DualSDF: Semantic Shape Manipulation using a Two-Level Representation – Supplementary Material –

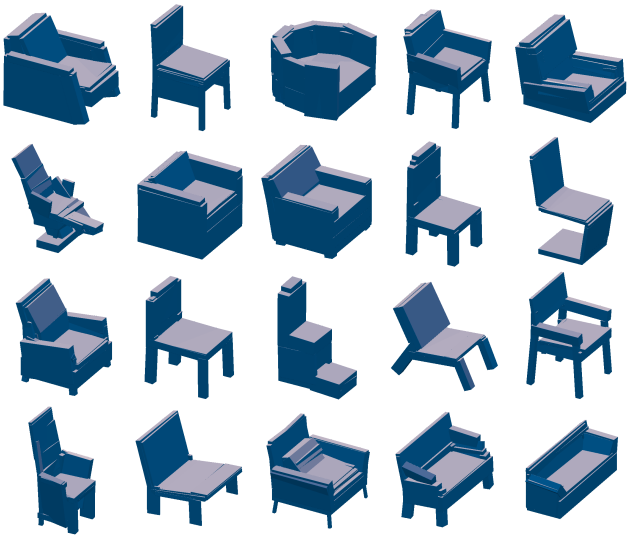

Figure 1: Learning with box primitives. Our technique is directly applicable for geometric shapes which can be represented with SDFs. Above we demonstrate coarse shape reconstructions learned by our model with box primitives.

1. DualSDF with Box Primitives

In our work, we use sphere primitives for our coarse representation. In the main paper, we also show reconstruction results obtained with capsule primitives. In Figure 1, we demonstrate that box primitives can be utilized in our framework as well. In fact, our framework is very flexible in terms of primitive choice. Any primitive that can be represented with signed distance function can be incorporated into the coarse representation.

2. Effect of Latent Code Regularization on Shape Manipulation

Figure 3 shows the effect of latent code regularization term $L_{\texttt{REG}}$ on the shape manipulation process. Empirically, a latent code with high likelihood under the prior $p(\mathbf{z})$ usually decodes to more plausible shapes. $L_{\texttt{REG}}$ keeps the latent code from deviating too far from the prior during the manipulation process, improving the quality of the result shape. From a user's perspective, the regularization term guide the user input towards more semantically meaningful shapes by moving the unconstrained primitives to the correct places and guarding the user input against unreasonable configurations.

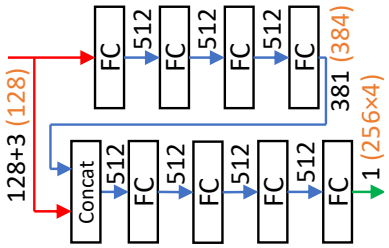

Figure 2: Network structure. The different settings used for the primitive-based representation are marked by parenthesis.

3. Analysis of Running Time

As mentioned in the main paper, our shape manipulation framework is able to run at real time. Here we provide a more comprehensive analysis on the running time of our model to further back up our claim. All of the benchmarks are implemented with PyTorch and run on a single GTX 1080ti GPU. We assume a single-user scenario, where the batch size is only one.

For the primitive-based representation section, it takes an average of 0.66ms to obtain the primitive attributes from latent code, and it takes an average of 2.74ms to perform one gradient descent step (including forward and backward) to update the latent code and to obtain the attributes of the updated shape, after receiving the user input. Even with our less-that-optimal implementation, this is already fast enough to provide the user with real-time feedback. Once the primitive attributes are obtained, the shape can then be rendered conveniently and rapidly with real time rendering engines and hardware acceleration.

For the high-resolution representation, assuming we are rendering the SDF with ray-marching method, there are two parameters that determine the render quality and speed: image resolution and number of ray-marching steps. The

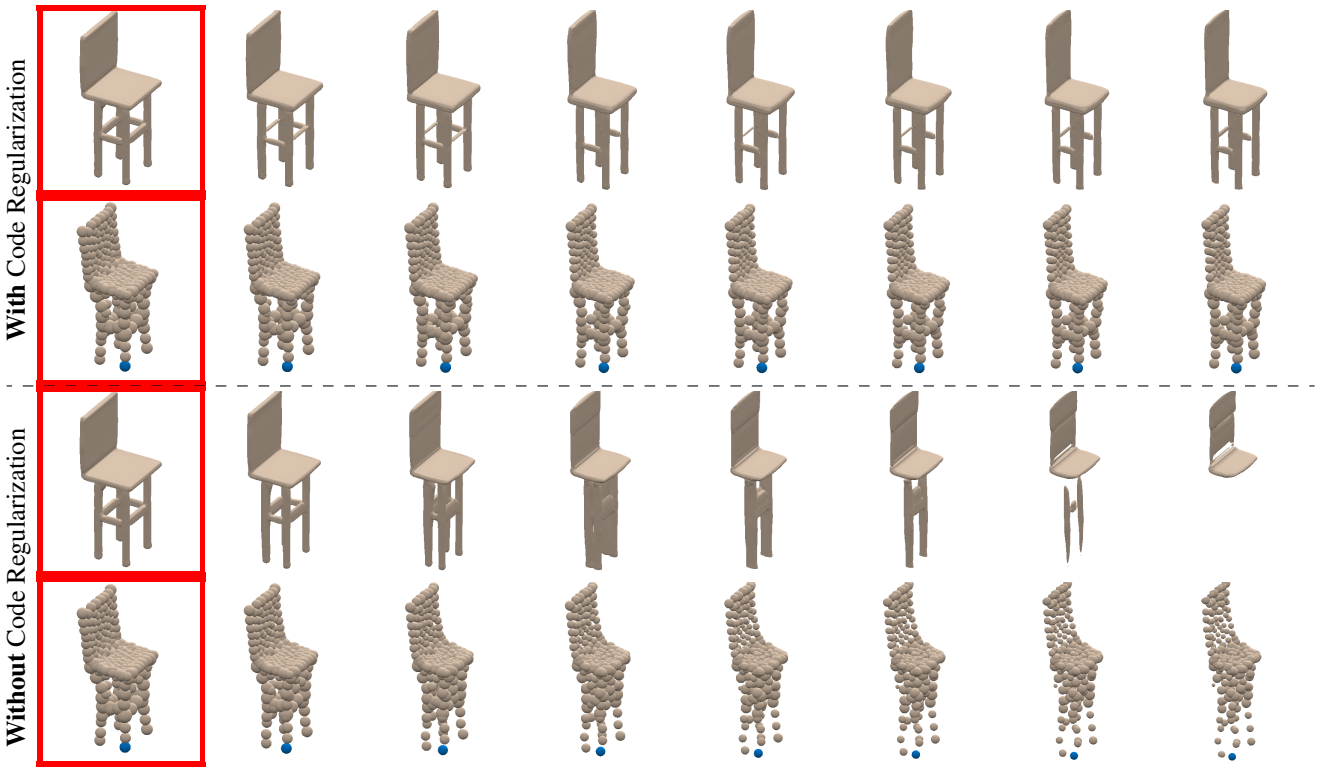

Figure 3: The effect of the latent code regularization term in the shape manipulation objective on the quality of the resulting shape. Here we move the blue sphere downwards in an attempt to make the legs of the chair longer. From left to right we show the original shape (marked in red boxes) and the intermediate shapes during the manipulation process.

effects of the two parameters on image quality and rendering time are presented in Figure 4. All the renderings shown in the main paper are rendered with 64 iterations at a resolution of 480×480. Although at full resolution and highest quality, the rendering of high-resolution representation is not as fast as the primitive-based representation, a reasonable trade-off between time and quality can be obtained. During interactive manipulation, we can present the user with reduced resolution rendering at a reduced rate, in addition to the real-time rendering of the primitive-based representation, and render the full resolution result only when needed.

4. Detailed Experimental Setting

We use a 128-dimensional latent code $\mathbf{z}_j$ throughout the experiments. For the high-resolution SDF representation, we use a 8-layer MLP with one cross-connection. The 131-dim input is the concatenation of the 128-dim latent code $\mathbf{z}_j$ and the 3D coordinate $\mathbf{p}$. The output is the predicted SDF value at this 3D coordinate. For the primitive-based representation, we use the same network architecture with $\mathbf{z}_j$ as the only input, and the attributes (center coordinates and log radii, denoted by $\boldsymbol{\alpha}_j$ in the main paper) of 256 spheres as output. The networks are shown in Figure 2. Weight normalization is used on all the fully connected layers. We use ReLU activation on all but the last fully connected layers. We use dropout with a probability of 0.2 on the output of all but the last fully connected layers, only in the high-resolution network g_θ.

For all the experiments, we use Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rates are 5e-9 for θ, ϕ and 1e-8 for $\boldsymbol{\mu}_j, \boldsymbol{\sigma}_j$. Each batch consists of 64 shapes; for each shape we sample 2048 SDF values for the high-res representation and 1024 for the primitive-based representation. We train the model for 2800 epochs and drop the learning rate by 50% after every 700 epochs. We empirically set $\lambda_1 = \lambda_2 = 1\text{e}5$. Their values affect the trade-off between latent space compactness and reconstruction quality due to D_{KL}.

5. SDF Losses

We use truncated SDF loss on both coarse and fine shape representations during training and shape encoding. This has been shown beneficial in DeepSDF. For high resolution representation, we truncate the SDF on both inside and

Output							
#Steps	16	24	32	40	48	56	64
Resolution	480 × 480	480 × 480	480 × 480	480 × 480	480 × 480	480 × 480	480 × 480
Time (s)	2.35	3.36	4.36	5.37	6.39	7.40	8.41

Output							
#Steps	64	64	64	64	64	64	64
Resolution	80 × 80	160 × 160	240 × 240	320 × 320	400 × 400	480 × 480	560 × 560
Time (s)	0.26	0.95	2.11	3.73	5.85	8.44	11.49

Figure 4: Trade-off between rendering quality and speed on the fine-scale representation. Higher resolution and larger number of ray-marching iterations lead to better image quality, at the cost of longer running time. Our model is capable of adjusting the trade-off on the fly to adjust to different scenarios. For example, we can render shapes in lower quality during interactive manipulation, and render a high quality result once the manipulation is done.

outside:

$$L_{\text{SDF}}^{\text{fine}}(d,s) = \begin{cases} \max(d,-\delta)+\delta & s < -\delta, \\ |d-s| & -\delta \le s \le \delta, \\ \delta - \min(d,\delta) & s > \delta. \end{cases} \quad (1)$$

For primitive-based representation, we only truncate the SDF inside the shape to zero, as the SDF outside the shape is guaranteed to be valid (metric):

$$L_{\text{SDF}}^{\text{coarse}}(d,s) = \begin{cases} \max(d,0) & s < 0, \\ |d-s| & s \ge 0. \end{cases} \quad (2)$$

6. Additional Results

In Figure 5, we show additional shape manipulation results on Chair and Airplane collections. Please refer to the accompanying video for *full* sequences on additional shapes. Note that while these results are obtained using simple editing operations (such as dragging a single primitive along one axis), more complicated operations on multiple primitives can be achieved in a similar manner (as we show in the main paper).

We demonstrate shape interpolation results obtained in two ways. Figure 6 shows results on linearly interpolating the latent code, while Figure 7 illustrates a novel way of partially interpolating between two shapes by optimizing the primitive parameters, as we propose in the main paper. The latter method allows selectively interpolating certain characteristics of the shapes, such as the outline of the shape.

As illustrated in Figure 7, in the top two rows, we perform optimization on the primitive attributes to encourage the coarse representation of the left chair to match the coarse representation of the right chair. This effectively interpolates the outlines of the chairs while keeping the fine details on the left chair intact. This cannot be achieved with standard interpolation (Figure 6, top two rows). Similarly, in the bottom two rows, we use L1 loss to encourage the heights (y-coordinates) of the primitives on the left chair to match the heights of the corresponding primitives on the right chair, while allowing other attributes to change freely during the optimization. Note that finding the correspondences of primitives between two shapes is trivial since each primitive generally stays at the same position across different shapes within a single class, as we illustrate in the main paper.

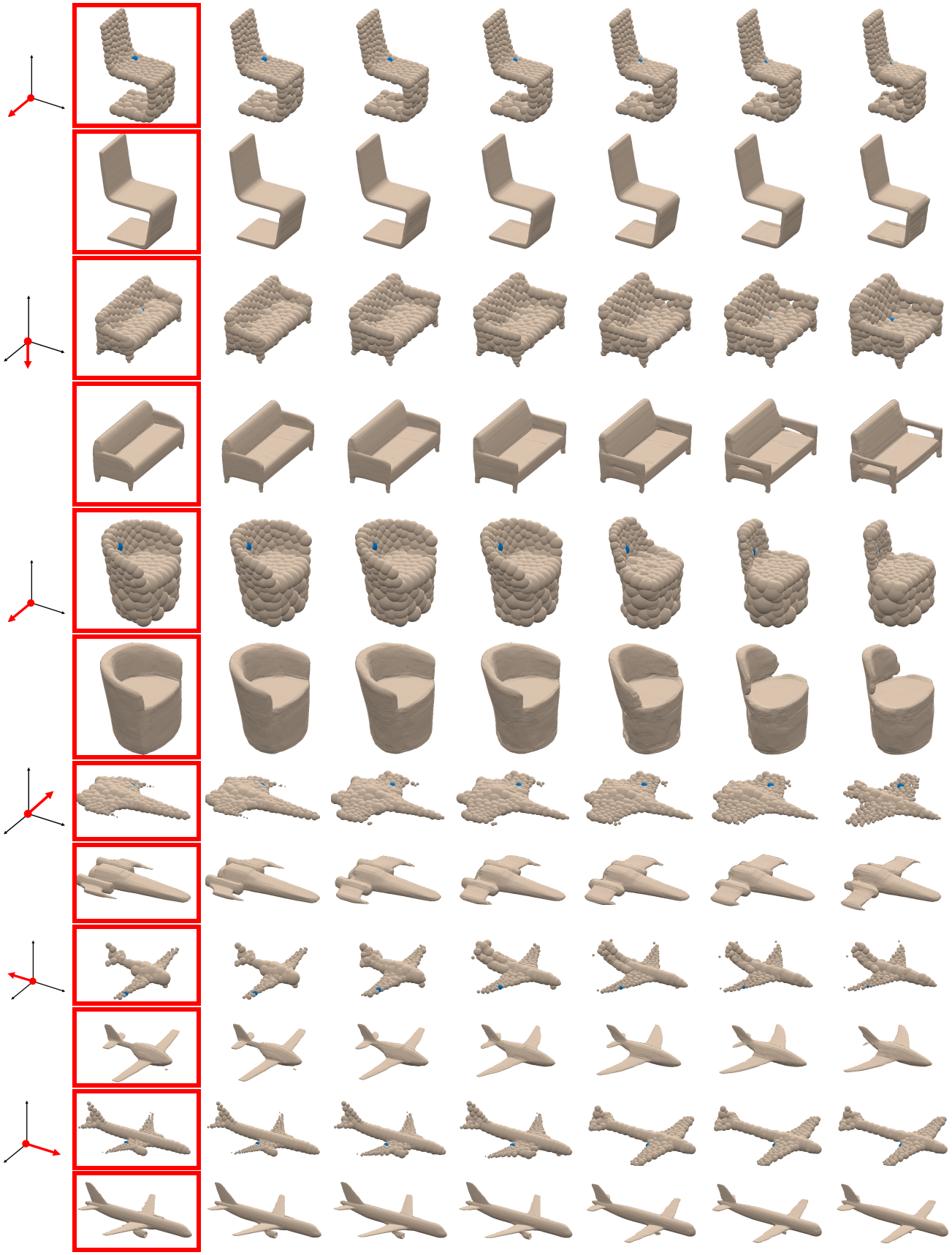

Figure 5: Manipulating shapes (on the left) by dragging a single primitive (colored in blue) along a specified direction (red arrow on the left). Please refer to the accompanying video for full manipulated sequences.

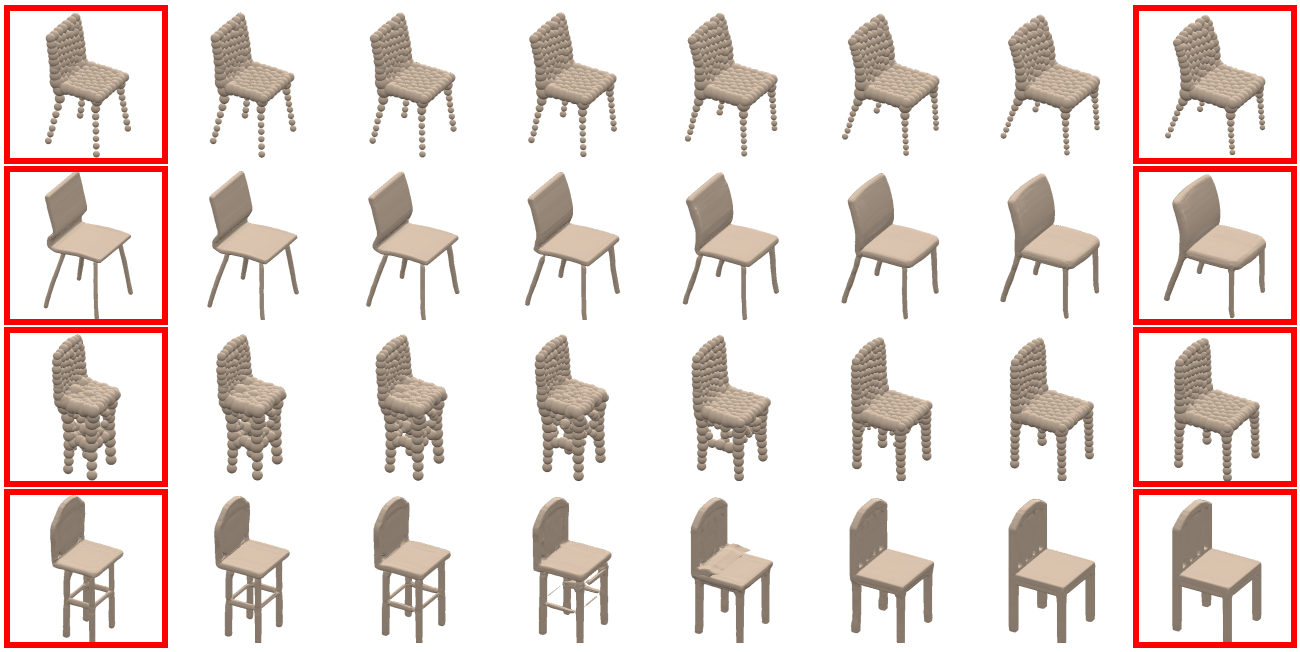

Figure 6: Linear interpolation in the latent space between pairs of shapes. The original shape pairs are marked with red boxes while the shapes in between are generated from interpolated latent code. Accompanying each shape is its corresponding primitive-based representation.

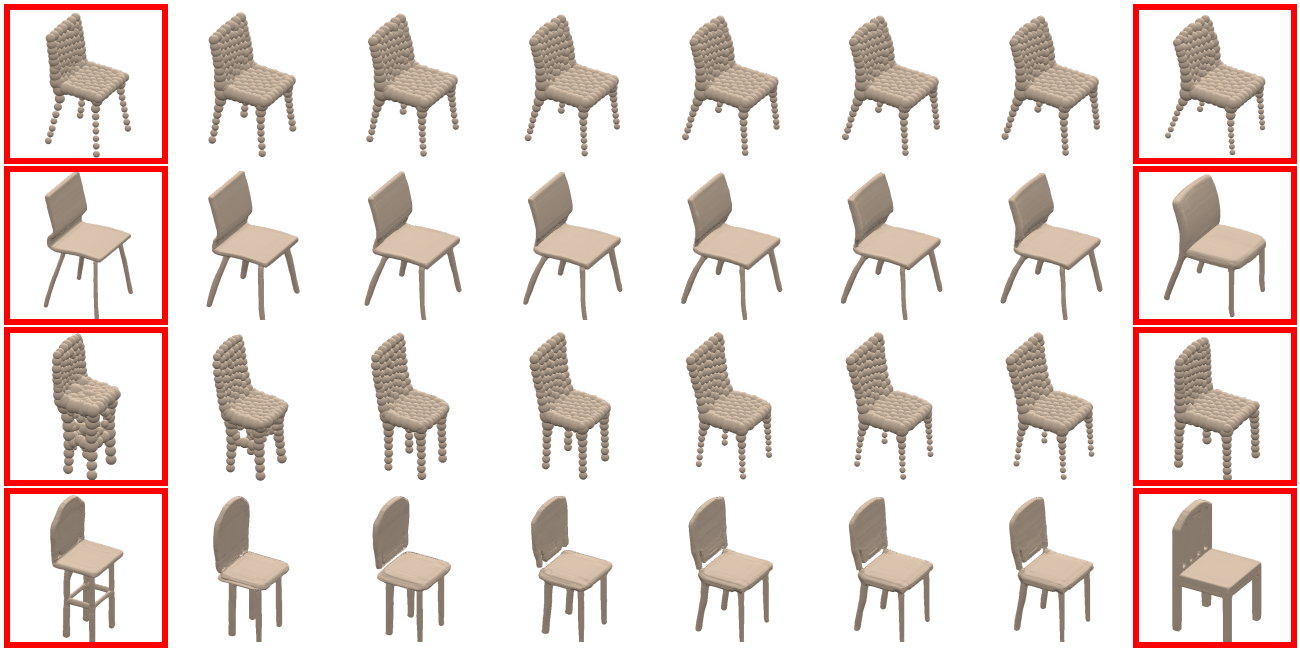

Figure 7: Selective primitive-space interpolation on the same pairs of shapes shown in Figure 6. This type of interpolation supports selectively interpolating some of the attributes. For the first two rows, the outlines of the chairs are interpolated by gradually matching the coarse representation of the left chair to the right chair. Note how the fine details of the left chair are preserved. Similarly, for the bottom two rows, the heights of the primitives of the left chair are gradually matched to the right chair, while everything else is allowed to change freely. Many key features of the left chair are preserved in this process.