Evolutionary Generation of Primitive-Based Mesh Abstractions

Markus Friedrich, Felip Guimerà Cuevas, Andreas Sedlmeier, André Ebert
Institute for Computer Science
LMU Munich
Oettingenstr. 67
80538 Munich, Germany
{markus.friedrich|andreas.sedlmeier|andre.ebert}@ifi.lmu.de, felip.guimera@campus.lmu.de

ABSTRACT
The procedural generation of data sets for empirical algorithm validation and deep learning tasks in the area of primitive-based geometry is cumbersome and time-consuming while ready-to-use data sets are rare. We propose a new and highly flexible framework based on Evolutionary Computing that is able to create primitive-based abstractions of existing triangle meshes favoring fast running times and high geometric variation over reconstruction precision. These abstractions are represented as CSG trees to widen the scope of possible applications. As part of the evaluation, we show how we successfully used the generator to create a data set for the evaluation of neural point cloud segmentation pipelines and additionally explain how to use the system to create artistic abstractions of meshes provided by publicly available triangle mesh databases.

Keywords
Evolutionary Algorithms, Geometry Processing, CAD, CSG, Deep Learning

1 INTRODUCTION
A plethora of empirical algorithm validation and deep learning tasks in the field of primitive-based 3D geometry processing require a diverse and sufficiently large set of 3D models as test or training input. For models represented as triangle meshes, these data sets exist and are available for free (e.g. ShapeNet [CFG+15], ModelNet [WSK+14], ABC [KMJ+18], etc.). However, if model representations based on a composition of geometric primitives (e.g. spheres, cylinders, cuboids, etc. combined by Boolean set operations) are needed, data sets are rare. For example, the ABC data set contains 1000,000+ Computer Aided Design (CAD) models with primitive information but lacks cuboids as one of the considered primitive types and also does not account for compositional information like the arrangement of Boolean operators in a CSG tree.

An example use case where such data sets are needed would be a CSG tree detection pipeline based on neural networks (see [SGL+18] for example). It requires a 3D point cloud as input and delivers a CSG tree that fits the 3D point cloud best, together with the parameters of detected primitives. Traditionally, deep learning tasks need huge training sets, which are in this case hard to find or cumbersome to generate manually with off-the-shelf CAD tools.

In order to fill this gap, we propose a CSG tree generator framework based on Evolutionary Computing that transforms a triangle mesh model together with a set of constraints (e.g. frequency distribution of primitive types) into a CSG tree representation which combines a set of fitted primitives with Boolean set operations (e.g. union, intersection, difference). Note that the primary goal here lies not in finding the CSG tree which matches the input geometry as perfectly as possible but in generating a sufficiently accurate abstraction, allowing for the generation of tens of thousands of models within an acceptable time frame on currently available hardware. Another, completely different use case worth to consider is the artistic abstraction of geometry for visually appealing renderings and animations appearing in entertainment products and multimedia installations.

The proposed processing pipeline starts with sampling the input model, resulting in a 3D point cloud that is then clustered for better computational efficiency. For each cluster, primitives are fitted. Which primitive types to use for fitting is determined by sampling a user-defined frequency distribution that specifies the desired distribution of primitive types in the resulting CSG tree. Then, per-cluster CSG trees are extracted using a specific variant of an Evolutionary Algorithm (EA). Finally, resulting CSG trees are merged to a combined result.

In summary, this paper presents the following main contributions:

- A highly flexible and configurable framework for the generation of primitive-based mesh abstractions that are represented as CSG trees.
• An evaluation of three neural network architectures considering the task of primitive detection from 3D point clouds. Necessary training, validation and test data sets were generated with our proposed framework.

2 BACKGROUND
2.1 CSG Trees and Signed Distance Functions
A CSG tree represents a 3D model as a hierarchical combination of Boolean set operations and primitives (e.g. cubes, spheres, cylinders, ...). Set operations are thereby inner nodes of the tree whereas primitives are always leaves. In our case, a primitive $p$ is described by a signed distance function $f_p$, where the surface of $p$ is the zero set of $f_p: \{x \in \mathbb{R}^3 : f_p(x) = 0\}$. The Boolean set operations are represented using min- and max-functions [Ric73]:

- Intersection: $p_1 \cap p_2 := \max(f_{p_1}, f_{p_2})$
- Union: $p_1 \cup p_2 := \min(f_{p_1}, f_{p_2})$
- Complement: $\overline{p} := -f_p$
- Subtraction: $p_1 \setminus p_2 := p_1 \cap \overline{p_2}$

The surface normal for a certain point $x \in \mathbb{R}^3$ can be retrieved by $\nabla f_p(x)$.

2.2 Genetic Algorithms
A Genetic Algorithm is a population-based metaheuristic for solving optimization problems and belongs to the class of Evolutionary Algorithms. The concept is inspired by the biological phenomenon of natural selection. Initially, a randomly generated population of possible solutions is created and ranked using a problem-specific objective function. In the following iteration, the best solutions from the last iteration are selected and changed using domain-dependent modification operators (mutation and crossover). This procedure is repeated until a certain stop criterion is met (e.g. a certain objective function value or maximum iteration count has been reached). The extraction of a CSG tree from a set of fitted primitives and a shape-describing point cloud is a combinatorial optimization problem (see e.g. [FFPF18]) which we solve using a Genetic Algorithm.

3 RELATED WORK
3.1 Available Data Sets
Large model databases of triangle meshes exist (ShapeNet [CFG+15], ModelNet [WSK+14]) but do not describe distinct primitives. The data set introduced in [KMJ+18] contains primitives but does not include cuboids and CSG tree descriptions. For an exhaustive overview of available data sets, see [KMJ+18].

3.2 Procedural Model Fitting & Modeling
Procedural Model Fitting (PMF) describes the task of finding a geometric representation that fits a certain input data set (e.g. a point cloud) as precisely as possible. The primitive generator proposed by Zou et al. [ZYy+17] takes point clouds as input and uses a variant of the Iterative Closest Point method (ICP) [BM92] to fit cuboids but is restricted to that single primitive type. A constrained-based PMF technique employing a Genetic Algorithm is proposed in [HSS17]. While results look promising, all models are represented using triangle meshes, which is not suitable for our use case. Other approaches use Sequential [RMGH15] or Markov Chain Monte Carlo [TLL+11] methods as well as Reinforcement Learning [TKS+13, SGL+18]. Our approach is different in that we accept arbitrary 3D meshes (not just point clouds) as input and focus on generation speed rather than precise fitting.

A related research field is Procedural Modeling (PM). There, visual content (3D models, textures, ...) is generated based on specialized algorithms with user-controlled parameters. In recent years, there has been vivid research activity in the field of procedural content generation using Machine Learning approaches (PCGML). See [SSG+18] for a comprehensive survey. For a survey on the procedural generation of complete worlds (landscapes, buildings, creatures, ...), see [FE17].

4 PROBLEM STATEMENT
The problem that is solved by our proposed generator can be described as follows: Given a 3D model represented as a closed triangle mesh and a frequency distribution of primitive types initially defined by the user, generate a CSG tree which matches the input mesh as closely as possible while the set of primitives corresponds to the selected distribution. Important to note is that the computational effort of the generation process should be kept low in order to allow for the creation of large model data sets ($>10,000$ objects) in a reasonable time frame. Speed is therefore more important than visual quality.

5 CONCEPT
The proposed pipeline as depicted in Figure 1 starts with a closed 3D triangle mesh together with a user-defined primitive frequency distribution $H_p$ as input and
results in a CSG tree together with a set of primitives and their parameters as output. Each of the following sub sections is dedicated to a particular pipeline step.

5.1 Sampling
In this step, the input mesh is sampled resulting in a 3D point cloud $S$. The sample points are later used to measure how well a CSG tree matches the mesh’s shape. Each point in $S$ receives a label that indicates whether the point is located inside, outside or on the surface of the mesh. See Figure 2 for an example.

The points and their corresponding assigned labels are retrieved using a special raycasting approach, in which rays do not detect meshes for which the origin of the raycast lies inside the mesh: First, random points within the bounding box of the mesh are selected. These points serve as origins for rays that are cast in random directions. If a ray hits the mesh, the hit point $p_{hit}$ is added to $S$ with an "on-surface" label. If no mesh was hit during the raycast, then any point along that cast outside the bounding box can be marked as $p_{hit}$ with an "outside" label. In addition, the ray's origin $p_{org}$ is added to $S$. Its corresponding label ("inside", "outside") is determined by casting a second ray back from $p_{hit}$ through $p_{org}$ and comparing the lengths of both rays. If the first ray is longer, then the label "inside" is assigned, if it is shorter, the label "outside". In case of equal ray lengths the point is marked as "on-surface". If $p_{hit}$ and $p_{org}$ are the same, the origin is not added to $S$. The label of $p_{org}$ is therefore always uniform regardless of the initial direction of the raycast.

5.2 Clustering
In order to reduce the computational complexity of primitive fitting and CSG tree extraction, a two-level clustering approach is applied to the point cloud $S$ ("inside", "outside" and "on-surface" points are considered). First, it is clustered into a set of $n_c$ clusters using the k-means algorithm [Llo82]. Then, each cluster is again clustered into $m_c$ sub-clusters using the same technique. The method results in a set of $n_c \cdot m_c$ sub-clusters $C_{sub}$, which is then the basis for primitive fitting. Please note that $m_c$ and $n_c$ are user-controlled parameters.

5.3 Primitive Fitting
For each sub-cluster $c_{sub}$ in $C_{sub}$, a single primitive is fitted. The primitive type is determined by sampling the user-defined primitive frequency distribution $H_p$, which assigns a probability to each supported primitive type (sphere, box, cylinder, torus, cone, extruded pentagon and triangle). This results in the primitive set $P_{sub}$. See Algorithm 1 for all details.

Important to note is that the primitive fitting method does not strive for high matching precision but for fast running times and high geometric variation.

5.4 Per-Cluster CSG Tree Extraction
For each cluster $c_i$ in $C$, all sub-clusters are merged resulting in a set of primitives $P_i$ and points $S_i$. The CSG tree extraction process is conducted for each cluster in
Algorithm 1: The primitive fitting algorithm. The method createP(·) generates a primitive instance based on a primitive type (e.g. sphere or box), a center position and a size value. The size value for a sphere would be its radius. For primitives with more than a single size dimension (e.g. boxes), each dimension is set to the size value.

input : Set of sub-clusters $C_{\text{sub}}$, primitive frequency distribution $H_p$
output: Fitted primitive for each sub-cluster $\in C_{\text{sub}}$

$P_{\text{sub}} \leftarrow \{\}$

foreach $c_{\text{sub}} \in C_{\text{sub}}$ do

$\begin{align*}
    d_{\text{min}} &\leftarrow \text{minimum of the largest distances per axes in } c_{\text{sub}} \\
    d_{\text{max}} &\leftarrow \text{diameter}(c_{\text{sub}}) \\
    \text{size}_p &\leftarrow \text{random}(d_{\text{min}}, d_{\text{max}} \cdot 0.5) \\
    \text{center}_p &\leftarrow \text{center}(c_{\text{sub}}) \\
    \text{type}_p &\sim H_p \\
    P_{\text{sub}} &\leftarrow P_{\text{sub}} \cup \text{createP(type}_p, \text{center}_p, \text{size}_p) \\
\end{align*}$

return $P_{\text{sub}}$

C and uses a Genetic Algorithm to solve the CSG tree extraction problem.

Initialization. The GA initializes the CSG tree population with randomly generated CSG trees that use primitives from $P_t$. This is done exactly once while the following steps are executed repeatedly.

Ranking. All CSG trees in a population are ranked using the objective function

$$E(t) = \sigma(t) \cdot \sum_{j=1}^{|S_j|} \begin{cases} f_i(s_{ij}) & l(s_{ij}) = "\text{on-surf.}" \\
    \min(f_i(s_{ij}), 0) & l(s_{ij}) = "\text{outside}" \\
    \max(f_i(s_{ij}), 0) & l(s_{ij}) = "\text{inside}" \end{cases}$$

where $\sigma(t) = \log_2(\text{size}(t))$ is a tree size penalty term, $f_i(·)$ is the signed distance function of tree $t$ and $l(·)$ assigns a label $\in \{"\text{on-surface}", "\text{outside}", "\text{inside}"\}$ to each point in $S_i$. The GA-based CSG tree extraction aims for minimizing Equation 1. See Figure 4 for getting an intuition of the objective function.

Selection. After the whole population was ranked and sorted in ascending order, the best (with respect to their objective function value as defined in Equation 1) $n_t$ CSG trees are selected for the steps “Enhancement”, “Crossover” and “Mutation” (collectively referred to as variation steps in the following). Variation steps are applied to each of the $n_t$ selected CSG trees $m_t$ times, resulting in a constant population size of $n_t \cdot m_t$ CSG trees (see Figure 1).

Enhancement. The idea of the enhancement operator is to improve the geometry score before the actual classic variation operators (mutation, crossover) are applied. It can be seen as a local hill-climbing strategy for faster convergence.

It works as follows: Each CSG tree in the population stores the sample point $p_w$ with the worst objective function value. The enhancement operator modifies the CSG tree in the following way: If $l(p_w) = "\text{outside}"$, i.e., the sample point $p_w$ should be outside, but -as it has a bad objective value- is wrongfully placed inside, a randomly generated primitive is cut out by adding it to the CSG tree together with a difference operation. If $l(p_w) = "\text{inside}"$, then a randomly generated primitive is added to the CSG tree together with a union operation. In case of $l(p_w) = "\text{on surface}"$, a randomly generated primitive is either cut out or added to the solid induced by the CSG tree. In this case, operation choice (cut out or add) is random with both operations having a probability of 0.5.

Crossover & Mutation. The currently used crossover and mutation operators for a particular CSG tree are chosen randomly and applied to the CSG tree $m_t$ times together with the enhancement operator. Each combined enhancing, crossover and mutation operation results in a single new CSG tree. See Figure 5 and Figure 6 for an overview and description of used crossover and mutation operators.

Termination. In order to determine whether or not the GA should terminate, the average score of improvement $\delta_k$ for GA iteration $k$

$$\delta_k = \frac{1}{n_b} \left[ \min_{0 \leq j \leq k} \left( \sum_{t \in T_i} E(t) \right) - \sum_{t \in T_i} E(t) \right]$$

is evaluated, where $T_i$ is the set of the $n_b$ best CSG trees in the population of iteration $k$ and $\min_{0 \leq j \leq k}(·)$ is the best accumulated score reached so far in that particular cluster. $\delta_k$ is then compared to a pre-defined average score threshold $\delta^{av}_k$ which is multiplied by the best average score reached. This checks whether $\delta_k$ has improved by a certain amount $\delta^{av}_k$ compared to the current
(a) Crossover 1: Merges the first tree with a randomly chosen sub-tree from the second tree.

(b) Crossover 2: Replaces a proper, randomly chosen sub-tree from one tree with a randomly chosen, not necessarily proper sub-tree (the sub-tree could also be the entire CSG tree), from the second tree and adds a randomly chosen operation (union, intersection or difference) to the parent of the sub-tree of the first tree.

(c) Crossover 3: Selects a sub-tree randomly from both trees and combines them arbitrarily using the union, intersection or difference operator.

Figure 5: All used crossover operators.

best average score. If this is not the case, an iteration counter is incremented. If the number of iterations surpasses a user-controlled upper bound, the algorithm terminates.

5.5 Merge

The result of the steps described in Section 5.4 is a CSG tree for each cluster in C. In order to combine these per-cluster trees into a single one representing the complete model, we apply a hierarchical merge scheme based on the nearest neighbor information obtained by the clustering mechanism. See Figure 7 for an explanation of the algorithm by example. The proposed merge procedure combines per-cluster trees that are close by, which has positive effects on tree editability and tree balance.

5.6 Blending

For certain use cases (e.g. artful model abstraction), additional blending of neighboring per-cluster CSG trees can be applied using the blending operator [Ric73]

\[ b(f_{t1}, f_{t2}, x) = f_{t2}(p) \cdot (1 - h) + f_{t1}(p) \cdot h - \alpha \cdot h \cdot (1 - h) \]

where \( f_{t1} \) and \( f_{t2} \) are the signed distance functions of the two trees that should be blended together, \( h = \min(1, \max(0, 0.5 + \frac{f_{t2}(x) - f_{t1}(x)}{2 \alpha})) \) and \( \alpha \) is a user-defined parameter controlling blending smoothness.
See Figure 15 and 16 for a visualization of different smoothness strengths. Note that a specific $\alpha$-value is not given since it depends on model size and thus is not generalizable.

6 EVALUATION

The evaluation consists of two parts: The first part describes and explains the performance characteristics of the generator framework and the second part details a possible use case.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_t$</td>
<td>15</td>
</tr>
<tr>
<td>$m_t$</td>
<td>70</td>
</tr>
<tr>
<td>$n_b$</td>
<td>15</td>
</tr>
<tr>
<td>$H_p$</td>
<td>uniform distribution</td>
</tr>
<tr>
<td>$\delta_E$</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 1: Parameters used throughout the evaluation.

6.1 Generator Framework

The generator framework was evaluated using a machine with an Intel(R) Core(TM) i7 CPU @ 3.06GHz and 12GB of RAM. Experiments were conducted with a varying number of clusters and different sample point cloud sizes with seven 3D models taken from the Google Poly data set [pol] (see Figure 14 for an overview). See Table 1 for a list of parameter values that were used throughout the evaluation and Figure 13 for an exemplary CSG tree result.

For a meaningful quality evaluation, an extra sampling of the triangle mesh’s surface with a fixed number of samples is conducted. This point set is then used to evaluate the objective function for a specific CSG tree resulting in its geometry score.

6.1.1 Number of Clusters

The impact of the total number of sub-clusters $|C_{sub}| = n_c \cdot m_c$ for a fixed $|S| \approx 2000$ on the running time is shown in Figure 8 ($n_c = 5$, $m_c \in \{1,2,...,7\}$). It is clearly visible that running time and $|C_{sub}|$ have an approximate linear relationship. The number of fitted primitives (which is equal to $|C_{sub}|$) positively affects the quality of the model approximation. As visible in Figure 9, this effect weakens significantly starting from $|C_{sub}| = 15$. This is a good hint for a running time/quality trade-off. Note that geometry scores are not normalized and thus an inter-model comparison is not possible. Figure 15 and 16 show results for models Elephant and Giraffe for different values of $|C_{sub}|$.

6.1.2 Point Cloud Size

We evaluated the impact of the size of the sampling point cloud $|S|$ on the running times and result quality using all seven models and 25 sub-clusters ($n_c = 5$, $m_c = 5$). Results are depicted in Figure 10 and 11.

As expected, running times grow linearly with the number of sampling points. The results also show a significant improvement of result quality until a sampling point cloud size of ca. 1000 – 1500 points. This can be explained by the geometric complexity of the input triangle mesh: There is no positive effect if more samples are used than are required for the representation of the input mesh in all its details.

Figure 8: $|C_{sub}|$ and running times.

Figure 9: $|C_{sub}|$ and reached geometry score (smaller score means better quality).

Figure 10: Sample point cloud size and running times.
6.2 Deep Primitive Segmentation

We tested the usability of our generator framework in an evaluation of three different neural network architectures for point cloud segmentation (PointNet [QSMG16], PointNet++ [QYSG17] and PointCNN [LBSC18]). Our main interest was their performance (Accuracy and Intersection over Union (IoU)) in the task of primitive segmentation (cuboids, spheres, cylinders and cones) as well as their robustness to noise and differences between training and test sets.

The training set consisted of 15,000 point clouds, each representing 20 – 80 primitives using 2048 points. In order to evaluate the networks’ performance, its robustness and ability to generalize, we used six different test data sets, each containing 500 point clouds:

- **No Rotation**: Generated with the same parameters as the training set.
- **Rotation**: Same as *no rotation* but with additional random rotation.
- **Reduced Points**: Same as *no rotation* but containing only 50% of the points.
- **Noise (low/high)**: Same as *no rotation* but with added low ($\mu = 0, \sigma = 0.5$, *noise low*) and high ($\mu = 0, \sigma = 1.5$, *noise high*) Gaussian noise.
- **More Primitives**: Same as *no rotation* but with an increased minimum primitive count of 40.

The data generation was fully automated and took 5 days on the system described in Section 6.1. Please note that instead of the usual generator output (a CSG tree), a point cloud sampled from it was used here.

Table 2 lists the results. PointNet++ performs best by far on all test data sets which makes it the preferred candidate for further research on enhanced primitive fitting pipelines. Figure 12 shows the result of an exemplary point cloud segmentation for all evaluated networks.

<table>
<thead>
<tr>
<th>Test set</th>
<th>Metric</th>
<th>PointNet</th>
<th>PointNet++</th>
<th>PointCNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>No rotation</td>
<td>Acc</td>
<td>42.91</td>
<td>99.46</td>
<td>36.32</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>18.66</td>
<td>98.73</td>
<td>15.28</td>
</tr>
<tr>
<td>Rotation</td>
<td>Acc</td>
<td>39.15</td>
<td>93.41</td>
<td>34.41</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>16.27</td>
<td>80.38</td>
<td>14.55</td>
</tr>
<tr>
<td>Red. points</td>
<td>Acc</td>
<td>42.83</td>
<td>99.45</td>
<td>34.25</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>18.59</td>
<td>98.34</td>
<td>13.51</td>
</tr>
<tr>
<td>Noise low</td>
<td>Acc</td>
<td>42.88</td>
<td>99.47</td>
<td>35.23</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>18.70</td>
<td>98.65</td>
<td>14.80</td>
</tr>
<tr>
<td>Noise high</td>
<td>Acc</td>
<td>42.72</td>
<td>99.30</td>
<td>35.23</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>18.67</td>
<td>97.54</td>
<td>15.06</td>
</tr>
<tr>
<td>More prim.</td>
<td>Acc</td>
<td>43.21</td>
<td>99.60</td>
<td>42.16</td>
</tr>
<tr>
<td></td>
<td>IoU</td>
<td>19.32</td>
<td>99.43</td>
<td>19.88</td>
</tr>
</tbody>
</table>

A possible explanation for the very strong performance of PointNet++ might be its approach of separating the point cloud into local regions first. Using a density adaptive layer, hierarchical features are then extracted in a next step. This seems to be working especially well for the task of classifying single, isolated primitives which are located in regions of otherwise low point density. This is a task in which the other networks show particularly low performance. The approach PointNet++ takes also proves to be robust against noise and it shows good generalization performance when the test set differs from the training set.

7 CONCLUSION & FUTURE WORK

In this paper, we described a flexible and fast generator framework for creating primitive-based abstractions of triangle meshes. The proposed pipeline’s output is a CSG tree which is extracted by a Genetic Algorithm. As an intended use case, we presented an evaluation of deep primitive segmentation networks which use training and test sets created by the generator.
Next steps include the integration of PointNet++ as a semantic clustering tool, replacing the currently used k-means approach. In addition, more sophisticated methods for primitive fitting like RANSAC [SWK07] could further improve resulting visual quality.

8 REFERENCES


[ZYY+17] Chuhang Zou, Ersin Yumer, Jimei Yang, Duygu Ceylan, and Derek Hoiem. 3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks.
Figure 14: All used models (©Poly by Google, CC-BY-License).

Figure 15: Extracted CSG tree models of the elephant model with different cluster sizes and blending strengths.

Figure 16: Extracted CSG tree models of the giraffe model with different cluster sizes and blending strengths.