsc{Base}(b) - \textsc{Closest}(g, \textsc{Base}(b))| \tag{1}$$

Cost is added based on the distance from b to γ:

$$\textsc{Dist} := |\textsc{Base}(b) - \textsc{Closest}(\gamma, \textsc{Base}(b))| \tag{2}$$

$$\textsc{Cost}_d := \sum_{i=0}^{2} (\gamma.\texttt{scale\_d}[i])\textsc{Dist}^i \tag{3}$$

The $\gamma.\texttt{scale\_d}[i]$ coefficients are user-defined.

Cost is also added based on how closely aligned b is to the tangent at the closest point on γ, shown in Figure 3b. This is done by taking the dot product of the tangent and the normalized direction vector to b from its parent:

$$\hat{d}_{bone} := -\frac{\textsc{Tip}(b) - \textsc{Base}(b)}{|\textsc{Tip}(b) - \textsc{Base}(b)|} \tag{4}$$

$$\hat{d}_{guide} := \textsc{Dir}(\gamma, \textsc{Closest}(\gamma, \textsc{Base}(b))) \tag{5}$$

$$\textsc{Cost}_a := \left(\hat{d}_{bone} \cdot \hat{d}_{guide} + \gamma.\texttt{delta\_a}\right) \gamma.\texttt{scale\_a} \tag{6}$$

The cost coefficient $\gamma.\texttt{delta\_a}$ and alignment offset $\gamma.\texttt{scale\_a}$ are both user-defined. $\gamma.\texttt{delta\_a}$ must be in $[-1, 1]$ and represents how aligned a new bone needs to be with the guiding curve's tangent for it to receive a negative cost and thus for there to be incentive to add it. It affects the region of beneficial alignment shown in Figure 3b. With no offset, the unscaled alignment cost is the negative dot product of the normalized direction of the bone and the normalized direction of the guiding curve. Perfectly aligned bones have an alignment cost of -1 (meaning this bone will lower the overall cost of a model), bones perpendicular to the curve have a cost of 0, and bones in the exact opposite direction have a cost of 1. When a positive offset is introduced, the structure needs to be closer to being perfectly aligned in order to receive a negative cost. The higher the offset, the closer it must be to perfect alignment. A negative offset means that alignment is more lenient and bones can be incentivized even if they are facing away from the tangents of their guiding curves. This can be useful if adding *any* new structure should be incentivized over only adding well-aligned structure.

To compute *utilization* cost, each curve is divided into segments of a pre-defined length. We define $\gamma.\texttt{segments}(b)$ (or S for brevity) to be range of segments between the closest point to $\textsc{Base}(b)$ and the closest point to $\textsc{Tip}(b)$, inclusive. Each $s \in S$ gives the number of bones which previously used that segment to calculate its cost, as shown in Figure 3d. The overall cost decreases by an exponentially decreasing factor of the number of utilizations of that segment:

$$\textsc{Cost}_u := -\gamma.\texttt{scale\_u} \sum_{s \in S} e^{-s} \tag{7}$$

$\gamma.\texttt{scale\_u}$ is a positive user-defined constant, which makes the $\textsc{Cost}_u$ negative. The final cost is then defined as:

$$\textsc{Cost} := \textsc{Cost}_d + \textsc{Cost}_a + \textsc{Cost}_u \tag{8}$$

`scale_a`, `scale_d`, and `scale_u` are picked to balance how selective an ideal model should be with bone placement. Increasing `scale_a` relative to `scale_d` and `scale_u` favours well-aligned bones regardless of where they are. Increasing `scale_u` encourages uniform growth along the curves. `scale_d` limits the benefits from the previous costs to bones close to the guides.

The cost of a whole model x is the sum of the costs for every bone in its skeleton. Because addition is commutative, the alignment and

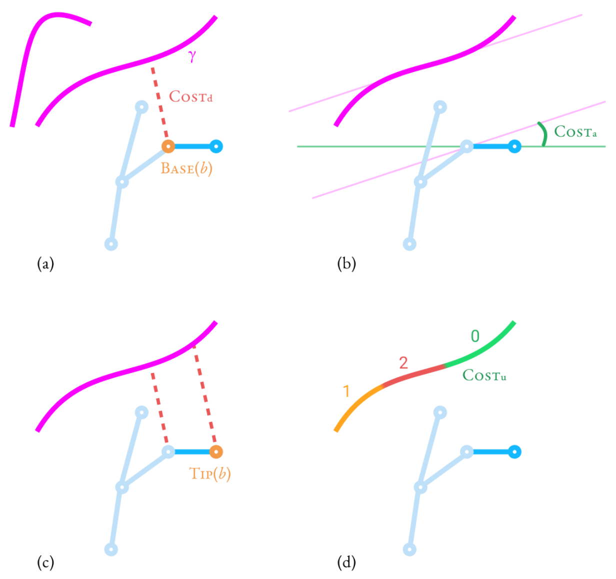


Figure 3: The steps to evaluate the cost function. (a) shows the selection of a the closest guide and the measure used in the distance cost; (b) shows the measure used in the alignment cost; (c) shows the mapping of the bone tip onto the selected guide; (d) shows the number of bones previously aligned with segments of the guide, used to determine the utilization cost.

distance costs per bone are independent of execution order, as well as the total utilization costs per bone segment. Therefore, the entire $\textsc{Cost}(x)$ term of the likelihood function is independent of execution order, as required by SOSMC.

3.4 Heuristic Cost

Since performance is a primary goal, it is beneficial to use more SOSMC samples on unfinished models that are more likely to produce low-cost complete models. To better accomplish this, we introduce a heuristic cost to try to predict the future cost of the model when more non-terminal symbols have been expanded.

The alignment cost, $\textsc{Cost}_a$, of the future model is estimated. When a grammar is first created, for each non-terminal symbol $n_s \in N_s$ in the grammar, a number of models X_{n_s} are generated using n_s as their starting symbol. When generating these models, M is initially an identity matrix. We then define the "expected" vector for n_s, which is the average sampled direction vector over the set of generated models X_{n_s}:

$$\textsc{Expected}(n_s) := \sum_{x \in X_{n_s}} \sum_{b \in x} \frac{\textsc{Tip}(b) - \textsc{Base}(b)}{|X_{n_s}||x|} \tag{9}$$

This cost intentionally generates a single average direction so that only one vector per non-terminal needs to be transformed into local coordinate space to compute the heuristic cost of a model: if it takes too long to compute, it becomes more useful to spend that time sampling more models without any heuristic.

For a partially complete model x created by string Z, the heuristic cost is defined as the sum of the alignment costs for the expected vectors of each symbol $Z[i]$ in Z that contains a non-terminal symbol in N_s. The expected vectors are transformed from their initial coordinate spaces into the coordinate space M_i defined by the affine transformations, pushes, and pops that occur in the prefix of Z ending at index i. We also use $\textsc{Dir}_{\mathscr{O}_i} = \textsc{Dir}(\gamma, \textsc{Closest}(\gamma, M_i[0,0,0]^T))$ as shorthand to denote the tangent at the closest point on γ to the origin in M_i in world space.

$$\begin{aligned} &\textsc{HeuristicCost}(x) := \\ &\sum_{i\,:\,Z[i]\in N_s} \left(-\frac{M_i \textsc{Expected}(Z[i])}{|M_i \textsc{Expected}(Z[i])|} \cdot \textsc{Dir}_{\mathscr{O}_i} + \gamma.\texttt{delta\_a} \right) \gamma.\texttt{scale\_a} \end{aligned} \tag{10}$$

The heuristic cost is then multiplied by a scaling factor, $\textsc{HeuristicScale}$. This factor starts high and then ramps down to zero in each successive generation. This is done to provide more heuristic guidance initially, when there is the least information available. The number of samples per generation is additionally ramped down in each successive generation under the assumption that there will be less variance between the costs of samples in a generation the further one goes in SOSMC and the more data is present.

The heuristic cost does not depend on the order in which symbols are rewritten. Take two models x_1 and x_2 with identical derivation trees but different orders of execution. Since the same number of symbols are rewritten in each iteration of SOSMC, x_1 and x_2 are also in the same generation. Models in the same generation use the same value of $\textsc{HeuristicScale}$, so because everything else about them is the same as well, x_1 and x_2 have the same cost. Therefore, the heuristic function preserves the independence between the likelihood function and the model's execution order, as required by SOSMC.

To demonstrate that the heuristic cost helps find lower cost samples given a time constraint, we compared the final costs of models generated with different approaches. We used the same generating grammar for trees (see Appendix A) and guiding curves for all samples. First, we generated models without using any heuristic. Then, we generated models without any heuristic, but with successive generation sizes ramped down. Then, we compared the results with both generation size ramping and heuristic cost enabled. We picked the generation sizes so that generating a model using each method takes at most 200ms, with ramping simply shifting the distribution of samples in the optimization such that there are more in earlier generations. The distributions of $\textsc{Cost}(x)$ for each final model x are compared in Figure 4. Note that these cost values do not include a heuristic cost, since they are cost values for complete models, which have no non-terminal symbols and therefore have no heuristic cost.

The heuristic manages to find, in general, lower-cost final models than are found without the heuristic, given the same time constraints, when the guiding curves have stricter alignment offsets. However, the less aligned one requires a model to be, the less important the heuristic cost's estimate is, so the heuristic becomes less useful with greater emphasis on distance. In these cases, it is not worth using, and may be turned off.

4 Implementation and Editor

To demonstrate the utility of our system, we created a procedural model editor that lets a user specify a generating grammar and draw curves to explore the generator's model space. We call our editor "Calder" in honor of Alexander Calder, the American sculptor who is known for using wire to construct three-dimensional abstract line drawings of various objects. Our implementation and editor are open-source at `github.com/calder-gl/{calder, playground}`.

4.1 Architecture

Our implementation aims to ensure minimal computation time. For a given partially-generated model, it may spawn several more-complete models in the next generation of SOSMC that share all the same pieces except for the one new piece. Our model data structure is therefore optimized to support efficient copy-and-add. A model is a tree structure of bones, where each bone has a pointer to its parent bone. A new bone can then be added in one sample without needing to copy any of the pieces from the previous generation. When

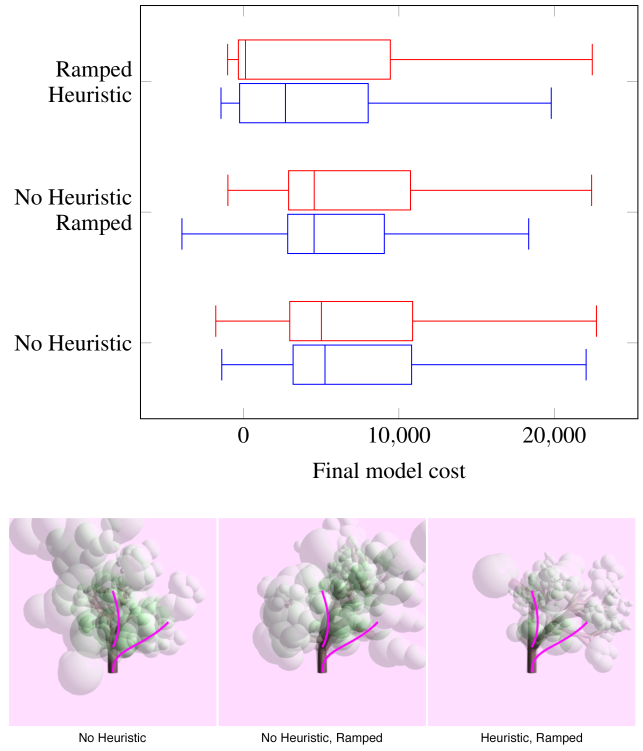
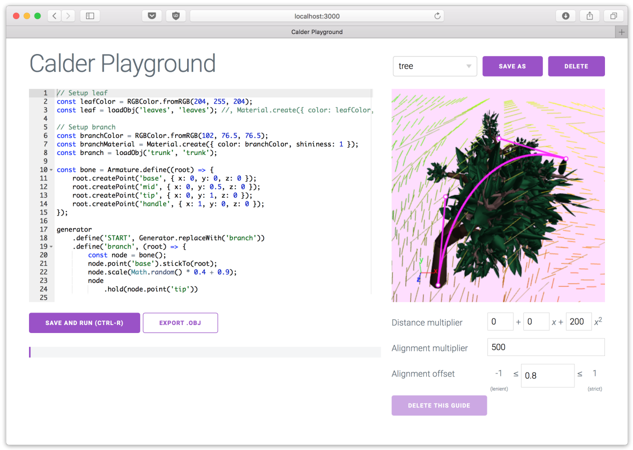


Figure 4: Comparison of final model cost distributions with and without using heuristic costs in a time limit of 200ms. Top: alignment offsets of 0.6 and 0.5 were used and displayed in red and blue, respectively. 500 models were used to create each distribution. Bottom: each tree is the average of the four models closest to the mean from each 0.6-offset distribution.

rendering a final model, a reference to every piece is needed. This is accomplished by creating a singly linked list of recently added pieces, where the most recent is the head. Nodes are given a pointer to the bone added before it in addition to the pointer to its parent bone. A given model needs to hold a reference only to the most recently added element to then be able to render itself in its entirety.

During rendering, a parent's absolute transformation matrix must be computed before a child's, since the child represents its transformation in terms of its parent. We create a stack out of the list of pieces in the model. We traverse the stack, maintaining a mapping from piece to absolute transform for quick lookup when one of its children is encountered. If we encounter a piece for which we need a parent transformation that has not yet been computed, we leave the child at the top of the stack, and then add its parent on top. The same will apply to the parent recursively until a processed node or the root is reached, creating an ordered dependency chain. Since nodes added in this chain have not been computed yet, they must not have been traversed yet, and must also occur later in the stack. Since the child is left on the stack, it will then be processed after its dependencies, so it will not need to be added to the stack again. Thus, each node is seen at most twice in the stack, so we can use child-to-parent pointers and still maintain $O(n)$ runtime for n items.

While our optimization process is fast enough to enable iterative design, it is not so fast that it can be done within a single frame and maintain 30 frames per second. Our editor is also a web application, where we must either use a single thread, or serialize data to pass between a main thread and a worker thread. We opt to use one thread, and do as much optimization as possible within a time budget. The optimization task is resumed every frame until it is complete, at which point the reference to the model being shown onscreen is swapped out with the model resulting from the optimization. The SOSMC algorithm is implemented in a JavaScript generator function. This produces a coroutine that yields between each sample in each generation, allowing it to check if its time budget has been exceeded and if it needs to continue in the next frame.



```
// Setup leaf
const leafColor = RGBColor.fromRGB(204, 255, 204);
const leaf = loadObj('leaves', 'leaves'); //, Material.create({ color: leafColor,

// Setup branch
const branchColor = RGBColor.fromRGB(102, 76.5, 76.5);
const branchMaterial = Material.create({ color: branchColor, shininess: 1 });
const branch = loadObj('trunk', 'trunk');

const bone = Armature.define((root) => {
    root.createPoint('base', { x: 0, y: 0, z: 0 });
    root.createPoint('mid', { x: 0, y: 0.5, z: 0 });
    root.createPoint('tip', { x: 0, y: 1, z: 0 });
    root.createPoint('handle', { x: 1, y: 0, z: 0 });
});

generator
    .define('START', Generator.replaceWith('branch'))
    .define('branch', (root) => {
        const node = bone();
        node.point('base').stickTo(root);
        node.scale(Math.random() * 0.4 + 0.9);
        node
            .hold(node.point('tip'))

```

Distance multiplier 0 + 0 x + 200 x^2

Alignment multiplier 500

Alignment offset -1 (lenient) ≤ 0.8 ≤ 1 (strict)

Figure 5: Our procedural model editor. The left pane contains the grammar in Appendix A. The right pane shows a generated model and guiding curves the user has drawn. Parameters for the selected guiding curve are shown below the 3D view.

4.2 User Interface

Our editor, shown in Figure 5, is split into two panes: a left pane for specifying a generating grammar, and a right pane for drawing guiding curves.

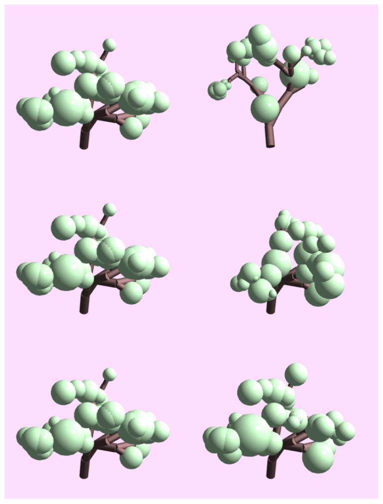
To specify a generating grammar, users write code in a domain-specific language built on top of JavaScript. It provides ready-made abstractions for specifying grammar rewrite rules for bones using a `defineWeighted(symbol, weight, generatingFunction)` syntax. A generating function allows the rule it defines to be written in terms of other rules through a "spawn point" abstraction using the `addDetail(symbol, position)` method. Positions are points in 3D space, referenced by named control points on bone primitives. Bones can be attached to other bones using the `stickTo(parent)` method, and can then be transformed using a standard set of geometric operations. To specify how to "finish up" skeleton-phase symbols early, the `wrapUp(symbol, generatingFunction)` construct is provided. Finally, grammar symbols that add geometry are specified using `thenComplete(symbols)`, or can be added implicitly from another rule in a `decorate(() => {...})` block. A full example of the grammar used to generate the tree in Figure 9 is shown in Appendix A.

Guiding curves can be drawn directly onscreen. Each drawn curve gets converted to a single Bézier path to emphasize the fact that guiding curves are meant to be coarse. Manipulating curves in three dimensions becomes tedious when there are too many points to control, so simpler curves also maintain ease of editing. Curves can be clicked on directly to show their control points, which can be dragged around in screen-space. The camera can be moved by dragging, if one wants to change the plane upon which control points are moved. When a curve is selected, controls appear below the 3D viewport where cost function parameters for the curve can be edited.

5 Discussion

5.1 Analysis

To analyze the efficacy of guiding curves at constraining model shape, we needed a way to efficiently represent a model's features, often called a *shape descriptor*. Several shape descriptors in the literature [6, 11, 16] are "global" in that they encapsulate the shape of the entire model. Global shape descriptors usually try to group models with similar shapes but different macro structures. In our analysis, however, since we compare models created from the same grammar, the shape descriptor distances will be constrained in their range of possible values. We benefit from a shape descriptor that can distinguish "local" features, such as branch positions along a tree trunk, and provide a richer range of shape descriptor distances. Our shape descriptor achieves this by leveraging the global D2 shape descriptor [11], a histogram of distances between two random points on the surface, but adjusted to add locality. Rather than computing the D2 distribution across the entire model, we first subdivide the model using a coarse voxel grid, and compute the D2 distribution for each subdivision. The end result is a voxel grid of histograms, which we will call the *subdivided-D2* shape descriptor. This makes the shape descriptor pose-dependent, which is acceptable for our purposes, because artists designing with these sets of tools are inherently trying to create models relative to drawn guides and not invariant to the viewing direction. To compare two shape descriptors, we compute the Kolmogorov-Smirnov distance [1] between the histograms at each voxel, and combine the results from all voxels to form a set of Kolmogorov-Smirnov distances. From here, we will call this set the *subdivided-D2 distance* of the two input models. Properties of this set, such as its median, act as indicators of similarity between the two input models. Figure 6 shows several examples of model pairs with their median Kolmogorov-Smirnov distances to demonstrate how it translates to visual difference.



Target Model	Model	Median Kolmogorov-Smirnov Distance	Grammar Mutation Distance
		0.53	60
		0.20	35
		0.10	5

Figure 6: Examples of Kolmogorov-Smirnov distances between pairs of models sampled 100 times from the grammar in Appendix A. When generating the models in each pair, both start from a common ancestor, and then diverge. The grammar mutation distance refers to the number of sampling iterations after this divergence.

We tested the efficacy of guiding curves by computing the subdivided-D2 distance of several pairs of models, and the results are shown in Figure 7. In these box plots, we used the median as an indicator of similarity between the models, and interquartile range as an indicator of similarity variance across all the subdivisions.

We observe that, as expected, a model is very similar to itself. The median Kolmogorov-Smirnov distance is not exactly 0 because the shape descriptors are computed non-deterministically and have slight variations between executions. Furthermore, we also see that models created from the same guiding curves are more similar than models created from different guiding curves, and are more similar than models created without any guiding curves at all, which suggests that guiding curves are effective at producing similar results. Models created from different guiding curves are less similar than models created without any guiding curves, which makes sense, as the former pair of models were intended to be different from each other, and we put no constraints on the latter pair. This further supports guiding curves' effectiveness.

Examining interquartile ranges, we see that the subdivided-D2 distance between a model and itself is small; since all corresponding subdivisions are identical, this variance measures the noise in the shape descriptor. We also see that the pairs of models created with guiding curves all have smaller interquartile ranges than the pair of models created without guiding curves. This provides more evidence that guiding curves are effective in making intentionally-similar models similar and intentionally-different models different; the models that were created without such intention had the freedom to be very similar in some subdivisions and very different in others.

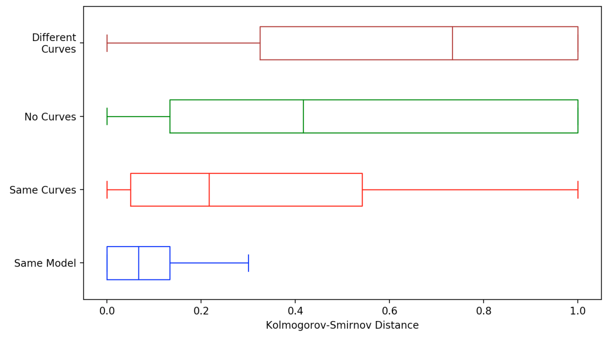


Figure 7: A comparison of shape descriptor distances (shown as box plots) for a variety of model pairs, generated using the grammar in Appendix A.

For models created using guiding curves, we also generated similar models using a silhouette matching cost function [13] to compare result similarity and the running time needed to achieve it. Models created with guiding curves each took less than 200ms to create, while models created with silhouette matching took roughly 45s each, using a coarsely detailed, 100×100 pixel silhouette image. A box plot comparison of Kolmogorov-Smirnov distances is shown in Figure 8. Note that the median Kolmogorov-Smirnov distance is smaller for the guiding curve comparison than the silhouette matching comparison, suggesting that our method constrains the output models more tightly than silhouette matching when providing coarse structure for models. Although silhouettes provide a richer set of constraining information than a small set of Bézier curves, they do not distinguish between models that "mostly" fill the silhouette, and make it slow to converge to the exact silhouette. Using a more detailed silhouette to mitigate this problem requires significantly more effort on the user's part and works against the goal of providing only a coarse structure, while comparable results can be achieved more easily with the specification of a small number of guiding curves.

5.2 Limitations

Specifying target shapes with guiding curves is particularly well suited for the sorts of objects that follow clear paths as they grow. Plants in general work well because they have a clear underlying structure. However, it is less intuitive to use guiding curves for areas and volumes, as might be used to create indoor scenes. A volume

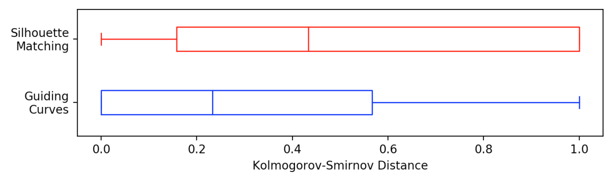


Figure 8: A comparison of subdivided-D2 distances for two models created with the same guiding curves, and two models created with the same silhouette. All models were created from the grammar in Appendix A.

throughout which model growth should be encouraged cannot be directly specified in our system. It can only be tediously approximated with many cross-hatched lines, with lenient alignment offsets to prevent directional artifacts due to the cross-hatching pattern.

Alignment with guiding curves makes the most sense as a cost function when generating grammars allow for a variety of bone angles to be produced. If grammars can only generate fixed angles, then it may be impossible for the grammar to produce results that align well with the curves. For cases where, for example, growth is restricted to cardinal axes, sketched curves are functional but awkward.

There are cases where discontinuities in the alignment vector field might become visible in models generated densely with geometry. While this is less of an issue in models with a strong distance cost, where bones prefer to stay out of these in-between regions, models that emphasize alignment may produce artifacts.

This method is also not well suited for complex skeletal structures. The more bones could potentially be added, the more iterations of SOSMC sampling are required. To accommodate, our method makes a clear separation between the skeleton of a model and the concrete geometry placed upon this skeleton. The intention is to use skeleton bones conservatively, with a high importance per bone, and then geometry can “loosely” be applied on top without using probabilistic sampling. Some applications, perhaps finely detailed fractal patterns, may require a more detailed skeletal structure than our method is optimized for. Models requiring this may yield results where more deeply nested elements lack detail or accuracy because there were not enough sampling opportunities in these regions.

Our method’s focus on uses that have a simpler skeleton makes it well-suited for the generation of structures with clear macro patterns and freedom in generating micro patterns. The micro patterns can be left for the computer to fill in when geometry is placed on top of the model skeleton. Plants again tend to work well under this setup. It is more difficult to apply our method to patterns where more detail is required in the macro structure and less detail in the micro structure, as this shifts work from the geometry phase to the skeleton optimization phase. We encountered friction of this sort when attempting to guide the generation of cityscapes.

6 Conclusion

We presented a definition of model skeletons and a cost function that operates on these skeletons to apply SOSMC sampling at interactive rates. This enables artists to search the space of models produced by their grammar to find results aligned with curves they draw. We believe this method is a step towards making controlled procedural modelling practical by allowing the control mechanism to be a part of the feedback loop between artists and their tools.

There are enhancements to the cost function used in this work that would be valuable for artists. Applications requiring control of areas and volumes could receive better support. Future work could give guiding curves finite, specifiable thickness over their lengths. With a suitable user interface, this could provide a more intuitive way to specify regions that uniformly encourage growth. Another extension might include a target density to discourage too many bones from being added in the same region.

Another direction for future work is to focus on the human-computer interaction aspects of the procedural modelling workflow. While this work assists in searching the model space of a grammar, creating the grammar in the first place is still an art requiring a

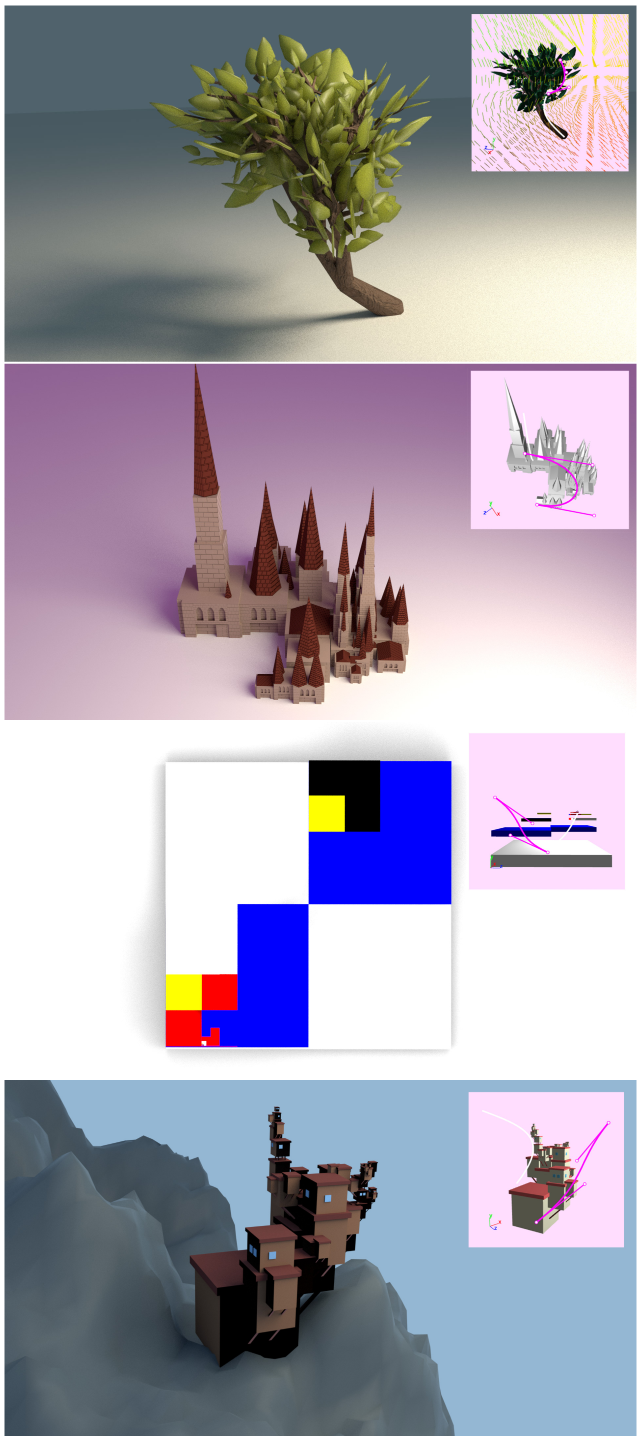


Figure 9: A tree, a cathedral, a Mondrian painting, and a modular house using imported 3D models as building blocks in the grammar, rendered in Blender after being generated and exported by the editor.

fair amount of skill. When creating a grammar to use with guiding curves, one must currently possess an intuition for many abstract concepts. Relative coordinate spaces must be kept in mind when specifying where to place bones, as well as the distribution of bones that will be created when introducing random variation. In addition to visualizing the bones and geometry of a single generated model, it would be useful to visualize the range in which bones could potentially be spawned. Perhaps the editor pane could detect which grammar rule is currently being edited and display visual information in the 3D view corresponding to the changes actively being made.

We hope that applications of the work presented help create new and interesting computer-generated art.

Acknowledgments

We would like to give thanks to Craig Kaplan for his discussions, advice, and well-selected choices of inspirational work, all of which helped guide this project to its current state. We would also like to thank the anonymous reviewers for their feedback and suggestions.

References

[1] *The Concise Encyclopedia of Statistics*, pp. 283–287. Springer New York, New York, NY, 2008. doi: 10.1007/978-0-387-32833-1_214

[2] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956. doi: 10.1109/TIT.1956.1056813

[3] I. M. Dart, G. De Rossi, and J. Togelius. Speedrock: Procedural rocks through grammars and evolution. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pp. 8:1–8:4. ACM, New York, NY, USA, 2011. doi: 10.1145/2000919.2000927

[4] S. R. Eddy and R. Durbin. Rna sequence analysis using covariance models. *Nucleic Acids Research*, 22(11):2079–2088, 1994. doi: 10.1093/nar/22.11.2079

[5] T. Ijiri, S. Owada, and T. Igarashi. The sketch l-system: Global control of tree modeling using free-form strokes. In A. Butz, B. Fisher, A. Krüger, and P. Olivier, eds., *Smart Graphics*, pp. 138–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[6] M. Kazhdan, T. Funkhouser, and S. Rusinkiewicz. Rotation invariant spherical harmonic representation of 3D shape descriptors. In *Symposium on Geometry Processing*, June 2003.

[7] A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280 – 299, 1968. doi: 10.1016/0022-5193(68)90079-9

[8] B. Mark, T. Berechet, T. Mahlmann, and J. Togelius. Procedural generation of 3d caves for games on the gpu. In *FDG*, 2015.

[9] J.-E. Marvie, J. Perret, and K. Bouatouch. Fl-system : A functional l-system for procedural geometric modeling. *The Visual Computer*, 21:329–339, 06 2005. doi: 10.1007/s00371-005-0289-z

[10] C. A. Monk, J. G. Trafton, and D. A. Boehm-Davis. The effect of interruption duration and demand on resuming suspended goals. *Journal of Experimental Psychology: Applied*, 14:299–213, 2008. doi: 10.1037/a0014402ER

[11] R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin. Shape distributions. *ACM Trans. Graph.*, 21(4):807–832, Oct. 2002. doi: 10.1145/571647.571648

[12] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1996.

[13] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan. Controlling procedural modeling programs with stochastically-ordered sequential monte carlo. *ACM Trans. Graph.*, 34(4):105:1–105:11, July 2015. doi: 10.1145/2766895

[14] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjlander, R. C. Underwood, and D. Haussler. Stochastic context-free grammers for trna modeling. *Nucleic Acids Research*, 22(23):5112–5120, 1994. doi: 10.1093/nar/22.23.5112

[15] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2):11:1–11:14, Apr. 2011. doi: 10.1145/1944846.1944851

[16] D. Vranic and D. Saupe. 3d shape descriptor based on 3d fourier transform. In K. Fazekas, ed., *Proceedings of ECMCS- 2001, the 3 rd EURASIP Conference on Digital Signal Processing for Multimedia Communications and Services, 11 - 13 September 2001, Budapest, Hungary*, pp. 271–274. Scientific Assoc. of Infocommunications, Budapest, 2001.

[17] L. Xu and D. Mould. Procedural tree modeling with guiding vectors. *Comput. Graph. Forum*, 34(7):47–56, Oct. 2015. doi: 10.1111/cgf.12744

A Tree Grammar

The code for the tree grammar used throughout the paper is provided below.

```
// Load leaf.obj and leaf.mtl
const leaf = loadObj('leaves', 'leaves');

// Load trunk.obj and trunk.mtl
const branch = loadObj('trunk', 'trunk');

const bone = Armature.define((root) => {
  root.createPoint('base', {x: 0, y: 0, z: 0});
  root.createPoint('mid', {x: 0, y: 0.5, z: 0});
  root.createPoint('tip', {x: 0, y: 1, z: 0});
  root.createPoint('handle', {x: 1, y: 0, z: 0});
});

generator
  // define == defineWeighted(., 1, .)
  .define('START', Generator.replaceWith('branch'))
  .define('branch', (root) => {
    const node = bone();
    node.point('base').stickTo(root);
    node.scale(Math.random() * 0.4 + 0.9);
    node.hold(node.point('tip'))
      .rotate(Math.random() * 360)
      .release();
    node.hold(node.point('handle'))
      .rotate(Math.random() * 80)
      .release();
    node.scale(0.7);

    // Add a post-skeleton phase rule
    Generator.decorate(() => {
      const trunk =
        node.point('mid').attachModel(branch);
      trunk.scale({ x: 0.2, y: 0.5, z: 0.2 });
    });

    Generator.addDetail({
      component: 'branchOrLeaf', at: node.point('tip')
    });
  })
  .defineWeighted(
    'branchOrLeaf', 1, Generator.replaceWith('leaf'))
  .defineWeighted('branchOrLeaf', 4, (root) => {
    // Add one required branch
    Generator.addDetail({
      component: 'branch', at: root
    });
    // Add two optional branches
    range(2).forEach(() => Generator.addDetail({
      component: 'maybeBranch', at: root
    }));
  })
  .define('leaf', (root) => {
    const leafBone = bone();
    leafBone
      .hold(leafBone.point('base'))
      .grab(leafBone.point('tip'))
      .pointAt({x: 0, y: 0, z: -20})
      .release();
    leafBone.createPoint(
      'leafAnchor', {x: 0.6, y: 0.2, z: 0.9});
    leafBone.point('leafAnchor').attachModel(leaf);
    leafBone.scale(Math.random() + 0.4);
    leafBone.point('base').stickTo(root);
  })
  // maybe == define(., .).define(., () => {})
  .maybe('maybeBranch', Generator.replaceWith('branch'))
  .wrapUpMany(
    ['branch', 'branchOrLeaf', 'maybeBranch'],
    Generator.replaceWith('leaf')
  )
  .thenComplete(['leaf']);
```