Pacific Graphics 2023
R. Chaine, Z. Deng, and M. H. Kim
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 7

Data-guided Authoring of Procedural Models of Shapes

Ishtiaque Hossain[1] I-Chao Shen[2] Takeo Igarashi[2] Oliver van Kaick[1]

[1] Carleton University, Canada
[2] The University of Tokyo, Japan

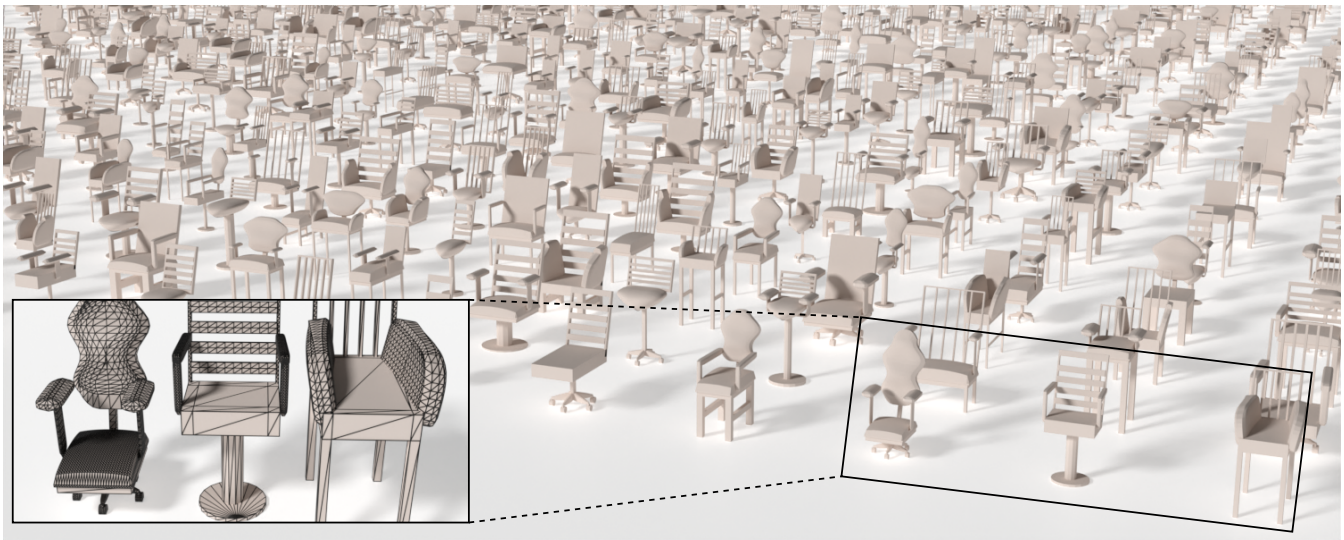

Figure 1: *A large collection of shapes created by a procedural model developed with our data-guided method. The inset figure shows the clean structure of the procedurally-created shapes, i.e., no self-intersections or defects are present in the shapes.*

Abstract

Procedural models enable the generation of a large amount of diverse shapes by varying the parameters of the model. However, writing a procedural model for replicating a collection of reference shapes is difficult, requiring much inspection of the original and replicated shapes during the development of the model. In this paper, we introduce a data-guided method for aiding a programmer in creating a procedural model to replicate a collection of reference shapes. The user starts by writing an initial procedural model, and the system automatically predicts the model parameters for reference shapes, also grouping shapes by how well they are approximated by the current procedural model. The user can then update the procedural model based on the given feedback and iterate the process. Our system thus automates the tedious process of discovering the parameters that replicate reference shapes, allowing the programmer to focus on designing the high-level rules that generate the shapes. We demonstrate through qualitative examples and a user study that our method is able to speed up the development time for creating procedural models of 2D and 3D man-made shapes.

CCS Concepts

• ***Computing methodologies*** → ***Computer graphics; Shape modeling;***

1. Introduction

3D content is important in a variety of areas, including animation, games, simulation, and virtual worlds. However, creating 3D shapes is a difficult task and one of the main barriers for the generation of large volumes of diverse 3D content. The manual approach where a person models a shape through an interface is time-consuming and requires skilled artists. Recently, much interest has been devoted to developing deep neural networks that auto-

matically synthesize 3D objects. Networks that represent shapes as implicit functions are able to learn models for collections of shapes [PFS*19, CTZ20], but the quality of the results is not optimal; the created shapes are not clean, having unnecessary bumps and dents, while lacking fine details. Recent methods based on transformers or diffusion models provide results that are visually smoother [HPG*22, YLM*22, ZVW*22, HLHF22], but depend on the availability of enough data resembling the target shapes.

In contrast, procedural generation is a powerful tool for automatically synthesizing a large amount of clean shapes. Procedural models have been proposed for a variety of shapes [STBB14], such as plants, architecture, and man-made objects. A procedural model is essentially a set of operations implemented as a computer program that can generate shapes from a set of input parameters. By varying the parameters, a large amount of shapes with a variety in fine details can be generated (Figure 1). And, by writing new rules, more diversity can be added to the structure of the shapes. In addition, the output shapes can be programmed to be *clean*, i.e., devoid of self-intersections and artifacts, and having a complexity that matches the amount of details present on the shapes. The generated shapes can also be further annotated with information like semantic labels, and creating a procedural model does not require the availability of a large training set of shapes. The main downside of procedural generation is that writing procedural models is difficult. The programs have to be manually written by experienced programmers, requiring significant time.

To reduce the time invested in the creation of procedural models, some work in the literature has investigated the automatic prediction of model parameters or rules. For example, earlier methods for inverse procedural modeling infer the procedural rules of a set of shapes based on geometric analysis, such as symmetry [BWS10, SPK*14]. Some approaches assume that a procedural model is given and predict the parameters of instances, e.g., from sketches [HKYM17]. Other approaches allow for shape editing based on discovered editing handles or directions [GGC*20, ZAESB20, WSS*20]. Differentiable rendering has been applied for model fitting [LLCL19], and networks have been proposed for learning a space of programs that generate shapes [JBX*20, JCG*21]. However, in most approaches, the user does not have direct control over the complexity of the rules and representations that are generated, possibly leading to highly-complex and redundant models which are difficult to be further edited by a user.

In this paper, we introduce a data-guided method for facilitating manual creation of procedural models. In our problem setting, a programmer is given a large set of shapes and is tasked with writing a procedural model to replicate the shapes in the set. We focus on this problem setting since a reference input set of shapes can be a good communication tool between artists (or directors) and programmers when crafting the procedural model. Artists can create reference shapes using arbitrary modeling tools, while directors can collect reference models from online repositories. The programmers then generalize the reference shapes by writing code. The reference shapes can also be scanned shapes that are not clean, i.e., not directly usable in products such as movies and games. Developing procedural models to replicate shapes is often a good way to improve the quality of references shapes. In addition, the reference collections do not have to be as large as the typical sets required for training neural networks for 3D synthesis.

We assume that the programmer is familiar with the procedural modeling language and has enough experience to create new shape generation rules. The input shapes may have been obtained from on-line repositories and thus may be non-manifold triangle soups. Hence, the goal is to create a set of rules that can recreate the given shapes with proper triangle meshes while allowing for further generation of additional shapes by varying the model parameters.

Our method works in an iterative fashion (Figure 2). The user starts by writing an initial set of rules, and the method automatically predicts the rule parameters for the shapes in the given set, also highlighting shapes that are not approximated well by the procedural model. The computation takes a few minutes to complete. The user can then add additional rules based on the given feedback and iterate the process. Thus, our method is designed to best combine the strengths of human and computer intelligence. Given a set of shapes, humans can easily derive high-level rules to roughly approximate an object. However, setting the parameters to exactly replicate the shapes can be tedious and labor-intensive. Thus, for this task, we employ a learning-based approach to automatically predict shape parameters.

Our method can be seen as an active learning tool where the existing rules are applied to all the shapes that can be approximated well, while only the shapes that are not replicated well are brought to attention to the user for inspection. In contrast to the existing methods, our method allows the user to maintain full control over the creation of the procedural model, while saving development time by automating the tedious steps of the process. In principle, our method is not limited to a specific representation of shapes or by the syntax used for developing the procedural model. Also, our method does not necessarily apply to only one specific type of shapes such as organic or architectural shapes, although our experiments focus on the generation of man-made shapes. Our method enables a generalized workflow that allows the user to decide the details of the procedural model for particular use cases.

We demonstrate our method for the generation of man-made shapes that can compose indoor environments. Our main results include the development of procedural models for four classes of 3D shapes. We evaluate our system with a user study for the development of a procedural model of 2D shapes, which demonstrates the advantages of our method over manual creation, and perform ablation studies to justify the components of our method.

2. Related work

Forward and inverse procedural modeling. Procedural modeling systems have been designed for generating a variety of geometry, such as organic shapes and man-made shapes [STBB14]. Different strategies can be used to guide the generation of shapes with a procedural model, such as the use of geometric constraints [MM11, KMG*21] or high-level specifications [TLL*11]. On the other hand, only a few systems have been proposed for facilitating the creation of the procedural model itself: Lipp et al. [LWW08] introduce a system for visual editing of procedural

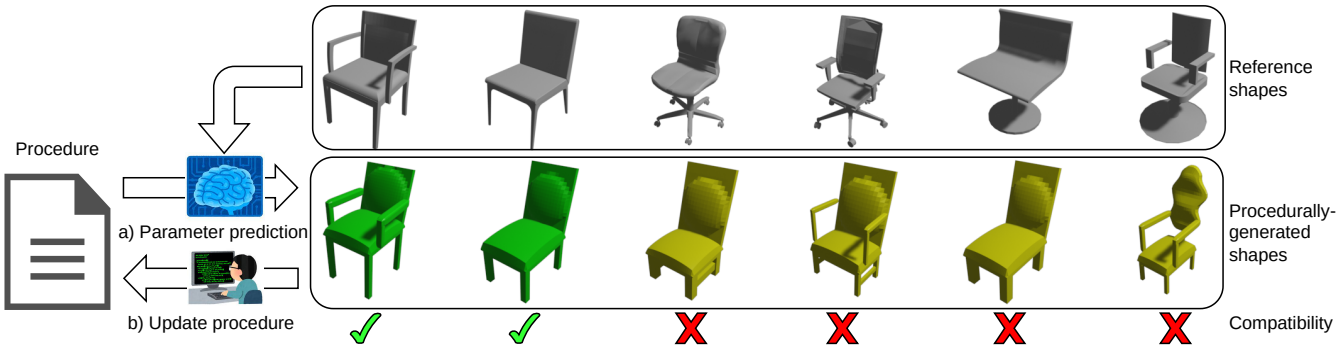

Figure 2: *Overview of our data-guided authoring of procedural models. (a) The system predicts procedural model parameters for reference shapes, and estimates the compatibility between procedurally-generated shapes and reference shapes. (b) The user manually updates the procedure to improve shape compatibility.*

rules, while Patow [Pat12] proposes the use of a graph-based visual interface for writing model rules. However, these tools focus mainly on visual representations of the rules, rather than on providing feedback during the creation of the model. On the other hand, our method is not tied to the specific form used for the procedural model. Instead, the user can decide the implementation details of the procedural model.

Some recent works [HLHF22,ZVW*22,LMW*22] enable shape generation and its counterpart problem *shape inversion* using latent representations of shapes. However, this approach is applicable in specific use cases such as shape editing, where the procedural model is hidden from the user. The latent space is difficult to explain and since the user does not have direct access to the procedural model, extending it by addition of new rules is not possible. In comparison, our method does not map shapes onto a latent space. Shapes are replicated by first predicting their parameters and then invoking the procedure.

There has also been considerable work on *inverse procedural modeling*. Some methods assume that a procedural model is given and infer the parameters for generating a given shape. For example, Stava et al. [SPK*14] identify parameters that replicate organic trees based on optimization with a tree similarity measure, Nishida et al. [NGDA*16] identify the grammar snippets that recreate a user sketch of a building, and Huang et al. [HKYM17] infer the parameters for generating a shape that resembles an input drawing. Moreover, Wei et al. [WSS*20] learn the parameters of a template that approximates a given shape, Gaillard et al. [GKG*22] introduce a system that identifies the procedural parameters of a shape based on user edits, while Sharma et al. [SGL*22] predict the sequence of constructive solid geometry statements that replicate an input shape. There have also been methods for finding good designs based on parametric models [KG01, TGY*09, KSI14], or exploring the space of shapes generated by an existing procedural model [YAMK15]. However, these methods assume that a model is already given and do not aid in the incremental design of the procedural model as our method does.

Some recent studies focus on learning inverse procedural models in the 2D domain. Hu et al. [HDR19] study graph-based procedural generation of materials and proposed a framework that automatically selects graphs from a predefined database to create procedural materials from images. Shi et al. [SLH*20] propose a method that optimizes the parameters used by graph nodes when generating materials from images. Hu et al. [HGH*22] extends this idea by proposing a technique that involves differentiable proxies for graph nodes that are otherwise not differentiable. In contrast, our workflow can be applied to both 2D images and 3D shapes. We show examples of the 2D case in a user study in Section 5 and present results for the 3D case in Section 4.

Another group of work investigates prediction of parameters for procedural models as a means to enable shape editing. Michel et al. [MB21] propose to represent shapes as directed acyclic graphs and edit shapes via amendments made to the graphs. Mathur et al. [MPZ20] introduces an approach that uses a decision-tree based optimization to edit parametric CAD models. Cascaval et al. [CSQ*22] put forth a CAD-based Domain Specific Language which allows for bidirectional editing. Hertz et al. [HPG*22] and Yan et al. [YLM*22] investigate shape editing and shape completion, respectively, using transformer-based approaches. The primary focus of these methods is to enable exploration of shapes via edit operations, while the underlying procedural models are static. In contrast, our method allows a user to dynamitcally create a procedural model.

Only a few methods seek to infer both the rules of a procedural model and parameters for generating a given shape. Some of the earlier studies attempt solving this problem by formulating a hierarchical grammar of rules that define a shape's geometry. For instance, Wu et al. [WYD*14] utilize a Split Grammar to define façade layouts procedurally and determine the least-cost grammar for given façades in an iterative way. Demir et al. [DAB14] define their grammar as various transformations applied to tree nodes of basic components, in order to procedurally model cityscapes. In a subsequent study [DAB16], they adapt a Split Grammar to represent 3D architectural models. Lipp et al. [LSL*19] also use a Split Grammar to create procedural approximations of buildings, and add the ability of local edits. Alternatively, Bokeloh et al. [BWS10] generate a procedural model of a shape based on an analysis of the shape's symmetry, Nishida et al. [NBA18] infer the rules for gener-

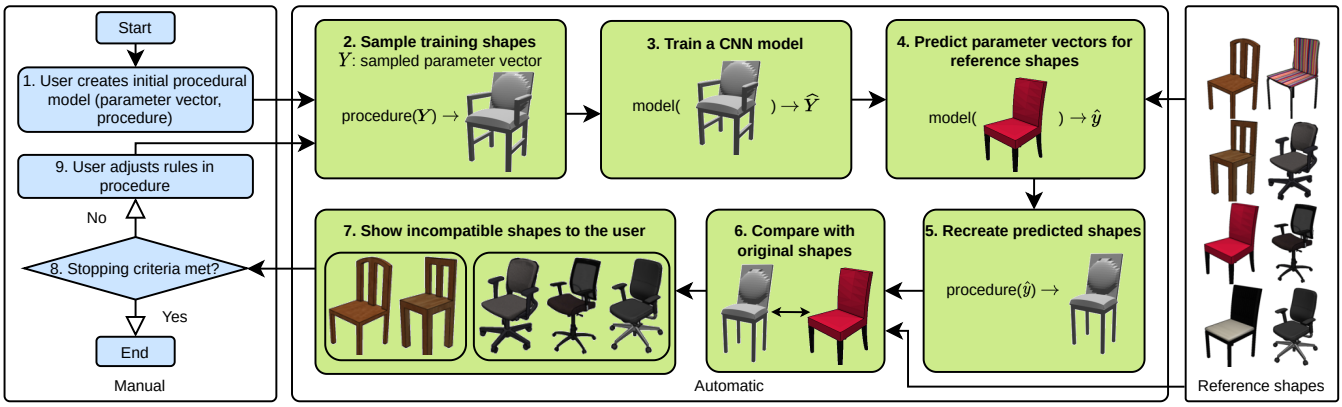

Figure 3: *Workflow and details of our procedural model authoring method, showing the method's manual and automatic learning-based steps.*

ating buildings and their façades from a photograph, while Guérin et al. [GDGP16] introduce a system for inverse procedural modeling of terrains based on a sparse representation. Using generative models to learn inverse procedural models has been investigated by researchers as well [TLS*19, JWR22]. Despite the advances on this front, inverse procedural modeling remains practical mainly for more constrained domains such as buildings and terrains, and is difficult to apply for shapes in general. Since our method does not restrict the user to a particular domain of shapes or a specific grammar, our method can be applied to a wider range of use cases, although the user is responsible for creating the rules of the model.

Recently, Jones et al. [JBX*20] introduced a method that learns a generative model of programs. Although this generative model can be used to automatically predict a program for an input shape, to learn the model, the method requires a training set of existing programs or shapes hierarchically organized into parts, while the feature set of the programming language is also limited. In contrast, our method addresses a different problem statement, where our goal is to aid the user in freely designing a procedural model.

3. Data-guided authoring of procedural models

Problem setting. Given an input collection of reference shapes, the goal of our method is to help an expert programmer in creating a procedural model that replicates the given shapes. The procedural model takes as input a vector that collects a set of multiple parameters, and outputs a shape corresponding to the parameters. The procedural model can then be used to generate a large number of shapes by varying the parameters. In this paper, we experiment with procedural models that generate polygonal mesh models. However, the basic framework is applicable to other shape representations.

The process for developing the procedural model can be decomposed into a workflow with the following three steps:

1. Define a vector of input parameters;
2. Define a procedure that generates shapes based on an input parameter vector;
3. Find the parameter values to replicate each reference shape.

While humans are proficient at the programming part of the workflow, i.e., examining a set of shapes and writing a procedure for reproducing the shapes, estimating specific parameters to replicate given shapes is tedious and time consuming. The latter task becomes impractical when a large number of shapes need to be replicated by means of trial and error. An additional challenge in this scenario is to identify shapes from the large collection of reference shapes that cannot be replicated by the current procedural model, unless the model is amended to replicate these shapes as well. This task is also labour-intensive for humans, but crucial for improving the procedural model. In this work, we introduce an iterative method to make this process more efficient. The user is responsible for the first two steps (programming), while we delegate the third step (parameter prediction) entirely to an automatic method. Moreover, the method also gives feedback to the user to provide assistance with the first step (programming).

Method overview. Figure 2 gives an overview of our procedural model authoring method. The process starts with an initial procedure/program crafted by the user. Next, the method predicts parameter values for all the shapes in the reference collection and procedurally generates shapes corresponding to the reference shapes. Then, the method identifies which shapes from the collection can be approximated well with the current procedural model and which shapes not. For the sake of brevity, throughout the rest of the paper, we will refer to the well-approximated shapes as *compatible shapes* and the shapes that are not well-approximated as *incompatible shapes*. If the input collection is large, inspecting the results of the procedural model for the entire collection can be a laborious and error prone task. Thus, we delegate this task to our method.

The method measures how closely each replicated shape resembles the corresponding shape in the collection, presenting the incompatible shapes to the user. Specifically, the method groups similar shapes together, so that the user mainly has to inspect a representative of each group rather than all of the incompatible shapes. By inspecting these shapes, the user can then adjust the procedural model accordingly. Once the user made changes to the procedural model to better approximate additional shapes, the entire process is

repeated. The method continues iteratively until a replication accuracy is achieved for all the shapes in the collection.

Figure 3 illustrates the authoring method in more detail. We use a learning-based method to predict the parameter vectors of unknown shapes. Thus, to create the training data for this purpose, the method obtains multiple random samples of the parameter vector and generates the corresponding sample shapes. This data composed of pairs of shapes and their parameter vectors is used for training a neural network that predicts the parameter vectors of reference shapes. After training, the method predicts the parameters of the unknown shapes, generates the corresponding procedural shapes, and then compares the generated shapes with the reference ones. If the replicated shapes are dissimilar to their references, the automatic method flags them as incompatible and presents them to the user, at which point the user can inspect these shapes and decide how to improve the procedural model further.

A possible alternative approach for predicting parameter vectors is to use search/optimization methods that do not involve learning. However, these methods typically require multiple evaluations of the function being optimized, which in our context is the invocation of the procedural model. Thus, the execution time for predicting the parameters of a shape can easily become prohibitive when using complex procedural models. In contrast, our method requires the generation of a training set with a pre-defined size and training of a neural network, but inference times are fast once the network has been trained. Thus, our method is faster than optimization-based methods in situations where parameters need to be estimated for a large number of reference shapes.

In the next sections, we provide details on the procedural model created by the programmer and discuss the parameter prediction, shape comparison, and feedback methods in more detail.

3.1. Procedural Model

We test our method with the task of writing procedural models of 2D and 3D man-made shapes such as tables and chairs. In both cases, the model has a similar structure: the model encodes a set of rules which define the grammar for one category of shapes. In 3D, the rules operate with primitives such as 3D boxes and surface patches. To generate a shape, the rules start by recursively splitting an initial 3D box to roughly represent the finer structure of the target shape. The finer boxes are then transformed into surface patches to generate a triangle mesh for the shape. Deformation is applied to surface patches when curved geometry is desired. In this process, a set of tags are assigned to the boxes or patches, where the tags dictate which rules should be applied to the primitives. In 2D, for simplicity, the program draws primitive shapes such as boxes and circles directly in their final position.

In the procedural models, the parameter vector is allowed to have a mix of four different kinds of parameters. A parameter can be a *continuous scalar*, e.g., to encode the height of a shelf. A parameter can be an *integer value*, e.g., to indicate how many rows a shelf has. A parameter can also be an *item from a set*, e.g., to indicate what type of legs a shelf has, such as column-like legs or legs spanning an entire side of the furniture. Lastly, a parameter can be a *binary value*, e.g., to define whether a shelf has a back surface closing the furniture or not.

Since the procedural model used in our experiments is fairly complex and is not suitable for illustration within limited space, we also present an experiment with a simpler procedural model in the supplementary material, showing the code and results of the model for each iteration of our method. We note that our method can be used with these different procedural models without requiring any modifications.

3.2. Parameter Prediction

We use a supervised learning approach to predict the parameters of shapes in a collection. Thus, we require training data in the form of shapes and their corresponding parameter values, which can be automatically generated from the procedural model. Then, we train a neural network to learn the parameter prediction. Note that, since there can be a non-linear relationship between parameters and the output of a procedural model, there are no guarantees that a learning-based method will capture all of these relationships. However, with sufficient training data, learning-based methods have been shown to perform reasonably well for difficult regression problems. We discuss the data generation and neural network model as follows.

Training data generation. Once the user defined the entries of the parameter vector and a procedure, we can generate a dataset of shapes with their corresponding parameter vectors by randomly sampling the parameter vector and invoking the procedure with the sampled vectors. For instance, if the parameter vector has 3 elements and they are sampled into 4, 6 and 5 steps, then we take random samples from $4 \times 6 \times 5 = 120$ parameter vectors in total. For scalar elements in the parameter vector, we find that it is best if the user defines the number of steps. For example, the user can decide to sample a "bed width" parameter according to semantically-meaningful values, e.g., twin, double, and queen size. Integer elements are sampled into $M - m + 1$ steps where M and m are the maximum and minimum possible value of the element, respectively. The binary elements are sampled into two steps. With the procedural model used in this study, it takes approximately 90 ms on average to generate a shape.

Neural network architecture. The training dataset is a collection of shapes with their corresponding parameter vectors. Thus, we can pose the prediction of a parameter vector for a shape as a regression task learned from the generated dataset, and use a neural network to learn the regression function. The shapes created by the procedural model are represented as triangle meshes, while the shapes from the reference collection can be represented in any desired format. Thus, in order to accept shapes with multiple formats as input, we use a more general prediction approach based on rendered images.

Specifically, we render the triangle meshes into multiple images, and then the images are processed through a convolutional neural network to predict parameter vectors. In our method, multiple cameras render a given shape into a stack of images, where the positions of the cameras are also learnable parameters optimized

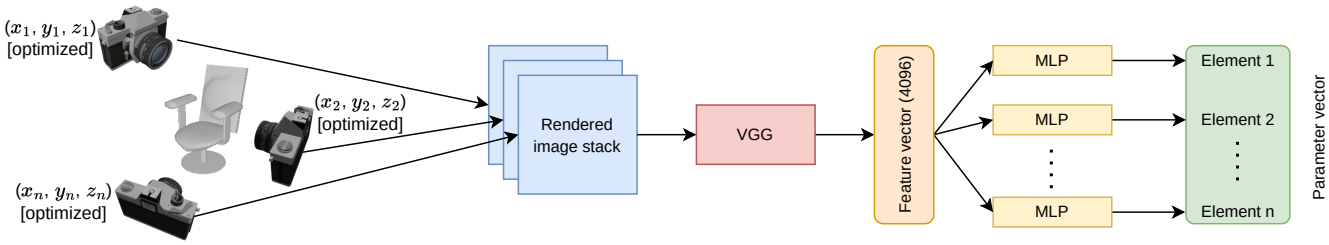

Figure 4: *Model architecture for parameter prediction, where the camera positions are also optimized during training.*

during training. Optimizing the camera positions allows us to select the viewpoints that potentially provide the best prediction accuracy. For optimizing the camera positions during training, we used the scheme proposed by Loper et al. and subsequent studies [LB14, LLCL19, RRN*20].

Figure 4 shows the architecture of the prediction network. We use VGG19 [SZ15] for extracting features from the data. Although the VGG19 model has been superseded by other architectures in certain domains, we find that in our setting it performs similarly as other models such as ResNet and MobileNet. Thus, we opt to use VGG19 due to its simplicity. Each parameter is predicted by an independent MLP, as it leads to a simpler MLP architecture for each parameter, as opposed to a single, more complex MLP that predicts all the parameters. Our chosen approach allows for fewer weights that need to be adjusted during training.

Loss function. We define the network's loss function as:

$$L = \sum_{s \in S} \sum_{i=1}^{n} L_i^s, \tag{1}$$

where L_i^s represents the loss for the i-th parameter of shape s, and n is the total number of parameters. L_i is an L1 loss for real-valued scalars, and a cross-entropy loss for integer and binary parameters.

Optimization. In order to minimize the loss, the Adam optimization algorithm is used with a learning rate of 10^{-5} and a weight decay of 10^{-8} for 50 epochs. Training is performed on an NVIDIA RTX 3090 GPU. In this specific configuration of hardware, the training process takes approximately 16 minutes to complete. Inference can then be completed in approximately 16 ms for each shape.

3.3. Shape Comparison

Our method compares shapes for two reasons: first, to determine if a given shape is compatible with the procedural model, and secondly, to collect shapes with similar characteristics into groups. Since the number of shapes in the collection can be large, we show to the user only representative shapes from different shape groups to avoid visual clutter. For compatibility testing, we predict parameters for unseen shapes using the neural network described in Section 3.2, replicate the shapes from the predicted parameters using the procedure, and finally measure how similar the shapes are to their corresponding references. For grouping incompatible shapes that look similar, we apply a clustering method and show to the user the shape from each cluster which is closest to the cluster centroid.

In both tasks, we measure shape similarity by calculating the distnace between the Light Field Descriptors (LFD) [CTSO03] computed for two shapes according to the L1 metric. We choose this descriptor instead of other metrics such as IoU or Chamfer distance since LFDs are more suited for quantifying visual similarity. The automatic method identifies shapes to be incompatible with the procedural model if the light field distances between the recreated shapes and their references fall below a predetermined threshold of 0.18. For grouping the incompatible shapes, we apply a hierarchical clustering algorithm where the maximum light field distance is used as the linkage criterion. Specifically, we use the Agglomerative Clustering algorithm available with the Python *scikit-learn* package [PVG*11]. We determine the number of clusters dynamically by normalizing all shape distances between $[0, 1]$ and using a threshold of 0.4 for hierarchical merging of clusters.

4. Evaluations

In this section, we discuss the experiments and studies we performed to evaluate our method and its components. Our implementation can be found at `https://github.com/ishtiaquehossain-sys/SimpleProcShapes`.

4.1. Datasets

We evaluate our method for the creation of procedural models on four categories of 3D shapes: bed, chair, shelf, and table. Training data for parameter prediction is generated by sampling parameter vectors for each category and using the procedure to create shapes based on the sampled vectors. The number of all possible samples can be high when the parameter vector has many elements. Thus, we limit the training data by randomly sampling $3,000$ vectors. The resulting dataset is composed of (shape, parameter vector) pairs.

We use a subset of the ShapeNet dataset [CFG*15] to compose our reference collections of 3D shapes. Note that the shapes from ShapeNet are not procedurally created and do not have a parameter vector associated with them. We included 200 examples for each of chair and table, and 20 examples for each of bed and shelf.

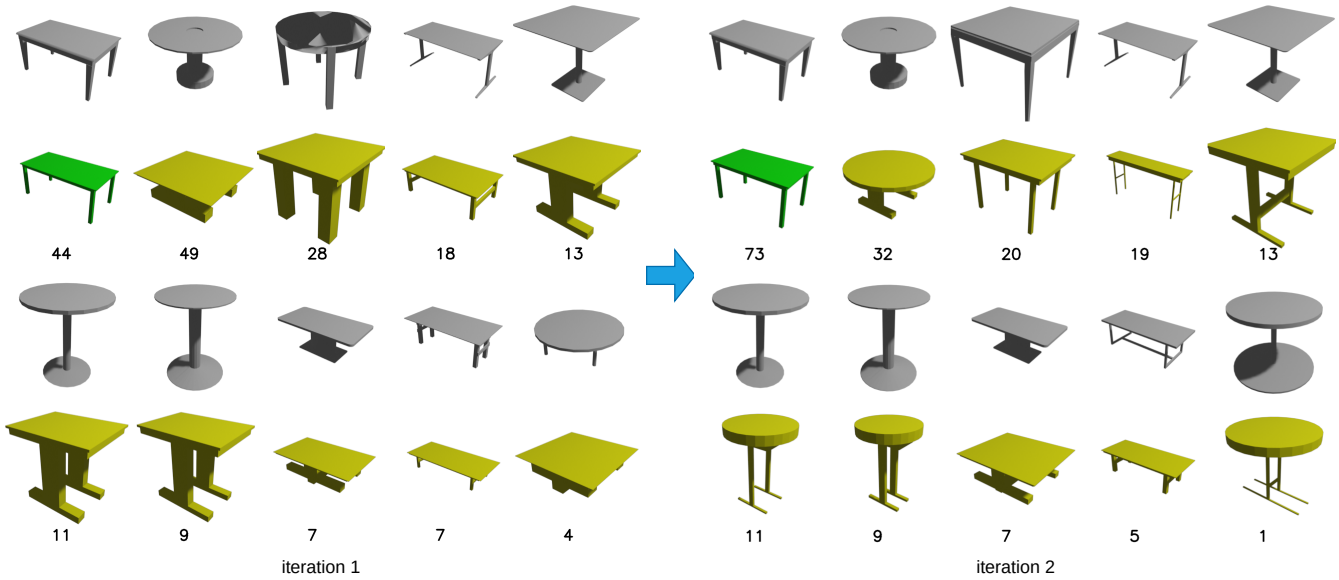

Figure 5: *Example of the output of the procedural model authoring system after parameter prediction and shape grouping in two iterations. Compatible shapes are green and incompatible shapes are yellow. Each colored shape is the representative of a group, where the number beneath denotes the quantity of additional shapes in the group.*

4.2. Qualitative Evaluation of Authoring System

Figure 6 demonstrates the data-guided workflow for authoring procedural models for the four classes of 3D shapes. We start with a basic procedural model and improve it over two iterations. The rows of shapes colored gray are reference shapes selected from different classes of ShapeNet. The following three rows show the same shapes represented by one of the four procedural models, starting from the results of the initial procedural model to the final model. The colors represent the level of visual similarity between the reference shapes and the shapes recreated by the procedural model, where the colormap ranges from red (most dissimilar) to green (most similar). It can be seen that the number of compatible shapes increases with each iteration. At the end of the iterative authoring, we obtain procedural models with 25, 35, 29, and 15 rules, for bed, chair, shelf, and table classes, respectively.

Figure 5 depicts two iterations of the the iterative authoring process in more detail for the table class. The reference shapes from ShapeNet are colored gray and the recreated compatible and incompatible shapes are colored green and yellow, respectively. Figure 5 also shows how the incompatible shapes are collected into groups of shapes that look similar. A representative shape from each group is shown, along with the number of additional shapes in each group.

We see that the number of compatible shapes increases in one iteration from 45 to 74. Thus, the interface groups all the compatible shapes into the cluster represented by the green shape, and shows new clusters of incompatible shapes. Specifically, in the first iteration, some of the groups show tables with round tops which the initial procedural model does not support. By adjusting the procedural model to support round tops, these shapes can be reproduced and the first iteration is complete. We see that improving the procedural model resulted in an increase of shapes that are compatible with the procedural model. Moreover, at the end of the first iteration, there are still groups of tables that have rounded bottoms that are not currently supported by the procedural model. Addition of this detail to the procedural model allows to reproduce these shapes and completes the second iteration.

We present a more complete example using a simpler procedural model and involving more iterations in the supplementary material.

4.3. Comparison with ShapeAssembly

Figure 7 shows a comparison of our method to the ShapeAssembly method of Jones et al. [JBX*20]. ShapeAssembly is able to automatically extract a set of rules that generate a given shape, after being trained with a related set of data. However, shape primitives are restricted to cuboids. In contrast, our semi-automatic method does not restrict the user in terms of syntax and semantics of the procedure. Thus, the user can also use more complex programming statements, and encode any type of shape primitive, including curved geometry. For the comparison, we predicted the parameters of selected ShapeNet objects and recreated them with our user-defined procedure. At the same time, using the ShapeAssembly point-cloud encoder, we converted the same shapes into latent vectors and then generated ShapeAssembly programs from the latent vectors, obtaining shapes from the programs using their pre-trained model.

It is worth mentioning here that the objective of this comparison is not to highlight limitations of ShapeAssembly in terms of visual details reproduced. Rather, the intention here is to delineate how the problems addressed by ShapeAssembly and our method are fundamentally different. Given a procedural model, ShapeAssembly can create a procedural replication of unseen shapes. However,

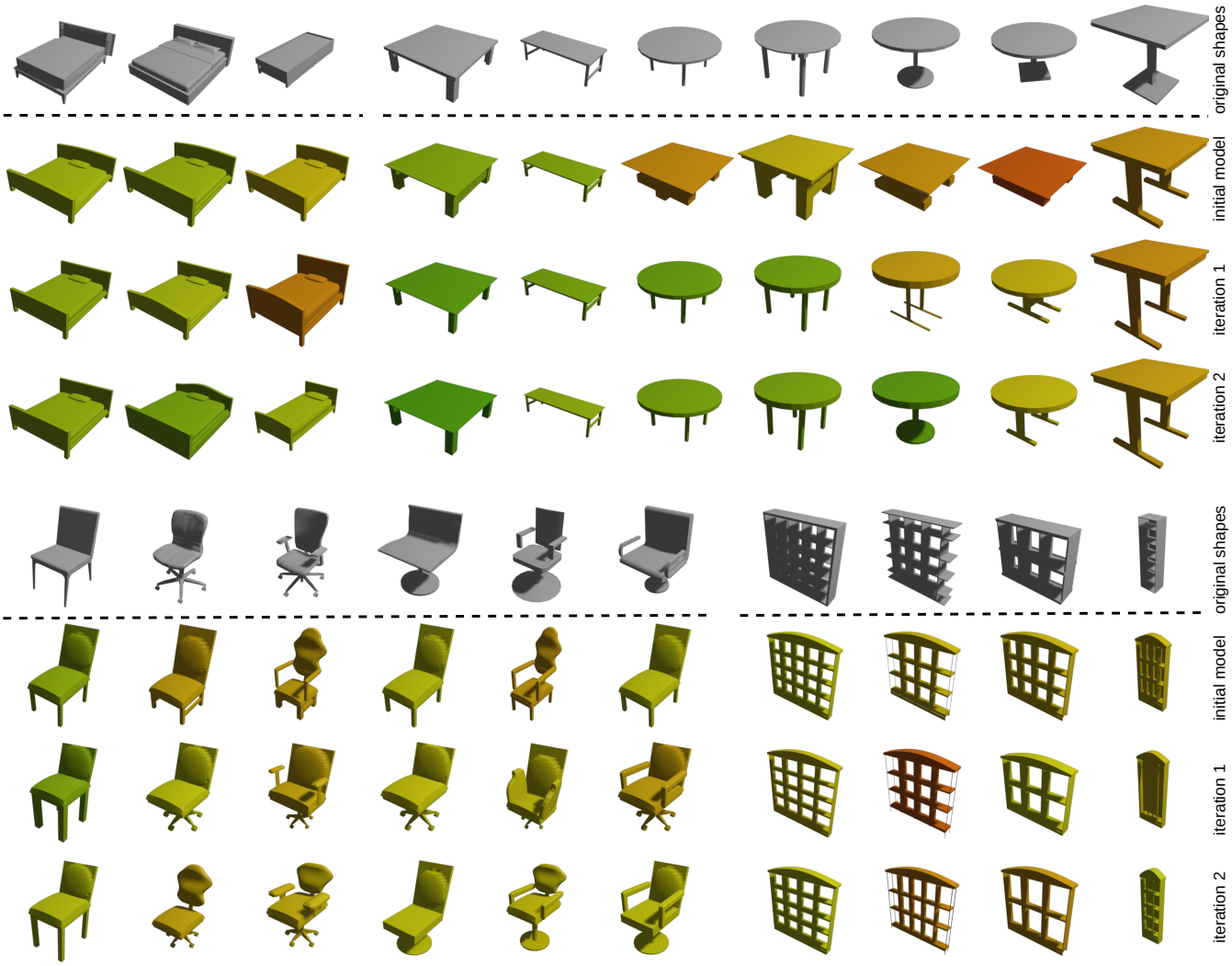

Figure 6: *Improvement of the shapes generated by the procedural model after each iteration, where the colormap ranging from green to red denotes the level of similarity of the shapes to the reference shapes (rows of shapes in gray).*

it neither takes into account that there can be shapes that are not replicated well, nor does it provide feedback for extending the procedural model, which itself is an essential and a time-consuming part of developing a procedural model. Our method aims to solve this problem by providing the user with assistance that shortens the time for refining the model.

4.4. Parameter Prediction Model

We evaluate how well the neural network presented in Section 3.2 predicts the parameters for given shapes. For training data, we take the four 3D datasets of shapes discussed in Section 4.1 and the parameters given by the complete procedural model developed in Section 4.2. The training data is split into train, validation, and test sets at a ratio of 0.8 : 0.1 : 0.1. For the normalized scalar elements in the parameter vector, we report the average of mean absolute errors (MAE) for all parameters, and for integer and binary elements, we report the average F1 scores, since a measurement of fractional improvements over the parameter values is not meaningful in the case of integers or booleans. Table 1 shows the prediction results for different categories of shapes. We see that the model is able to predict the parameter vectors reasonably well, with errors for scalar parameters of 0.1 or lower.

We also investigate how the number of cameras used by the network influences the prediction performance. In our experiments, increasing the number of cameras generally results in better performance. However, increasing the number of cameras also introduces more parameters to optimize and results in longer training time. In such a situation, a trade-off can be made by finding the point of diminishing returns. For instance, we observed that in our specific setting involving 3D shapes, using more than three cameras does not result in any significant improvement in performance. In prin-

Table 1: *Performance of the parameter prediction model. (a) Average parameter prediction errors (MAE) for normalized scalar parameters and average prediction scores (F1) for integer and binary parameters, for each category of shapes. The number of parameters predicted for each class/type are reported next to each prediction score. Lower MAE and higher F1 are better. (b) Changes in prediction scores when camera positions are optimized as opposed to using fixed camera positions. Improvements are shown in bold.*

		Bed	Chair	Shelf	Table
(a)	Scalar (MAE↓)	0.07(6)	0.05(4)	0.08(4)	0.10(6)
	Integer (F1↑)	1.00(1)	1.00(4)	0.91(1)	1.00(2)
	Binary(F1↑)	-	-	0.89(3)	0.70(1)
(b)	Scalar (MAE↓)	+0.04	+0.01	**-0.01**	**-0.09**
	Integer (F1↑)	0	**+0.01**	**+0.02**	**+0.03**
	Binary(F1↑)	-	-	**+0.01**	**+0.21**

Figure 7: *Comparison of ShapeNet objects (top) recreated with ShapeAssembly (middle) and our method (bottom). Our method involves manual editing of the procedural model, but provides feedback on how to improve the model, and allows for more flexibility in the program syntax, such as adding curved parts.*

ciple, the number of cameras can be treated as a hyper-parameter of the prediction model that can be tuned for optimizing the results.

Table 1 also shows the results of an ablation study where we investigated the effectiveness of optimizing the camera positions described in Section 3.2. We trained the model on the same dataset twice; first with the cameras at fixed positions and then with optimization of the camera positions. We kept all other settings the same. When the camera positions were not optimized, we used three camera positions randomly sampled around a sphere. We then tested the two models on the same test set and compared the results. Table 1 shows the changes in parameter prediction scores when the camera position is optimized. For some shapes (chair, shelf, and table), optimizing the camera positions results in better prediction. For others (bed), a fixed camera position is more suitable. Thus, the effectiveness of the camera optimization depends on the type of shape. From an application standpoint, both options can be made available to the user who ultimately chooses the one that performs the best in specific scenarios.

Figure 8: *Examples of compatible (green) and incompatible (red) shapes predicted by our method for the reference shapes (in gray).*

Figure 9: *Examples of table projections that users were asked to recreate procedurally in the user study.*

4.5. Visual Similarity of Replicated ShapeNet Objects

To give examples of how the parameter prediction scores translate to visual results, Figure 8 shows some examples of ShapeNet objects and their replications obtained with the procedural model. We show examples where the replicated shapes closely resemble the reference shapes (colored in green), as well as shapes that are not replicated very well (colored in red). The reference shapes are colored gray. The reason for some of the shapes not being approximated well is that the procedural model still needs additional rules to be able to fully replicate these shapes.

5. User Study

We performed a user study where we tasked users with creating a procedural model for a reference collection of shapes. To keep the development time within reasonable bounds, we chose a simplified task for the user study. We asked users to create a procedural model for a collection of 2D silhouettes of tables, illustrated in Figure 9. The shapes are derived from 2D front projections of ShapeNet objects. The programming environment is based on the Python programming language and allows to write instructions that create shapes based on a combination of geometric primitives such as lines, polygons, boxes, circles, and arcs, which are sufficient for replicating the shapes in the collection. The user is also able to employ programming constructs such as loops, conditionals, etc. The user starts with a white background and writes a program that draws multiple geometric primitives of black color over the background. The placement and size of the primitives can be controlled by parameters that are input to the program. The objective of the

user study task is for the user to write a complete program that is able to replicate the shapes from a reference collection. Figure 10 shows the graphical user interface that we designed for the user study. The pane on the right side shows a summary of the original shapes and replicated shapes. The user can click on any shape from the summary pane and view the enlarged pair of shapes for side-by-side comparison. The shapes colored in green are shapes that are replicated well. The bench shapes are provided as examples the users can refer to and the table shapes are for the users to complete. The users from the baseline group have the same interface, except it does not have any automatic assistance.

We compared different aspects of the creation process in two settings: when using our method and a baseline method. When using the baseline method, users develop the procedural model and are also required to determine the parameter values for each shape in the collection. The users also need to visually inspect if the generated shapes replicate the shapes in the collection. On the other hand, our method provides assistance for parameter selection and visual inspection. The parameter prediction involves the training of the neural network described in Section 3.2. For the simpler 2D setting, the training time is substantially lower than the 3D case. In the hardware configuration used for the experiment, the training time is approximately 5 minutes.

To conduct the user study, we allocate the same amount of time for users in both settings. We evaluate each resulting procedural model by counting the number of shapes within the reference 2D shape collection that are accurately replicated. A shape is considered to be accurately replicated when the visual similarity between the input shape and its procedurally recreated counterpart exceeds a predefined threshold. For measuring visual similarity between two shapes in 2D, we use Visual Information Fidelity [SB06] and a threshold of 0.05, which captures reasonably-similar shapes. In addition to the number of well-replicated shapes, we also record the average similarity between the reference shapes and the procedurally recreated shapes using the shape IoU metric.

After obtaining institutional ethics approval, we recruited a group of 10 participants (8 upper-year undergraduate students and 2 graduate students). Other than the graduate students, the participants did not have prior experience with procedural modeling. The participants were divided into two groups (between-subjects study), one to test our method (5 users) and one for the baseline (5 users). We divided the graduate students into two groups of participants to avoid bias. Each participant had 2 hours to complete the task which did not include the time to brief the participants.

When analyzing the results, we find that the users of our method were able to replicate an average of 34 out of 50 shapes within the given time. Almost all of them were able to complete at least two iterations, improving their procedural model in an incremental manner. In comparison, the users of the baseline method replicated only 12 shapes on average. Meanwhile, both groups achieved a comparable similarity (measured by IoU) of 92% between the replicated shapes and the reference shapes.

We observed that the variance in the number of replicated shapes for the baseline group is 141.8, while the variance is 26.2 for the group that used our method. Most of the users did not have prior experience with procedural modeling, and different users implement the procedural models at a different pace, which is reflected by the high variance. We also investigated the difference in the average number of replicated shapes between the two groups and performed a T-test. The resulting P-value is 0.01, indicating statistical significance and implying that our method does allow users to replicate more shapes on average in the time given.

To further verify the effectiveness of our method, we applied our parameter prediction model to the procedures written by the users of the baseline setting, so that the parameters of shapes are set automatically. In this manner, for the same procedures, our prediction model is able to successfully replicate 27 shapes on average. This is a significant increase over the number of shapes replicated manually and demonstrates how our method can facilitate development of procedural models.

We also asked the participants to complete questionnaires at the end of the study. Participants commented that our method would benefit from the addition of measurement tools to the user-interface and quicker feedback.

6. Conclusions

We introduced a data-guided method for authoring a procedural model in order to replicate a collection of shapes. Starting from an initial procedural model, our method enables an iterative workflow where a programmer can refine the model according to feedback provided by our method on how well the shapes in the collection are replicated. We showed with a user study and qualitative examples that the method speeds up the work of authoring a procedural model, and makes it easier to verify how well the shapes in the collection are replicated by the model.

Limitations and future work. Our method has limitations stemming from the components that compose the method. First, the quality of the parameter prediction can be suboptimal when dealing with complex procedural models that require many parameters, since the limited sample of training data may not capture sufficient parameter variations. The prediction in such scenarios could be improved by increasing the training data sample, which will incur in an increase of the training time, or developing an adaptive sampling that selects the representatives of the parameter space capturing the most variation. In addition, we use Light Field Descriptors for shape comparison, which we found to be sufficient in our evaluation. Nevertheless, other shape comparison methods such as ones based on metric learning could be used. Finally, our method currently only provides feedback on which shapes are not replicated well by the model. An interesting direction for future work is to develop methods that provide more detailed feedback, such as providing localized feedback on shape parts not approximated well, or automatically suggesting procedural rules (code snippets [NGDA*16]) that can then be refined by the programmer.

7. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant, JSPS Grant-in-Aid JP23K16921, and JST CREST

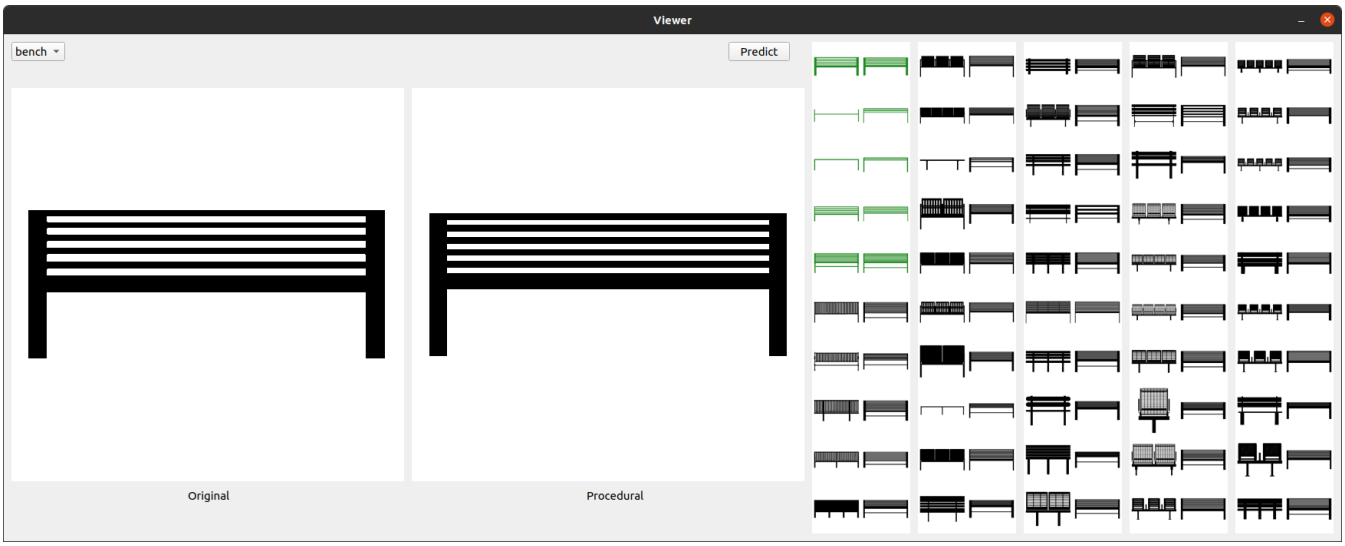

Figure 10: *Graphical user interface employed in the user study.*

Grant Number JPMJCR17A1, Japan. Ishtiaque Hossain would also like to thank Carleton University and the Provincial Government of Ontario, Canada, for partially funding this research through an Ontario Graduate Scholarship.

References

[BWS10] Bokeloh M., Wand M., Seidel H.-P.: A connection between partial symmetry and inverse procedural modeling. *ACM Trans. on Graphics 29*, 4 (2010), 104:1–10. 2, 3

[CFG*15] Chang A. X., Funkhouser T., Guibas L., Hanrahan P., Huang Q., Li Z., Savarese S., Savva M., Song S., Su H., Xiao J., Yi L., Yu F.: Shapenet: An information-rich 3d model repository, 2015. arXiv:1512.03012. 6

[CSQ*22] Cascaval D., Shalah M., Quinn P., Bodik R., Agrawala M., Schulz A.: Differentiable 3d cad programs for bidirectional editing. *Computer Graphics Forum 41*, 2 (2022), 309–323. 3

[CTSO03] Chen D.-Y., Tian X.-P., Shen Y.-T., Ouhyoung M.: On visual similarity based 3d model retrieval. *Computer Graphics Forum 22*, 3 (2003), 223–232. 6

[CTZ20] Chen Z., Tagliasacchi A., Zhang H.: BSP-NET: Generating compact meshes via binary space partitioning. In *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2020), pp. 45–54. 2

[DAB14] Demir I., Aliaga D. G., Benes B.: Proceduralization of buildings at city scale. In *Proc. 3D Vision (3DV)* (2014), vol. 1, pp. 456–463. 3

[DAB16] Demir I., Aliaga D. G., Benes B.: Proceduralization for editing 3d architectural models. In *Proc. 3D Vision (3DV)* (2016), pp. 194–202. 3

[GDGP16] Guérin E., Digne J., Galin E., Peytavie A.: Sparse representation of terrains for procedural modeling. *Computer Graphics Forum 35*, 2 (2016), 177–187. 4

[GGC*20] Gadelha M., Gori G., Ceylan D., Mech R., Carr N., Boubekeur T., Wang R., Maji S.: Learning generative models of shape handles. In *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2020), pp. 402–411. 2

[GKG*22] Gaillard M., Krs V., Gori G., Měch R., Benes B.: Automatic differentiable procedural modeling. *Computer Graphics Forum 41*, 2 (2022), 289–307. 3

[HDR19] Hu Y., Dorsey J., Rushmeier H.: A novel framework for inverse procedural texture modeling. *ACM Trans. on Graphics 38*, 6 (2019), 186:1–14. 3

[HGH*22] Hu Y., Guerrero P., Hasan M., Rushmeier H., Deschaintre V.: Node graph optimization using differentiable proxies. In *ACM SIGGRAPH 2022 Conference Proceedings* (2022), pp. 5:1–9. 3

[HKYM17] Huang H., Kalogerakis E., Yumer E., Mech R.: Shape synthesis from sketches via procedural models and convolutional networks. *IEEE Trans. Visualization & Computer Graphics 23*, 8 (2017), 2003–2013. 2, 3

[HLHF22] Hui K.-H., Li R., Hu J., Fu C.-W.: Neural wavelet-domain diffusion for 3d shape generation. In *SIGGRAPH Asia 2022 Conference Papers* (2022), pp. 24:1–9. 2, 3

[HPG*22] Hertz A., Perel O., Giryes R., Sorkine-Hornung O., Cohen-Or D.: SPAGHETTI: editing implicit shapes through part aware generation. *ACM Trans. on Graphics 41*, 4 (2022), 106:1–20. 2, 3

[JBX*20] Jones R. K., Barton T., Xu X., Wang K., Jiang E., Guerrero P., Mitra N. J., Ritchie D.: ShapeAssembly: learning to generate programs for 3D shape structure synthesis. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 39*, 6 (2020), 234:1–10. 2, 4, 7

[JCG*21] Jones R. K., Charatan D., Guerrero P., Mitra N. J., Ritchie D.: Shapemod: Macro operation discovery for 3d shape programs. *ACM Trans. on Graphics (Proc. SIGGRAPH) 40*, 4 (2021), 153:1–16. 2

[JWR22] Jones R. K., Walke H., Ritchie D.: Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2022), 9871–9880. 4

[KG01] Kovar L., Gleicher M.: Simplicial families of drawings. In *Proc. Symp. on User Interface Software and Technology* (2001), pp. 163–172. 3

[KMG*21] Krs V., Měch R., Gaillard M., Carr N., Benes B.: PICO: procedural iterative constrained optimizer for geometric modeling. *IEEE Trans. Visualization & Computer Graphics 27*, 10 (2021), 3968–3981. 2

[KSI14] Koyama Y., Sakamoto D., Igarashi T.: Crowd-powered

parameter analysis for visual design exploration. In *Proc. Symp. on User Interface Software and Technology* (2014), pp. 65–74. 3

[LB14] LOPER M. M., BLACK M. J.: Opendr: An approximate differentiable renderer. In *Proc. Euro. Conf. on Computer Vision* (2014), pp. 154–169. 6

[LLCL19] LIU S., LI T., CHEN W., LI H.: Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2019), pp. 7708–7717. 2, 6

[LMW*22] LIN C., MITRA N., WETZSTEIN G., GUIBAS L. J., GUERRERO P.: Neuform: Adaptive overfitting for neural shape editing. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022), vol. 35, pp. 15217–15229. 3

[LSL*19] LIPP M., SPECHT M., LAU C., WONKA P., MÜLLER P.: Local editing of procedural models. *Computer Graphics Forum 38*, 2 (2019), 13–25. 3

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. In *Proc. SIGGRAPH* (2008), pp. 102:1–10. 2

[MB21] MICHEL E., BOUBEKEUR T.: Dag amendment for inverse control of parametric shapes. *ACM Trans. on Graphics 40*, 4 (2021), 173:1–14. 3

[MM11] MERRELL P., MANOCHA D.: Model synthesis: A general procedural modeling algorithm. *IEEE Trans. Visualization & Computer Graphics 17*, 6 (2011), 715–728. 2

[MPZ20] MATHUR A., PIRRON M., ZUFFEREY D.: Interactive programming for parametric cad. *Computer Graphics Forum 39*, 6 (2020), 408–425. 3

[NBA18] NISHIDA G., BOUSSEAU A., ALIAGA D. G.: Procedural modeling of a building from a single image. *Computer Graphics Forum (Proc. Eurographics) 37*, 2 (2018), 415–429. 3

[NGDA*16] NISHIDA G., GARCIA-DORADO I., ALIAGA D. G., BENES B., BOUSSEAU A.: Interactive sketching of urban procedural models. *ACM Trans. on Graphics 35*, 4 (2016), 130:1–11. 3, 10

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32*, 2 (2012), 66–75. 3

[PFS*19] PARK J. J., FLORENCE P., STRAUB J., NEWCOMBE R., LOVEGROVE S.: DeepSDF: Learning continuous signed distance functions for shape representation. In *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2019), pp. 165–174. 2

[PVG*11] PEDREGOSA F., VAROQUAUX G., GRAMFORT A., MICHEL V., THIRION B., GRISEL O., BLONDEL M., PRETTENHOFER P., WEISS R., DUBOURG V., VANDERPLAS J., PASSOS A., COURNAPEAU D., BRUCHER M., PERROT M., DUCHESNAY E.: Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830. 6

[RRN*20] RAVI N., REIZENSTEIN J., NOVOTNY D., GORDON T., LO W.-Y., JOHNSON J., GKIOXARI G.: Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501* (2020). 6

[SB06] SHEIKH H., BOVIK A.: Image information and visual quality. *IEEE Transactions on Image Processing 15*, 2 (2006), 430–444. 10

[SGL*22] SHARMA G., GOYAL R., LIU D., KALOGERAKIS E., MAJI S.: Neural shape parsers for constructive solid geometry. *IEEE Trans. Pattern Analysis & Machine Intelligence 44*, 5 (2022), 2628–2640. 3

[SLH*20] SHI L., LI B., HAŠAN M., SUNKAVALLI K., BOUBEKEUR T., MECH R., MATUSIK W.: Match: Differentiable material graphs for procedural material capture. *ACM Trans. on Graphics 39*, 6 (2020), 196:1–15. 3

[SPK*14] STAVA O., PIRK S., KRATT J., CHEN B., MĚCH R., DEUSSEN O., BENES B.: Inverse procedural modelling of trees. *Computer Graphics Forum 33*, 6 (2014), 118–131. 2, 3

[STBB14] SMELIK R. M., TUTENEL T., BIDARRA R., BENES B.: A survey on procedural modelling for virtual worlds. *Computer Graphics Forum 33*, 6 (2014), 31–50. 2

[SZ15] SIMONYAN K., ZISSERMAN A.: Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations* (2015), pp. 730–734. 6

[TGY*09] TALTON J. O., GIBSON D., YANG L., HANRAHAN P., KOLTUN V.: Exploratory modeling with collaborative design spaces. *ACM Trans. on Graphics 28*, 5 (2009), 1–10. 3

[TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM Trans. on Graphics 30*, 2 (2011), 11:1–14. 2

[TLS*19] TIAN Y., LUO A., SUN X., ELLIS K., FREEMAN W. T., TENENBAUM J. B., WU J.: Learning to infer and execute 3d shape programs. In *Proc. Int. Conf. on Learning Representations* (2019), pp. 1–21. 4

[WSS*20] WEI F., SIZIKOVA E., SUD A., RUSINKIEWICZ S., FUNKHOUSER T.: Learning to infer semantic parameters for 3D shape editing. In *Proc. 3D Vision (3DV)* (2020), pp. 434–442. 2, 3

[WYD*14] WU F., YAN D.-M., DONG W., ZHANG X., WONKA P.: Inverse procedural modeling of facade layouts. *ACM Trans. on Graphics 33*, 4 (2014), 1–10. 3

[YAMK15] YUMER M. E., ASENTE P., MĚCH R., KARA L. B.: Procedural modeling using autoencoder networks. In *Proc. Symp. on User Interface Software & Technology* (2015), pp. 109–118. 3

[YLM*22] YAN X., LIN L., MITRA N. J., LISCHINSKI D., COHEN-OR D., HUANG H.: Shapeformer: Transformer-based shape completion via sparse representation. In *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2022), pp. 6239–6249. 2, 3

[ZAESB20] ZEKUN H., AVERBUCH-ELOR H., SNAVELY N., BELONGIE S.: DualSDF: semantic shape manipulation using a two-level representation. In *Proc. IEEE Conf. on Computer Vision & Pattern Recognition* (2020), pp. 7631–7641. 2

[ZVW*22] ZENG X., VAHDAT A., WILLIAMS F., GOJCIC Z., LITANY O., FIDLER S., KREIS K.: Lion: Latent point diffusion models for 3d shape generation. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022), vol. 35, pp. 10021–10039. 2, 3

Pacific Graphics 2023
R. Chaine, Z. Deng, and M. H. Kim
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 7

Data-guided Authoring of Procedural Models of Shapes Supplementary Material

Ishtiaque Hossain[1] I-Chao Shen[2] Takeo Igarashi[2] Oliver van Kaick[1]

[1] Carleton University, Canada
[2] The University of Tokyo, Japan

1. Additional result of our method on 3D shapes

Since the procedural model used in our paper for the main results on 3D shapes is too complex for illustration purposes, in this section, we show an example of a complete result of our method obtained with a simplified procedural model. The procedural model used here combines primitive shapes such as cubes and cylinders to create tables. The primitive shapes are created with the Blender API for Python. Figure 1 shows the cube and the cylinder functions, which are parameterized by the location and the scale of the corresponding primitive.

For the set of reference shapes, we selected 20 shapes from the table category in ShapeNet. Figure 2 shows the selected shapes. The figure also shows how the clustering step of our method collects similar-looking shapes into groups for the user to inspect. Each group is assigned a different color. In this example, after looking at the groups in Figure 2, the user decided to write an initial procedural model that creates a rectangular table-top and four straight legs, which can be created using cubes at various locations and scales. Figure 3 shows the initial procedural model.

Figure 4 shows the result of one iteration of our method after using the initial procedural model written by the user. We see how our method identifies the compatible shapes, i.e. the reference shapes that can be replicated well using the initial procedural model, as well as the incompatible shapes. Good replications are colored green and other approximations are colored yellow. Again, similar shapes are grouped together and looking at these groups, the user decided to add an additional parameter and code to create rectangular or rounded table-tops. Figure 5 shows the second version of the procedural model after this change.

Figure 6 shows the second iteration of our method. While inspecting the groups of incompatible shapes, the user noticed that some tables have a support structure among the four legs and decided to introduce another parameter and code to specify the type of leg to be constructed. Figure 7 shows the revised procedural model that takes this change into account.

The next iteration of our method is shown in Figure 8. Among the groups of incompatible shapes, there are tables that have only one leg with a rounded base. The user then included this type of leg into the model and revised the procedural model accordingly. Figure 9 shows the fourth version of the procedural model.

Figure 10 shows the next iteration, where the only remaining incompatible shapes have another type of leg. Then, the user modifies the procedural model to accommodate the additional leg type. Figure 11 shows the fifth version of the procedural model.

Figure 12 shows the final iteration of our method, where all the reference shapes are replicated well. At this point, the procedural model has grown in both complexity and its ability to generate variations of tables similar to the reference shapes.

```python
import bpy


def cube(location=(0.0, 0.0, 0.0), scale=(1.0, 1.0, 1.0)):
    bpy.ops.mesh.primitive_cube_add(size=1.0, location=location, scale=scale)


def cylinder(location=(0.0, 0.0, 0.0), scale=(1.0, 1.0, 1.0)):
    bpy.ops.mesh.primitive_cylinder_add(radius=0.5, depth=1.0, location=location, scale=scale)
```

Figure 1: *Methods for creating cube and cylinder primitives.*

Figure 2: *Initial grouping performed automatically by our method on selected shapes from ShapeNet.*

```python
def create_table(height, breadth, top_thickness, leg_thickness):
    h, b, t_th, l_th = height, breadth, height * top_thickness, breadth * leg_thickness
    # create table-top
    top_loc, top_scl = (0.0, 0.0, (h - t_th) / 2.0), (1.0, b, t_th)
    cube(location=top_loc, scale=top_scl)
    leg_scl = (l_th, l_th, h)
    # create four legs
    x = (1.0 - l_th) / 2.0
    y = (b - l_th) / 2.0
    leg_locs = [(x, y, 0.0), (x, -y, 0.0), (-x, y, 0.0), (-x, -y, 0.0)]
    for leg_loc in leg_locs:
        cube(location=leg_loc, scale=leg_scl)
```

Figure 3: *Initial procedural model.*

Figure 4: *Replication result using the initial version of the procedural model.*

```python
def create_table(height, breadth, top_thickness, leg_thickness, roundtop):
    h, b, t_th, l_th = height, breadth, height * top_thickness, breadth * leg_thickness
    rt = roundtop
    # create table-top
    top_loc, top_scl = (0.0, 0.0, (h - t_th) / 2.0), (1.0, b, t_th)
    if rt:
        cylinder(location=top_loc, scale=top_scl)
    else:
        cube(location=top_loc, scale=top_scl)
    leg_scl = (l_th, l_th, h)
    # create four legs
    a = 1.41 if rt else 1.0
    x = (1.0 - a * l_th) / (2 * a)
    y = (b - a * l_th) / (2 * a)
    leg_locs = [(x, y, 0.0), (x, -y, 0.0), (-x, y, 0.0), (-x, -y, 0.0)]
    for leg_loc in leg_locs:
        cube(location=leg_loc, scale=leg_scl)
```

Figure 5: *Second version of the procedural model.*

Figure 6: *Replication result using the second version of the procedural model.*

```python
def create_table(height, breadth, top_thickness, leg_thickness, roundtop, leg_type):
    h, b, t_th, l_th  = height, breadth, height * top_thickness, breadth * leg_thickness
    rt, l_tp = roundtop, leg_type
    # create table-top
    top_loc, top_scl = (0.0, 0.0, (h - t_th) / 2.0), (1.0, b, t_th)
    if rt:
        cylinder(location=top_loc, scale=top_scl)
    else:
        cube(location=top_loc, scale=top_scl)
    leg_scl = (l_th, l_th, h)
    # create four legs
    a = 1.41 if rt else 1.0
    x = (1.0 - a * l_th) / (2 * a)
    y = (b - a * l_th) / (2 * a)
    leg_locs = [(x, y, 0.0), (x, -y, 0.0), (-x, y, 0.0), (-x, -y, 0.0)]
    for leg_loc in leg_locs:
        cube(location=leg_loc, scale=leg_scl)
    # create support between legs
    if l_tp == 'support':
        if rt:
            leg_sprt_scl = (l_th, b / a, l_th)
        else:
            leg_sprt_scl = (l_th, b, l_th)
        cube(location=(x, 0.0, -h / 3.0), scale=leg_sprt_scl)
        cube(location=(-x, 0.0, -h / 3.0), scale=leg_sprt_scl)
        cube(location=(0.0, 0.0, -h / 3.0), scale=((1.0 - a * l_th) / a, l_th, l_th))
```

Figure 7: *Third version of the procedural model.*

Figure 8: *Replication result using the third version of the procedural model.*

```python
def create_table(height, breadth, top_thickness, leg_thickness, roundtop, leg_type):
    h, b, t_th, l_th  = height, breadth, height * top_thickness, breadth * leg_thickness
    rt, l_tp = roundtop, leg_type
    # create table-top
    top_loc, top_scl = (0.0, 0.0, (h - t_th) / 2.0), (1.0, b, t_th)
    if rt:
        cylinder(location=top_loc, scale=top_scl)
    else:
        cube(location=top_loc, scale=top_scl)
    leg_scl = (l_th, l_th, h)
    if l_tp == 'round':
        # create rounded leg
        cylinder(location=(0.0, 0.0, 0.0), scale=leg_scl)
        cylinder(location=(0.0, 0.0, -(19.0 * h) / 40.0), scale=(0.5, 0.5, h / 20.0))
    else:
        # create four legs
        a = 1.41 if rt else 1.0
        x = (1.0 - a * l_th) / (2 * a)
        y = (b - a * l_th) / (2 * a)
        leg_locs = [(x, y, 0.0), (x, -y, 0.0), (-x, y, 0.0), (-x, -y, 0.0)]
        for leg_loc in leg_locs:
            cube(location=leg_loc, scale=leg_scl)
        # create support between legs
        if l_tp == 'support':
            if rt:
                leg_sprt_scl = (l_th, b / a, l_th)
            else:
                leg_sprt_scl = (l_th, b, l_th)
            cube(location=(x, 0.0, -h / 3.0), scale=leg_sprt_scl)
            cube(location=(-x, 0.0, -h / 3.0), scale=leg_sprt_scl)
            cube(location=(0.0, 0.0, -h / 3.0), scale=((1.0 - a * l_th) / a, l_th, l_th))
```

Figure 9: *Fourth version of the procedural model.*

Figure 10: *Replication result using the fourth version of the procedural model.*

```python
def create_table(height, breadth, top_thickness, leg_thickness, roundtop, leg_type):
    h, b, t_th, l_th  = height, breadth, height * top_thickness, breadth * leg_thickness
    rt, l_tp = roundtop, leg_type
    # create table-top
    top_loc, top_scl = (0.0, 0.0, (h - t_th) / 2.0), (1.0, b, t_th)
    if rt:
        cylinder(location=top_loc, scale=top_scl)
    else:
        cube(location=top_loc, scale=top_scl)
    leg_scl = (l_th, l_th, h)
    if l_tp == 'round':
        # create rounded leg
        cylinder(location=(0.0, 0.0, 0.0), scale=leg_scl)
        cylinder(location=(0.0, 0.0, -(19.0 * h) / 40.0), scale=(0.5, 0.5, h / 20.0))
    elif l_tp == 'split':
        # create split legs
        for x in [0.5 - l_th / 2.0, l_th / 2.0 - 0.5]:
            cube(location=(x, 0.0, 0.0), scale=(l_th, l_th, h))
            cube(location=(x, 0.0, (l_th - h) / 2.0), scale=(l_th, b, l_th))
    else:
        # create four legs
        a = 1.41 if rt else 1.0
        x = (1.0 - a * l_th) / (2 * a)
        y = (b - a * l_th) / (2 * a)
        leg_locs = [(x, y, 0.0), (x, -y, 0.0), (-x, y, 0.0), (-x, -y, 0.0)]
        for leg_loc in leg_locs:
            cube(location=leg_loc, scale=leg_scl)
        # create support between legs
        if l_tp == 'support':
            if rt:
                leg_sprt_scl = (l_th, b / a, l_th)
            else:
                leg_sprt_scl = (l_th, b, l_th)
            cube(location=(x, 0.0, -h / 3.0), scale=leg_sprt_scl)
            cube(location=(-x, 0.0, -h / 3.0), scale=leg_sprt_scl)
            cube(location=(0.0, 0.0, -h / 3.0), scale=((1.0 - a * l_th) / a, l_th, l_th))
```

Figure 11: *Fifth version of the procedural model.*

Figure 12: *Replication result using the fifth version of the procedural model.*