PICO: Procedural Iterative Constrained Optimizer for Geometric Modeling

Vojtěch Krs, Radomír Měch, Mathieu Gaillard, Nathan Carr, and Bedrich Benes *IEEE Senior Member*

Abstract—Procedural modeling has produced amazing results, yet fundamental issues such as controllability and limited user guidance persist. We introduce a novel procedural model called PICO (Procedural Iterative Constrained Optimizer) and PICO-Graph that is the underlying procedural model designed with optimization in mind. The key novelty of PICO is that it enables the exploration of generative designs by combining both user and environmental constraints into a single framework by using optimization without the need to write procedural rules. The PICO-Graph procedural model consists of a set of geometry generating operations and a set of axioms connected in a directed cyclic graph. The forward generation is initiated by a set of axioms that use the connections to send coordinate systems and geometric objects through the PICO-Graph, which in turn generates more objects. This allows for fast generation of complex and varied geometries. Moreover, we combine PICO-Graph with efficient optimization that allows for quick exploration of the generated models and the generation of variants. The user defines the rules, the axioms, and the set of constraints; for example, whether an existing object should be supported by the generated model, whether symmetries exist, whether the object should spin, etc. PICO then generates a class of geometric models and optimizes them so that they fulfill the constraints. The generation and the optimization in our implementation provides interactive user control during model execution providing continuous feedback. For example, the user can sketch the constraints and guide the geometry to meet these specified goals. We show PICO on a variety of examples such as the generation of procedural chairs with multiple supports, generation of support structures for 3D printing, generation of spinning objects, or generation of procedural terrains matching a given input. Our framework could be used as a component in a larger design workflow; its strongest application is in the early rapid ideation and prototyping phases.

Index Terms—Computational Geometry and Object Modeling, Three-Dimensional Graphics and Realism

1 Introduction

Procedural modeling has been successfully applied in a wide variety of areas such as vegetation, texturing, architecture, and decorative design. One of its most important strengths is the ability to encapsulate a large variety of shapes into a concise formal description that can be efficiently parameterized. This, in effect, allows for the generation of variants of the structures by changing the procedural model parameters or rules. This approach is also called forward procedural modeling.

While procedural modeling has been recognized as a strong and expressive methodology with many application areas, its strength has not been fully harnessed because of its disadvantages. Probably the most important problem is the difficulty to fully comprehend the procedural model derivation from an initial state (axiom). Procedural models often exhibit complex behavior and non-linearity between input parameters and the output shape [1], [2], because the rules can exponentially amplify some features while diminishing others. The designer is usually left with trial-and-error experimentation. Due to the lack of controllability, practical applications of procedural models usually hide the procedural rules and show just an interface [3], [4]. Alternatively, they may provide sets of examples that are reused which is a common strategy adopted by many commercial products [5], [6].

- *V. Krs, M. Gaillard, and B. Benes were with the Department of Computer Graphics Technology, Purdue University, West Lafayette, IN, 47907.*
- *N. Carr and R. Měch were with Adobe Research, San Jose, CA, 95110*

Procedural models can be targeted to a specific goal by using optimization. User-defined constraints have been used to control procedural models as detailed in Section 2. These approaches attempt to automatically find the parameters of the procedural system that generate results matching the user-specified requirements. However, existing systems target narrow domains and usually focus on a single procedural model. More general approaches capable of working with multiple procedural representations often lack interactivity due to the high-dimensionality of parameter space that must be explored and lack of any domain-specific information that could otherwise speed up the process. Furthermore, they optimize a predefined function that cannot be changed during the optimization. In fact, prior work focuses predominantly on optimizing the *derivation* of a *predefined* model, *e.g.,* a tree grammar that grows into a desired shape. However, there has been little work on optimizing the procedural rules themselves.

Several observations motivate this work. First, a *set* of user-defined constraints can be used together to impose complex requirements on the generated objects. For example, the *function* of the object can be specified with a handful of geometric constraints, as we demonstrate by generating a variety of free-form chairs (Figure 1). The second key observation is that a *broader procedural system* can be created that encompasses the commonly used hand-crafted grammars. This generic system can then be optimized to produce a wider range of objects that a single hand-crafted grammar cannot express. User's intent can then be approximately achieved by simply defining a set of constraints and se-

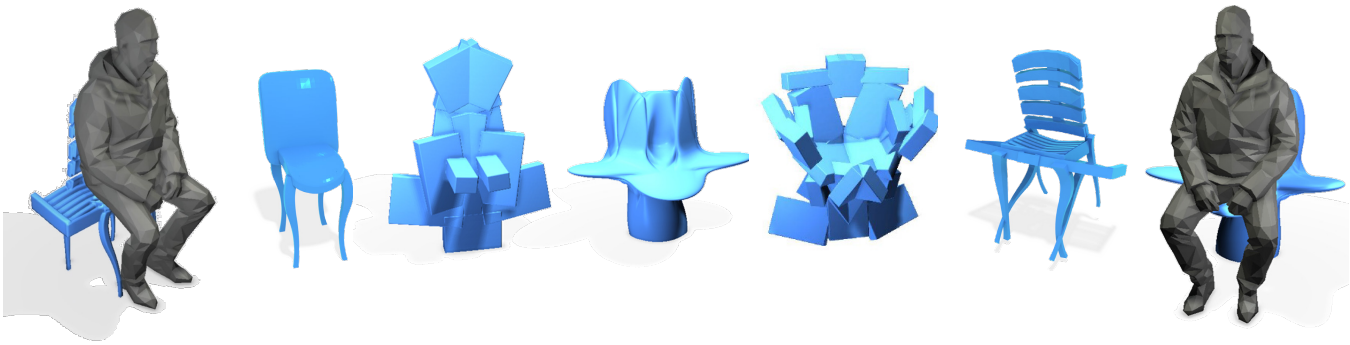

Fig. 1. **Chairs**. PICO automatically generated various procedural models of a chair. The user first marks the bottom of the character as an active area and PICO generates the procedural chair that supports it. Additionally, models were constrained to be stable, i.e., not to tip over.

lecting parts from which to build the object, be it simple primitives or existing geometry. This is again demonstrated by Figure 1, where no chair-specific grammar has been created, and the procedural models were evolved automatically. The last observation is that, contrary to existing off-line approaches, instant visual feedback and the ability to interactively control the optimization process significantly improves the expressiveness of procedural modeling.

We introduce PICO (Procedural Iterative Constrained Optimizer), a framework for procedural geometry optimization and interactive modeling. At its heart is a novel procedural model which we call PICO-Graph. This model uses a data-flow paradigm, where nodes represent geometry generation operations and edges define travel paths of objects. Objects travel through this graph between source (axiom) and sink (scene output) nodes, triggering operations that generate more geometry whenever they arrive at any node in the graph. This representation supports branching, recursion, and instancing. The definition of the geometry generation operations and traveling objects is flexible and supports arbitrary 2D and 3D geometry such as user-defined meshes. The PICO framework itself consists of an interactive constraint definition system and an optimization engine that refines the PICO-Graph to match the given constraints. The optimization uses a multi-objective evolutionary algorithm which is capable of optimizing graphs with cycles, as opposed to only derivation trees.

An important property of our semi-automatic approach is that it is a bridge between two very orthogonal worlds: manual editing that provides absolute control over the generated model and the procedural modeling that produces semi-random structures. Our generated models share the randomness of the procedural models but allow certain level of control defined by the optimization constraints. Such generated models are a good starting point for manual editing or ideation of further models.

We demonstrate the capabilities of PICO on a variety of examples, including automatically generated geometries of chairs, trees, 3D printing supports, and terrains. We show that many of these examples can be controlled interactively by using simple and intuitive constraints. Furthermore, we demonstrate that our optimizer coupled with the PICO-Graph representation outperforms previous work in terms of speed. We claim the following main contributions:

1) A novel procedural model, called PICO-Graph, that generates a wide range of 3D and 2D geometry. A simple design with fast evaluation makes it suitable for optimization.
2) We couple PICO-Graph with a novel optimization technique that allows for interactive user-controlled structure generation.
3) We introduce a novel procedural workflow at a high level of abstraction, where the user provides building blocks but the system finds their relationships to generate the desired object.

An example in Figure 1 shows an application of our framework. The input is a 3D mesh model of a person. The user interaction consists of marking areas that require support, specifying additional constraints, *e.g.,* stability and mass minimization, and choosing the building blocks for the model. Our optimization then evolves models that satisfy the given constraints.

2 Related Work

Procedural modeling is a broad topic that has been applied in a variety of contexts. Early explorations leveraged fractals, focused on generation of terrains [7], [8], and vegetation [9]. Shape grammars and split grammars [10] were successfully applied into architectural models in [11]. Split grammars were extended in various directions including procedural buildings [12] and just recently into a procedural model called `CGA++` in [13]. Numerous examples of purely procedural models exist such as the approach of Merrell et al. [14] that generates infinite architectural structures by using only procedural rules. For readers interested in more broad coverage of the topic, we reference a number of state-of-the-art surveys. These include the generation of procedural worlds [15], [16], optimization of procedural models for games [17], and inverse procedural modeling [18].

Control for Procedural Models: The drawback of most procedural modeling systems is the lack of artistic control, which motivates an active research interest. Ijiri et al. [19] introduced a system that can encode a simple user sketch as L-system and Palubicky et al. [20] used sketching of attraction particles to interactively control growth of simulated vegetation. Closely related is the work of Mitra and Pauly [21] who optimize 3D structures so that they match user-defined shadows. Guided procedural modeling [22]

generalizes the concept of environments by closing procedural models into guides that can communicate by message passing and Krecklau and Kobbelt [23] introduced a procedural model that allows for generation of interconnected structures. Many design tools have been designed around specific tasks such as the design specific shape classes (*e.g.*, chairs [24] or terrains [25]) or handling the arrangement or placement of shapes [26]. By targeting the system with some level of domain specificity, more compelling results can often be achieved. In contrast, our system is agnostic to the class of shapes allowing it to be used on a a variety of tasks.

There has been a focus on neural approaches in recent years. Ritchie et al. suggests controlling procedural models by stochastically-ordered sequential Monte Carlo programs in [27], but later introduced neurally-guided procedural models in [28]. Recently, procedural models were coupled with sketching and deep learning to provide a more natural interface for artists [3], [4], where a neural network recognizes the sketch and predicts the procedural model and its parameters. Neural based methods have been recently applied to program optimization as well. One of the most recent examples is the work of Ellis et al. [29], who proposed a method that is able to take simple hand-drawn images and translate them into a graphic programs able to generate LaTeX-style figures. The graphics programs follow a simple grammar that include simple primitive drawing, loops and conditional statements. Similarly, Sharma et al. [30] showed a neural approach that infers a simple program, equivalent to a CSG hierarchy, that constructs a given 2D or 3D shape. Their method uses reinforcement learning and encoder-decoder architecture, where the input is an image of an object, the output is a program that generates an object, and the reward is the difference of the two in image space. Conversely, Du et al. [31] introduced an analytic method of synthesizing a CSG tree from existing geometry using a search of possible CSG programs.

Finally, procedural models give rise to an often exponential space of variation. While approaches have been suggested for navigating and exploring these spaces [32], more direct artistic control remains challenging.

Procedural Modeling and Optimization: Procedural models were coupled with various optimization approaches in the past. Sims used a combination of genetic algorithms along with competition for resources to evolve virtual creatures in an environment with simple physics in his seminal paper [33]. Hornby et al. [34] used L-systems and evolutionary algorithms to generate various shapes and Talton et al. [35] used L-systems to parse states of expression of a rule set to find an optimal geometry by using Metropolis Hasting variant called Reversible Jump Monte Carlo Markov Chains (MCMC). Contrary to the previous work, our approach does not require fixed set of rules and the rules and their dependencies are generated automatically during the optimization step. Merrell et al. [36] used similar approach to organize furniture and MCMC was also used to layout synthesis in [37]. Yu at al. [38] used environmentally-sensitive optimization to organize furniture in a room and Peng et al. [39] used high-level specification of goals to optimize transportation network that was demonstrated on furniture layout and street traffic design. Localized learning of stochastic procedural models for virtual terrains has been used in brush-like approach in [40]. Although MCMC approaches provide good results, they tend to be very slow for large scenes.

Structurally sound masonry buildings were achieved via optimization in [41] and our approach shares analogy with this work in that it attempts to use functional constraints. However, our definition of function does not encompass only the structure, but also other aspects such as volume, touching, proximity, etc.

Měch and Miller [42] introduced Deco that uses a scripting language to generate 2D or 3D patterns by guiding the growth of the procedural model to follow the user input. In our approach, we control the model indirectly by modifying the constraints and by painting on the objects. Also similar to our method is the work of [43] who leverage graph grammars to evolve 3D shapes. However, the control of their method is low as opposed to our approach that allows using constraints to guide the procedural optimization to a desired output. Bergen et al. [44] used aesthetic criteria to evolve L-systems and Xu et al. [45] optimized shape collections of genetic algorithms by using a higher semantic representation. Finally, Haubenwallner et al. [46] used genetic algorithms to find procedural grammar expansion to match given constraints and was an inspiration for this work . In Section 6, we compare their approach to ours.

Most previous works use a fixed procedural model or provide a direct control for its definition. We were inspired by the seminal work [33] and ours is closest in spirit to [44], [46], [47]. Compared to Jacob [48], we propose a new expressive class of procedural models that can create a variety of shapes, without having to use predefined shapes like flowers or leaves [48], or use of voxel representation [44]. We also introduce a novel optimization system enabling an interactive control during the evolution process that allows for incremental updates.

Procedural model representations: Numerous representations for procedural models have been proposed, whose formalism is rooted in programming language design. These include data flow models as well as stream processing [49], [50]. Here we mention only the most relevant systems from which our work takes inspiration. Lindenmayer introduced L-systems [51] that were extended by geometric interpretation and recursion by Prusinkiewicz [52]. L-systems are linear, while our approach aims at volumetric objects and allows for geometric operations on them. In general, L-system rules are not easy to evolve directly, and as a result only a relatively simple cases of L-systems have been evolved so far [48], [53]. Stellar grammars [54] were used to generate subdivision structures and this approach is similar to ours, except we attempt to expand each vertex. Similarly, `vv-system` allows vertex-vertex expansion to simulate subdivision surface in [55]. Our procedural model is close to the operator graph representation [56] with the most important difference being that we use a mix of coordinate frames and 2D/3D primitives as the traveling objects among the rules that control the generated shape. Moreover, we also provide novel optimization approach that allows for reconnecting the rules, their mutations, and cross-over.

Our system shares similarities with approaches used in existing software, eg. Grasshopper in Rhinoceros 3D [57],

Houdini [6] and Substance Designer [58]. These systems use a data-flow paradigm, however they do not support optimization of the graph and instead rely on manual specification of the procedural generation.

3 Method Overview

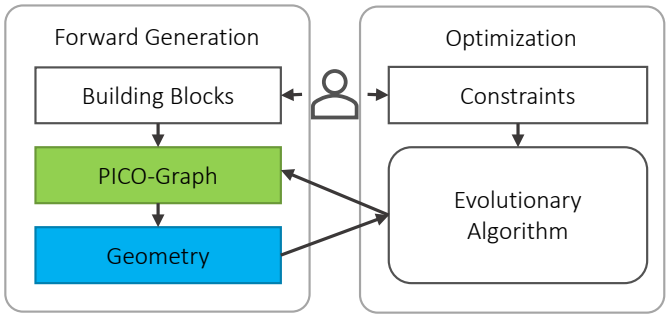

Fig. 2. Overview of PICO. User defines the *building blocks* which represent parameterized geometry generating operations with connectivity information. During the *Forward Generation* the user can connect the operations into a *PICO-Graph* that generates the output *geometry*. Alternatively, the user can define a set of *constraints* that are *optimized* for by an evolutionary algorithm. The constraints can be modified interactively while various geometries are generated and shown to the user.

The input to our method (see Figure 2) is a set of building blocks, *i.e.,* definition of geometry generating operations, and constraints, *i.e.,* requirements from the user how should the generated geometry look like. Geometry-generating operations can be either simple geometric objects, such as spheres or boxes, or user-defined geometries imported from existing meshes. These operations may be parameterized (size, orientation, recursion limit) and must contain information on how they can be connected to other building blocks. The connectivity information, in examples shown in this work, is a set of coordinate frame transformations.

The building blocks are connected into a *PICO-Graph*, which is the underlying procedural representation in our system. Although our framework supports manual definition of PICO-Graph, this may quickly become an overwhelming task when modeling complex objects. The key contribution of our work is the automatic generation of procedural models by using user-defined constraints and evolution. Some constraints can be specified by a simple toggle (*e.g.,* that the object should be stable), some require manual input (*e.g.,* sketching of support surfaces or image to match), and some require loading external geometry (for example for object avoidance). Each constraint has an associated importance that allows the user to control various design trade-offs. An important feature of our system is the fast evolution algorithm that allows for dynamic and interactive modifications of constraints by the user during the model generation.

PICO can be used for *forward generation* to generate geometry by manually defining the PICO-graph, *i.e.,* connecting individual building blocks. The PICO-graph is a dataflow graph in which objects travel from a source node (axiom) to a sink node (scene output). The objects traveling in our implementation are *coordinate frames* and *2D or 3D geometry*. The geometry-generating operations are therefore defined as taking either frames or geometry as input and outputting further frames or geometries or a combination of both. The actual procedural output geometry-generation starts by sending initial objects from the source nodes. The objects trigger the geometry-generating operations on the nodes they travel to. These operations generate new objects which are sent further into the graph. Finally, the objects accumulated at sink node(s) can be gathered into the final geometry (see an example of forward generation in Figure 3 and the accompanying video).

The *optimization* iteratively evaluates geometry against the user-specified constraints and modifies the PICO-Graph such that the generated geometry satisfies the constraints. Constraints that cannot be enforced directly are combined into a fitness that is maximized by solving a weighted multi-objective optimization problem by using a novel evolutionary algorithm. The algorithm maintains a population of individuals which are defined by using PICO-Graph as their genotype and the generated geometry as their phenotype. New solutions are generated using mutation and crossover operators defined over the PICO-Graph. Furthermore, we use niching, *i.e.,* we maintain different species in a population, to maintain diversity and to explore fitness landscape that may be multi-modal.

4 Forward Generation

PICO-Graph is the procedural model used in our system. It is built by using *building blocks*, *i.e.,* geometry-generating nodes that take other geometry as input and create more geometry. The PICO-Graph defines both the geometry-generation operations along with the order in which the operations should be applied to produce the final model. Figure 4 shows an overview of the PICO-Graph.

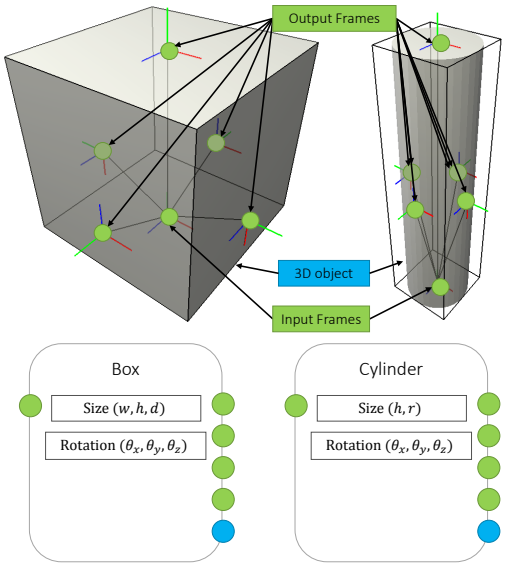

Fig. 3. An example of two building blocks generating a box and a cylinder respectively. The input and output frames can be positioned and orientated arbitrarily and define how the primitive will connect to others. The size and orientation with respect to an incoming frame are parameterized.

4.1 Building Blocks

The building blocks are geometry-generating operations Op that take in a spatial object S_{in} (triangle meshes, 3D coordinate frames, Gaussians, and Constructive Solid Geometry (CSG) trees in our implementation). The operation Op generates a new set of spatial objects $S_i, i \in (0, n)$ (which can be of different types), subject to the operation's parameters $p_j, j \in (0, k)$:

$$Op : (S_{in}, p_0, p_1, \cdots, p_k) \rightarrow (S_0, S_1, \cdots, S_n). \quad (1)$$

Figure 4 (bottom) shows a graphical representation of this general operation. We use two common forms of spatial objects in our implementation: *coordinate frames* and *2D/3D objects*. The coordinate frames F describe a linear transformation as a 4×4 matrix. The 2D/3D objects are either defined parameterically, for simple primitives such as spheres or cuboids, or using data, *e.g.,* a mesh or a signed distance field. The locations of the coordinate frame is parameterized and defined by the user. The number of coordinate frames depends on the number of outgoing connections.

Figure 3 shows two examples of the building blocks, one generating a box and the other a cylinder. Both take coordinate frames as input and output a 3D object and four more coordinate frames. The new coordinate frames can then be used to generate further objects. The operation generating a box is written as:

$$\begin{aligned} Box : (F_{in}, w, h, d, \theta_x, \theta_y, \theta_z) \rightarrow \\ (F_{front}, F_{top}, F_{left}, F_{bottom}, F_{right}, O_{box}), \\ F_{front} = T(0, h, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\ F_{top} = T(0, d/2, 0) R(-\pi/2, 0, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\ F_{left} = T(0, w/2, 0) R(0, 0, \pi/2) R(\theta_x, \theta_y, \theta_z) F_{in} \\ F_{bottom} = T(0, d/2, 0) R(\pi/2, 0, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\ F_{right} = T(0, w/2, 0) R(0, 0, -\pi/2) R(\theta_x, \theta_y, \theta_z) F_{in}, \end{aligned} \quad (2)$$

where T and R are translation and rotation matrices, respectively. The generated object O_{box} is a box of size (w, h, d) at the origin, transformed by $R(\theta_x, \theta_y, \theta_z) F_{in}$; in our implementation, we use θ to adjust the frame of every generated geometry. Furthermore, consistent in the notation in L-systems, the y axis is direction of procedural generation (growth) and it corresponds to the frame F_{front}.

Each of the building block's parameters $p_j \in P_j$ has an associated domain P_j; for example, the rotation angle parameters can be restricted to a certain range, *e.g.,* $-\pi/4 \leq P_{\theta_x} \leq \pi/4$. This equips the user with a degree of control over the general style of the generated geometry during the optimization (Section 5) .

4.2 PICO-Graph

The PICO-Graph is a data-flow graph that allows geometrical objects (coordinate frames and 2D/3D geometry) to flow through the graph. The PICO-Graph is a directed multi-graph G consisting of nodes $v_i \in V$ and directed edges $e_i \in E$:

$$G = (V, E). \quad (3)$$

Each node has a set of inputs I_{v_i} and outputs O_{v_i}, corresponding to I and O in Eqn(1). Edges connect individual outputs to inputs, providing one-to-one mapping:

$$E : \{O_{v_i} \; \forall v_i \in V\} \rightarrowtail \{I_{v_i} \; \forall v_i \in V\}. \quad (4)$$

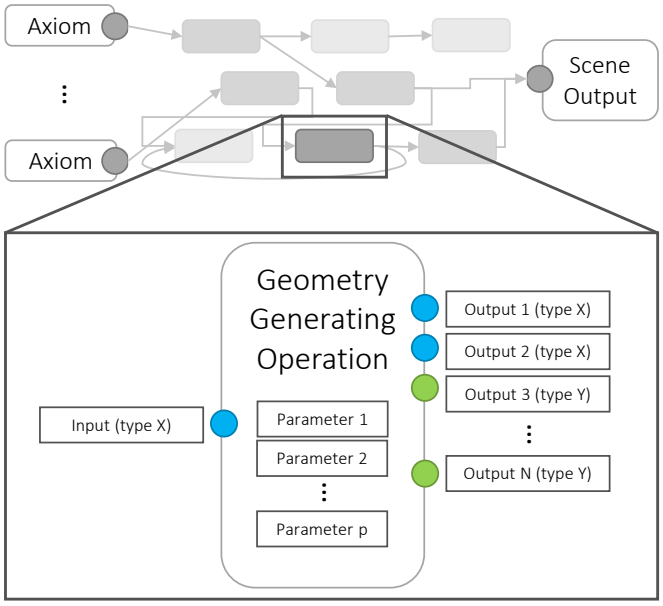

Fig. 4. Schematic of the PICO-Graph (top). The graph includes source (axiom) nodes, geometry generating nodes, and sink nodes (scene output). Objects travel through this graph from sources to sinks, invoking geometry generating operations, which create and send more objects down the graph. A general template for a geometry generating operation is shown in the zoomed portion. Each operation has a single input and multiple outputs (shown in different colors) and it has multiple parameters that influence the objects generation.

Note that this mapping allows multiple outputs connected to a single input. The set of all nodes V consists of three subsets: set of source (axiom) nodes, set of building block (geometry generating) nodes, and a set of sink nodes. Figure 4 shows a diagram of the graph, as well as a general building block node (inset). The source nodes (axioms) have no inputs. The sink nodes have no outputs and collect objects that traveled to them.

To generate geometry from PICO-Graph, we use the following iterative process:

1) Initialize queue Q with tuples (S, O_v), where S are objects produced by axiom nodes and O_v are node outputs.
2) While Q is not empty
   a) Remove tuple (S, O_v) from Q
   b) Find node u such that $I_u = E(O_v)$
   c) Execute operation associated with u, i.e., $S' = Op_u(S)$
   d) If u is a sink node, accumulate the result, otherwise add (S', O_u) to Q
3) Collect accumulated objects from sink nodes

In our implementation, we use Constructive Solid Geometry hierarchy to accumulate the 3D objects, and a flat array for 2D objects. We denote the generated geometry as $\mathcal{G}$.

The PICO-Graph may contain directed cycles (Figure 5) leading to a *recursive* generation. The recursion is tracked by counting each time an object, or its descendants, visit a given node, where S' is a descendant of S if $S' = Op(S)$. A recursion limit is enforced to stop further execution of an object (1-3 in our experiments).

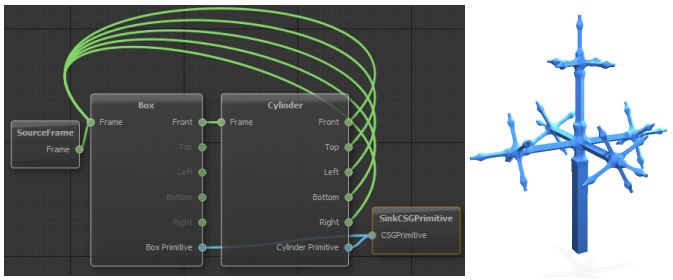

Fig. 5. An example of a PICO-Graph with cycles (left) and the generated recursive structure (right). Blue edges represent Constructive Solid Geometry (CSG) primitives and green edges 3D coordinate frames. The sink operation blends the incoming primitives and outputs them to the scene.

If a node has multiple incoming edges, the node is executed for each object that travels through it. Similarly to recursion, this produces an *instance* of the same geometry (at a different position with a different frame), but contrary to recursion, it only happens once (unless the node is part of a cycle as well).

5 Optimization

We have designed PICO so that the PICO-Graph can be generated and efficiently optimized automatically by using an evolutionary approach; providing immediate visual feedback (see the accompanying video) . To guide the optimization we use user-defined constraints. Some constraints can be enforced directly, for example symmetry, while others have to be quantified as objective functions that are minimized.

5.1 Hard Constraints

Hard constraints must always be met and they can be specified and enforced directly by modifying the PICO-Graph. Our current implementation supports a number of hard constraints including symmetry, spin and parameter spaces.

Parameter spaces: P_j for parameters p_j are defined by the user and the optimization is constrained to sample values from these spaces. Each space is defined by specifying minimum and maximum values. The optimization samples these spaces uniformly for initialization and perturbs them by a value sampled from a normal distribution.

Plane and axis *symmetry* can be set by the user interactively. If the building blocks contain two symmetrical frames F_0 and F_1, we modify the graph such that the outputs $O^0_{v_i}$ and $O^1_{v_i}$ of a node v_i that correspond to these frames are routed to the same input I_{v_j} of a node v_j. Because the objects output from node v_i are oriented according to the symmetric frames F_0 and F_1, the two sets of objects created further down the graph (in v_j and further) will be symmetric as well.

Spinning objects (Figure 13) have their center of mass aligned to the spinning axis and the spinning axis itself should be parallel to the maximal axis of inertia [59]. We transform the geometry to have its center of mass at the origin and we rotate it by using rotation Q that is computed by using the eigen-decomposition $Q\Lambda Q^T = I$ where I is the inertia tensor.

The **3D printing supports** in Figure 12 are constrained to have a maximum angle (45° in our example) and the overhang points are automatically connected to the nearest geometry. If there's no geometry in the cone specified by the above maximum angle, the overhang is connected directly to the ground to ensure printability.

5.2 Soft Constraints

In addition to hard constraints our system also supports the modeling of soft constraints. We model each soft constraint by an associated objective function. The optimization then minimizes all of the objective functions to find a Pareto optimal solution, subject to the hard constraints outlined above. The user can modify the importance of each constraint to further control the optimization. We categorize the objective functions into two types: **environmental** and **intrinsic**.

The **environmental** objective functions encompass extrinsic properties of the model including 3D protected volumes P_i, the scene bounding box Ω, ground plane G, and the points from the interacting surfaces from the input geometry Q_i.

The protected volumes are input by the user as 3D objects and they indicate 3D space that the generated geometry should avoid. The scene bounding box Ω limits the operational space of the generated geometry by defining its extent and making sure that the object does not become unreasonably large. The ground plane G makes sure the generated model touches the ground and is also used to optimize for stability of the objects that should not tip over.

Furthermore, the user can add user-defined objects to the scene and mark target areas by painting manually on their surfaces. We refer to them as *interacting surfaces* and they specify locations to which the generated geometry should grow. If the interaction surfaces are present, the goal of the optimization is to expand the procedural geometry so that it approximates the shape of the interacting surfaces, for example by generating a chair that follows the shape a person that sits on it. The interacting surfaces are sampled into a set of 3D points denoted by Q and the objective function attempts to minimize the distance between Q and the generated procedural geometry $\mathcal{G}$. If the goal is to generate an object that touches all points in Q, the objective function is

$$\frac{1}{|Q|}\sum_{q\in\mathcal{Q}}\frac{d(q,\mathcal{G})}{|\Omega_{diag}|}, \tag{5}$$

where $|\Omega_{diag}|$ denotes the length of the diagonal of the domain's bounding box, *i.e.,* the largest possible distance and $d(p,\mathcal{G})$ is the distance between a point p and $\mathcal{G}$.

If the goal is to only *touch* the interaction surface, for example the ground plane G, the function is

$$\min_{q\in\mathcal{Q}}\frac{d(q,\mathcal{G})}{|\Omega_{diag}|}. \tag{6}$$

Protected volumes specify regions into which the generated objects should not grow. We chose to model this constraint as soft, as it facilitates intermediate solutions that

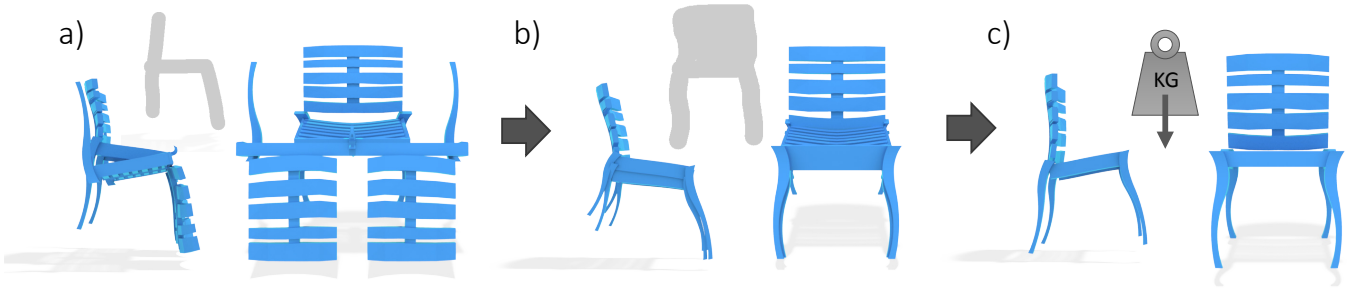

Fig. 6. Modeling a chair using several constraints. First, the user sketches a side view (a). However, the model is free to grow in the direction away or to the user. Therefore, a second sketch may be needed from another view (b). Finally, to remove unnecessary parts, a mass minimizing constraint is applied (c).

eventually lead to a solution without any collisions. The objective function is defined as

$$\frac{V(P \cap \mathcal{G})}{V(P)} \tag{7}$$

where P is the protected volume and V denotes the user-defined volume that should not be entered.

Sketching: To control the shape of the generated geometry more finely, we introduce a sketch matching constraint. The sketch is defined as a binary mask $\mathcal{I}_s$ that is either sketched or downloaded and it is compared to a perspective projection of the generated geometry $\mathcal{I}_g$. The objective function is defined as

$$smoothstep(N_0, 0, N_g) - smoothstep(N_1, 0, N_g), \tag{8}$$

where $smoothstep$ is the Hermite interpolation as implemented in GLSL [60], N_g is the number of set pixels in $\mathcal{I}_g$, N_0 is the number set in $\mathcal{I}_g$ but not in $\mathcal{I}_s$, and N_1 is the number set in both $\mathcal{I}_g$ and $\mathcal{I}_s$.

The *stability* of the generated geometry $\mathcal{G}$ is also optimized. For an object to be stable the following equation must hold:

$$m' \in \mathrm{Conv}(\mathcal{G} \cap G), \tag{9}$$

where m' is the center of mass m projected along the gravity vector to the ground plane G, and Conv denotes a convex hull. The objective function that maximizes stability is:

$$\frac{|m' - \mathrm{Conv}(\mathcal{G} \cap G)_{centroid}|}{|\Omega_{diag}|}. \tag{10}$$

Note that we assume constant density throughout the object to compute its center of mass .

The *spinnability* of the object can be guaranteed by the hard constraints outlined above, but the quality of the spin can be further improved by minimizing the ratio of its moments of inertia (Eqn (3)) in [59].

The **intrinsic** members of the objective function consider various properties of the generated structure $\mathcal{G}$. Intrinsic members are the volume of the bounding box of $\mathcal{G}$ its mass, number of generated geometric primitives, and the total length of the graph induced by the tokens passed around in the PICO graph.

We control the size of the object by minimizing its bounding volume using the following objective function

$$\frac{V(\mathcal{G}_{BB})}{V(\Omega)}. \tag{11}$$

Furthermore, to avoid bulky objects that contain unnecessary parts (with respect to other objectives) we minimize mass using

$$\frac{\rho V(\mathcal{G})}{V(\Omega)}, \tag{12}$$

where ρ is the density of the structure. We keep $\rho = 1$ in our implementation.

Figure 6 shows the effect of applying several constraints in the modeling process. The user is free to apply them at once or subsequently as needed, as is shown in the figure. First a side sketch is created and then one from the front, which determines the desired shape of the object. Finally, a mass minimizing constraint is used to simplify the generated model.

5.3 Evolutionary Algorithm

The evolutionary approach optimizes the set of objective functions given by the user-defined constraints. The main steps of the algorithm are population *initialization*, *speciation*, *evaluation*, *selection*, and *reproduction*. Our overall algorithm shares commonalities with Genetic Algorithms, *i.e.*, we define a genotype and a phenotype, and Genetic Programming [61], *i.e.*, we evolve graphs that can be conceptualized as programs. Furthermore, we adapted techniques from Neuroevolution of Augmenting Topologies (NEAT) [62] that allow us to measure compatibility of individuals for reproduction versus keeping a separate species.

The population is **initialized** with a set of random individuals, each representing the minimal working PICO-Graph, *i.e.*, one axiom, one geometry generating node, and one sink, each with randomized parameters. The individual consist of a genotype and a phenotype. The genotype is a description of a single PICO-Graph G. The genotype includes a list of nodes, along with their parameters, and a list of edges, along with information whether they are enabled or disabled. We keep an *innovation number* associated with every edge, which tracks new topological changes within the broader population and assist in the crossover operator and speciation. An edge between nodes is considered to be a *gene*. The phenotype is defined as the generated geometry $\mathcal{G}$ and is used for evaluation.

The **evaluation** consists of computing the fitness $F(I)$ for each individual I. Because the objective functions may

have different ranges and distributions, we use the *sum of weighted global ratios* [63] to compute the fitness:

$$F(I) = \frac{1}{\sum_{i=0}^{N-1} w_i} \sum_{i=0}^{N-1} w_i \frac{f_i(I) - f_i^{min}}{f_i^{max} - f_i^{min}}, \quad (13)$$

where f_i is the i-th objective function out of N, f_i^{min} and f_i^{max} are the minimum and maximum values of f_i for the entire population throughout all past generations, and w_i is the user-defined *importance* of a member function f_i. The importance of individual constraints is controlled by user via sliders, such that the sum of all importance values is equal to one.

Speciation is a process of dividing the population into multiple distinct species based on a similarity metric, called *compatibility*, such that genotypically similar individuals are grouped together and reproduce only within the species. This ensures diversity in the population and helps explore multi-modal fitness landscapes. We use a modified definition of compatibility from [62] which differs in the term quantifying identical genes. Our compatibility between two genotypes g_a and g_b is defined as:

$$\delta(g_a, g_b) = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}, \quad (14)$$

where N is the total number of genes (edges in the graph) and D and E is the number of disjoint and excess genes respectively (appearing in only one of the genotypes). The term $\overline{W}$ is computed as a distance between parameters p_j of the nodes $v \in V$ in the graph. Thanks to the innovation numbers, we can track nodes that occupy the same position in the graph topology, but are parameterized differently in different individuals. Therefore we sum over the all differences in parameters of nodes that are connected by the genes that appear in both genotypes. The difference between two parameters p_a and p_b are calculated by using an L_2-norm. The coefficients c_1, c_2 and c_3 are used to weight the contributions of genes in compatibility ($c_1 = 2$, $c_2 = 2$, and $c_3 = 1$ in our implementation). Finally, to decide if two individuals belong to same species, we use a threshold t_δ. If $\delta > t_\delta$, individuals do not belong to the same species and a new species is created, unless there exists an individual within an existing species whose compatibility is below the threshold. We vary t_δ during the optimization process to keep the number of species constant, in our case $3-5$ species for population size of 150. Finally, we employ fitness sharing within the species.

We **select** individuals for reproduction from the top $5-15\%$ individuals in each species. The **reproduction** uses two operators, *mutation* and *crossover*, to produce children from selected individuals. We either use mutation only (5% of the time), crossover only (85% of the time), or crossover with subsequent mutation.

We use five distinct types of mutations: 1) add a node (after an existing node), 2) insert node (between connected nodes), 3) mutate parameter, 4) add an edge, and 5) toggle edge.

Add node finds an open output in the graph and adds a randomly initialized node, causing the geometry to grow outward. *Insert node* finds an existing edge and replaces it with a new node and two new edges, again causing the model the grow by prolongation. *Mutate parameter* randomly perturbs parameters of the nodes, with low probability (5%) but by a large amount ($\sigma = 80\%$ of parameter space P, using a normal distribution). *Add an edge* connects two nodes that were not previously connected. Note that if this mutation is not performed, the graph will remain cycle-free and will resemble a grammar derivation tree, similar to the work of [46]. Finally, *toggle edge* randomly disables and enables edges in the graph, allowing for pruning of unnecessary parts of the graph or reactivating parts that may be relevant to the current state of the optimization.

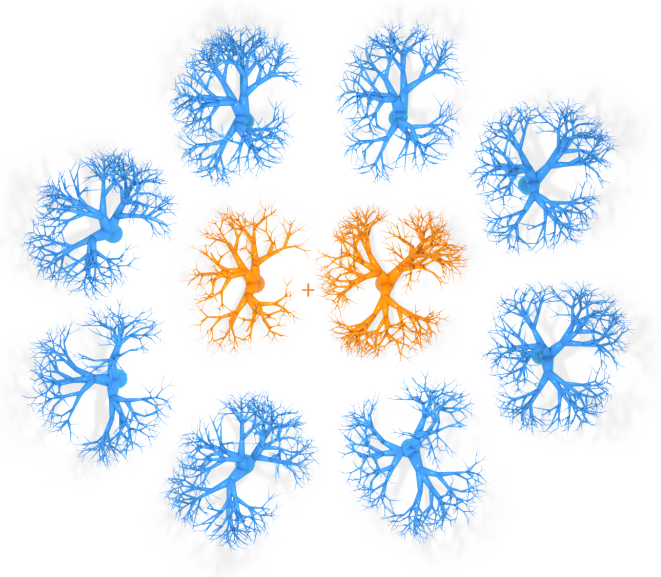

Fig. 7. Crossover on PICO-Graphs generating trees. The two middle orange trees are parents that generate blue off-springs by using the crossover operator.

We adapted the **crossover** operator from [62], with the main difference being that we need to transfer node parameters from parents to child. Two parent genotypes are first aligned using their innovation numbers, same as in the compatibility calculation (Eqn 14). Genes (*i.e.*, edges in the graph) that occur in both parents are randomly chosen from one parent to transfer to the child. Excess and disjoint genes are copied from the fitter parent. Finally, for whichever edge transfers to the child, the parameters associated with the nodes that the edge connects are transferred to the child as well. An example of the result of the crossover operator is shown in Figure 7.

Figure 7 shows an example of two generated structures resembling biological trees (top) that were combined into four by two separate crossover operations.

The children replace all the parents after the reproduction step and form a new generation. However, we keep the best individual for each species, ensuring that the best to-date solution survives. The new generation is divided into the species again and the process is repeated until a stopping criterion has been met. In our implementation, we stop if after 100 generations there is no improvement in the fitness, or, in interactive sessions, whenever user decides to stop the optimization.

Our algorithm starts from a minimal graph and pro-

gressively increases the number of nodes and edges in the graph, which increases the complexity of the generated model. The reproduction operators need to generate new solutions that would, ideally, be fitter than previous generation. However, in practice, the mutation and crossover operators often worsen the solution. We have observed that the rate of improvement gradually slows down with the increased complexity, particularly because there are a lot of mutations performed on parts of the graph that do not need to be mutated. For example, in case of tree growth in Figure 10, mutations to nodes generating the root and initial branches do not need to be changed after first few hundred of generations. For that reason we use *gene freezing*. We track whenever a gene mutation contributed to improvement in fitness. If there has been no improvement in a certain number of generations (50 in our implementation), the gene is frozen and cannot be mutated by parameter mutations and node insertion mutations. We randomly unfreeze frozen genes with a probability of 0.5%. Finally, we unfreeze all genes if any of the constraints have changed, so that the system can adapt to the new environment.

5.4 Convergence

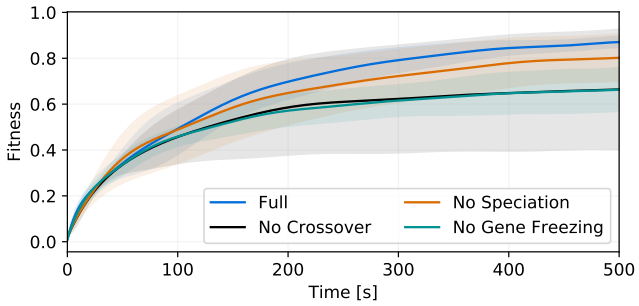

Fig. 8. Average fitness and its standard deviation through time for the tree sketch example (Figure 10). Individual curves show convergence for variations of the algorithm without crossover, speciation, or gene freezing. Values are an average of five runs.

Individual parts of the algorithm influence the overall convergence of the algorithm. Figure 8 shows an ablation experiment where we disabled different parts of the algorithm. We use the structures from Figure 10, where we try to grow a tree model that matches a sketch. Besides the full algorithm, we ran a variation without crossover (*i.e.*, asexual reproduction through mutation), speciation, and gene freezing. The fitness improves best over time if all parts are used and we conclude that all of these parts contribute to better convergence.

Figure 9 shows the influence of the population size on the convergence and time. The results are aligned with common behavior of genetic algorithms [64], [65]: increasing the size of the population improves the convergence significantly (left). However, at a cost of increased computation time (right). We chose the population size of 150 for our examples, because it gave us a good middle ground between speed and convergence.

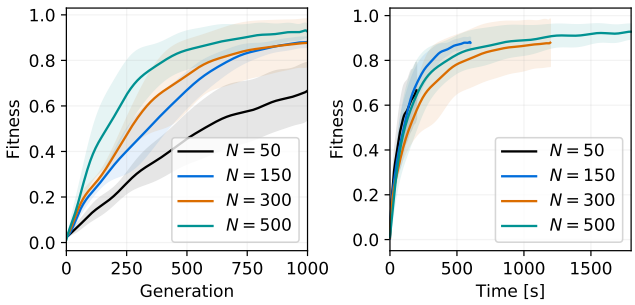

Fig. 9. Effect of population size N on the average fitness and its standard deviation through time for the tree sketch example (Figure 10). The experiment was run for 1,000 generations, and five runs per curve.

6 Implementation and Results

6.1 Implementation

We have implemented PICO in $\mathcal{C}$++ with support of OpenGL, GLSL and CUDA for rendering. Results were generated on a desktop computer with an Intel i7 processor clocked at 4.0 Ghz, 16 GB of RAM, and an NVIDIA Titan Xp graphics card.

We represent the 3D objects by a constructive solid geometry (CSG) tree, *i.e.,* a tree with set operations as inner nodes and geometric primitives as leafs. Because many of our objective functions require distance estimation, we represent geometric primitives by an analytic signed distance function or a signed distance field. Set operations are then performed on the signed distance d. For example, the union operation between two primitives a and b is defined as $min(d_a(p), d_b(p))$ for a point p.

We render objects by using ray-marching on the GPU implemented in CUDA, where tree traversals are expensive. Therefore we convert the CSG tree into a custom program representation using the Sethi-Ullman [66] algorithm. The resulting program is then uploaded to the CUDA constant memory and evaluated on the GPU, or evaluated directly on the CPU.

In order to quickly detect collisions, we convert all meshes into a signed distance field (SDF) representation by first voxelizing using ray-casting and then using the Fast Marching Method [67]. This conversion happens offline for each individual mesh (including building blocks) used in the system, using a 128^3 grid to store the distance values. The collision volume is then calculated as the volume of the intersection of the SDF of the mesh and SDF of the CSG tree. We do this calculation recursively, by subdividing the domain's axis aligned box (AABB), evaluating the SDF at the box's center. If the absolute distance is greater than half the diagonal of the AABB, the entire AABB is either completely inside or outside of the volume, depending on the sign of the distance. Otherwise we subdivide further until we reach a certain depth ($6-8$ in our implementation). The same algorithm is used to compute the mass, volume, and moment of inertia tensor of the generated object.

6.2 Results

We used the evolutionary optimization to interactively generate the following results. The various examples differ in

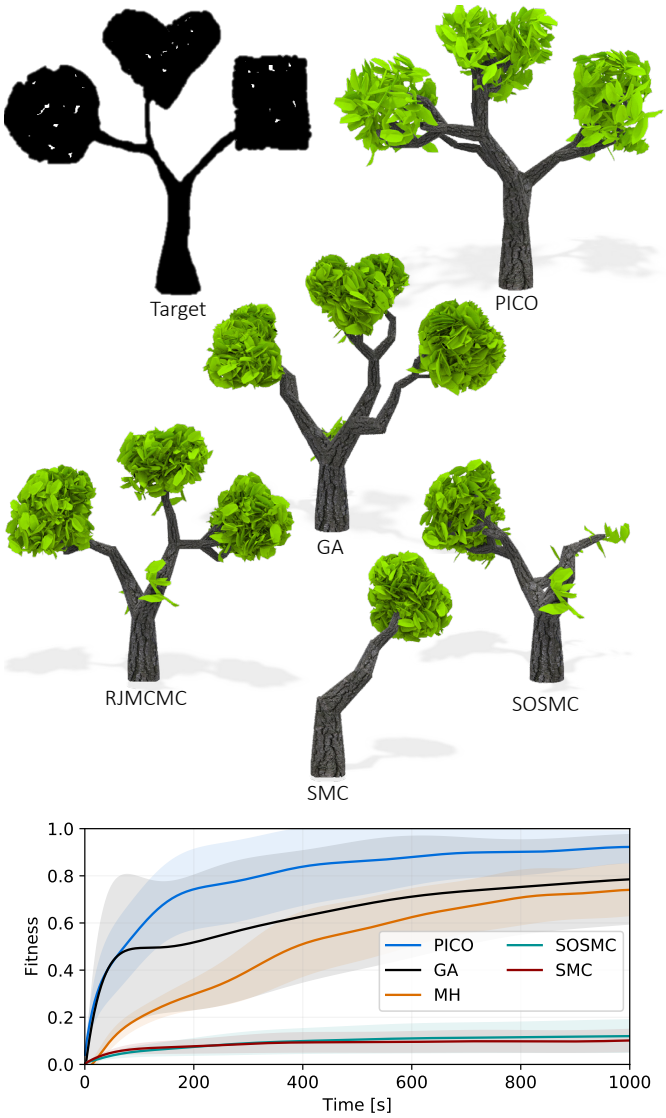

Fig. 10. **Tree sketch**. Evaluation of tree sketching from ShapeGenetics [46]. Target image specifies areas where the tree should grow. We show a comparison of generated models from our system and models generated using ShapeGenetics implementation of various algorithms. We also show that our system achieves higher fitness faster (bottom). Curves correspond to are average fitness over 10 runs and bands show the standard deviation.

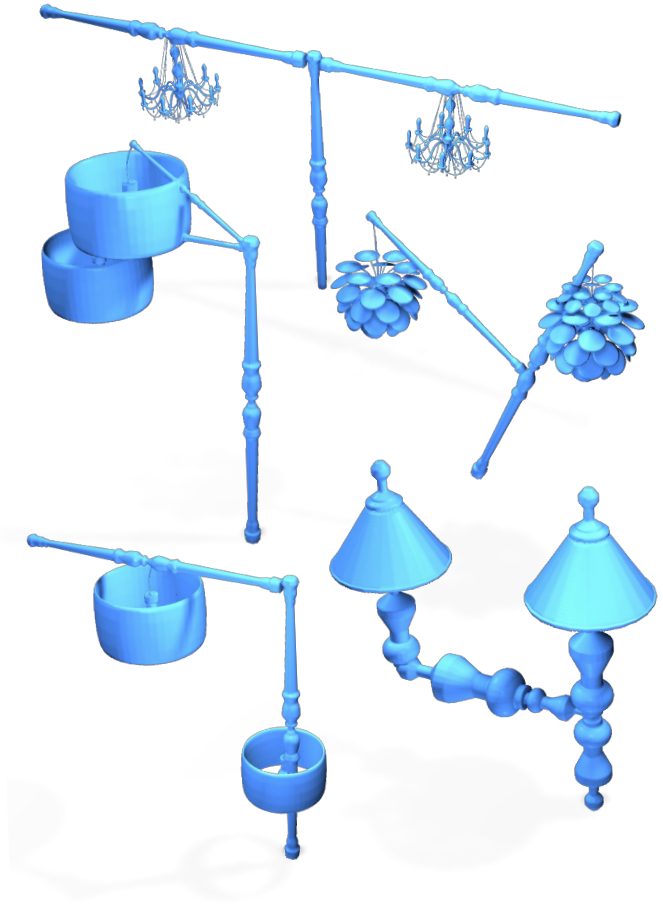

Fig. 11. **User generated lamps.** We conducted a pilot study where users were asked to sketch lamps and pick building blocks out of which the lamps were built. The figure shows a subset of the generated lamps.

the types of buildings blocks and the set of constraints used.

The first example in Figure 1 shows an array of generated 3D chairs. The bottom and back part of the person are marked and a 3D chair is fully automatically generated by optimizing for touching the marked areas, stability, and small mass of the entire structure. We have building blocks defined from actual chair meshes to make the result visually plausible.

To evaluate our approach against existing methods, we chose to recreate the tree grammar from ShapeGenetics (see Figure 8a in the paper [46]). We used a single type of building block that generated branch geometry and branched three-fold, or generated leaf geometry if none of its outputs were being used. The constraint in this experiment was matching a sketch shown in Figure 10, and we used identical fitness and experiment setup to ShapeGenetics. We ran PICO and the implementation of Genetic Algorithm (GA) [46], Reversible Jump Monte Carlo Markov Chain (RJMCMC) [32], Sequential Markov Chain (SMC) [69] and Stochastically Ordered Markov Chain (SOSMC) [27]. The result geometry is shown in Figure 10 (top). Figure 10 (bottom) shows the mean fitness and its standard deviation as a function of time for individual methods. The SMC and SOSMC methods have issues converging from the start and are unable to cover the entire space of the sketch. The RJMCMC and GA methods converge to satisfactory results. Our method outperforms them, especially in the first half of the optimization process. This is likely because our framework searches the candidate space more efficiently, focusing on several search directions concurrently using speciation, and spending less time on sub-optimal search directions thanks to gene freezing and gene alignment in crossover.

We conducted a **small pilot user study** with four participants who were asked to create a simple model of a lamp. The users were asked to sketch a desired shape and pick building blocks for the lamp's body and shade. The building blocks were created by segmenting several ShapeNet [70] lamp models. We also added an objective minimizing number of generated parts. The participants filled a small survey on a four point Likert-scale (2-strongly agree, 1-agree, -1-disagree, -2-strongly disagree). The results to our questions were: *This system is easy to use: 0.75*, *I can*

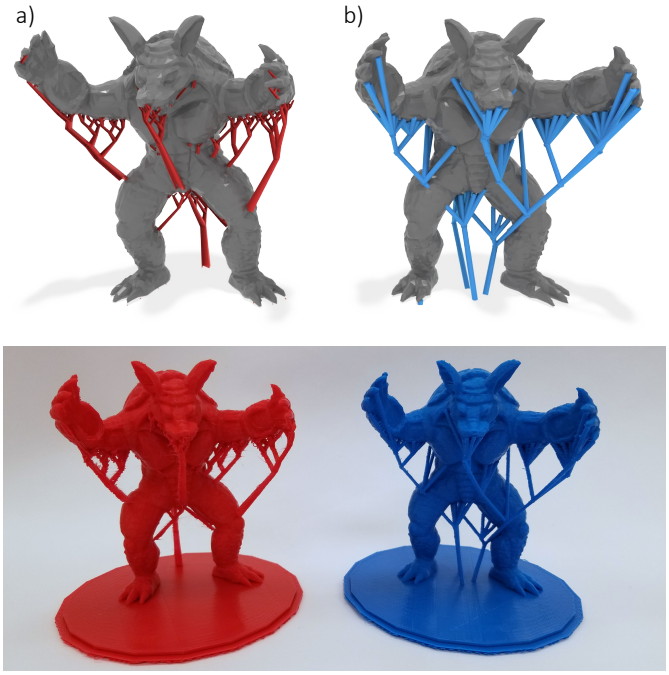

Fig. 12. **Armadillo supports**. PICO can automatically generate organic supports for 3D printing. We compared our generated supports (b) to the work of [68] (a). We used the same overhang points and same dimensions of the print and we achieved a comparable resulting weight of the used material.

achieve my intent quickly: 1.5, *I can control the design easily: 0.5*, *The response is fast: -0.25*, and *I need to understand procedural modeling: -0.75* indicating that the need to understand procedural rules is not necessary in order to generate models by using our framework. Moreover, the participants identified themselves as *I have previous experience in procedural modeling: -0.75*, and *I have previous experience in computer design: -0.25*. Figure 11 shows examples generated by the users.

An interesting application of PICO is for generating organic **support structures for 3D printing** (Figure 12). We compared our approach to CleverSupport [68], a method that grows tree-like supports from overhang points. We took the same model and used the same sampling and generated the supports using PICO. The building blocks used were simple cylinders, branching up to four-fold. There were two hard constraints used: angle with gravity vector had to be less than 45° and all the overhang points had to be connected to our generated object. The main optimization goal was minimization of the length of the structure. Note that we use multiple axiom nodes in this example to grow multiple tree-like structures at the same time. We printed the object and compared the resulting weight of used material. Ours being 85.10g, compared to 86.54g achieved by [68]. There are factors that were not considered, for example, structural strength, tips for easy removal or optimized profile of the supports. However, we show that we achieve the same task with a comparable amount of material.

Another example shows an automatically generated structure that is able to be spun and stay balanced while **spinning** (Figure 13); which is an application inspired by [59]. The main objective of the optimization is to achieve distribution of mass such that ratio of lateral axes of inertia to the principal axis is as small as possible. Furthermore, to control the shape of the spinning top, we sketch a rough shape from a side view. The center of mass and alignment of the principal axis of inertia with the spinning axis are enforced as hard constraints. Note that we intentionally disabled the hard symmetry constraint and we did not use symmetrical frames. Nevertheless, the system found symmetrical models automatically through optimization.

Although there are many methods for **procedural terrain** generation and we do not claim a contribution to this field, we wanted to show the expressiveness of our method by matching three real terrains taken as a sample of digital elevation map of Alps (resolution 64×64 pixels, 30 meters per pixel) by the PICO procedural model (Figure 14). The geometry-generating operations were 2D Gaussians modulated by Perlin noise. The resulting height map is the sum of the contributions of all the Gaussians primitives. We used Mean Square Error (MSE) to compare the height maps. The Gaussians cannot capture all the fine details of the terrain, especially erosion patterns, but they work for the overall appearance. The optimization time was about two minutes. The accompanying video shows the optimization process.

Figure 15 shows an example of interactive design. A procedural **hat hanger** is automatically generated and then expanded each time a new object is added to the scene.

Table 1 shows statistics of generation of the results. The time to generate the shown examples ranges from seconds to couple minutes, depending on the complexity of the object and the type of constraints used. The input includes the number of different constraints and number of different building blocks. The optimization consists of the generation time [ms], evaluation of fitness in [ms], reproduction time [ms], number of generations in the evolutionary optimization, and the total optimization time in seconds. The output includes number of generated coordinate frames, geometric objects (*e.g.*, primitives, meshes, Gaussians) and the total number of used geometry generating operations. The most expensive fitness calculation was for the Spinning objects in Figure 13, which includes calculating the moment of inertia tensor, and took on average 1,746.3 [ms] for the entire population. Concerning terrains, the generation operation takes the most time due to the cost of evaluating Perlin noise, which is the bottleneck for this application.

7 Conclusion

We have introduced PICO that uses PICO-Graph, which is a novel procedural model that is coupled with an evolutionary algorithm. PICO-Graph is a flexible graph representation that defines procedural generation by connecting simple geometry-generating operations. Geometric objects, in our examples coordinate frames and 2D/3D geometry, are sent from axiom nodes down the graph, triggering further geometry-generation in other nodes. We couple this representation with an evolutionary algorithm and we guide it using various user-defined hard and soft constraints as a means of control of the procedural generation. The evolutionary algorithm uses reproduction operators and genome compatibility defined over the PICO-Graph. Mutations are implemented as topological or parameter changes of the graph. We adapted the crossover and speciation for our

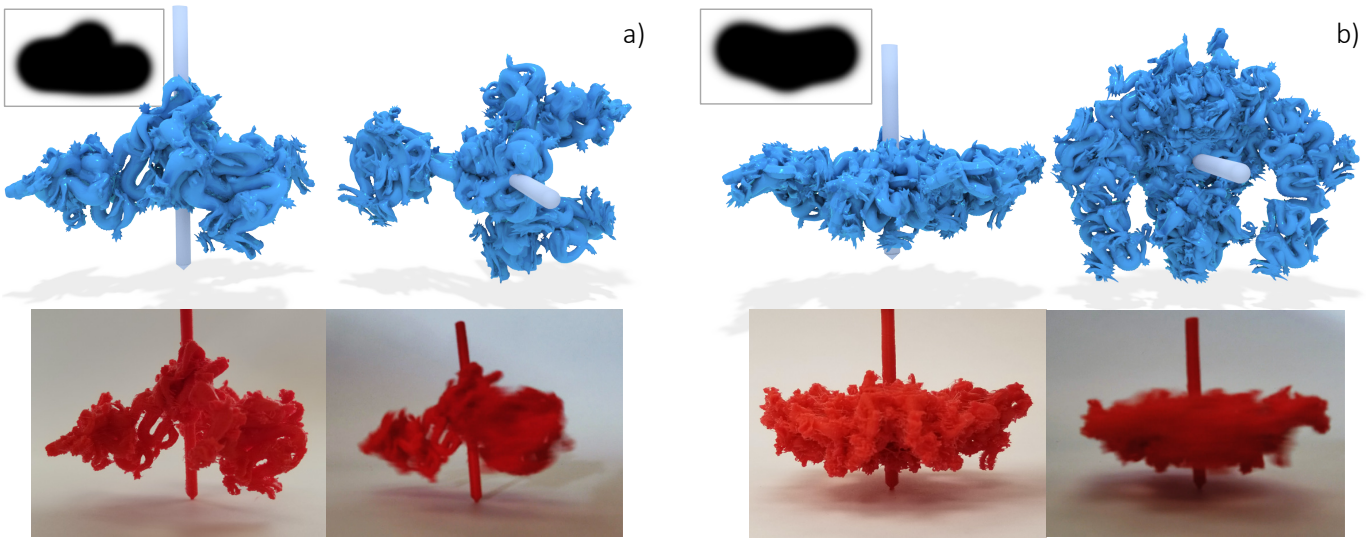

Fig. 13. **Spinnable objects**. Two spinnable objects a) and b), shown from a side and top view, have been generated from Stanford dragon building blocks. The shape was guided by a sketch constraint from a single view, shown in insets and corresponding to the view on the left. The center of mass and maximal axis of inertia have been aligned using hard constraints. The system optimized the placement of building blocks to improve the quality of the spin. No symmetries were enforced but the optimization process discovered a symmetrical geometry nevertheless.

	Input		Optimization					Output		
Model	Constraints	Op. Types	Gen. Op. [ms]	Fitness eval. [ms]	Reproduction [ms]	Generations	Total [s]	Frames	Geom. Obj.	Geom. Gen. Op.
Chairs (Fig 1)	6	3-5	72.97	27.15	29.08	82.25	24.23	100	22	7
Tree sketch (Fig 10)	1	1	134.75	678.26	138.35	2000	1747	494	495	164
User generated lamps (Fig 11)	2	2-4	4.80	133.81	38.12	183	37.72	13	9	6
Armadillo supports (Fig 12)	4	1	248.03	728.6	195.23	207	224.67	220	429	47
Spinning objects (Fig 13)	2	1	9.74	1746.3	21.51	402	2216.02	533	212	38
Terrains (Fig 14)	1	1	248.3	0.13	14.6	169	111.5	151	86	16
Hanger (Fig 15)	7	1	5.75	27.43	17.67	162	41.17	81	80	14

TABLE 1
Statistics for the generated examples. *Input* includes number of applied constraints, and number of different types of building blocks. *Optimization* shows timing per each part: Gen. Op is the time for executing the PICO-Graph and generating geometry, Fitness evaluation, Reproduction, number of generations, and average time per iteration. The *Output* shows the number of coordinate frames passed through the graph, final geometric objects and the number of used geometry generating operations that represent the final model.

procedural graph representation. The optimization allows for interactive guidance of the procedural model, but also for offline generation of complex geometry.

We have shown PICO on a variety of examples including procedural trees, automatically generated chairs, generation of supports for 3D printing, spinning objects, and even terrains. We believe that the flexibility and generality of the PICO system makes it a very powerful modeling tool for a wide range of applications. We have also evaluated PICO by comparing to the state-of-the art algorithms. Contrary to the existing approaches, PICO can generate existing models without the need of hand writing the underlying procedural model that is generated automatically by evolution.

Our work has a number of notable limitations. If the procedural evolution discovers an interesting pattern, it can be forgotten in next iterations or modified because of the mutations. It would be interesting to evaluate the time each structure stays in the iteration and its effect on the overall fitness. The objective function includes various criteria that can compete with each other and this can lead to a poor convergence rate in specific cases. In most cases however, PICO finds a solution very quickly that follows the user's intuition. Thanks to the interactive generation, PICO can produce results quickly and does not require any knowledge of procedural modeling as suggested by our user study. Although the constraints provide good control over the generated structure, it is not always entirely clear what the result will be. This is one of the main problems of procedural modeling and we bring a partial solution by using stochastic evolutionary algorithm with high level user guidance. Exploring finer levels of control would be beneficial. Furthermore, the space of possible solutions is non-trivially dependent on the geometry generating operations and the set of constraints. We found that for interactive generation, there has to be balance between too few and too many degrees of freedom (e.g., number of node types and node parameters). If there are too few, the optimization fails to find a path to an acceptable solution, while too many cause the optimization to no longer be interactive and to generate unintended results. Finally, the soft constraints have to be defined over the entire spatial domain for the optimization to move toward an optimal solution for every possible generated geometry.

Future work. In this work we have demonstrated PICO working with a specific set of constraints and a small set of building blocks. We think there is a potential in exploring

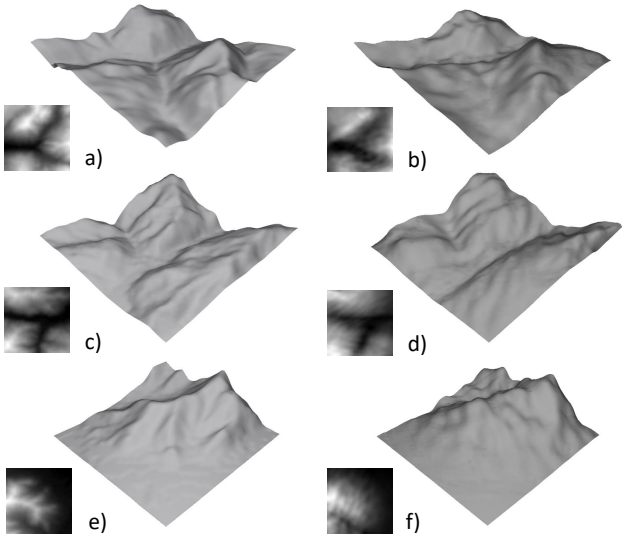

Fig. 14. **Terrains**. PICO matched the real terrain from the left by a set of Gaussians (right).

this direction further and adapting PICO to even more domains. It would be interesting to conduct additional studies with both artists and designers to better understand workflow patterns that can enable further system refinements. While our current optimization process is efficient, we believe there still exists opportunity to improve the convergence and responsiveness of our method. This includes not only the raw performance of our system, but its ability to find high quality solutions in the large search space.

ACKNOWLEDGMENTS

This research was funded in part by National Science Foundation grant #10001387, *Functional Proceduralization of 3D Geometric Models*.

REFERENCES

[1] G. W. Flake, *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*, ser. A Bradford book. MIT Press, 1998.
[2] M. Mitchell, *Complexity: a guided tour*. Oxford [England] ; New York: Oxford University Press, 2009.
[3] G. Nishida, I. Garcia-Dorado, D. G. Aliaga, B. Benes, and A. Bousseau, "Interactive sketching of urban procedural models," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 130:1–130:11, 2016.
[4] H. Huang, E. Kalogerakis, E. Yumer, and R. Mech, "Shape synthesis from sketches via procedural models and convolutional networks," *IEEE TVCG*, vol. 23, no. 8, pp. 2003–2013, 2017.
[5] ESRI, "City engine," http://www.esri.com/software/cityengine, 2019.
[6] SideFx, "Houdini," https://www.sidefx.com/products/houdini, 2019.
[7] A. Fournier, D. Fussell, and L. Carpenter, "Computer rendering of stochastic models," *Commun. ACM*, vol. 25, no. 6, pp. 371–384, Jun. 1982.
[8] E. Galin, E. Guérin, A. Peytavie, G. Cordonnier, M.-P. Cani, B. Benes, and J. Gain, "A Review of Digital Terrain Modeling," *Comp. Gr. Forum*, vol. 38, no. 2, 2019.
[9] M. Aono and T. L. Kunii, "Botanical tree image generation," *IEEE Computer Graphics and Applications*, vol. 4(5), pp. 10–34, 1984.
[10] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," in *Segmentation of Buildings for 3D Generalisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*, 1971.
[11] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 669–677, 2003.
[12] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 614–623, 2006.
[13] M. Schwarz and P. Müller, "Advanced procedural modeling of architecture," *ACM Trans. on Graph.*, vol. 34, no. 4 (Proceedings of SIGGRAPH 2015), pp. 107:1–107:12, 2015.
[14] P. Merrell and D. Manocha, "Model synthesis: A general procedural modeling algorithm," *IEEE TVCG*, vol. 17, no. 6, pp. 715–728, 2011.
[15] M. Natali, E. M. Lidal, J. Parulek, I. Viola, and D. Patel, "Modeling terrains and subsurface geology." in *Eurographics (STARs)*, 2013, pp. 155–173.
[16] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds," *Comp. Gr. Forum*, vol. 33, no. 6, pp. 31–50, 2014.
[17] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.
[18] D. G. Aliaga, I. Demir, B. Benes, and M. Wand, "Inverse procedural modeling of 3d models for virtual worlds," in *ACM SIGGRAPH 2016 Courses*, ser. SIGGRAPH '16. New York, NY, USA: ACM, 2016, pp. 16:1–16:316.
[19] T. Ijiri, S. Owada, and T. Igarashi, *The Sketch L-System: Global Control of Tree Modeling Using Free-Form Strokes*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 138–146.
[20] 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. 1–10, 2009.
[21] N. J. Mitra and M. Pauly, "Shadow art," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 156:1–156:7, 2009.
[22] B. Benes, O. Šťava, R. Měch, and G. Miller, "Guided procedural modeling," *Comp. Gr. Forum*, vol. 21, no. 3, pp. 325–334, September 2011.
[23] L. Krecklau and L. Kobbelt, "Procedural Modeling of Interconnected Structures," *Comp. Gr. Forum*, 2011.
[24] S. Garcia and L. Romão, *A Design Tool for Generic Multipurpose Chair Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 600–619.
[25] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352 – 363, 2011.
[26] P. Guerrero, S. Jeschke, M. Wimmer, and P. Wonka, "Learning shape placements by example," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 108:1–108:13, 2015.
[27] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, "Controlling procedural modeling programs with stochastically-ordered sequential monte carlo," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 105:1–105:11, 2015.
[28] D. Ritchie, A. Thomas, P. Hanrahan, and N. Goodman, "Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks," in *Advances in Neural Information Processing Systems 29*, 2016, pp. 622–630.
[29] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum, "Learning to infer graphics programs from hand-drawn images," in *NIPS 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 6060–6069.
[30] G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, and S. Maji, "Csgnet: Neural shape parser for constructive solid geometry," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
[31] T. Du, J. P. Inala, Y. Pu, A. Spielberg, A. Schulz, D. Rus, A. Solar-Lezama, and W. Matusik, "Inversecsg: Automatic conversion of 3d models to csg trees," *ACM Trans. Graph.*, vol. 37, no. 6, pp. 213:1–213:16, Dec. 2018.
[32] J. O. Talton, D. Gibson, L. Yang, P. Hanrahan, and V. Koltun, "Exploratory modeling with collaborative design spaces," in *Proceedings of the 2nd Annual ACM SIGGRAPH Conference and Exhibition in Asia*. ACM Press, 2009.
[33] K. Sims, "Evolving 3d morphology and behavior by competition," *Artificial life*, vol. 1, no. 4, pp. 353–372, 1994.
[34] G. S. Hornby and J. B. Pollack, "The advantages of generative grammatical encodings for physical design," in *Proceed-*

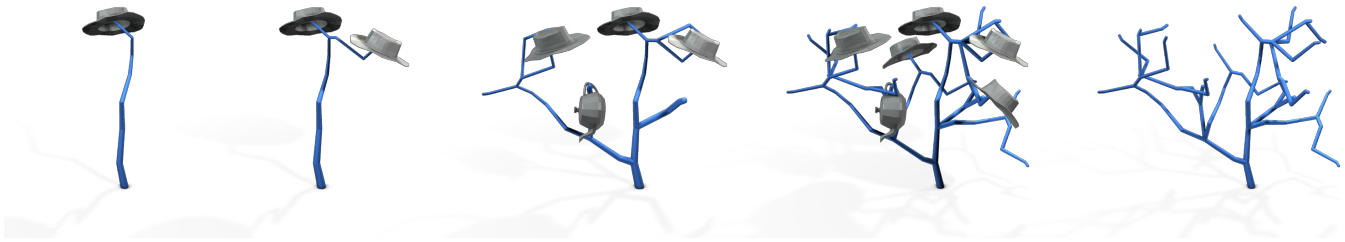

Fig. 15. A procedural hat hanger is automatically expanded every time a new hat is added.

ings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), vol. 1, 2001, pp. 600–607 vol. 1.

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

[36] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun, "Interactive furniture layout using interior design guidelines," *ACM Trans. Graph.*, vol. 30, no. 4, pp. 87:1–87:10, 2011.

[37] Y.-T. Yeh, L. Yang, M. Watson, N. D. Goodman, and P. Hanrahan, "Synthesizing open worlds with constraints using locally annealed reversible jump mcmc," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 56:1–56:11, 2012.

[38] L.-F. Yu, S.-K. Yeung, C.-K. Tang, D. Terzopoulos, T. F. Chan, and S. J. Osher, "Make it home: Automatic optimization of furniture arrangement," *ACM Trans. Graph.*, vol. 30, no. 4, pp. 86:1–86:12, Jul. 2011. [Online]. Available: http://doi.acm.org/10.1145/2010324.1964981

[39] C.-H. Peng, Y.-L. Yang, F. Bao, D. Fink, D.-M. Yan, P. Wonka, and N. J. Mitra, "Computational network design from functional specifications," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 131:1–131:12, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2897824.2925935

[40] A. Emilien, U. Vimont, M.-P. Cani, P. Poulin, and B. Benes, "Worldbrush: Interactive example-based synthesis of procedural virtual worlds," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 106:1–106:11, Jul. 2015.

[41] E. Whiting, J. Ochsendorf, and F. Durand, "Procedural modeling of structurally-sound masonry buildings," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 112:1–112:9, 2009.

[42] R. Měch and G. Miller, "The *Deco* framework for interactive procedural modeling," *Journal of Computer Graphics Techniques (JCGT)*, vol. 1, no. 1, pp. 43–99, Dec 2012.

[43] J. McDermott, *Graph Grammars as a Representation for Interactive Evolutionary 3D Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 199–210.

[44] S. Bergen and B. J. Ross, "Aesthetic 3d model evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 3, pp. 339–367, 2013.

[45] K. Xu, H. Zhang, D. Cohen-Or, and B. Chen, "Fit and diverse: Set evolution for inspiring 3d shape galleries," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 57:1–57:10, Jul. 2012.

[46] K. Haubenwallner, H.-P. Seidel, and M. Steinberger, "Shapegenetics: Using genetic algorithms for procedural modeling," *Comp. Gr. Forum*, vol. 36, no. 2, pp. 213–223, 2017.

[47] C. Jacob, "Genetic l-system programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 333–343.

[48] C. Jacob', "Genetic l-system programming: Breeding and evolving artificial flowers with mathematica," in *IMS'95 - First International Mathematica Symposium*, 06 1996, pp. 215–222.

[49] W. W. Wadge and E. A. Ashcroft, *LUCID, the Dataflow Programming Language*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[50] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[51] A. Lindenmayer, "Mathematical models for cellular interaction in development," *Journal of Theoretical Biology*, vol. Parts I and II, no. 18, pp. 280–315, 1968.

[52] P. Prusinkiewicz, "Graphical applications of l-systems," in *Proceedings on Graphics Interface '86/Vision Interface '86*, 1986, pp. 247–253.

[53] J. McCormack, *Aesthetic Evolution of L-Systems Revisited*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 477–488.

[54] L. Velho, "Stellar subdivision grammars," in *SGP '03*. Eurographics Association, 2003, pp. 188–199.

[55] C. Smith and P. Prusinkiewicz, "Simulation modeling of growing tissues," in *Proceedings of the 4th International Workshop on Functional-Structural Plant Models,*, 2004, pp. 365–370.

[56] P. Boechat, M. Dokter, M. Kenzel, H.-P. Seidel, D. Schmalstieg, and M. Steinberger, "Representing and scheduling procedural generation using operator graphs," *ACM Trans. Graph.*, vol. 35, no. 6, pp. 183:1–183:12, 2016.

[57] R. McNeel and Associates, "Grasshopper," https://www.grasshopper3d.com, 2019.

[58] Adobe, "Substance designer," https://www.substance3d.com, 2019.

[59] M. Bächer, E. Whiting, B. Bickel, and O. Sorkine-Hornung, "Spin-it: Optimizing moment of inertia for spinnable objects," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 96:1–96:10, 2014.

[60] J. Kessenich, D. Baldwin, and R. Rost, *The OpenGL Shading Language*, Khronos Group, 2014, rev. 9.

[61] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[62] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[63] P. J. Bentley and J. P. Wakefield, "Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms," in *Soft computing in engineering design and manufacturing*. Springer, 1998, pp. 231–240.

[64] L. Davis, *Handbook of genetic algorithms*. Cumincad, 1991.

[65] R. L. Haupt and S. E. Haupt, *Practical genetic algorithms*. John Wiley & Sons, 2004.

[66] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *Journal of the ACM (JACM)*, vol. 17, no. 4, pp. 715–728, 1970.

[67] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, 1996.

[68] J. Vanek, J. A. G. Galicia, and B. Benes, "Clever support: Efficient support structure generation for digital fabrication," *Comp. Gr. Forum*, vol. 33, no. 5, pp. 117–125, 2014.

[69] A. Doucet, S. Godsill, and C. Andrieu, "On sequential monte carlo sampling methods for bayesian filtering," *Statistics and computing*, vol. 10, no. 3, pp. 197–208, 2000.

[70] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su *et al.*, "Shapenet: An information-rich 3d model repository," *arXiv preprint arXiv:1512.03012*, 2015.

Vojtěch Krs is a Research Engineer at Adobe Research. He received his B.S. in Software Engineering from Czech Technical University in Prague in 2014 and his Ph.D. in Computer Graphics from Purdue University in 2019. His research interests include geometrical and procedural 3D modeling, simulation of natural phenomena and human-computer interaction.

Radomír Měch leads a Procedural Imaging Group at Adobe Research. The researchers span areas of 2D and 3D design, modeling, HCI, machine learning for image processing, inverse rendering, and object acquisition. His areas of research are procedural modeling, with a particular focus on interaction with procedural models and casual modeling, 3D printing, and imaging algorithms.

Mathieu Gaillard is a PhD student at Purdue University focusing on geometric and procedural modeling as well as simulation of natural phenomena. He received an Engineer's degree from INSA Lyon and a master's degree from the University of Passau.

Nathan Carr is a director of Adobe Research in San Jose California whose lab focuses on computer graphics and vision. His research spans the field of both 2D and 3D geometric modeling as well as rendering.

Bedrich Benes is George McNelly professor of Technology and 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 140 research papers in the field.