Supplementary Material for Generating Part-Aware Editable 3D Shapes without 3D Supervision

Konstantinos Tertikas[*1,3] Despoina Paschalidou[2] Boxiao Pan[2] Jeong Joon Park[2]
Mikaela Angelina Uy[2] Ioannis Emiris[3,1] Yannis Avrithis[4] Leonidas Guibas[2]
[1]National and Kapodistrian University of Athens [2]Stanford University
[3]Athena RC, Greece [4]Institute of Advanced Research in Artificial Intelligence (IARAI)

Abstract

*In this **supplementary document**, we first present a detailed overview of our network architecture, the training and generation procedure, in Sec. 1. We then provide ablations on how different components of our pipeline impact the performance of our model, in Sec. 2. Subsequently, in Sec. 3 we provide additional results both on the shape generation task as well as on the shape editing task. Next, we demonstrate various applications of our model such as shape and image inversion, in Sec. 4 and Sec. 5, respectively. Finally, in Sec. 8, we analyse the limitations, future research directions and in Sec. 9 we discuss potential negative impact of our method on society.*

1. Implementation Details

In this section, we provide a detailed description of the several components of our network architecture (Sec. 1.1). Next, we describe our training procedure (Sec. 1.2) and the generation protocol (Sec. 1.3). Then, we provide details regarding the various editing operations demonstrated in the main submission (Sec. 1.4). Finally, we detail our metrics computation (Sec. 1.5) and discuss our baselines (Sec. 1.6).

1.1. Network Architecture

Here, we describe the architecture of each individual component of our model, as illustrated in Fig. 2 in the main submission. As already discussed in Sec 3.3 of our main submission, we implement our generative model using an auto-decoder [3, 29]. In particular, the input to our model is a set of embedding vectors $\mathcal{Z} = \{\mathbf{z}_i\}_{i=1}^{n}$, uniquely identifying each one from the n training samples. Each $\mathbf{z}_i = \{\mathbf{z}^s, \mathbf{z}^t\}$, comprises two learnable embeddings that capture the shape and appearance properties of each sample. Note that both the shape $\mathbf{z}^s$ and texture $\mathbf{z}^t$ codes are initialized by sampling from the normal distribution $\mathbf{z}^s, \mathbf{z}^t \sim N(\mathbf{0}, I)$. To avoid notation clutter, we omit the object index i in the rest of this section.

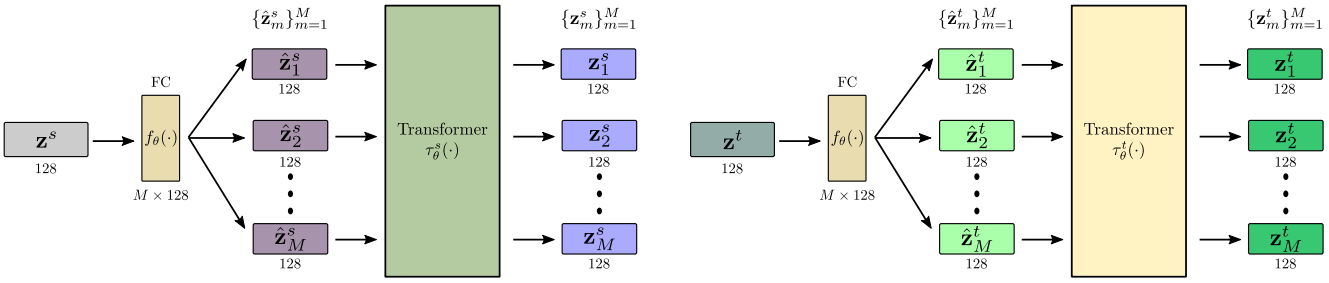
Our network consists of three main components: (i) the *decomposition network* that maps the instance-specific latent codes $\{\mathbf{z}^s, \mathbf{z}^t\}$ to M per-part shape $\{\mathbf{z}^s_m\}_{i=1}^{M}$ and texture $\{\mathbf{z}^t_m\}_{i=1}^{M}$ embeddings, (ii) the *structure network* that takes the per-part shape code $\mathbf{z}^s_m$ and predicts an affine transformation $\{\mathbf{R}_m, \mathbf{t}_m\}$ and scale $\mathbf{s}_m$ that controls the pose and the area of influence of this part, and (iii) the *neural rendering module* that renders a 2D image using M NeRFs. As supervision, we use the observed RGB color $C(r) \in \mathbb{R}^3$ and the object mask $I(r) \in \{0, 1\}$ for each ray $r \in \mathcal{R}$. We also associate r with a binary label $\ell_r = I(r)$, as shown in Fig. 5. Namely, we label a ray r as *inside*, if $\ell_r = 1$ and *outside* if $\ell_r = 0$.

Decomposition Network: The decomposition network maps the instance-specific shape and texture $\{\mathbf{z}^s, \mathbf{z}^t\}$ latent codes into a set of M latent codes that control the shape and appearance of each part. We implement the decomposition network using two multi-head attention transformers without positional encoding [38], τ^s_θ and τ^t_θ, that predict M shape $\{\mathbf{z}^s_m\}_{m=1}^{M}$ and texture $\{\mathbf{z}^t_m\}_{m=1}^{M}$ codes, as follows:

$$\begin{aligned} \tau^s_\theta &: \mathbb{R}^{L_d} \rightarrow \mathbb{R}^{L_d \times M} \quad \{\mathbf{z}^s_m\}_{m=1}^{M} = \tau^s_\theta(f_\theta(\mathbf{z}^s)) \\ \tau^t_\theta &: \mathbb{R}^{L_d} \rightarrow \mathbb{R}^{L_d \times M} \quad \{\mathbf{z}^t_m\}_{m=1}^{M} = \tau^t_\theta(f_\theta(\mathbf{z}^t)) \end{aligned} \tag{1}$$

[*]Work done during internship at Stanford.

where L_d is the dimensionality of the instance-specific embeddings and is set to 128. The input to transformer τ_θ^s is the set of M per-part shape codes $\{\hat{\mathbf{z}}^s\}_{m=1}^M$, where $\hat{\mathbf{z}}_m^s \in \mathbb{R}^{128}$. Likewise, the input to transformer τ_θ^t is the set of M per-part texture codes $\{\hat{\mathbf{z}}^t\}_{m=1}^M$, where $\hat{\mathbf{z}}_m^t \in \mathbb{R}^{128}$. Note that the M shape $\{\hat{\mathbf{z}}^s\}_{m=1}^M$ and texture $\{\hat{\mathbf{z}}^t\}_{m=1}^M$ codes are produced from $\mathbf{z}_s$ and $\mathbf{z}_t$ respectively using M linear projections $f_\theta(\cdot)$. The output of each transformer is the set of per-part shape $\{\mathbf{z}_m^s\}_{i=1}^M$ and texture $\{\mathbf{z}_m^t\}_{i=1}^M$, where $\mathbf{z}_m^s \in \mathbb{R}^{128}$ and $\mathbf{z}_m^t \in \mathbb{R}^{128}$. Both transformers consist of 2 layers with 4 attention heads. The queries, keys and values have 128 dimensions and the intermediate representations for the MLPs have 1024 dimensions. To implement the transformer architecture we use the transformer library provided by Wightman [40][1].



(a) $\tau_\theta^s(\cdot)$ predicts the M per-part shape codes $\{\mathbf{z}_m^s\}_{i=1}^M$.

(b) $\tau_\theta^t(\cdot)$ predicts the M per-part shape codes $\{\mathbf{z}_m^t\}_{i=1}^M$.

Figure 1. **Decomposition Network.** The decomposition network consists of two two multi-head attention transformers, τ_θ^s and τ_θ^t, without positional encoding, that map the object-specific shape and texture embeddings $\mathbf{z}^s, \mathbf{z}^t$ to M learnable codes $\{\mathbf{z}_m^s\}_{m=1}^M$ and $\{\mathbf{z}_m^t\}_{m=1}^M$ that control the shape and texture of each part respectively.

Structure Network: The structure network learns a function $s_\theta : \mathbb{R}^{L_d} \rightarrow \mathbb{R}^4 \times \mathbb{R}^3 \times \mathbb{R}^3$ that maps the per-part shape code $\mathbf{z}_m^s$ to an affine transformation, defined through a rotation matrix $\mathbf{R}_m \in SO(3)$ and a translation vector $\mathbf{t}_m \in \mathbb{R}^3$, and a scale vector $\mathbf{s}_m \in \mathbb{R}^3$. We follow [30] and parametrize the rotation matrices using quaternions [13], whereas the translation and the scale vectors are represented using 3 scalars along the axes. Specifically the per-part $\mathbf{R}_m, \mathbf{t}_m$ and $\mathbf{s}_m$ are predicted from an MLP with shared weights across parts. A pictorial representation for the structure network is provided in Fig. 2. Starting from the shape latent code for part m, we feed it to a linear layer with 128 hidden dimensions that predicts 4 values

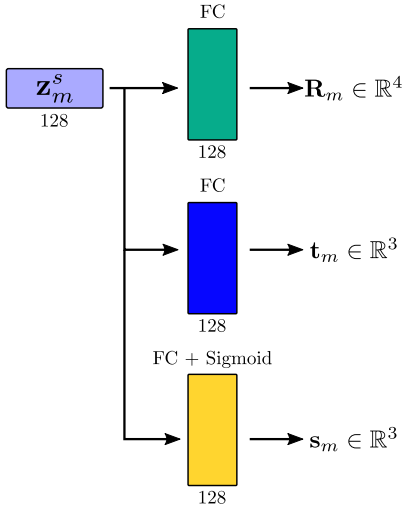


Figure 2. **Structure Network.** The structure network s_θ maps the shape latent code $\mathbf{z}_m^s \in \mathbb{R}^{L_d}$ to a translation vector $\mathbf{t}_m$, a rotation matrix $\mathbf{R}_m$ and a scale vector $\mathbf{s}_m$, using an MLP with shared weights across parts.

that define the rotation matrix $\mathbf{R}_m$ as a quaternion. Similarly, we feed $\mathbf{z}_m^s$ to another another linear layer with 128 hidden dimensions that predicts the 3 values that define the translation vector $\mathbf{t}_m$ of the m-th part. Finally, to predict the scale vector $\mathbf{s}_m$, we use a linear layer with 128 hidden dimensions, followed by a sigmoid activation function. As discussed in Sec. 3.2 of

[1] https://github.com/rwightman/pytorch-image-models

our main submission, we associate every ray with a specific part and transform it to its coordinate system using the predicted rotation and translation matrices. To transform a 3D point $\mathbf{x}$ to the local coordinate system of the m-th part, we simply apply $\mathcal{T}_m(\mathbf{x}) = \mathbf{R}_m(\mathbf{x} + \mathbf{t}_m)$, whereas to transform the ray direction $\mathbf{d}$ it simply suffices to multiply it with the rotation matrix $\mathbf{R}_m$. The scale vector $\mathbf{s}_m$ represents the spatial extent of each part. To ensure that each part captures continuous regions of the object, we multiply its occupancy function with the occupancy function of an axis-aligned 3D ellipsoid centered at the origin of the coordinate system, with axis lengths given by the scale vector $\mathbf{s}_m$. Specifically the parametric function of the m-th ellipsoid is $f_\theta^m(\mathbf{x}) = f(\mathcal{T}_m(\mathbf{x}), \mathbf{s}_m)$, where

$$f(\mathbf{x}, \mathbf{s}) = \left\| \mathrm{diag}(\mathbf{s})^{-1}\mathbf{x} \right\|^2, \tag{2}$$

is less than 1 when a 3D point $\mathbf{x}$ is inside the ellipsoid, greater than 1 when $\mathbf{x}$ is outside and equal to one when $\mathbf{x}$ is on the surface. We convert the parametric function of each ellipsoid to an occupancy function $g : \mathbb{R}^3 \rightarrow [0, 1]$ with a sigmoid function $\sigma(\cdot)$, as follows:

$$g(\mathbf{x}, \mathbf{s}) = \sigma\left(\beta\left(1 - f(\mathbf{x}, \mathbf{s})\right)\right), \tag{3}$$

where $\beta = 100$ controls the sharpness of the transition of the occupancy function. As already discussed in Sec. 3.2 of our main submission, note that before estimating the occupancy function of the m-the ellipsoid $g_\theta^m(\mathbf{x}) = g(T_m(\mathbf{x}), \mathbf{s}_m)$, we first transform $\mathbf{x}$ to the coordinate frame of the m-th part. This ensures that any transformation of the part, also transforms its ellipsoid.

Neural Rendering: The last component of our pipeline performs volumetric rendering using M NeRFs using the assignment of rays to parts from Eq. 9 in our main submission and the per-part rendering equation

$$\hat{C}_m(r) = \sum_{i=1}^{N} h_\theta^m(\mathbf{x}_i^r) \prod_{j<i} \left(1 - h_\theta^m(\mathbf{x}_i^r)\right) c_\theta^m(\mathbf{x}_i^r, \mathbf{d}^r), \tag{4}$$

as follows:

$$\hat{C}(r) = \sum_{m=1}^{M} \mathbb{1}_{r \in \mathcal{R}_m} \hat{C}_m(r), \tag{5}$$

where $\hat{C}(r)$ is the predicted color value for ray r and $h_\theta^m(\mathbf{x}) = o_\theta^m(\mathbf{x}) g_\theta^m(\mathbf{x})$ is the joint occupancy function of the m-th part. Namely, we use the m-th NeRF to render ray r if it is assigned to the m-th part. Whereas, if a ray is not associated with any part its color is simply black, namely $\hat{C}(r) = 0$.

We follow [28] and employ two separate networks: an *occupancy network* $o_\theta(\cdot)$ and a *color network* $c_\theta(\cdot)$ to predict the color and occupancy values. The occupancy network o_θ maps a 3D point $\mathbf{x}$ and the shape code $\mathbf{z}_m^s$ to an occupancy value o. For the m-th part, the predicted occupancy value of a 3D point $\mathbf{x}$ is defined as:

$$o_\theta : \mathbb{R}^3 \times \mathbb{R}^{128} \rightarrow [0, 1] \qquad o = \sigma(\tau o_\theta^m(\mathcal{T}_m(\mathbf{x}), \mathbf{z}_m^s)), \tag{6}$$

where $\sigma(\cdot)$ is the sigmoid function and τ is a hyperparameter that controls the sharpness of the transition of the occupancy function and is set to 100. Likewise, the color network c_θ maps the 3D point $\mathbf{x}$ and the viewing direction $\mathbf{d}$ after applying positional encoding on its components, as well as the shape code $\mathbf{z}_m^s$ and the texture code $\mathbf{z}_m^t$ into a color value $\mathbf{c} \in \mathbb{R}^3$. For the m-th part the predicted color value for a 3D point $\mathbf{x}$ along a ray with direction d can be defined as:

$$c_\theta : \mathbb{R}^{L_x} \times \mathbb{R}^{L_d} \times \mathbb{R}^{128} \times \mathbb{R}^{128} \rightarrow \mathbb{R}^+ \qquad \mathbf{c} = c_\theta^m(\gamma(\mathcal{T}_m(\mathbf{x})), \gamma(\mathbf{R}_m\mathbf{d}), \mathbf{z}_m^s, \mathbf{z}_m^t). \tag{7}$$

Here, $\gamma(\cdot)$ is the positional encoding of [26] that is applied element-wise as follows

$$\gamma(p) = [p; \sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)], \tag{8}$$

where $p \in \mathbb{R}$ can be any dimension of the input point $\mathbf{x}$ and the corresponding ray direction $\mathbf{d}$ and $[\cdot;\cdot]$ denotes concatenation. In our experiments, we set $L = 10$, hence $L_x = L_d = 3(2L + 1) = 63$.

Occupancy Network: The occupancy network is implemented using 2 residual blocks as shown in Fig. 3 and is similar to [25]. The input to our occupancy network is the per-part shape latent code $\mathbf{z}_m^s \in \mathbb{R}^{128}$ and a batch of 3D points sampled along rays that are transformed to the local coordinate system of the m-th part. Note that we do not apply positional encoding on

the input as we empirically observed that projecting the 3D points into a higher dimensional space using (8) does not improve our model's performance. The input points are passed through a fully-connected layer that produce a 128-dimensional feature vector for each point. This feature vector, together with the per-part shape code $\mathbf{z}_m^s$, is then passed to 2 residual blocks. Each residual block first applies Conditional Scaling and Translation (CST) to the input features followed by a ReLU. The output is then fed into a fully-connected layer, a second CST, a ReLU activation and another fully-connected layer. The output of this operation is then added to the input of the residual block. The Conditional Scaling and Translation layer (CST) is the same as Conditional Batch-Normalization (CBN) [7], using in [25], but we replace the normalization layer with the identity function. The output of the two residual blocks is fed to a fully-connected layer that produces a 257-dimensional output. This is then split into a 1-dimensional feature vector that we use to compute the occupancy value, simply by passing it to a sigmoid function, as defined in (6), and a 256-dimensional feature vector, which we pass to the color network that produces the corresponding color.

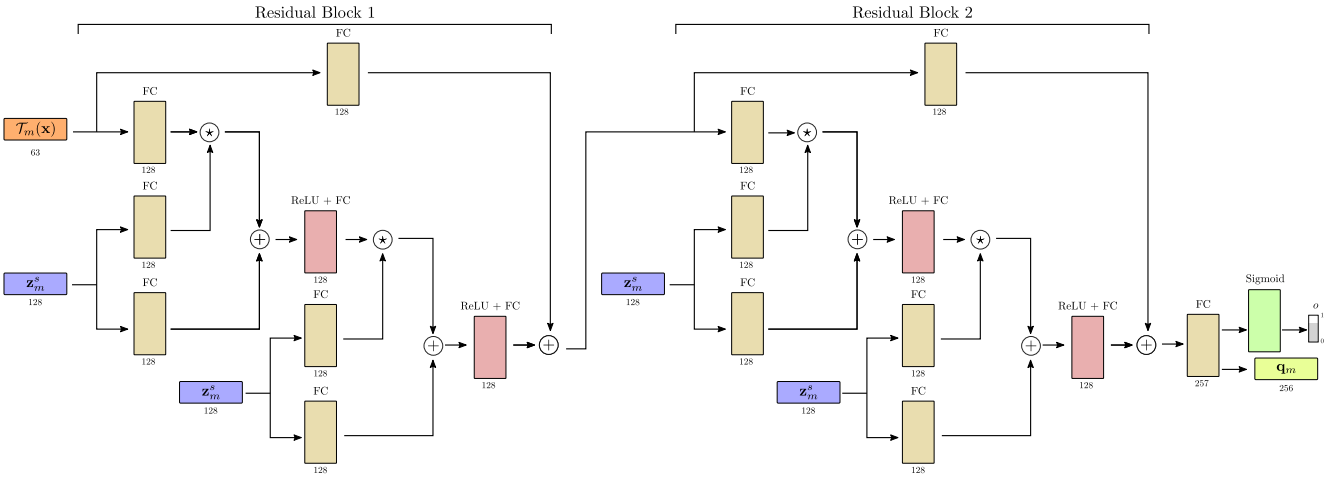


Figure 3. **Occupancy Network.** The occupancy network $o_\theta(\cdot)$ maps a 3D point $\mathbf{x}$ transformed in the local coordinate system of the m-th part and its shape code $\mathbf{z}_m^s$ to an occupancy value $o \in [0, 1]$ and an intermediate feature vector $\mathbf{q}_m \in \mathbb{R}^{256}$ which is passed to the color network $c_\theta(\cdot)$.

Color Network: The color network is implemented using 3 residual blocks, as shown in Fig. 4. Each residual block [15] is implemented as a 3-layer MLP with ReLU non-linearities. The hidden dimensions of the 3 linear layers of each block are: $510, 256, 256$ for the first, $256, 256, 256$ for the second and $256, 128, 64$ for the third. The inputs to the color network are: the per-part texture code $\mathbf{z}_m^t \in \mathbb{R}^{128}$, the 3D point $\gamma(\mathcal{T}_m(\mathbf{x})) \in \mathbb{R}^{63}$ and the ray direction $\gamma(\mathbf{R}_m\mathbf{d}) \in \mathbb{R}^{63}$ transformed in the coordinate system of the m-th part and projected in a higher dimensional space using (8), and the intermediate feature representation $\mathbf{q}_m \in \mathbb{R}^{256}$ that is predicted from the occupancy network $o_\theta(\cdot)$. We concatenate all inputs and produce a 510-dimensional feature vector. Each residual block first applies a linear projection to the current feature vector, followed by a ReLU activation. The output is then fed into a fully-connected layer, a ReLU activation and another fully-connected layer followed by a ReLU non linearity. The output of this operation is then added to the input of the residual block after feeding it to a fully connected layer. To predict the color, we use one linear layer with hidden size 64 and output size 3, followed by a sigmoid activation.

1.2. Training Protocol

In all our experiments, we use the Adam optimizer [20] with batch size of 32 and a learning rate that begins at $\eta = 5 \times 10^{-4}$ and decays to $\eta = 5 \times 10^{-6}$ over the course of optimization. We employ a cosine annealing learning rate schedule with warm up that is set to 500 steps. For the other hyperparameters of Adam, we use the PyTorch [33] defaults ($\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^7$) and we have no weight decay. Furthermore, our input images and object masks are rendered at 256^2 resolution and we approximate each image with 512 rays. Specifically, we adopt a similar strategy as [43] and sample an equal number of rays inside and outside the object mask, as shown in Fig. 5, during training. We observe that this sampling strategy results in faster convergence and prevents several local minima. In addition, during training, we sample 64 random points along each ray, whereas during inference, we sample 128 points. Note that unlike [26], we do not consider a coarse and fine volume of point coordinates along rays. While, we anticipate that this would improve the quality of our renderings, we were not able to

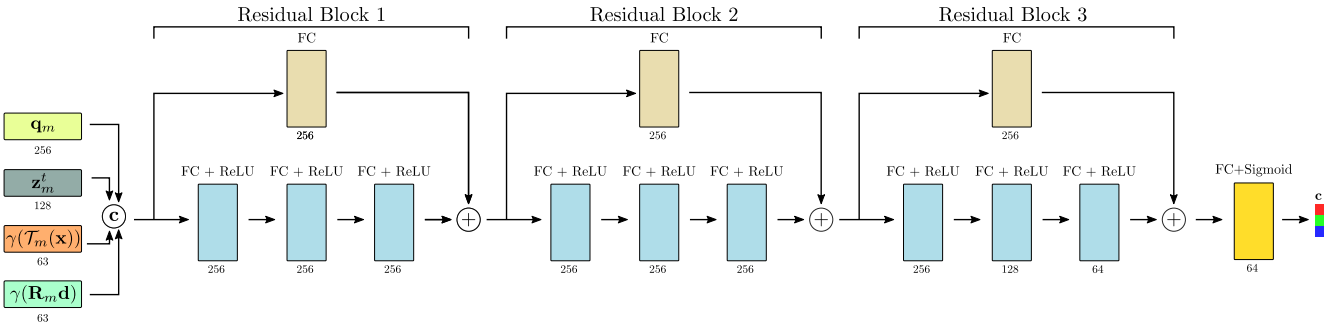


Figure 4. **Color Network.** The color network $c_\theta(\cdot)$ maps a 3D point $\mathbf{x}$ and a viewing direction $\mathbf{d}$ after applying element-wise positional encoding on their components, the texture code $\mathbf{z}^t_m$ and the feature vector $\mathbf{q}_m$ into a color value $\mathbf{c} \in \mathbb{R}^3$. Note that $\mathbf{q}_m$ is predicted from the occupancy network $o_\theta(\cdot)$ from $\mathbf{z}^s_m$.

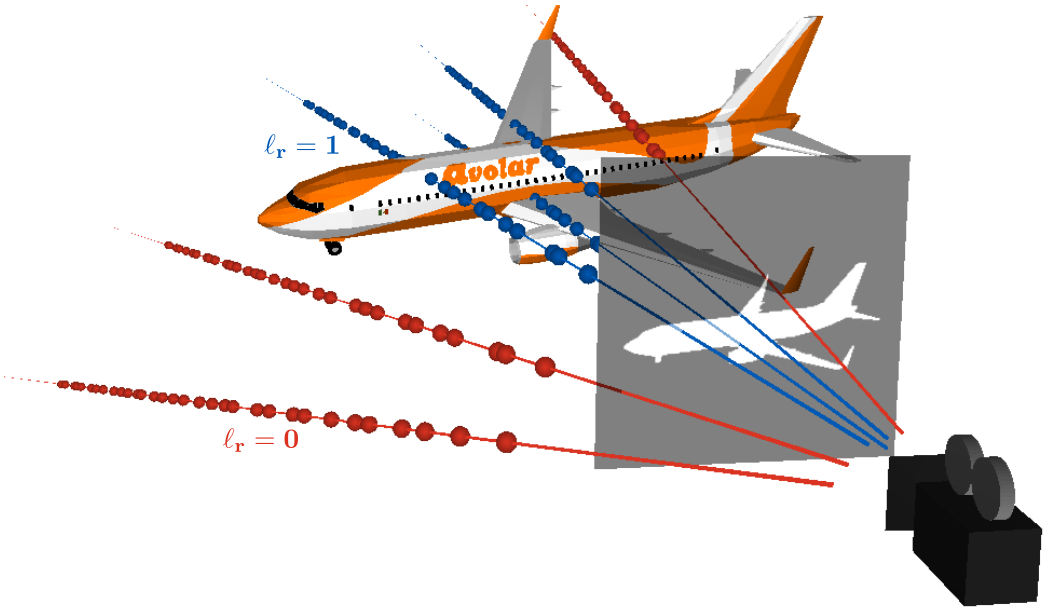


Figure 5. **Inside-Outside Rays.** Inside rays correspond to pixels within the object mask (*i.e.* white pixel), whereas outside rays correspond pixels outside the object mask (*i.e.* black pixel).

adopt this sampling strategy due to our limited computational resources, as it would significantly increase the computational requirements of our model. Finally, we train our model with 16 parts, for each object category, for approximately 125k iterations. All our experiments were conducted on a single NVIDIA RTX 3090, with 24GB and training per category takes approximately 1 week.

We weigh the loss terms of Eq. 14 in our main submission with $1.0, 1.0, 1.0, 0.01, 0.01, 1.0, 0.0001, 0.0001$. Note that we use smaller weights for $\mathcal{L}_{overlap}(\mathcal{R})$ and $\mathcal{L}_{cov}(\mathcal{R})$, as they act as regularizers on the recovered parts and we want our model to focus primarily on generating part arrangements that can render plausible objects from any novel camera. For $\mathcal{L}_{occ}(\mathcal{R})$, we weight the two terms of Eq. 18 in our main submission, by $0.1, 0.01$. We empirically observe that having both terms when computing the occupancy loss significantly helps performance. Finally, we weigh the control loss $\mathcal{L}_{control}$ with 1 as it takes relatively smaller values and weighting it with a smaller number would make its contribution insignificant. The impact of each term is discussed in detail in Sec. 2.1. In addition, the k term of $\mathcal{L}_{cov}(\mathcal{R})$ is set to 16, since we have 256 rays and 16 parts. This encourages that each part will "cover" approximately the same number of rays, hence parts would uniformly cover the generated shape. Having uniformly distributed parts across the generated shape is essential for our application, as we want to enable uniform control over the generated shape. Finally, we set parameter λ of $\mathcal{L}_{overlap}(\mathcal{R})$ to 3 as we empirically observe that it leads to good performance. Note that setting $\lambda = 1$ would enforce that each ray can intersect, *i.e.* be predicted

as internal, only with one part, thus yielding completely disjoint parts. However, this is quite limiting when trying to generate complex geometries. Therefore, we instead set $\lambda = 3$ in order to enable more flexibility when capturing complex geometries. We anticipate that setting $\lambda = 2$ would lead to similar performance, however in our experiments we use $\lambda = 3$ as it seemed to consistently yield plausible generations.

1.3. Generation Protocol

In this section, we describe the sampling process used for the generation of new shapes. In particular, we follow [16] and draw a set of shape and texture latent codes $\{\mathbf{z}^s, \mathbf{z}^t\}$ from a multivariate normal distribution. Namely,

$$\begin{aligned} \mathbf{z}^s &\sim \mathcal{N}(\boldsymbol{\mu}^s_{train}, \boldsymbol{\Sigma}^s_{train}) \\ \mathbf{z}^t &\sim \mathcal{N}(\boldsymbol{\mu}^t_{train}, \boldsymbol{\Sigma}^t_{train}). \end{aligned} \tag{9}$$

Here, the mean $(\boldsymbol{\mu}^s_{train}, \boldsymbol{\mu}^t_{train})$ and covariance $(\boldsymbol{\Sigma}^s_{train}, \boldsymbol{\Sigma}^t_{train})$ values are calculated from the shape and texture embeddings learnt during training. At inference/generation, we randomly sample $\mathbf{z}_s$ and $\mathbf{z}_t$ and for a given camera we cast rays and sample points along these rays. For each point we predict the corresponding color and opacity value. Note that during generation, our model does not need to condition on the object mask to be generated. To generate a novel shape from a novel camera, our model associates the casted rays with the learned parts as described in Eq. 9 of our main submission and performs rendering. Note that while for training we sample 64 points along each ray, for generation we increase this number to 128 in order to improve the generation quality.

1.4. Editing

In this section, we provide additional information regarding the editing capabilities of our model and analyze how the editing operations are performed. As we are interested in editing specific parts of the object, we want to be able to modify the pose, size and appearance of each part independently. This can be enforced by making sure that each NeRF/part receives geometric inputs in its own local coordinate system, which is enforced by augmenting each part with:

1. an *affine transformation* $T_m(\mathbf{x}) = \mathbf{R}_m(\mathbf{x} + \mathbf{t}_m)$ that maps a 3D point $\mathbf{x}$ to the local coordinate system of the part, where $\mathbf{t}_m \in \mathbb{R}^3$ is the translation vector and $\mathbf{R}_m \in SO(3)$ is the rotation matrix;
2. a *scale* vector $\mathbf{s}_m \in \mathbb{R}^3$, representing spatial extent;
3. latent codes: *shape* $\mathbf{z}^s_m \in \mathbb{R}^{128}$ and *texture* $\mathbf{z}^t_m \in \mathbb{R}^{128}$.

To select and edit a specific part of a 3D shape, it suffices to select its corresponding latent vectors $\mathbf{z}^s_m$ and $\mathbf{z}^t_m$ and manipulate them based on the type of the editing operation. Below, we provide more details regarding the editing operations presented in the main paper.

Rigid Transformations: To apply a rigid transformation on a specific part of a shape, it suffices to alter the per-part translation vector $\mathbf{t}_m$ and rotation matrix $\mathbf{R}_m$. For example, to translate a part we add an appropriate displacement vector to the $\mathbf{t}_m$. Similarly, to rotate a part, we can multiply $\mathbf{R}_m$ with the desired rotation matrix.

Non-Rigid Transformations: Similarly, a non-rigid transformation that changes the size of a specific part can be implemented by multiplying the part's rotation matrix with an appropriate scale matrix.

Geometry Mixing: To perform geometry mixing, we select part shape latent codes $\mathbf{z}^s_m$ from an arbitrary number of initial shapes, forming a set of part shape latent codes that describe the new mixed shape. To generate a shape with a desired texture, we allow selecting the part texture latent codes either from a single shape or from an arbitrary number of shapes. For example, to generate the results of the shape mixing experiment presented in Section 4.3 of our main submission and Fig. 10, we select texture codes only from Shape 1 for the chair and the airplane in the third column, whereas we select texture codes from both shapes to generate the mixed shapes for the chair and the airplane in the last column. As soon as the selection process is done, we feed the new part latent codes to the Neural Rendering module and render the new mixed shape. Note that the resulting number of parts in the generated shape can be arbitrary, as our model's occupancy and color networks can produce meaningful results for any given number of parts.

Texture Mixing: For texture mixing, we follow the same process as in the geometry mixing procedure, and select part shape and texture latent codes either from a single shape, or from multiple shapes and feed them to the Neural Rendering module to generate the new shape. To generate the results for the experiment presented in Section 4.3 and Fig. 10 of our main

submission, we select the shape codes from Shape 1 and for the legs of the chair, we select the texture codes of the legs of Shape 2.

Part Addition: Part addition can be seen as the equivalent operation of our shape mixing strategy, where we select all the part latent codes from an entire shape, and append extra part geometry and texture codes from other shapes. Similarly to the previously discussed editing operations, the per-part shape and texture codes of the new shape after the addition are fed to the Neural Rendering module that renders the new shape.

Part Removal: To remove parts from a shape, it suffices to remove the corresponding part shape and texture embeddings, and feed only the remaining part embeddings to the Neural Rendering module.

Color Change: Our hard ray-part assignment enables changing the colors of selected parts, by directly replacing the predicted colors of the associated part rays with a color of our preference.

Part-Level Interpolation: For the part-level interpolation operation, we select corresponding parts from two different shape instances, and linearly interpolate the part shape and texture embeddings, producing new part shape and texture latent codes. The interpolated vectors are then passed to the Neural Rendering module to generate a new shape.

1.5. Metrics

As mentioned in the main submission, we evaluate our model and our baselines using two metrics that measure the plausibility and the diversity of set of generated shapes G in comparison to a set of reference shapes R. In particular, we report the Coverage score (COV) and the Minimum Matching Distance (MMD) [1] using Chamfer Distance (CD). MMD measures the *quality of the generated shapes* by computing how likely it is that a generated shape looks like a shape from the reference set of shapes. On the other hand, COV measures how many shape variations are covered by the generated shapes, by computing the percentage of reference shapes that are closest to at least one generated shape.

To ensure fair comparison with our baselines, when we compare with NeRF-based generative models (*i.e.* Table 1 in our main submission), we follow [10], using the test set as the set of reference shapes and synthesizing five times as many generated shapes. In contrary, when comparing to part-based generative models (*i.e.* Table 2 in our main submission), we follow [16] and randomly generate 1000 shapes which are compared to 500 shapes from the training set and 500 shapes from the test set.

To estimate the similarity between two shapes from the two sets, we use the Chamfer Distance (CD), which is simply the distance between a set of points sampled on the surface of the reference and the generated mesh. Namely, given a set of N sampled points on the surface of the reference $\mathcal{X} = \{\mathbf{x}_i\}_{i=1}^{N}$ and the generated shape $\mathcal{Y} = \{\mathbf{y}_i\}_{i=1}^{N}$ the Chamfer Distance (CD) becomes

$$\text{CD}(\mathcal{X}, \mathcal{Y}) = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{X}} \min_{\mathbf{y} \in \mathcal{Y}} \|\mathbf{x} - \mathbf{y}\|_2^2 + \frac{1}{M} \sum_{\mathbf{y} \in \mathcal{Y}} \min_{\mathbf{x} \in \mathcal{X}} \|\mathbf{y} - \mathbf{x}\|_2^2 . \tag{10}$$

For both our NeRF-based and part-based baselines, we set $N = 2048$.

The Minimum Matching Distance (MMD) is the average distance between each shape from the generated set G to its closest shape in the reference set R and can be defined as:

$$\text{MMD}(G, R) = \frac{1}{|R|} \sum_{\mathcal{X} \in R} \min_{\mathcal{Y} \in G} \text{CD}(\mathcal{X}, \mathcal{Y}). \tag{11}$$

Intuitively, MMD measures how likely it is that a generated shape is similar to a reference shape in terms of Chamfer Distance and is a metric of the plausibility of the generated shapes. Namely, a high MMD score indicates that the shapes in the generated set G faithfully represent the shapes in the reference set R.

The Coverage score (COV) measures the percentage of shapes in the reference set that are closest to each shape from the generated set. In particular, for each shape in the generated set G, we assign its closest shape from the reference set R. In our measurement, we only consider shapes from R that are closest to at least one shape in G. Formally, COV is defined as

$$\text{COV}(G, R) = \frac{|\{\text{argmin}_{\mathcal{X} \in R} \text{CD}(\mathcal{X}, \mathcal{Y}) \mid \mathcal{Y} \in G\}|}{|G|} \tag{12}$$

Intuitively, COV measures the diversity of the generated shapes in comparison to the reference set. In other words, a high Coverage indicates that most of the shapes in the reference set R are roughly represented by the set of generated shapes G. To ensure a fair comparison, with our baselines, we use the evaluation code of [10].

Method	Rendering	Representation	Textures	Supervision	Parts
GET3D [10]	Differentiable	Mesh	✓	2D	✗
GRAF [35]	Volumetric	Neural Field	✓	2D	✗
Pi-GAN [5]	Volumetric	Neural Field	✓	2D	✗
EG3D [4]	Volumetric	Neural Field	✓	2D	✗
DualSDF [14]	-	Implicit SDF	✗	3D	✓
SPAGHETTI [16]	-	Implicit Occupancy	✗	3D	✓
Ours	Volumetric	Neural Field	✓	2D	✓

Table 1. **Comparison to Prior Work.** We introduce the first part-based generative model that does not require explicit 3D supervision. We compare our model with NeRF-based generative models [4, 5, 35], part-based generative models [14, 16] and the concurrent [10] that unlike our model relies on differentiable rendering.

1.6. Baselines

In this section, we provide additional details regarding our baselines. To the best of our knowledge, our work is the first part-aware generative model that does not require explicit 3D supervision, thus there are no other works that are directly comparable to our model. Therefore, in our evaluation, we consider three types of baselines: NeRF-based generative models [4, 5, 35] that are part agnostic, part-based generative models [14, 16] that require explicit 3D supervision in the form of watertight meshes and can only generate shapes without textures and finally the concurrent GET3D [10] that relies on differentiable rendering and again does not consider any parts. A concise comparison between our model and these baselines is provided in Tab. 1. Furthermore, we also compare our model with our baselines w.r.t. the editing capabilities of each model in Tab. 2.

GET3D: In concurrent work, GET3D [10][2], proposed a novel generative model for textured meshes that relies on a highly optimized differentiable graphics renderer [21]. In particular, their model comprises a geometry generator that employs DMTet [9] to recover surfaces of arbitrary topologies and a texture generator that generates textures as a texture field [28]. During training, they rely on two discriminator losses on the rendered image and the 2D silhouette. As the underlying representation of GET3D is a textured mesh, their generated textures and geometries are more detailed compared to ours. However, as they do not consider parts, they cannot perform several editing operations that our model is capable of. As already mentioned in our main submission, training GET3D requires approximately 16 GPU days (8 NVIDIA A100 for 2 days), whereas training our model takes approximately one third of the time (5 GPU days). Due to their rigorous evaluation and to ensure a fair comparison, we report their results in Table 1 and Figure 6 in our main submission.

GRAF: GRAF [35][3] was among the first 3D-aware generative models that combined a 2D GAN [12] with volumetric rendering as in NeRF [26]. In particular, they introduce a conditional NeRF parametrized as an MLP that maps the positional encoding of a 3D point and a viewing direction to a color and a volume density value, conditioned on a shape and an appearance latent codes. Unlike [26], GRAF is trained from a collection of unposed images and employs a PatchGAN [17] discriminator to improve the quality of the rendered images from a novel view. While GRAF's code is publicly available, to generate the qualitative results for Fig 6 and the quantitative comparison in Table 1 from our main submission, we do not train their model. Instead, we directly report the results from [10]. Unlike our model, GRAF employs a discriminator to improve the quality of the rendered images from novel view points. Furthermore, as [35] does not consider parts, it only enables few editing operations on the entire image, *i.e.* changing the color of the depicted object.

Pi-GAN: Similar to GRAF [35], Pi-GAN [5][4] addresses the 3D-aware image synthesis task and leverages a neural representation with periodic activation functions [36]. Unlike [35] that conditions on a shape and appearance latent code, Pi-GAN conditions the MLP on a single input noise vector, as in StyleGAN [18]. Also, Pi-GAN uses a discriminator to further improve the quality of the volumetric rendering. As they do not consider parts, [5] have limited editing capabilities. Again, to ensure a fair comparison, we do not train their model from scratch but instead report the results from [10].

[2] https://nv-tlabs.github.io/GET3D/
[3] https://github.com/autonomousvision/graf
[4] https://marcoamonteiro.github.io/pi-GAN-website/

Method	Image Inversion	Shape Inversion	Generation	Shape Mixing	Shape Editing	Texture Editing
GET3D [10]	✓	✓	✓	✗	✗	✗
GRAF [35]	✓	✓	✓	✗	✗	✗
Pi-GAN [5]	✓	✓	✓	✗	✗	✗
EG3D [4]	✓	✓	✓	✗	✗	✗
DualSDF [14]	✗	✓	✓	✗	✓	✗
SPAGHETTI [16]	✗	✓	✓	✓	✓	✗
Ours	✓	✓	✓	✓	✓	✓

Table 2. **Comparison to Prior Work w.r.t. Editing Operations.** We compare the editing capabilities of our model and our baselines. Note that *Shape Editing* and *Texture Editing* refer to **local** changes on the texture and the shape of the generated object.

EG3D: More recently, EG3D [4][5] proposed a StyleGAN-based generator [18] that relies on a tri-plane representation instead of positional encoding. Their model is capable of generating images at high resolutions using a CNN-based upsampling module. Despite its impressive results, EG3D does not consider parts, hence it cannot perform local edits on the generated shapes. Similar to [5, 35], for the qualitative and quantitative evaluation in Sec. 4.2 of our main submission, we report the results from [10]. In addition, note that as EG3D [4] requires training of approximately 8 days on 8 NVIDIA Tesla V100 GPUs, we are not able to run their code due to our limited GPU resources.

DualSDF: DualSDF [14][6] was among the first to introduce a model capable of editing neural implicit shapes. In particular, they proposed a representation with two levels of granularity, where a user can manipulate a 3D shape through the coarse primitive-based representation and these edits are reflected to a high-resolution implicit shape, via a shared latent code. Unlike our model, DualSDF assumes explicit 3D supervision in the form of a watertight mesh and cannot generate shapes with texture. To ensure a fair comparison, we report the results from [16] in Table 2 of our main submission. Although our model considers 2D supervision in the form of posed images and object masks, our model outperforms DualSDF in terms of MMD and COV both for airplanes and tables (see Table 2 in our main submission).

SPAGHETTI: SPAGHETTI [16][7] is the state-of-the-art part-based generative model that similar to [14] permits manipulating neural implicit shapes. Similar to our model, they implement their generative model as an auto-decoder [3, 29] that takes an instant-specific latent code as input and decomposes it to M per-part latent codes, representing different parts of the object. Unlike our model, SPAGHETTI [16], employs a transformer decoder that merges the parts and produces the final implicit shape as an occupancy field. Unlike our model, [16] assumes explicit 3D supervision in the form of a watertight mesh and cannot generate meshes with texture. As already mentioned previously, to ensure a fair comparison, the numbers in Table 2 of our main submission are from [16]. For the qualitative comparison in Fig. 7 of our main submission, we use the pre-trained models provided to us by the authors.

1.7. Mesh Extraction

To extract meshes from the predicted occupancy field, we employ the Multiresolution IsoSurface Extraction (MISE) technique introduced in [25]. In particular, we start from a voxel grid of 32^3 initial resolution for which we predict occupancy values. Next, we follow the process proposed in [25] and extract the approximate isosurface with Marching Cubes [24], which is then refined using 3 optimization steps. For the mesh extraction, we use the code provided by Mescheder *et al*. [25]. Note that for the individual parts, we only extract approximate surfaces with Marching Cubes [24].

[5] https://nvlabs.github.io/eg3d/
[6] https://www.cs.cornell.edu/ hadarelor/dualsdf/
[7] https://github.com/amirhertz/spaghetti

2. Ablation Study

In this section, we ablate various components of our model and demonstrate their impact on its generation capabilities. We conduct all our ablations on a subset of ShapeNet [6] Airplanes, which comprises approximately 10% of the original train set. Namely, we use 178 training samples and 100 views for training all our model variants. In Sec. 2.1, we discuss the effect of our loss terms on the overall performance of our model and in Sec. 2.2, we demonstrate the impact of the number of parts on the generation capabilities of our model. Finally, we provide additional information regarding the impact of our hard ray-part assignment, presented in our main submission (see Sec. 2.3)

2.1. Loss Functions

Our optimization objective $\mathcal{L}$ is the sum of six terms combined with two regularizers on the shape and texture embeddings $\mathbf{z}^s, \mathbf{z}^t$, namely

$$\mathcal{L} = \mathcal{L}_{rgb}(\mathcal{R}) + \mathcal{L}_{mask}(\mathcal{R}) + \mathcal{L}_{occ}(\mathcal{R}) + \mathcal{L}_{cov}(\mathcal{R}) + \mathcal{L}_{overlap}(\mathcal{R}) + \mathcal{L}_{control} + \|\mathbf{z}^s\|_2 + \left\|\mathbf{z}^t\right\|_2 . \tag{13}$$

As supervision, we use the observed RGB color $C(r) \in \mathbb{R}^3$ and the object mask $I(r) \in \{0, 1\}$ for each ray $r \in \mathcal{R}$. In addition, we associate r with a binary label $\ell_r = I(r)$, as shown in Fig. 5. Namely, we label a ray r as *inside*, if $\ell_r = 1$ and *outside* if $\ell_r = 0$. In this section, we discuss how the loss terms affect the performance of our model w.r.t. the plausibility of the generated shapes. In particular, we train 4 variants of our model and for each one we omit one of the loss terms. We provide both quantitative (see Tab. 3) and qualitative comparison (see Fig. 6) for these scenarios. For a fair comparison, all models are trained for the same number of epochs, which is set to 50. Furthermore, in this experiment, we set the number of parts to 10. To compute the metrics in Tab. 3, we consider a subset of the test set, which amounts to 46 shapes, namely roughly 10% of the original test set.

	w/o $\mathcal{L}_{mask}$	w/o $\mathcal{L}_{occ}$	w/o $\mathcal{L}_{cov}$	w/o $\mathcal{L}_{overlap}$	Ours
MMD-CD (↓)	1.787	1.823	1.737	1.757	1.732

Table 3. **Ablation Study on Loss Terms.** We investigate the impact of each loss term by training our model without each one of them. We report the MMD-CD (↓) for all variants.

w/o Mask Loss: The mask loss is the squared error between the observed and the rendered pixel value of the object mask for a set of rays. We observe that removing the mask loss $\mathcal{L}_{mask}(\mathcal{R})$ results in more noisy/blurry renderings, in particular around the surface boundaries (see first column in Fig. 6). We experimentally observed that $\mathcal{L}_{mask}(\mathcal{R})$ helps the network find the object boundaries faster, which naturally improves the overall rendering quality. On the part-level, we also observe that the generated parts are less smooth (*e.g.* see the tail parts of the two last airplanes in the first column of Fig. 6).

w/o Occupancy Loss: The occupancy loss makes sure that the generated shape does not occupy empty space. This is enforced by making sure that the generated parts only contain/cover *inside rays*, namely rays with $\ell_r = 1$. Therefore, removing $\mathcal{L}_{occ}(\mathcal{R})$ from our optimization objective results in renderings of worse quality, particularly around the object boundaries (see second column in Fig. 6). Note that in our evaluation, even if we remove this loss, we still use $\mathcal{L}_{mask}(\mathcal{R})$, which forces, to some extent, the network to capture the surface boundary.

w/o Coverage Loss: The coverage loss prevents degenerate parts, that are either too thin or too large. Therefore, when we remove $\mathcal{L}_{cov}(\mathcal{R})$ from our optimization, we note that the generated parts become either very small (first and second example in third column of Fig. 6) or too large (last example in the same column).

w/o Overlapping Loss: The overlapping loss encourages the generated parts to not capture the same regions of the object. We identify overlapping parts by a ray being internal to more than λ parts and we set $\lambda = 3$. Note that this done using the predicted label for ray r w.r.t. to part m, $\hat{\ell}_r^m = \max_{\mathbf{x}_i^r \in \mathcal{X}_r} h_\theta^m(\mathbf{x}_i^r)$. Removing $\mathcal{L}_{overlap}(\mathcal{R})$ results in generated parts that capture the same regions of the object, as shown in the fourth column of Fig. 6.

2.2. Number of Parts

In this section, we analyze the impact of the number of parts on the quality of the generated shapes. Specifically, we train our model with: 5, 10, 16 and 20 parts. Unlike the results presented in our main submission, for this experiment, we

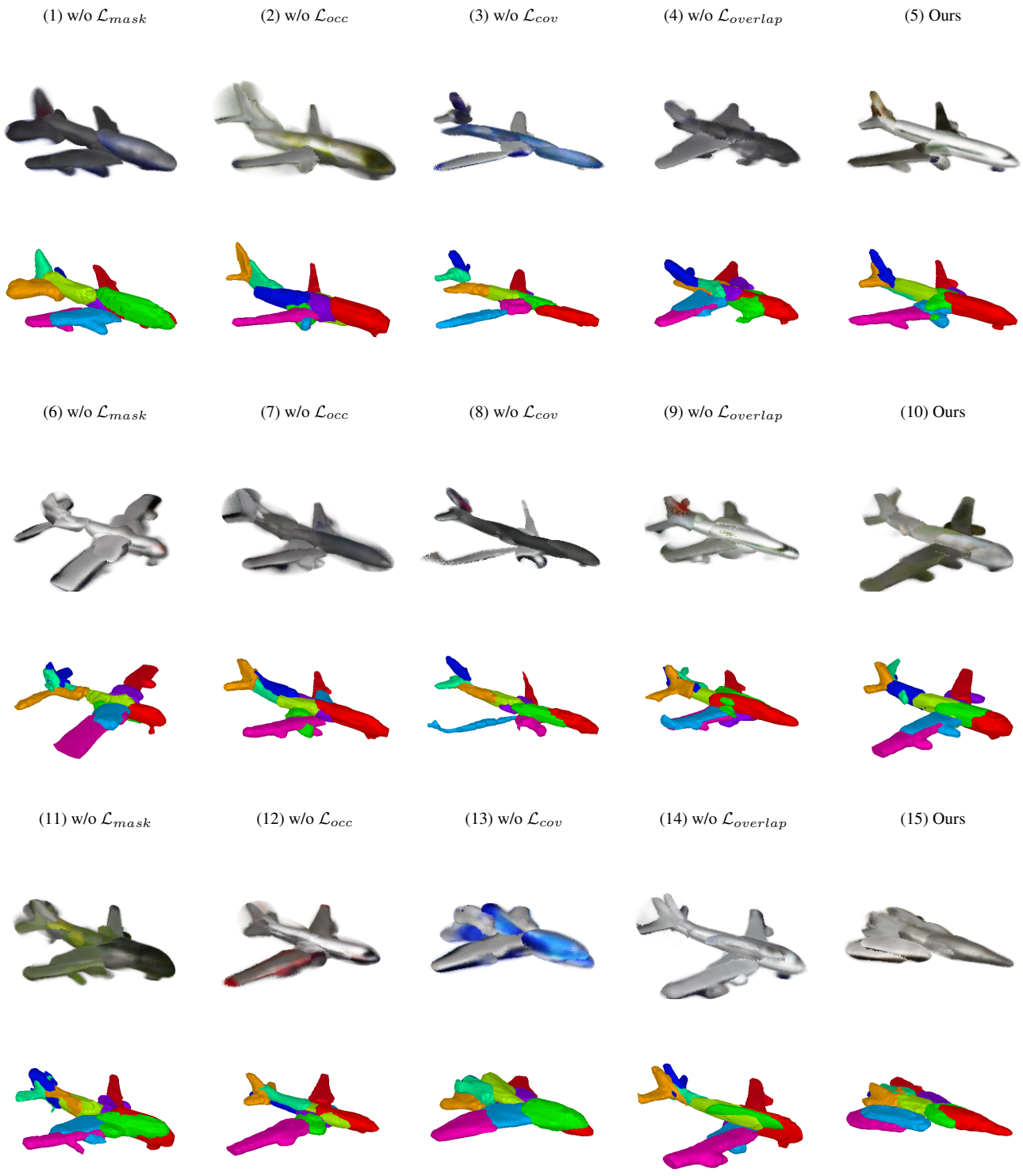


Figure 6. **Ablation Study on Loss Terms.** Qualitative evaluation of the impact of the 4 loss terms on the performance of our model. We train our model, without each loss term and generate novel shapes.

train all model variants on a subset of the Airplanes category, consisting of 178 training samples, for 50 epochs. To evaluate the plausibility of the generated shapes, we compute the Minimum Matching Distance (MMD-CD) score between shapes generated with a different number of parts and a set of reference shapes. As reference shapes, we take 10% of the entire test set, which amounts to approximately 46 shapes. Results are summarized in Tab. 4. Note that we do not compare our model

	5 Parts	10 Parts	16 Parts	20 Parts
MMD-CD (↓)	1.87	1.73	1.53	1.52

Table 4. **Ablation Study on the Number of Parts.** This table shows a quantitative comparison of our approach with different numbers of parts w.r.t. MMD-CD (↓).

variants w.r.t. Coverage (COV), which is a metric indicative of the diversity of the generated shapes, as we only consider a very small reference set of shapes and the results could have been misleading.

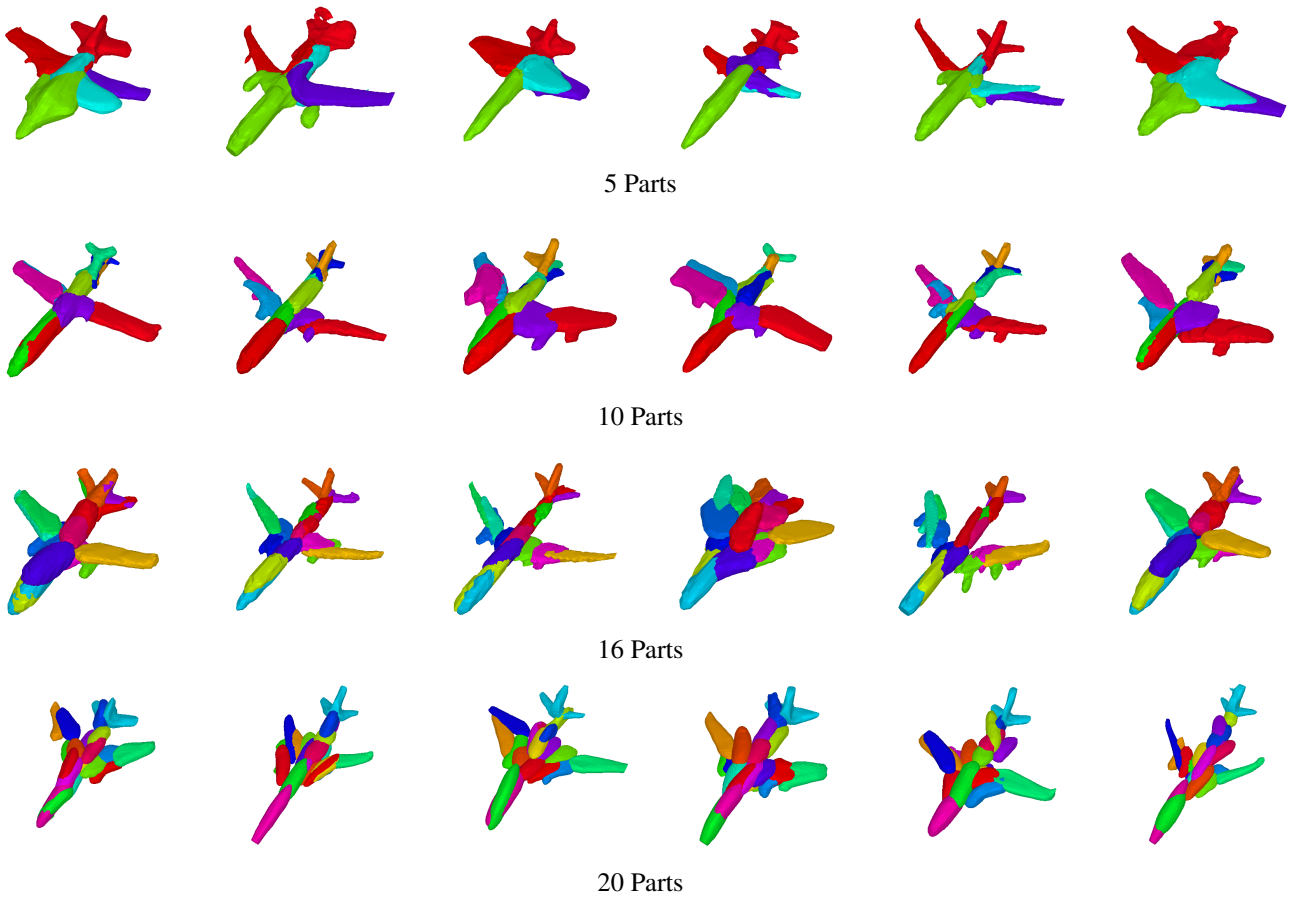


Figure 7. **Ablation Study on the Number of Parts**. Qualitative evaluation of the impact of the number of parts. The first row shows randomly generated airplanes with 5 parts, the second row with 10, the third with 16, and the last row with 20 parts. Note that we train a different model for the four different numbers of parts.

From the quantitative comparison in Tab. 4, we observe that the quality of the generated meshes increases for a larger number of parts, namely using more parts yields higher MMD-CD scores. From the qualitative comparison of Fig. 7, we note that using fewer parts results in parts that capture multiple regions of the object, *e.g.* the red part represents the left wing and the tail. Since our main goal is to enable editability and part-level control, we want to ensure that there is an adequate number of parts so that the user can choose what they want to edit/change. For the case of 10 parts, we note that now parts are capable of recovering more distinct regions of the object.

2.3. Hard Ray-Part Assignment

In Sec. 4.1 of our main submission, we demonstrate that enabling a soft assignment between parts and rays, namely the color of one ray being determined from multiple parts, prevents several editing operations (see Fig. 5 in our main submission). This is to be expected as for the case of the soft ray-part assignment, the colors and opacities of all rays are determined jointly by all NeRFs. As a result, changing one NeRF naturally affects the others, hence the colors and opacities of parts of the object that we did not intend to change.

3. Additional Experimental Results

In this section, we provide additional information regarding our experiments on ShapeNet [6]. In particular, we consider five categories: *Motorbike, Chair, Table, Airplane, Car*, which contain 337, 6678, 8509, 4045 and 7497 shapes respectively. For the *Car, Table, Airplane and Chair*, we use 24 random views, while for Motorbike we use 100, as it contains fewer training samples. To ensure fair comparison with our baselines, we use the train-test splits of [10] for the *Motorbike, Car, Chair* object categories and the train-test splits of [16] for the *Airplane, Table* category. To render our training data, we randomly sample camera poses from the upper hemisphere of each shape and render images at 256^2 resolution as in [10]. To render both the RGB images and the corresponding object masks we use *simple-3dviz* [2][8]. Examples of the rendered RGB images and object masks per category are provided in Fig. 8.

Figure 8. **Rendered RGB Images and Object Masks**. We provide examples of the rendered RGB images and object masks that we use for training.

3.1. Shape Generation

In this section, we provide additional qualitative results for our shape generation experiment on the five ShapeNet [6] categories: *Motorbike, Chair, Table, Airplane, Car*. In particular, we visualize several random samples per category, rendered from a novel view point, not seen during training. In addition, we also show the corresponding 3D meshes and parts. Specifically, in Fig. 9, we illustrate randomly generated motorbikes. We notice that all motorbikes are consistently plausible and the underlying parts meaningfully capture several motorbike's geometry. Likewise, for the case of generated cars (see Fig. 10) and airplanes (see Fig. 11), we observe that our model is capable of generating diverse geometries and textures that are consistently plausible. For the case of chairs and tables (see Fig. 12), we notice that the geometries are consistently plausible, as also evidenced from Table 1 in our main submission, however the generated textures are less crisp in comparison to the other categories. We hypothesize that this discrepancy stems from the fact that tables and chairs have thinner parts than the other objects. We believe that incorporating a coarse and a fine volume of 3D coordinates would potentially improve this.

[8] https://simple-3dviz.com/

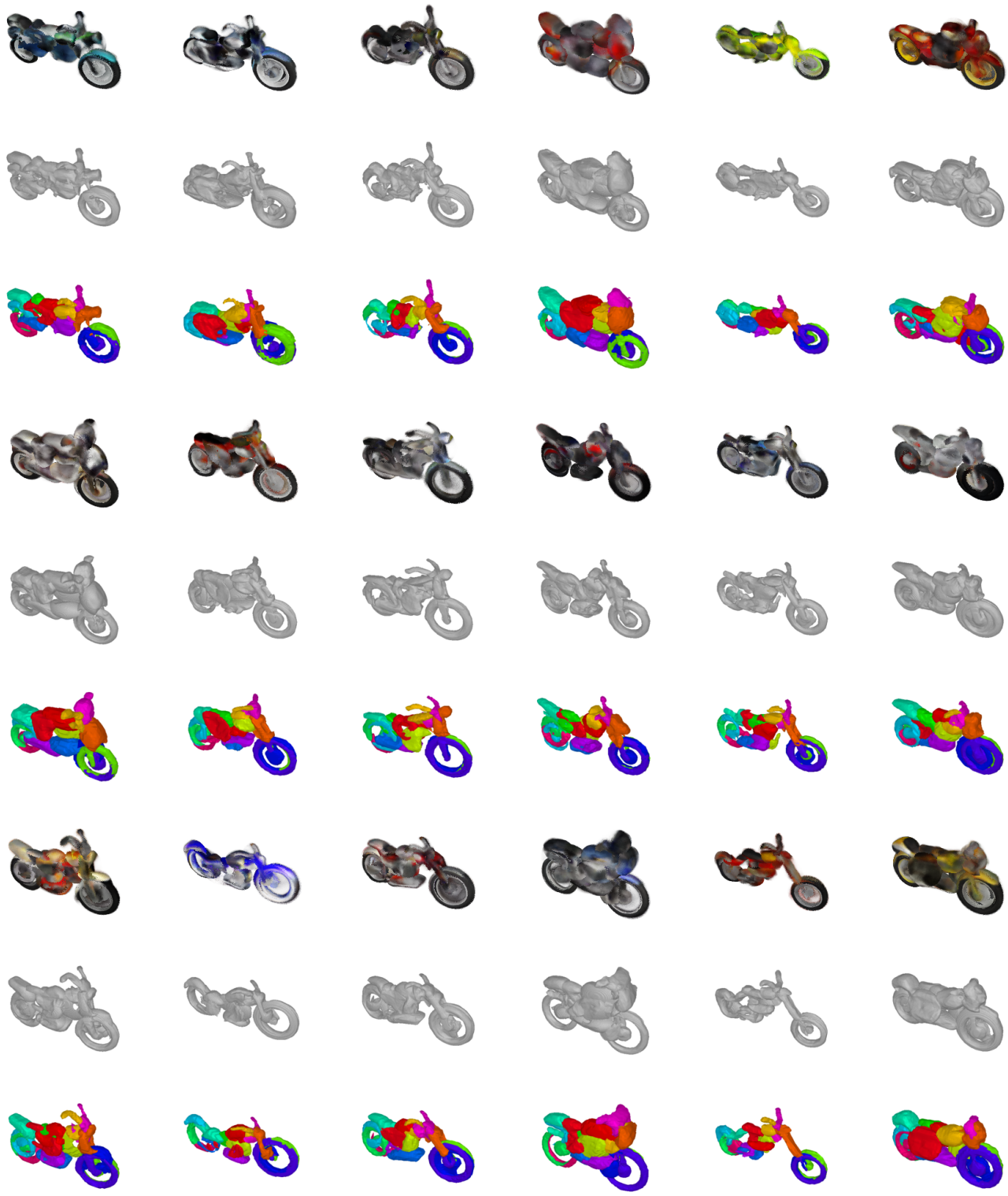


Figure 9. **Generated Motorbikes**. Generated motorbikes accompanied by their corresponding 3D geometries and part-based geometries.

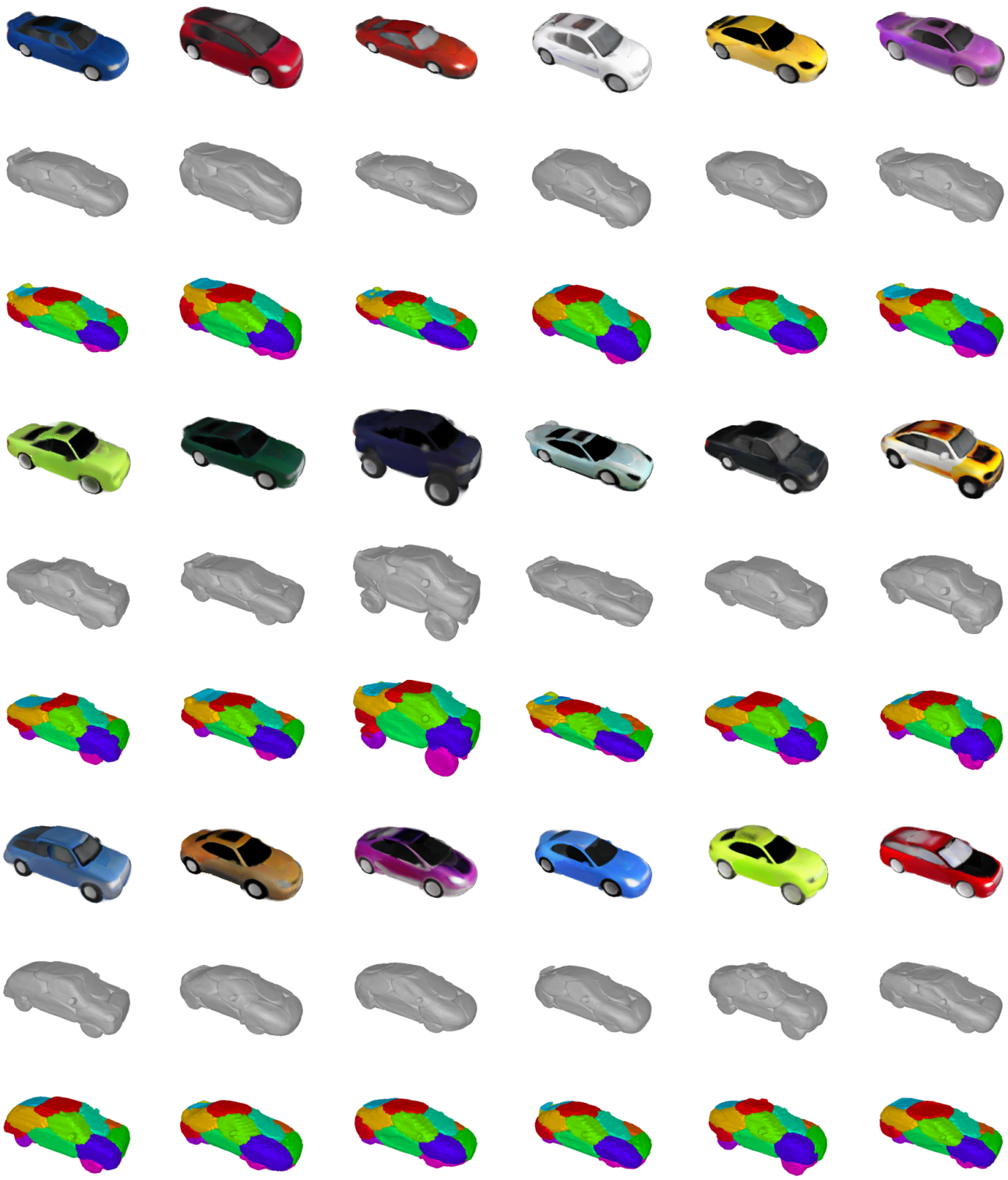


Figure 10. **Generated Cars**. Generated cars accompanied by their corresponding 3D geometries and part-based geometries.

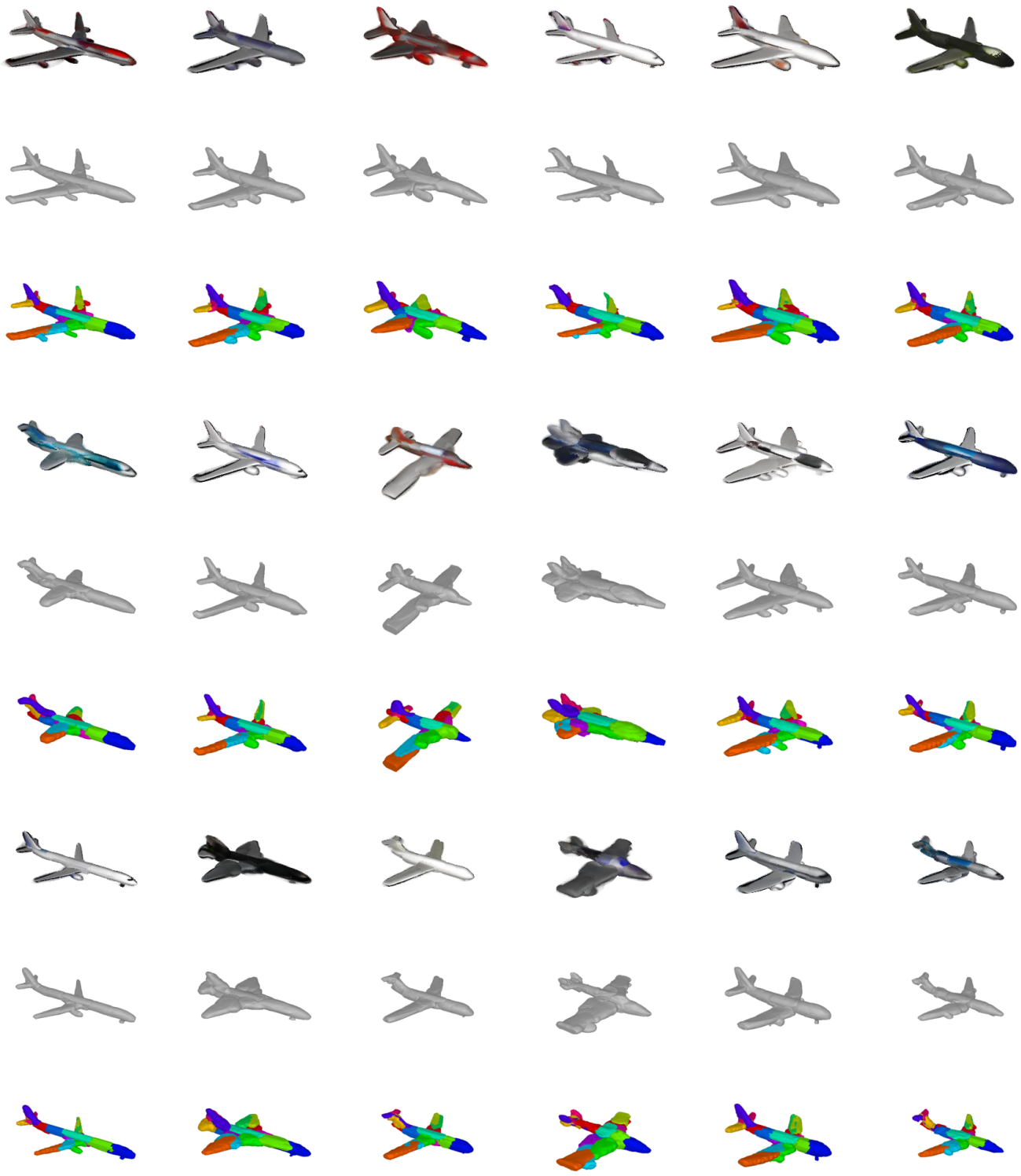


Figure 11. **Generated Airplanes**. Generated airplanes accompanied by their corresponding 3D geometries and part-based geometries.

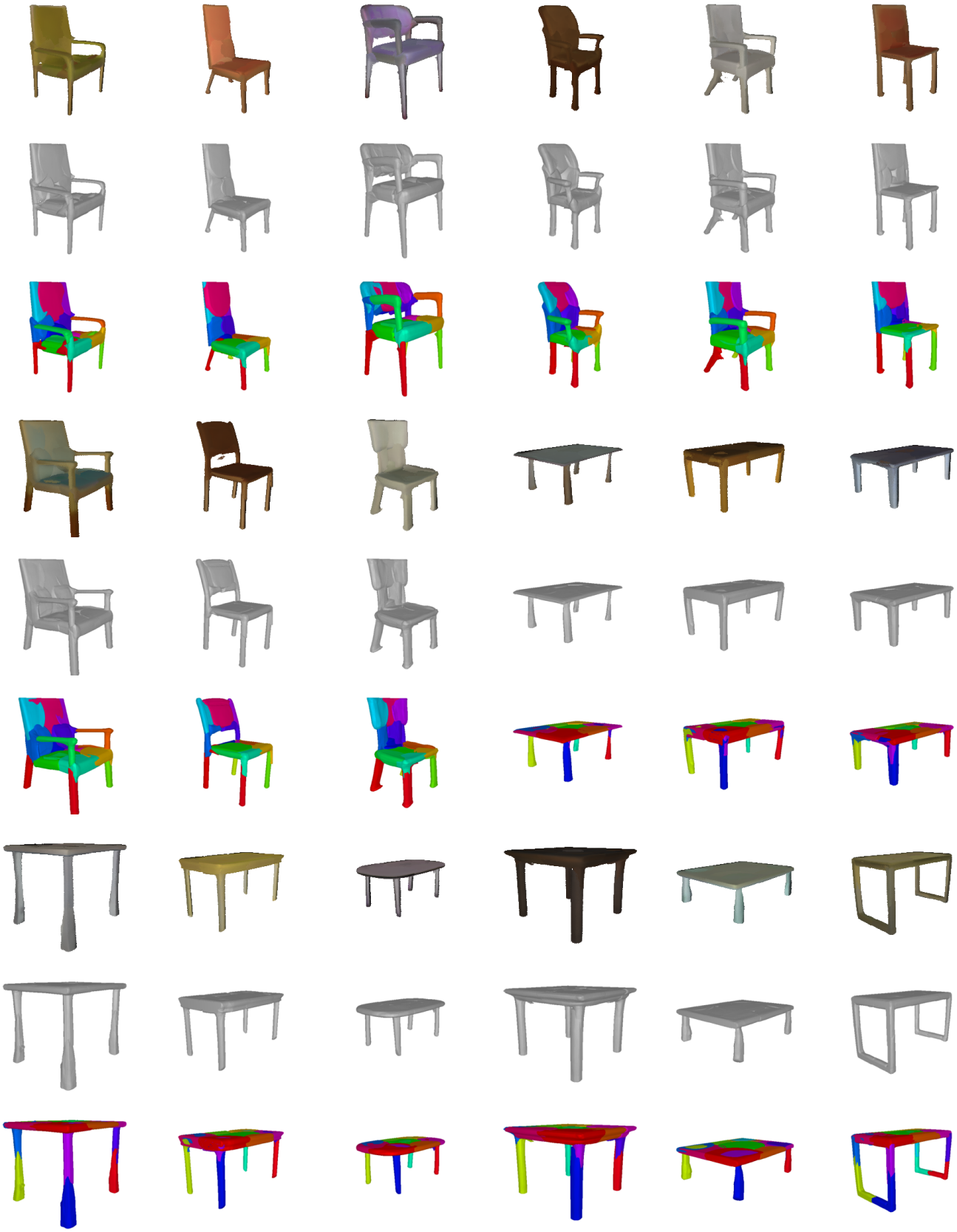


Figure 12. **Generated Chairs & Tables**. Generated chairs and tables accompanied by their corresponding 3D geometries and part-based geometries.

3.1.1 Comparison with Part-Based Generative Models

In this section, we provide more qualitative comparisons of our generated shapes with existing part-based generative methods [16], that require explicit 3D supervision in the form of occupancy labels. In particular, we visualize random samples for three ShapeNet [6] categories: *Table, Airplane, Chair*. Results are summarized in Fig. 13. Our method can generate geometries of similar quality to [16], without the need of explicit 3D supervision.

3.2. Shape Interpolation

In this section, we show interpolations between the shape and texture codes of two various objects for all object categories. In particular, we interpolate between the latent codes of two shapes and demonstrate that our model is able to smoothly interpolate between two shapes, while preserving the shape structure and the part-based structure for all object categories. Fig. 14 shows three interpolations from left to right between two airplanes. For the case of cars and motorbikes, interpolation results are summarized in Fig. 16 and Fig. 15. Note that for all interpolations the transitions from one shape to the other are consistently meaningful.

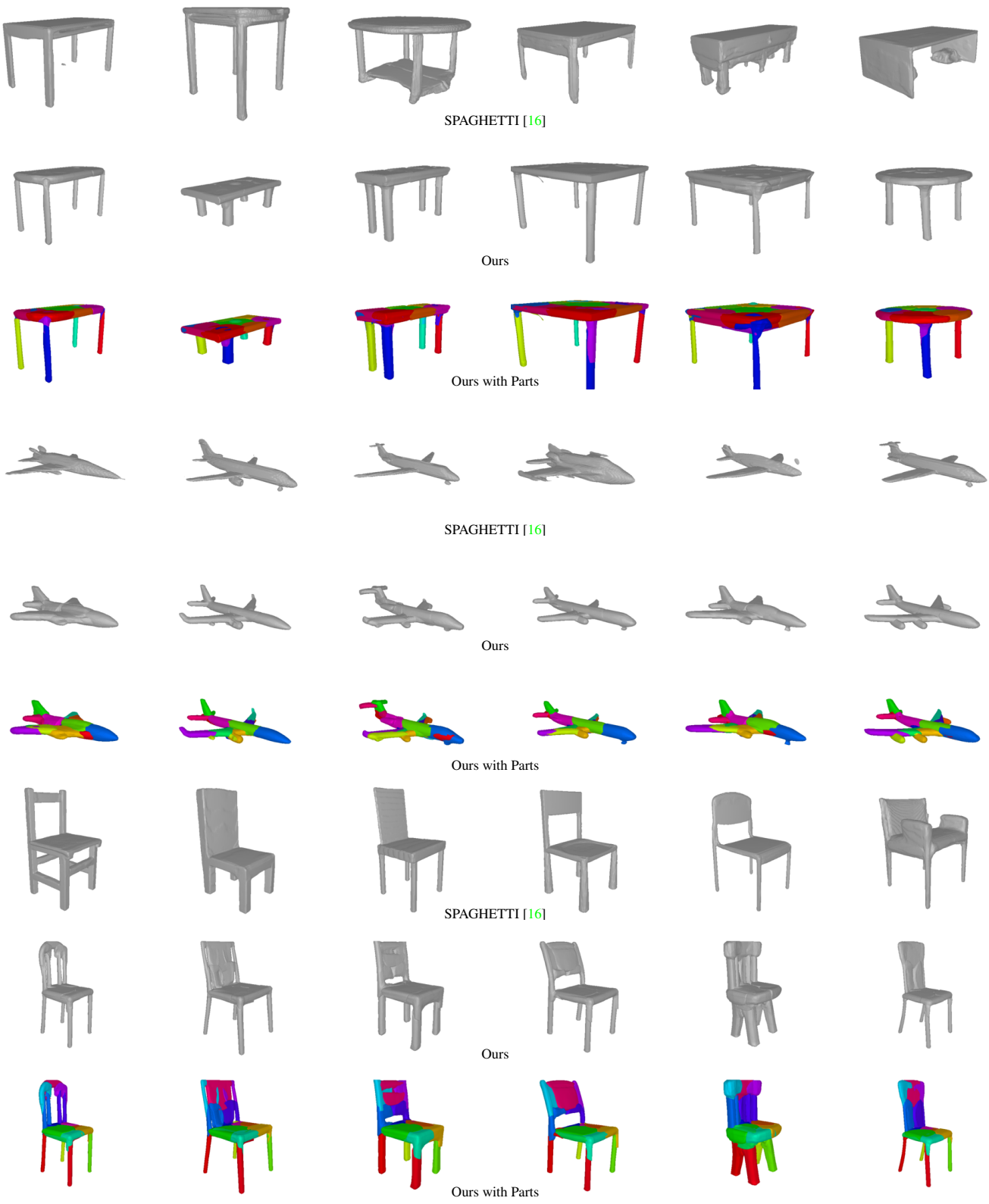


Figure 13. **Shape Generation Comparisons**. We compare our model with [16] and show six randomly generated samples per category. The first row per category corresponds to samples from [16], the second row to our results, and the third row to our part geometry results.

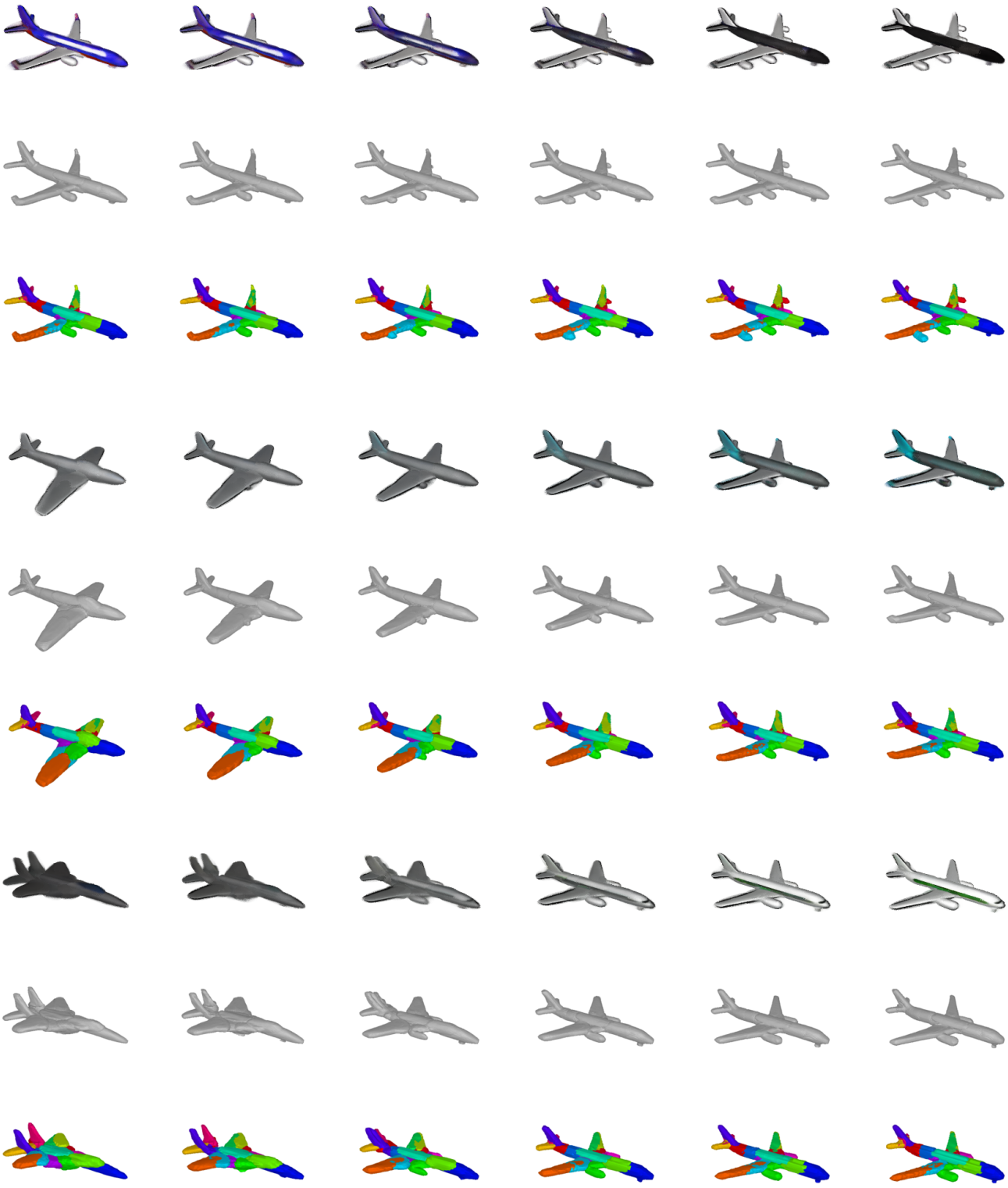


Figure 14. **Airplanes Shape Interpolation**. From left to right, we interpolate between the geometry and texture latent codes of the two airplanes.

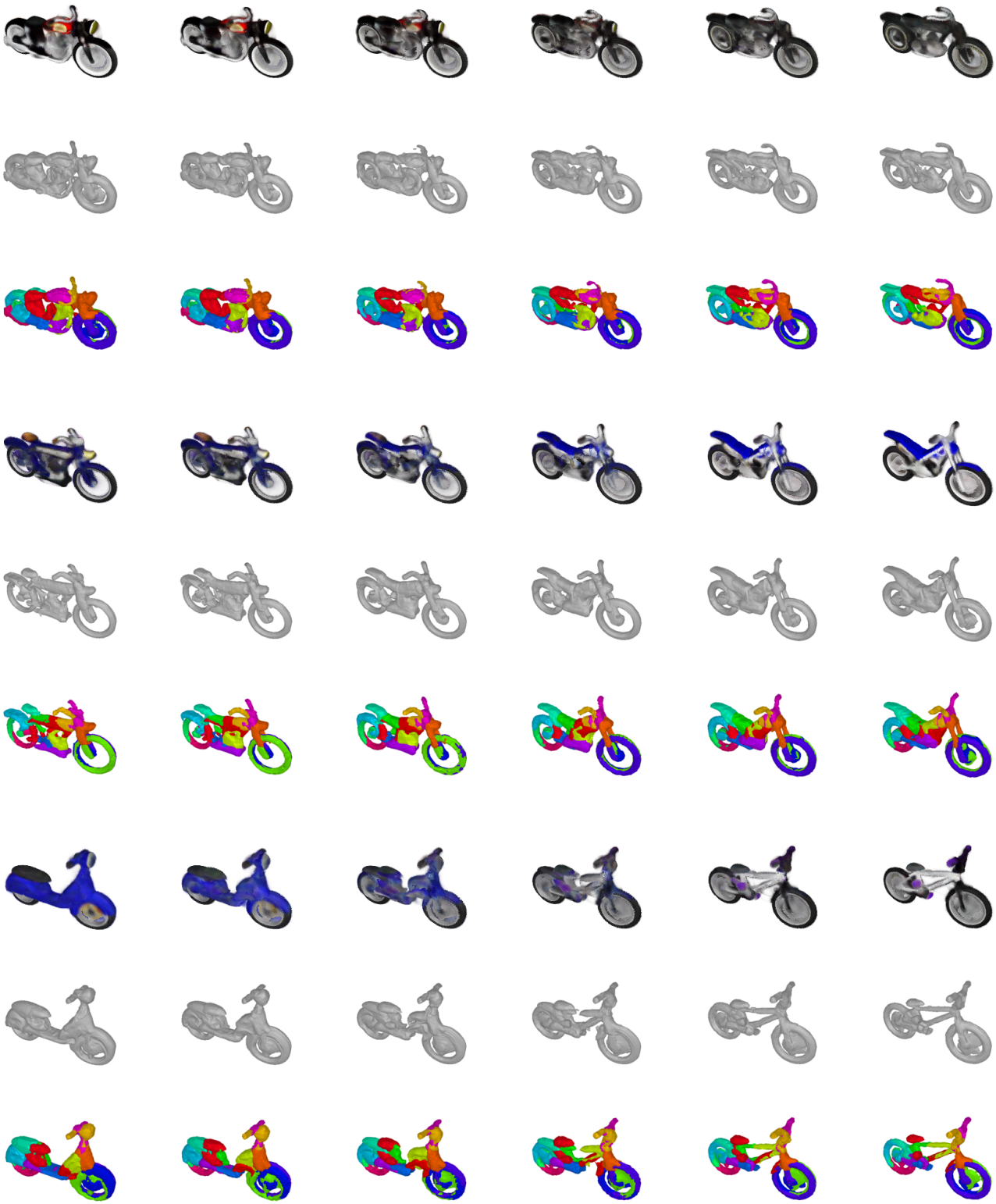


Figure 15. **Motorbikes Shape Interpolation**. From left to right, we interpolate between the geometry and texture latent codes of the two motorbikes.

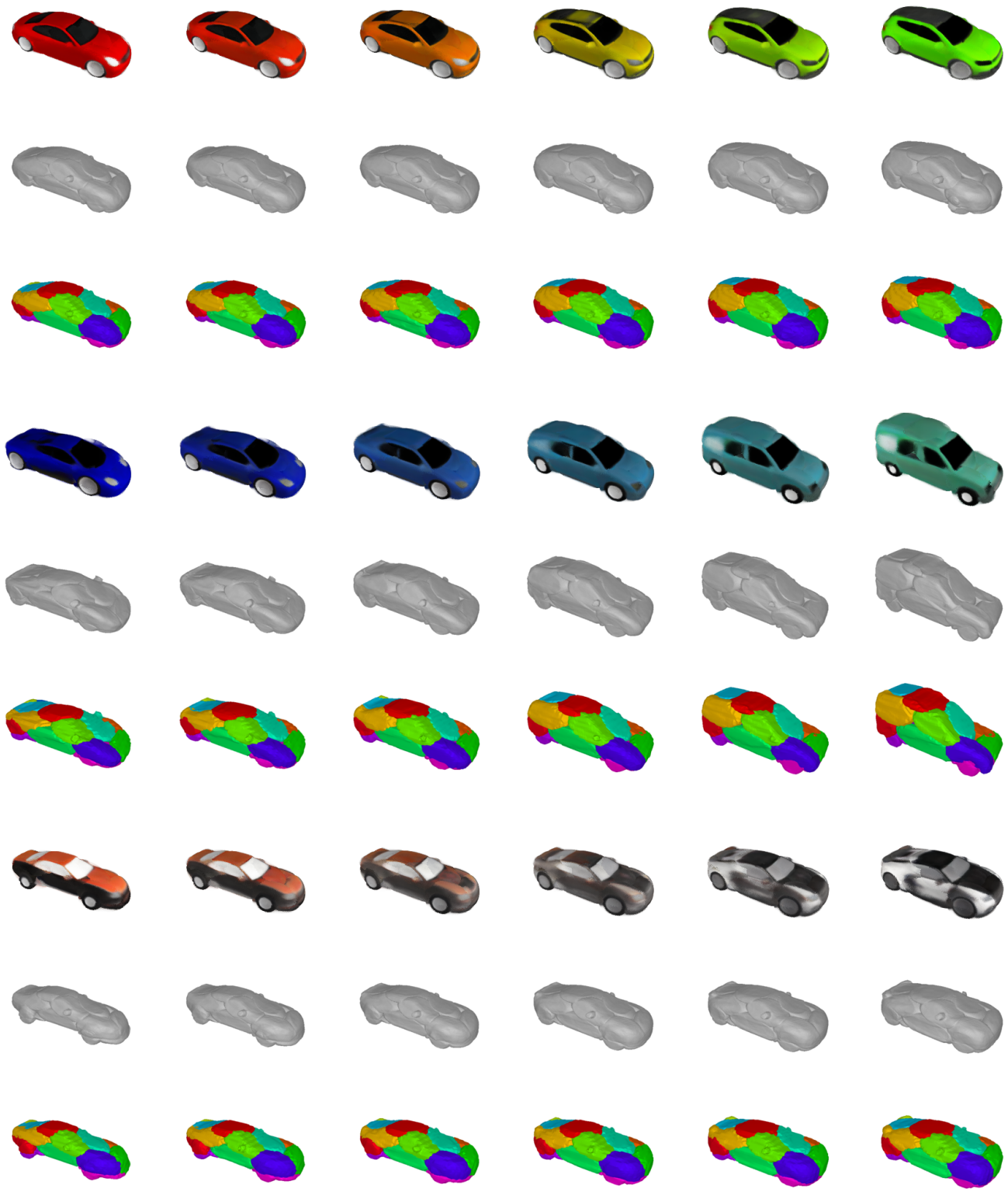


Figure 16. **Cars Shape Interpolation**. From left to right, we interpolate between the geometry and texture latent codes of the two cars.

4. Shape Inversion

Our generative model is realized by an auto-decoder architecture. In order to be able to edit a shape that does not belong to the training set, we seek to match the given shape to a shape embedding $\mathbf{z}^s$ that can closely reconstruct its geometry. Following [16], we randomly initialize the shape embedding $\mathbf{z}^s$, sampling from a multivariate normal distribution using the mean $\boldsymbol{\mu}^s_{train}$ and covariance $\boldsymbol{\Sigma}^s_{train}$ of shape embeddings seen during training:

$$\mathbf{z}^s \sim \mathcal{N}(\boldsymbol{\mu}^s_{train}, \boldsymbol{\Sigma}^s_{train}). \tag{14}$$

Then, we sample points along rays $\mathcal{R}$ from $N = 5$ different views of the object, and freeze our entire network, optimizing the latent code $\mathbf{z}^s$ using the following loss function:

$$\mathcal{L}_{inversion} = \mathcal{L}_{mask}(\mathcal{R}) + \mathcal{L}_{occ}(\mathcal{R}) + \|\mathbf{z}^s\|_2 \,, \tag{15}$$

using weights 1.0, 1.0 and 0.0001. We optimize the latent code $\mathbf{z}^s$ for a fixed number of 700 gradient update steps. Note that this experiment does not require any color information, as we are solely interested in extracting an accurate representation of the geometry of the target shape. We therefore utilize only object masks during optimization. We compare our method with DualSDF [14] and SPAGHETTI [16] on the test set of the ShapeNet Airplanes category. Our results are summarized in Tab. 5. To measure the Chamfer-L_1 distance, we follow [16] and sample $30,000$ points on the surface of the generated and the target mesh.

	DualSDF	SPAGHETTI	Ours
Chamfer-L_1	0.806	**0.050**	0.4536

Table 5. **Shape Inversion.** Quantitative evaluation of our method against DualSDF [14] and SPAGHETTI [16] w.r.t. to Chamfer distance (↓) on the test set of the *Airplanes* category.

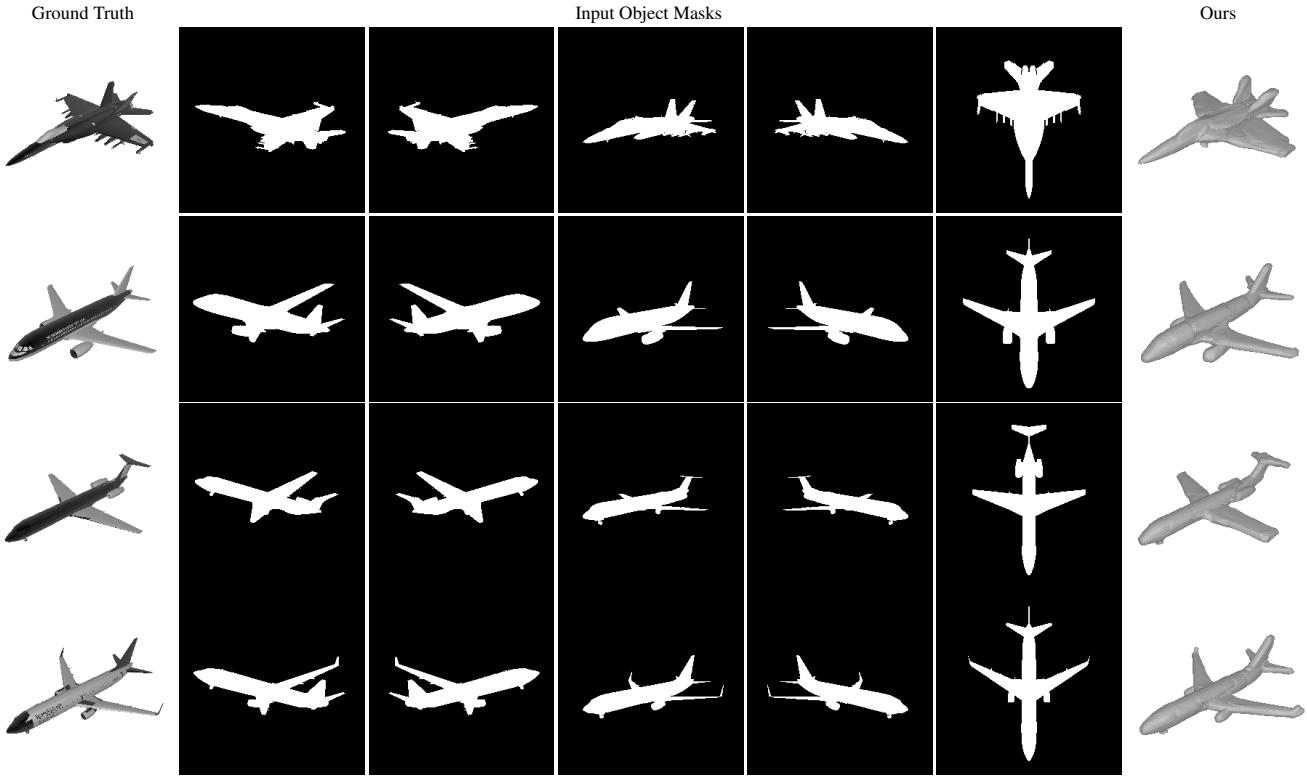


Figure 17. **Shape Inversion**. We show examples of our results when performing shape inversion on the ShapeNet airplanes.

Our model yields better reconstructions than DualSDF [14] in terms of Chamfer distance, even though [14] was trained with explicit 3D supervision. Compared to SPAGHETTI [16], our model yields worse reconstructions. We hypothesize that

conditioning our generations to more than 5 views would further improve the performance of our model. In Fig. 17, we show examples of our shape inversion experiment.

5. Image Inversion

The goal of this experiment is to recover both the appearance and geometry of a 3D shape, as opposed to only the shape as for the case of shape inversion (Sec. 4). To this end, we need to find an appropriate match for both the shape and texture embeddings $\{\mathbf{z}^s, \mathbf{z}^t\}$, given a set of posed images and masks as targets. In a similar manner to the sampling process of Sec. 1.3, we randomly initialize the shape and texture embeddings, as discussed in Sec. 1.3. Then, we freeze the network parameters and optimize the embeddings using the loss function:

$$\mathcal{L}_{image\_inversion} = \mathcal{L}_{rgb}(\mathcal{R}) + \mathcal{L}_{mask}(\mathcal{R}) + \mathcal{L}_{occ}(\mathcal{R}) + \|\mathbf{z}^s\|_2 + \left\|\mathbf{z}^t\right\|_2, \tag{16}$$

with weights of 1.0, 1.0, 1.0, 0.0001 and 0.0001. We use $N = 5$ posed images and masks as targets, and optimize the latent codes for a fixed number of 2000 gradient update steps. We showcase several inverted examples in Fig. 18.

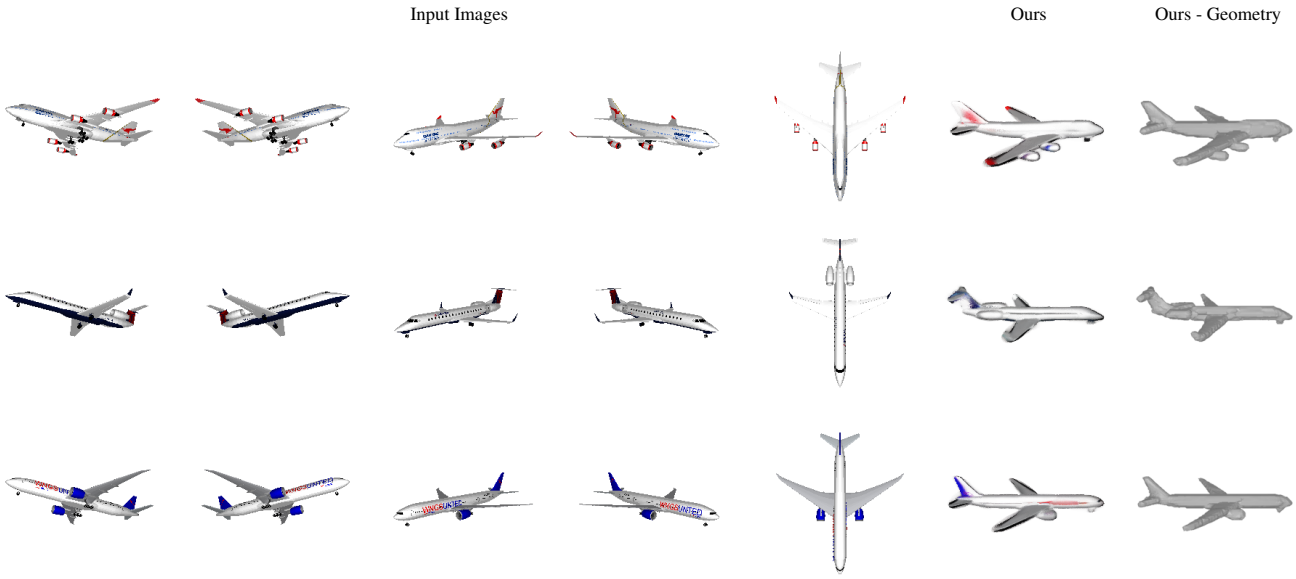
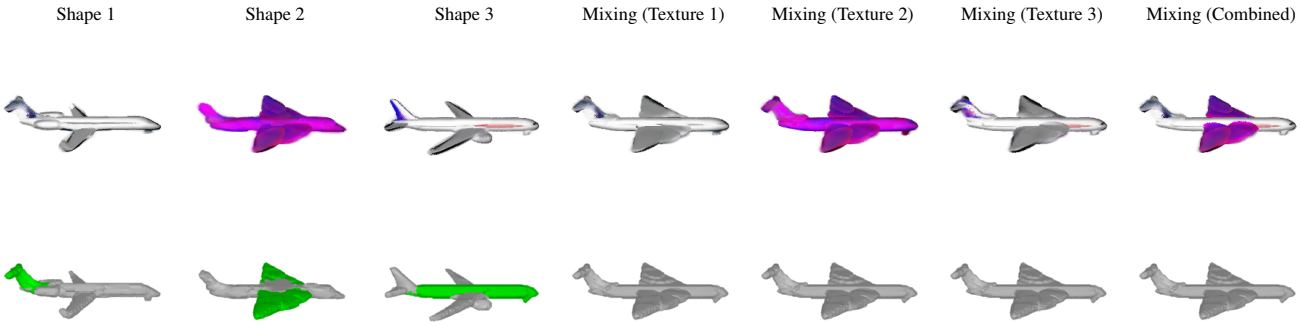


Figure 18. **Image Inversion**. We show examples of our results when performing image inversion on the ShapeNet airplanes.

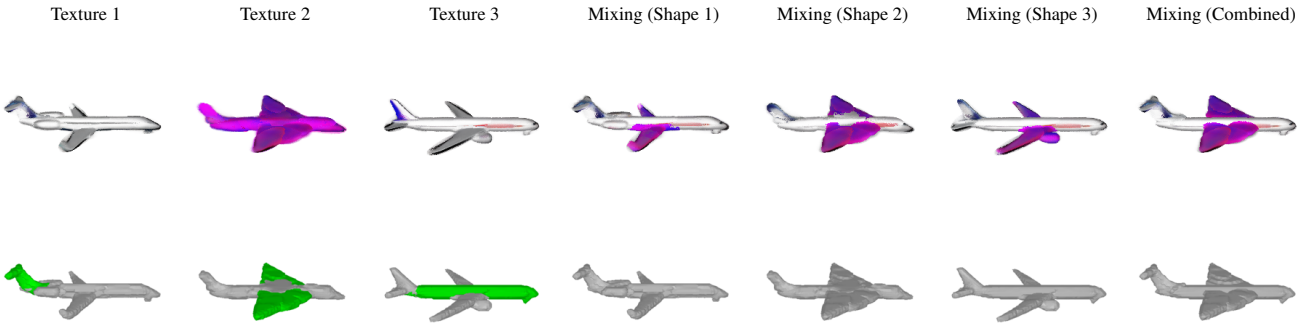
We observe that our model can faithfully recover the object geometry and appearance for various objects. Note that Image Inversion enables users to directly edit shapes, by just providing a few posed images and masks. In Sec. 6 we showcase several editing functionalities on the inverted shapes.

6. Shape Editing

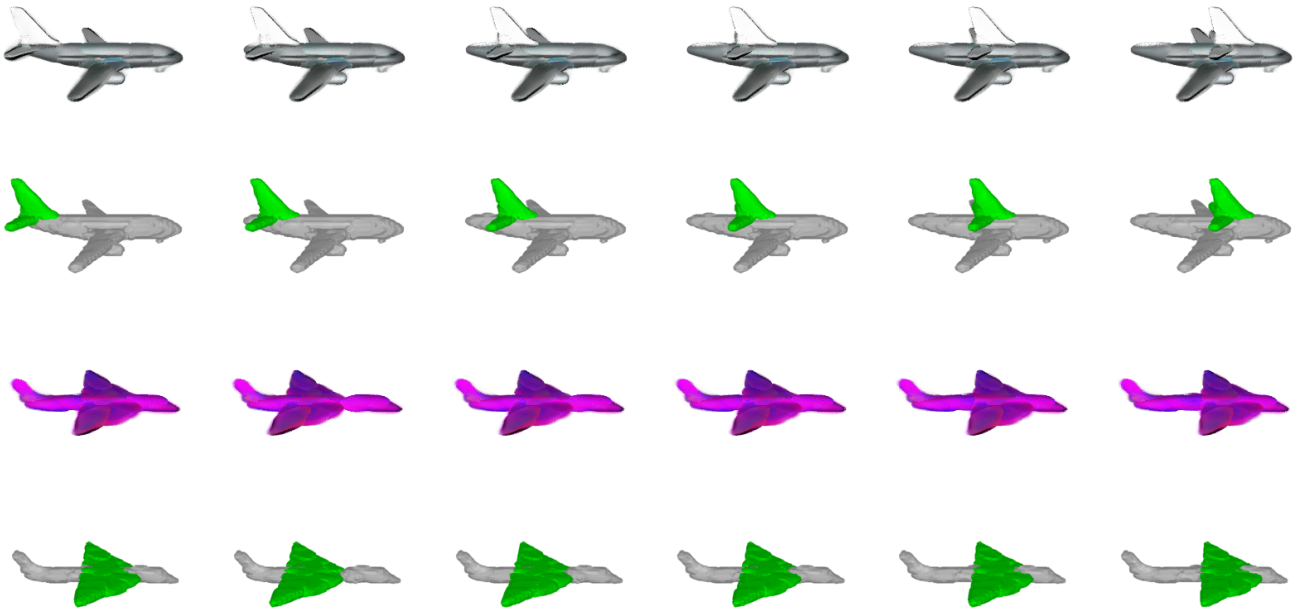
In this section, we demonstrate the editing capabilities of our model on inverted shapes from ShapeNet *Airplane*. To perform the image inversion, we follow the process described in Sec. 5. In Fig. 19, we show several examples of *geometry*, *texture* mixing and shape editing. In particular, we show two shape editing examples, (see last four rows in Fig. 19), where we select several parts from two airplanes and we translate them along the body of the two airplanes. The parts that we select correspond to the tail of the first airplane and the wings of the second, highlighted in green. Across both transformations, only the pose of the selected parts change, while their appearance/texture as well as the shape and appearance of the other parts does not change. For the case of texture mixing (see third and fourth rows in Fig. 19), we start from three airplanes and perform several texture mixing operations. For example, in the last column of the third row of Fig. 19, we show a generated airplane, whose texture and shape for the tail is from the first shape, while the texture and shape for the wings and the body are from the second and third shape respectively. Likewise, for the geometry mixing experiment (see two first rows in Fig. 19), we show examples where we mix the shapes of the three input airplanes, while keeping the texture of the first (fourth column), the second (fifth column) or the third input airplane (sixth column).



Geometry Mixing



Texture Mixing



Shape Editing

Figure 19. **Shape Editing**. We show examples of our results when performing geometry mixing, texture mixing and part editing operations on image inverted ShapeNet airplanes.

7. Comparison with Compositional NeRFs

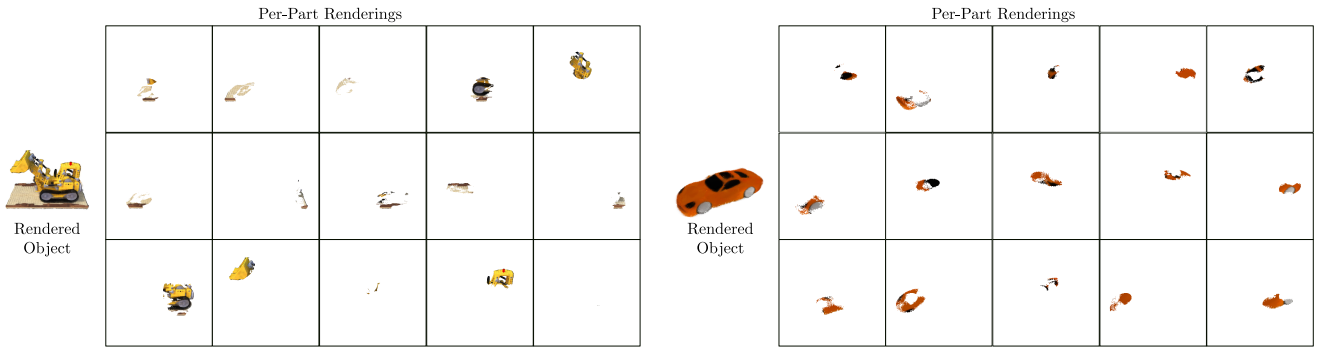
Although our work is the first part-aware generative model for editable 3D shapes, several previous works have considered decomposing a scene using multiple NeRFs to enable efficient rendering [34] or to improve the generation quality of scenes with several objects [27]. Our formulation differs from [27], as we explicitly associate rays with parts, instead of simply combining them. This is an essential property of our model that enables local control. In comparison to the scene-specific DeRF [34], that utilizes a set of independent NeRFs that describe different regions of space within a Voronoi cell with the goal to reduce the rendering time, we propose a generative model that parametrizes parts with NeRFs in order to enable local control and unlock several editing capabilities not previously possible.

8. Discussion and Limitations

In the primitive-based literature, the word part refers to geometric primitives that are typically simple shapes such as cylinders [22], cuboids [37], superquadrics [31,32], spheres [14] or more generally convex shapes [8]. More recently, the term part has also been used to describe parts that can capture complex geometries and are not limited to simple geometric shapes [11, 19, 30]. Regardless of the part's expressivity, primitive-based methods yield semantically consistent reconstructions, where the same part is consistently used for representing the same part of the object. Being semantically consistent does not necessarily mean that parts will also be semantically meaningful, namely refer to humanly identifiable parts.

As we parametrize parts by neural radiance fields, our parts can capture complex topologies that are not limited to simple geometric shapes. Examples of our generated parts can be found in Fig. 9−Fig. 16. We note that our parts can be coherent across some objects, e.g. the same part is used for representing the same region of the object, in particular when the parts have comparable size and shape. However, for two shapes with different sizes e.g a big and small car (see Fig. 10), we note that our parts are not semantically consistent, namely the part that is used to represent the tire and the top part of the car for a smaller car is now representing only the tire of a bigger car. Note that this is not a limitation only of our work but also of existing part-based methods, as they do not explicitly enforce the semantic consistency of parts.

In addition, in Fig. 20, we provide two examples of the renderings produced using our 16 NeRFs/parts, when rendering the tractor scene from a novel view and when rendering a ShapeNet car from a novel camera. Although the per-part renderings are consistently crisp and faithfully capture the appearance details of the object, they do not always correspond to semantically meaningful parts. For example, for the case of the tractor scene, our network has associated one part with the bucket of the tractor, however it uses multiple parts to capture regions of the floor, whereas ideally we would like one part to capture the entire floor. Similarly, for the case of the car, while the per-part renderings have sharp colors, they cover parts of the object that are not humanly interpretable, such as a single NeRF representing part of the tire and part of the car.



(a) Tractor Scene (b) ShapeNet Car

Figure 20. **Per-Part Renderings.** We show the per-part renderings produced by our 16 NeRF/parts for a novel view.

This is again a common issue observed in all existing part-based methods [8,30,31,37]. As these methods do not explicitly enforce the meaningfulness of their parts, oftentimes parts are associated with regions of objects that are not meaningful and do not correspond to humanly interpretable parts. We believe that in future research, it would be valuable to explore ways to enforce that parts are more interpretable, in order to unlock more editing operations, that better keep humans in the loop. Our main goal for this work, was to introduce a generative model that enables local control through parts. While increasing the number of parts enables more control on the generated shape, as we can select multiple small parts that compose a semantic part such as the wings of the airplane, we believe that it is useful to also explore other types of more semantic control, where

a user for example could select a part based on its semantic class e.g. wing of an airplane or saddle of a motorbike, without having to manually select multiple parts, as is the case for our model.

In contrast to existing NerF-based generative models [4, 5, 9, 35] that rely on adversarial losses to generate high quality renderings, our model does not employ such losses. We believe that incorporating triplane-based representations [4] or adversarial losses could further improve the quality of our generated textures. Moreover, in our current framework, moving one part of an object to a new location outside the object does not necessarily generate a contiguous shape. To address this, [16], introduced a blending network, implemented as a transformer decoder that merges the subsequent parts and synthesizes a novel object. Incorporating such a technique in our model is not possible, as we do not have access to 3D supervision in the form of a mesh. However, we could utilize [23, 42] to synthesize connections between parts.

To summarize, while our work makes an important step towards generating editable 3D shapes, it still has several limitations. In particular, during training, our supervision comes from posed images and object masks. Nevertheless, acquiring detailed masks for objects in the wild is not always possible. Therefore, we believe it would be useful to explore ways to alleviate the need for object masks. Likewise, adopting a formulation like [35] that enables training from unposed images could further extend the capabilities of our model. In addition, while our model can perform several editing operations, our current formulation does not support operations where a part is deformed using techniques such as Neural Cages [41] or Biharmonic Coordinates [39]. Incorporating deformations in our editing operations could unlock several editing applications.

9. Potential Negative Impact on Society

Our proposed model enables generating editable 3D meshes with textures. While, we see this as an important step towards automatic content creation and enabling a multitude of editing functionalities, it can also lead to negative consequences, when applied to sensitive data, such as human bodies or faces. Therefore, we believe it is imperative to always check the license of any 3D publicly available 3D model. In addition, we see the development of techniques for identifying real from synthetic data as an essential research direction that could potential prevent deep fake.

Note that throughout this work, we have only worked with publicly available datasets and did not use any data that involves privacy or copyright concerns. For future users that would like to train our model on new data, we recommend to first remove biases from the training data in order to ensure that our model can fairly capture the diversities in terms of shapes, sizes and textures.

References

[1] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J. Guibas. Learning representations and generative models for 3d point clouds. In *Proc. of the International Conf. on Machine learning (ICML)*, 2018. 7

[2] Katharopoulos Angelos and Despoina Paschalidou. simple-3dviz. https://simple-3dviz.com, 2020. 14

[3] Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks. In *Proc. of the International Conf. on Machine learning (ICML)*, 2018. 1, 9

[4] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas J. Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3d generative adversarial networks. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2022. 8, 9, 28

[5] Eric R. Chan, Marco Monteiro, Petr Kellnhofer, Jiajun Wu, and Gordon Wetzstein. Pi-gan: Periodic implicit generative adversarial networks for 3d-aware image synthesis. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 8, 9, 28

[6] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *arXiv.org*, 1512.03012, 2015. 10, 14, 19

[7] Harm de Vries, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron C. Courville. Modulating early visual processing by language. In *Advances in Neural Information Processing Systems (NIPS)*, 2017. 4

[8] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. Cvxnets: Learnable convex decomposition. *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020. 27

[9] Jun Gao, Wenzheng Chen, Tommy Xiang, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Learning deformable tetrahedral meshes for 3d reconstruction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 8, 28

[10] Jun Gao, Tianchang Shen, Zian Wang, Wenzheng Chen, Kangxue Yin, Daiqing Li, Or Litany, Zan Gojcic, and Sanja Fidler. GET3D: A generative model of high quality 3d textured shapes learned from images. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 7, 8, 9, 14

[11] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas A. Funkhouser. Local deep implicit functions for 3d shape. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020. 27

[12] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NIPS)*, 2014. 8

[13] William Rowan Hamilton. Xi. on quaternions; or on a new system of imaginaries in algebra. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 33(219):58–60, 1848. 2

[14] Zekun Hao, Hadar Averbuch-Elor, Noah Snavely, and Serge J. Belongie. Dualsdf: Semantic shape manipulation using a two-level representation. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020. 8, 9, 24, 27

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016. 4

[16] Amir Hertz, Or Perel, Raja Giryes, Olga Sorkine-Hornung, and Daniel Cohen-Or. SPAGHETTI: editing implicit shapes through part aware generation. *ACM Trans. on Graphics*, 2022. 6, 7, 8, 9, 14, 19, 20, 24, 28

[17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017. 8

[18] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of StyleGAN. 2020. 8, 9

[19] Yuki Kawana, Yusuke Mukuta, and Tatsuya Harada. Neural star domain as primitive representation. *arXiv.org*, 2020. 27

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. of the International Conf. on Learning Representations (ICLR)*, 2015. 4

[21] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Trans. on Graphics*, 2020. 8

[22] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas Guibas. Supervised fitting of geometric primitives to 3d point clouds. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019. 27

[23] Connor Z. Lin, Niloy J. Mitra, Gordon Wetzstein, Leonidas J. Guibas, and Paul Guerrero. Neuform: Adaptive overfitting for neural shape editing. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 28

[24] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Trans. on Graphics*, 1987. 9

[25] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019. 3, 4, 9

[26] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proc. of the European Conf. on Computer Vision (ECCV)*, 2020. 3, 4, 8

[27] Michael Niemeyer and Andreas Geiger. Giraffe: Representing scenes as compositional generative neural feature fields. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 27

[28] Michael Oechsle, Songyou Peng, and Andreas Geiger. UNISURF: unifying neural implicit surfaces and radiance fields for multi-view reconstruction. In *Proc. of the IEEE International Conf. on Computer Vision (ICCV)*, 2021. 3, 8

[29] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019. 1, 9

[30] Despoina Paschalidou, Angelos Katharopoulos, Andreas Geiger, and Sanja Fidler. Neural parts: Learning expressive 3d shape abstractions with invertible neural networks. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 2, 27

[31] Despoina Paschalidou, Ali Osman Ulusoy, and Andreas Geiger. Superquadrics revisited: Learning 3d shape parsing beyond cuboids. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019. 27

[32] Despoina Paschalidou, Luc van Gool, and Andreas Geiger. Learning unsupervised hierarchical part decomposition of 3d objects from a single rgb image. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020. 27

[33] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv.org*, 1606.02147, 2016. 4

[34] Daniel Rebain, Wei Jiang, Soroosh Yazdani, Ke Li, Kwang Moo Yi, and Andrea Tagliasacchi. Derf: Decomposed radiance fields. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 27

[35] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. Graf: Generative radiance fields for 3d-aware image synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 8, 9, 28

[36] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 8

[37] Shubham Tulsiani, Hao Su, Leonidas J. Guibas, Alexei A. Efros, and Jitendra Malik. Learning shape abstractions by assembling volumetric primitives. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017. 27

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, 2017. 1

[39] Yu Wang, Alec Jacobson, Jernej Barbic, and Ladislav Kavan. Linear subspace design for real-time shape deformation. *ACM Trans. on Graphics*, 2015. 28

[40] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019. 2

[41] Wang Yifan, Noam Aigerman, Vladimir G. Kim, Siddhartha Chaudhuri, and Olga Sorkine-Hornung. Neural cages for detail-preserving 3d deformations. In *CVPR*, 2020. 28

[42] Kangxue Yin, Zhiqin Chen, Siddhartha Chaudhuri, Matthew Fisher, Vladimir G Kim, and Hao Zhang. Coalesce: Component assembly by learning to synthesize connections. In *Proc. of the International Conf. on 3D Vision (3DV)*, 2020. 28

[43] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelnerf: Neural radiance fields from one or few images. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 4