End-to-End Fine-Tuning of 3D Texture Generation using Differentiable Rewards

Supplementary Material

A. Remarks on The Texture Generation Step

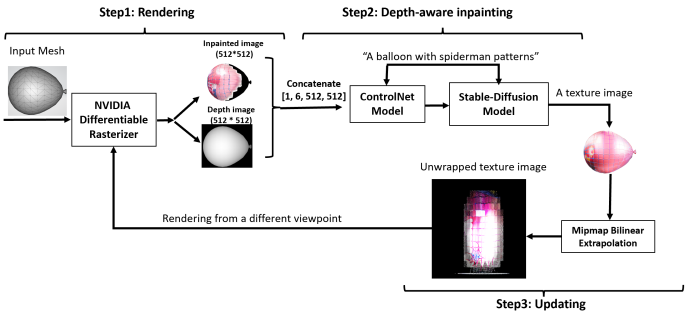


Figure 4. Visualization of three main stages in the *texture generation*: (i) rendering, render the object from a camera viewpoint using a differentiable renderer and extract a rendering buffer, (ii) depth-aware painting, given a text prompt, each viewpoint is painted using a depth-aware text-to-image diffusion model, guided by a pre-trained ControlNet that provides depth information. This ensures the generated textures align with both the text and depth (geometry) information, and (iii) update the final texture. We repeat this process iteratively across all camera viewpoints until the full 3D surface is painted.

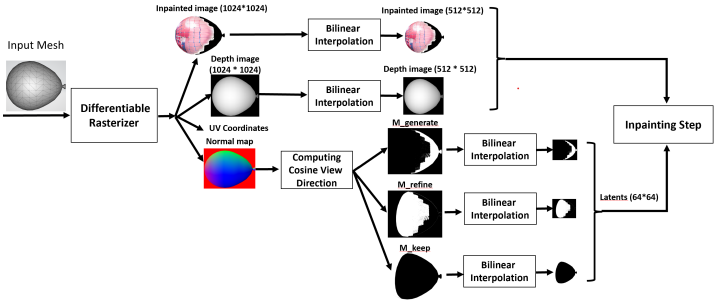


Figure 5. Visualization of *rendering* in the *texture generation* step. For each camera viewpoint, we render the object using a differentiable renderer and extract rendering buffer including painted viewpoint image, depth maps, normal maps, and UV coordinates. Then, using the normal map, obtained from the differentiable renderer, we compute the *view direction cosine*, and then generate three regions (masks), explained above, for each viewpoint: $M_{generate}, M_{refine}$ and M_{keep}. These three regions will serve as input to the next step of the texturing process to enforce the consistency in the output texture.

Due to the stochastic nature of the diffusion models, applying the iterative painting process for texturing process, explained in Sec. 3.1, would leads to having inconsistent textures with noticeable seams on the appearance of the object. To resolve this issue, we take a dynamic partitioning approach similar to [5, 29], where each rendered viewpoint is divided to three regions based on a concept called *view direction cosine*. The *view direction cosine* is utilized to determine how the surface is facing the viewer. Mathematically, the view direction cosine is represented as the cosine of the angle between two vectors: the *view direction vector* which is the direction of the viewer's (in our case, the camera) line of sight, and *mesh's surface normal vectors* associated with each face or vertex of the mesh. These normals indicate the direction perpendicular to the surface at that point (vertex):

$$\text{viewcos } = \cos(\phi) = \frac{\vec{V}.\vec{N}}{|\vec{V}| \cdot |\vec{N}|} \tag{11}$$

where $\vec{V}$ is the view vector, a vector starting from the camera position points toward a point on the surface of the 3D mesh object, $\vec{N}$ is the surface normal, vector perpendicular to the mesh surface, and ϕ is the angle between the view vector and surface normal. During the rendering process, this helps determine how to project and render 3D objects onto a 2D screen, ensuring that objects are viewed from the correct perspective. More specifically, the *view direction cosine* is computed for each rendered viewpoint, and the rendered viewpoint is partitioned into three regions based on the value of *view direction cosine*: 1) *generate* - the regions have been viewed for the first time by the camera and have never been painted during the texturing process, 2) *refine* - the regions that have already been viewed and painted from a viewpoint in previous iterations, but is now seen from a better camera angle (i.e. higher *view direction cosine* values) and should be painted again, and 3) *keep* - the regions that have already been viewed and painted from a good camera angle and should not be painted again. Appendix A.1 presents further details and the differentiable mathematical representation of our *texture generation* step.

A.1. Mathematical Representation

Mathematically, the texture-generation process can be formulated as steps below (see Fig. 4): (i) *rendering*, render the object using a differentiable renderer [17] and extract a rendering buffer h^i corresponding to the camera viewpoint i, including painted viewpoint image, depth maps, normal maps, and UV coordinates:

$$h^i = f_{render}(a, v_{gen}) \tag{12}$$

where a is the 3D asset's parameters including texture image, the *view direction cosine* for the current view, *view direction cosine* for previous iterations, v_{gen} is the camera

viewpoints for texture-generation process, and $i \in v_{gen}$ is the current viewpoint. Using the normal map, obtained from the differentiable renderer, we then compute the *view direction cosine*, and then generate three regions (masks), explained above, for each viewpoint: $M_{generate}, M_{refine}$ and M_{keep}. These three regions will serve as input to the next step of the texturing process to enforce the consistency in the output texture. Fig. 5 visually demonstrates the details of the output generated by the differentiable renderer. (ii) *Depth-aware painting*, at each denoising step t, the depth-aware diffusion process can be formulated as below:

$$\begin{aligned} y_{t-1}^i &= f_{controlnet}(h^i, t, c) \\ x_{t-1}^i &= \mu(x_t^i, y_{t-1}^i, t, c) + \sigma_t z, \quad z \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \end{aligned} \tag{13}$$

where $f_{controlnet}$ represents the ControlNet model whose parameters are fixed during the training, h^i is the differentiable renderer output, y_{t-1}^i is the control signal generated by the ControlNet model at timestep t for viewpoint i, and c is the prompt embedding obtained from the CLIP encoder model [26]. To enforce the consistency during the painting process, we follow [2] to incorporate latent mask blending in the latent space of the diffusion process. Specifically, by referring to the part that we wish to modify as $m_{blended}$ and to the remaining part as $m_{keep} = 1 - m_{blended}$, we blend the two parts in the latent space, as the diffusion progresses. This modifies the sampling process in a way that the diffusion model does not change the *keep* regions of the rendered image. Therefore, the latent variable at the current diffusion timestep t is computed as:

$$z_t^i \leftarrow z_t^i \odot m_{\text{blended}} + z_{O_t}^i \odot (1 - m_{\text{blended}}) \tag{14}$$

where $z_{O_t}^i$ is the latent code of the original rendered image with added noise at timestep t and for the viewpoint i and $m_{blended}$ is the painting mask defined below:

$$m_{blended} = \begin{cases} M_{\text{generate}}, & t \leq (1-\alpha) \times T \\ M_{\text{generate}} \cup M_{\text{refine}}, & t > (1-\alpha) \times T \end{cases} \tag{15}$$

where T is the total number of diffusion denoising steps and α is the refining strength which controls the level of refinement over the viewpoints - the larger it is, the less strong refinement we will have during the multi-viewpoint texturing process. Lastly, when the blending latent diffusion process is done, we output the image by decoding the resultant latent using the pre-trained variational autoencoder (VAE) existing in the stable diffusion pipeline.

A.2. Modifications To Make Texture Generation Step Differentiable

Differentiable Camera Pose Computation. Instead of using traditional camera positioning, to enable end-to-end training with backpropagation through the camera positioning steps, we provide a re-implementation of this procedure using PyTorch [23] operations, preserving gradient flow. More specifically, camera poses are computed using statically generated camera-to-world transformation matrices from spherical coordinates (elevation and azimuth). This process involved the following steps: (i) spherical to cartesian conversion - given azimuth ϕ, elevation θ, and radius r, the camera position $c \in \mathbb{R}^3$ is computed as:

$$\begin{aligned} x &= r\cos(\theta)\sin(\phi), \\ y &= -r\sin(\theta), \\ z &= r\cos(\theta)\cos(\phi), \\ \mathbf{c} &= [x, y, z]^\top + \mathbf{t}, \end{aligned} \tag{16}$$

where $\mathbf{t}$ is the target point. (ii) Look-at matrix construction - the rotation matrix $R \in \mathbb{R}^{3\times3}$ is computed using a right-handed coordinate system by constructing orthonormal basis vectors:

$$\begin{aligned} \mathbf{f} &= \text{ normalize } (\mathbf{c} - \mathbf{t}) \quad \text{(forward vector)}, \\ \mathbf{r} &= \text{ normalize } (\mathbf{up} \times \mathbf{f}) \quad \text{(right vector)}, \\ \mathbf{u} &= \text{ normalize } (\mathbf{f} \times \mathbf{r}) \quad \text{(up vector)}, \end{aligned} \tag{17}$$

where $\mathbf{up} = [0, 1, 0]^T$. The camera pose matrix $T \in \mathbb{R}^{4\times4}$ is then presented as:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{c} \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{18}$$

This modification transforms a previously non-trainable pre-processing step into a fully differentiable component, allowing camera positions and orientations to participate in gradient-based learning, particularly during texture preference learning where the texture is optimized to match geometry-aware reward signals.

Differentiable Viewpoint Dynamic Partitioning (Masking). In the original implementation of the texture generation step [33], dynamic viewpoint partitioning for multi-view texturing was performed using hard, non-differentiable binary masks based on thresholding and bitwise operations. Specifically, three disjoint masks were used to guide different behaviors across image regions during training: (i) a generation mask $M_{generate}$, (ii) a refinement mask M_{refine}, and (iii) a keep mask M_{keep}. Let $c \in \mathbb{R}^{H\times W}$ denotes the count map (indicating coverage across viewpoints), and $v, v_{old} \in \mathbb{R}^{H\times W}$ be the view-consistency scores for the current and cached viewpoints, respectively. The original masks were defined as:

$$\begin{aligned} M_{\text{generate}} &= 1[c < 0.1] \\ M_{\text{refine}} &= 1\left[v > v_{\text{old}}\right] \cdot \neg M_{\text{generate}} \\ M_{\text{keep}} &= \neg M_{\text{generate}} \cdot \neg M_{\text{refine}} \end{aligned} \tag{19}$$

where $1[\cdot]$ denotes the indicator function and logical operators like, $\neg$ produce discrete masks, making them non-differentiable and unsuitable for gradient-based optimization. To enable end-to-end differentiability, we replace these hard decisions with soft, continuous approximations using the sigmoid function. Let $\sigma_k(x) = \frac{1}{1+\exp(-kx)}$ denote the sigmoid function with steepness k. We define soft masks as follows: (i) soft generation mask -

$$M^{\text{soft}}_{\text{generate}} = 1 - \sigma_k(c - 0.1) \tag{20}$$

This approaximates $1[c < 0.1]$, where $k = 100$ ensures a sharp transition.(ii) soft refinement mask -

$$M^{\text{soft}}_{\text{refine}} = \sigma_k\left(v - v_{\text{old}}\right) \cdot \left(1 - M^{\text{soft}}_{\text{generate}}\right) \tag{21}$$

This replaces the hard bitwise AND with elementwise multiplication and soft comparison. (iii) soft keep mask -

$$M^{\text{soft}}_{\text{keep}} = \left(1 - M^{\text{soft}}_{\text{generate}}\right) \cdot \left(1 - M^{\text{soft}}_{\text{refine}}\right) \tag{22}$$

These soft masks retain the semantics of the original binary partitioning but allow gradients to propagate through all mask operations, enabling joint optimization with the diffusion model during our texture preference learning step. We also apply bilinear interpolation in the zooming operation to maintain spatial smoothness across render resolutions.

B. Remarks on Memory-saving Details in The Texture Preference Learning Step

Activation Checkpointing. Gradient or activation checkpointing [6] is a technique that trades compute for memory. Instead of keeping tensors needed for backward alive until they are used in gradient computation during backward, forward computation in checkpointed regions omits saving tensors for backward and recomputes them during the backward pass. In our framework, we apply two levels of gradient checkpointing to our pipeline. First, *low-level checkpointing*, where we apply the checkpointing technique to the sampling process in the diffusion model by only storing the input latent for each denoising step, and re-compute the UNet activations during the backpropagation. Second, *high-level checkpointing*, where we apply the checkpointing technique to every single-view texturing process in our multi-view texturing algorithm. Practically, what it means is that, for each viewpoint during the multi-view iterative texturing process, intermediate tensors inside the per-view texturing are not saved. Instead, only the inputs (camera pose, prompt embeddings, etc.) are saved. Then, during the backward pass, we recompute the forward pass of the per-view texturing again so gradients can flow backward. Overall, this technique allows to keep memory usage constant even across many viewpoints and diffusion steps which consequently leads to scaling to more views and using higher-resolution renders or more diffusion steps to have a more realistic and visually appealing texture image without any memory issues.

Low-Rank Adaptation (LoRA). LoRA is a technique, originally introduced in [14] for large language models, that keeps the pre-trained model weights fixed and adds new trainable low-rank matrices into each layer of the neural architecture which highly reduces the number of trainable parameters during the training process. Mathematically, for a layer with base parameters W whose forward pass is $h = Wx$, the LoRA adapted layer is $h = Wx + BAx$, where A and B are the low-rank trainable matrices and the W remains unchanged during the training process. In our case, we apply the LoRA techniques to the U-Net architecture inside the latent diffusion model (LDM) [19] which lies at the heart of our texture generation step. This allows us to fine-tune the LDM using a parameter set that is smaller by several orders of magnitude, around 1000 times fewer, than the full set of LDM parameters.

C. Remarks on Principal Curvature Directions

Principal curvature directions at a point on a surface describe how the surface bends in different directions. Curvature is defined as $k = \frac{1}{R}$ where R is the radius of the circle that best fits the curve in a given direction. Among all possible directions, two orthogonal directions, called the principal curvature directions, are of special interest: the maximum and minimum curvature directions. The minimum curvature direction corresponds to the direction in which the surface bends the least (i.e., the largest fitting circle), while the maximum curvature direction corresponds to the direction in which it bends the most (i.e., the smallest fitting circle). See Fig. 6 for an illustration. Additionally, the mean curvature at a point is defined as the average of the two principal curvatures: $H = \frac{1}{2}(k_1 + k_2)$. We use the Libigl library [15] to compute both the mean curvature values and the principal curvature directions on our 3D mesh surfaces.

D. Emergence of Repetitive Patterns through Geometry-Aligned Sampling.

An interesting observation in our results (see Fig. 3 and Fig. 12) is the emergence of repetitive texture patterns after fine-tuning with the geometry-texture alignment reward. This behavior stems from the mismatch between the number of curvature vectors and the number of texture gradients. Specifically, curvature is a vertex-level attribute defined per mesh vertex, whereas texture gradients are defined per pixel in the 2D UV space. For example, in the bunny mesh example, we have 4487 curvature vectors (one per vertex), but over one million texture gradients from a 1024×1024 texture image. To compute cosine similarity between curvature vectors and texture gradients, both must be in the same space and of the same dimension. We address this by

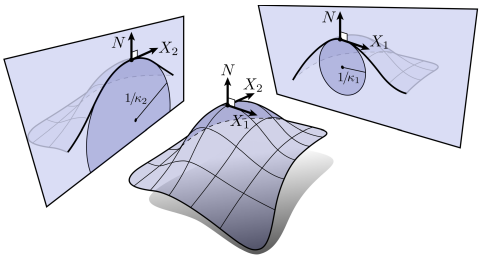


Figure 6. Visualization of principal curvatures (Image Credit: Keenan Crane). The minimum curvature direction corresponds to the direction in which the surface bends the least (i.e., the largest fitting circle), while the maximum curvature direction corresponds to the direction in which it bends the most (i.e., the smallest fitting circle).

sampling the texture gradients at the UV coordinates corresponding to mesh vertices, the same positions where curvature vectors are defined. This results in two aligned tensors (both of size 4487 in the bunny case), enabling the reward function to compare directional alignment directly. Over time, this differentiable sampling mechanism guides the model to place strong texture features (e.g., edges) at those specific UV coordinates. As these positions remain fixed (being tied to the mesh vertices), the model learns to reinforce edge features at the same UV locations across views and iterations. This alignment results in visually repetitive patterns in the generated textures, especially in regions of high curvature.

E. Remarks on Texture Features Emphasis Reward

This reward encourages texture patterns that emphasize 3D surface structure while maintaining perceptual richness through color variation. It consists of two components: (i) *texture-curvature magnitude alignment* which strengthens texture variations in high-curvature regions to enhance capturing geometric structures, and (ii) *colorfulness* inspired by [11, 13], which discourages desaturated textures by encouraging vibrant color use, measured through the standard deviation and mean distance of opponent color channels from neutral gray. Let $T(x, y) \in [0, 1]^3$ denote the RGB texture at pixel (x, y) and $C(x, y) \in [0, 1]$ be the normalized curvature map in UV space. The *texture-curvature alignment* term is mathematically defined as the negative mean squared error (MSE):

$$R_{magnitude} = -\frac{1}{N}\sum_{x,y}(\nabla \mathbf{T}(x, y) - C(x, y))^2 \quad (23)$$

where $\nabla \mathrm{T}(x, y) = \sqrt{\left\|\frac{\partial \mathrm{T}}{\partial x}(x, y)\right\|^2 + \left\|\frac{\partial \mathrm{T}}{\partial y}(x, y)\right\|^2}$ is the texture gradient magnitude and N is the total number of pixels in the texture image.

Then, we augment this reward with a colorfulness term based on opponent color channels [13]. Let $R(x, y), G(x, y), B(x, y) \in \mathbb{R}^{H\times W}$ denote the red, green, and blue channels of the texture. We compute the opponent axes:

$$rg = R - G, \quad yb = \frac{1}{2}(R + G) - B \quad (24)$$

the *colorfulness* term is then defined as:

$$R_{color} = \sigma_{rg} + \sigma_{yb} + 0.3\,(|\mu_{rg}| + |\mu_{yb}|) \quad (25)$$

where σ_{rg}, σ_{yb}, μ_{rg}, and μ_{yb} are the standard deviations and mean colors offsets corresponding to color opponent components $rg(x, y) = R(x, y) - G(x, y)$ and $yb(x, y) = \frac{R(x,y)+G(x,y)}{2} - B(x, y)$, respectively. The full *texture features emphasis* reward combines both components:

$$R_3 = \alpha_m R_{magnitude} + \alpha_c R_{color} \quad (26)$$

where α_m and α_c are weights for the *texture-curvature magnitude alignment* and *colorfulness* terms, respectively.

F. Remarks on Geometry-Aware Texture Colorization

F.1. Geometry-Aware Texture Colorization Reward Design

To encode surface curvature into texture color, we design a reward function that encourages warm colors (e.g., red, yellow) in regions of high curvature and cool colors (e.g., blue, green) in regions of low curvature. Given a per-pixel scalar curvature map $C(x, y) \in [-1, 1]$, and a curvature threshold $T \in [-1, 1]$, let $I_r(x, y)$ and $I_b(x, y)$ denote the red and blue channels of the predicted texture at pixel (x, y), respectively. For pixels where $C(x, y) > T$, we encourage warm colors by rewarding a higher red-blue channel difference $\Delta_{rb}(x, y) = I_r(x, y) - I_b(x, y)$. Conversely, for $C(x, y) < T$, we encourage cool colors by penalizing this difference. The final reward is the average Δ_{rb} across all pixels:

$$R_2 = \frac{1}{N}\sum_{x,y}\begin{cases}\Delta_{rb}(x, y), & \text{if } C(x, y) > T\\ -\Delta_{rb}(x, y), & \text{if } C(x, y) \le T\end{cases} \quad (27)$$

where N is the total number of pixels. This formulation captures the desired color-geometry correlation. However, its hard conditional logic makes it non-differentiable and unsuitable for gradient-based training. To resolve this, we replace hard thresholds with smooth, sigmoid-based approximations, enabling end-to-end differentiability.

Note that to compute the curvature-quided colorization reward, we need the mean curvature 2D map as a way to incorporate mesh's geometry information into texture learning objectives (Eq. (27)). To this end, we construct a mean

curvature 2D map, a 2D image in UV space that encodes the mean curvature of the underlying 3D mesh surface and then compute the rewards in the 2D UV space by comparing the texure image and the 2D curvature map. This process involves the following key steps.

UV-Pixel mapping. Each mesh vertex is associated with a UV coordinate $(u, v) \in [0, 1]^2$. These UV coordinates are scaled to a discrete $H \times W$ pixel grid representing the output texture space.

Per-face barycentric interpolation. For each triangular face of the mesh, we project its UV coordinates to the texture grid, forming a triangle in 2D space. We then compute a barycentric coordinate transform that expresses any point within the triangle as a convex combination of its three corners.

Compute barycentric coordinates. We iterate over each pixel inside the UV triangle's bounding box and compute its barycentric coordinates. If a pixel lies within the triangle, its mean curvature value is interpolated from the curvature values at the triangle's corresponding 3D vertices using:

$$\begin{aligned} \text{val} = \ & \text{bary}[0] \cdot \text{curv}_v[\text{tri\_v}[0]] + \\ & \text{bary}[1] \cdot \text{curv}_v[\text{tri\_v}[1]] + \\ & \text{bary}[2] \cdot \text{curv}_v[\text{tri\_v}[2]] \end{aligned} \tag{28}$$

where $\text{bary}[i]$ is the i-th barycentric coordinate of the pixel relative to the UV triangle, $\text{tri\_v}[\text{i}]$ is the index of the i-th 3D vertex corresponding to the UV triangle corner, and curv_v is the mean curvature array, containing one scalar curvature per 3D vertex. This produces a smoothly interpolated curvature value for each pixel within the UV triangle.

Accumulation and averaging. Each pixel may be covered by multiple UV triangles. Therefore, we accumulate curvature values and record the number of contributions per pixel. After processing all triangles, we normalize the curvature value at each pixel by the number of contributions, producing a final per-pixel average curvature. The output is then a dense 2D curvature texture defined over the UV domain. This texture captures the surface's geometric structure and is suitable for use in differentiable training objectives or geometric regularization.

F.2. Geometry-Aware Texture Colorization Experiment and Results

This task aims to colorize textures based on surface bending intensity, represented by mean curvature, an average of the minimal and maximal curvature directions, on the 3D mesh. Specifically, the model is encouraged to apply warm colors (e.g., red, yellow) in high-curvature regions and cool colors (e.g., blue, green) in low-curvature areas. Fig. 13 shows qualitative results on rabbit and cow mesh objects colorized with different textual prompts. As illustrated, our method consistently adapts texture colors, regardless of initial patterns, according to local curvature and successfully maps warmth and coolness to geometric variation (more results in Fig. 13).

G. Attempts to Overcome Reward Hacking

To prevent the fine-tuned diffusion model from diverging too far from the original pre-trained model and to mitigate issues such as overfitting and reward hacking, we incorporate three regularization strategies: KL divergence [12], LoRA scaling [35], and a colorfulness incentive [13]. These techniques are added as regularization terms to each reward function we design and are evaluated for their effectiveness during training. While KL regularization offers a principled approach to reducing reward overfitting, we find that LoRA scaling and the colorfulness incentive [13] consistently yield better empirical results. We detail each of these techniques in the following.

KL-Divergence. Inspired by [8] and similar to [12], we introduce a regularization penalty that encourages the fine-tuned model to stay close to the behavior of the pre-trained Stable Diffusion model. Specifically, we penalize the squared difference between the noise predictions of the pre-trained and fine-tuned U-Net models at the final denoising step. Let ϵ_{pt} and ϵ_{ft} denote the predicted noise from the pre-trained and fine-tuned models, respectively. We define the penalty term as:

$$\beta_{\text{KL}} \left\| \epsilon_{\text{ft}} - \epsilon_{\text{pt}} \right\|_2^2 \tag{29}$$

and add it to the overall reward during training. The coefficient β_{KL} controls the strength of this regularization and is varied across different experimental settings. Intuitively, this penalty discourages the fine-tuned model from diverging too far from its pre-trained counterpart, helping to mitigate issues such as catastrophic forgetting or overly narrow specialization. This approach is justified by the fact that, under fixed variance, the KL divergence between two Gaussian noise distributions simplifies to the squared difference between their means. In the context of conditioned diffusion models, noise prediction ϵ is conditioned on a noised latent x_t. We compute this penalty at the final denoising timestep $t = 1$, just before image reconstruction. For each training batch, we compute ϵ_{pt} by forwarding the frozen pre-trained model (with gradients disabled), and ϵ_{ft} by forwarding the fine-tuned model. The squared difference, scaled by β_{KL}, is then added to the reward signal.

LoRA scaling. Inspired by [8] and similar to [35], we scale the LoRA parameters by a factor in the range $(0, 1]$ to modulate their effect during fine-tuning. Empirically, we observe that factors close to 1, specifically within $[0.9, 1]$, consistently yield the best performance in our experiments.

Colorfulness incentive. Inspired by [11, 13], we discourage desaturated texture images by encouraging vibrant color use, measured through the standard deviation and

mean distance of opponent color channels from neutral gray. To this end, we augment our reward with a colorfulness term based on opponent color channels [13]. Let $R(x,y), G(x,y), B(x,y) \in \mathbb{R}^{H\times W}$ denote the red, green, and blue channels of the texture. We compute the opponent axes:

$$rg = R - G, \quad yb = \frac{1}{2}(R+G) - B \tag{30}$$

the *colorfulness* term is then defined as:

$$R_{color} = \sigma_{rg} + \sigma_{yb} + 0.3\,(|\mu_{rg}| + |\mu_{yb}|) \tag{31}$$

where σ_{rg}, σ_{yb}, μ_{rg}, and μ_{yb} are the standard deviations and mean colors offsets corresponding to color opponent components $rg(x,y) = R(x,y) - G(x,y)$ and $yb(x,y) = \frac{R(x,y)+G(x,y)}{2} - B(x,y)$, respectively. This term is incorporated into both our *texture feature emphasis* and *symmetry-aware texture generation* reward functions to promote visually rich and saturated outputs.

H. Ablation Studies

We include a focused ablation study to quantify the effect of regularization choices, LoRA scaling, and the memory/-computation trade-off of our proposed configuration. Results and qualitative examples are shown in Fig. 8 (regularization ablations) and Fig. 7 (LoRA scaling ablations), and a compact memory/throughput comparison is reported in Tab. 3.

Full vs. LoRA+Checkpointing fine-tuning. Tab. 3 summarizes the memory and throughput trade-offs. Using LoRA in combination with activation checkpointing reduces the peak GPU memory substantially (from an infeasible full-fine-tuning footprint to a modest working set) and reduces the number of trainable parameters by several orders of magnitude, while maintaining acceptable throughput. The table demonstrates that our default configuration provides a realistic balance between memory cost and training performance for high-resolution texture adaptation. Therefore, when training on limited GPU memory, it is recommended to enable activation checkpointing and LoRA to dramatically reduce peak memory, as shown in Tab. 3.

Fine-tuning with/without regularization strategies. Fig. 8 compares two regularizers that we found important in practice: a colorfulness regularizer (Fig. 8a) and a LoRA-based regularization term that penalizes LoRA parameters changes (Fig. 8b). Each subfigure shows three rows for the same example mesh: the input (before fine-tuning), the output after fine-tuning *without* the regularizer, and the output after fine-tuning *with* the regularizer. The colorfulness regularizer reduces undesirable saturations and preserves a more balanced color distribution across the texture image, which is visible in the bunny texture (Fig. 8a). The LoRA regularization term helps keep the fine-tuned model's distribution close to the pre-trained one while allowing parameter updates to satisfy the reward objective. This yields texture images that both exhibit natural patterns and meet specific objectives (e.g., symmetry patterns in Fig. 8b. Qualitatively, both regularizers improve perceptual texture quality compared to training without them, though the more beneficial choice depends on the reward and desired visual characteristics.

LoRA scaling. Fig. 7 reports reward curves for multiple LoRA scale values under two reward objectives: symmetry-aware (Fig. 7a) and aesthetic (Fig. 7b). The plots show smoothed rewards over training iterations for each LoRA scale value. Across both objectives we observe a trade-off: larger LoRA scales accelerate early gains (they provide more expressiveness and thus faster improvement on the target reward), but they can also increase variance and, in some settings, cause less stable long-term behaviour. Smaller scales train more conservatively and tend to be more stable, but require more iterations to reach the same reward level. The two rewards behave slightly differently: the symmetry reward benefits noticeably from higher early expressiveness, while the aesthetic reward shows a more gradual, steady improvement for moderate scales. Based on these results, we recommend a moderate-to-high default (e.g., 0.7–0.9) and advise practitioners to tune upward only when faster short-term gains are needed and the risk of added variance is acceptable.

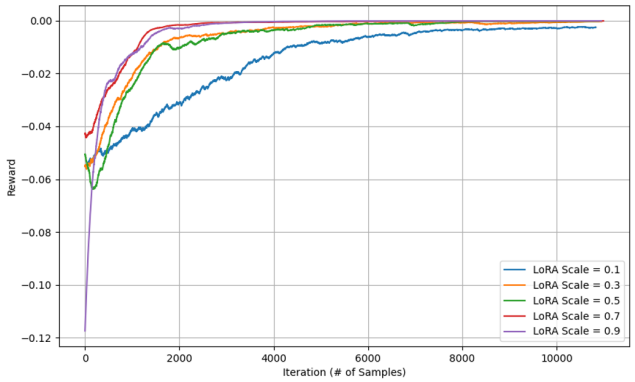
Table 3. Ablation study comparing LoRA+checkpointing fine-tuning with full fine-tuning. Results show a substantial reduction in memory usage and trainable parameters when using LoRA with activation checkpointing. (OOM = Out of Memory)

Configuration	Peak GPU mem (GB)	#Params (M)	Throughput (sec/iter)
Full fine-tuning	300 (OOM)	859.5	OOM
Ours (LoRA + Checkpointing)	20	0.797	40

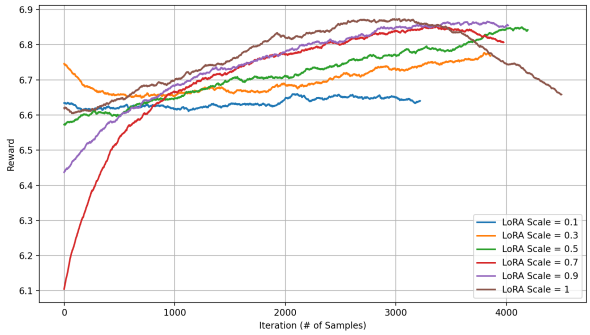
I. Additional Experiments and Results

We show expanded results from the main paper featuring more texture image and more viewpoints of different 3D objects. Fig. 11, Fig. 12, Fig. 13, and Fig. 14 shows the qualitative results of the symmetry-aware, geometry-texture alignment, geometry-guided texture colorization, and texture features emphasis experiments, respectively. Moreover, a quantitative comparison on the *geometry-texture alignment* is presented in Tab. 4, using various texture patterns on the bunny object. It evaluates our fine-tuned method against InTeX [33] (before fine-tuning) for geometry–texture alignment. Our approach yields significantly higher alignment counts, demonstrating that the generated textures respect the underlying geometric structure, whereas the InTeX [33] fails to achieve such alignment.

We also added extended results on complex objects in Fig. 15. Since our method modifies only texture (not geometry), self-occlusion and mesh resolution do not directly



(a) Ablation on LoRA scale measured with the symmetry-aware reward. Curves show smoothed reward versus training iterations for multiple LoRA scale values. Larger LoRA scales produce faster initial gains on the symmetry objective but can introduce higher variance; moderate-to-high scales achieve a good balance between speed and stability.



(b) Ablation on LoRA scale measured with the aesthetic reward. Curves show the smoothed aesthetic reward over iterations for the same set of LoRA scales. The relative ranking of scales differs slightly from the symmetry objective, indicating that reward choice interacts with LoRA expressiveness.

Figure 7. Ablation study on different values of LoRA scale. (Top) symmetry-aware reward. (Bottom) aesthetic reward. In both plots each series corresponds to a different LoRA scale (see legends); curves were smoothed for visual clarity.

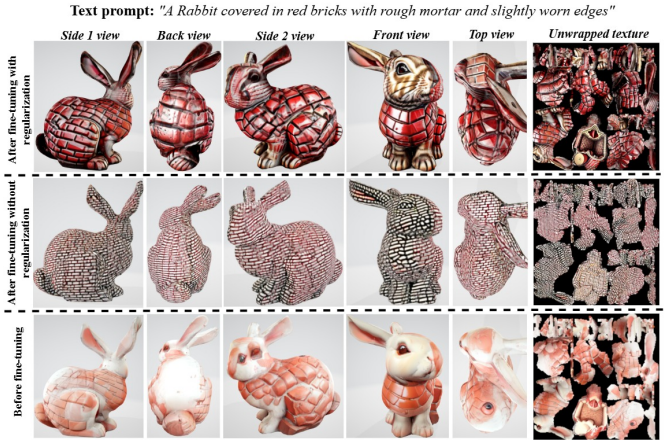
affect the outputs. Nevertheless, we include additional experiments with higher-poly, occluded, and complex meshes in Fig. 15.

J. Analysis of Failure Cases

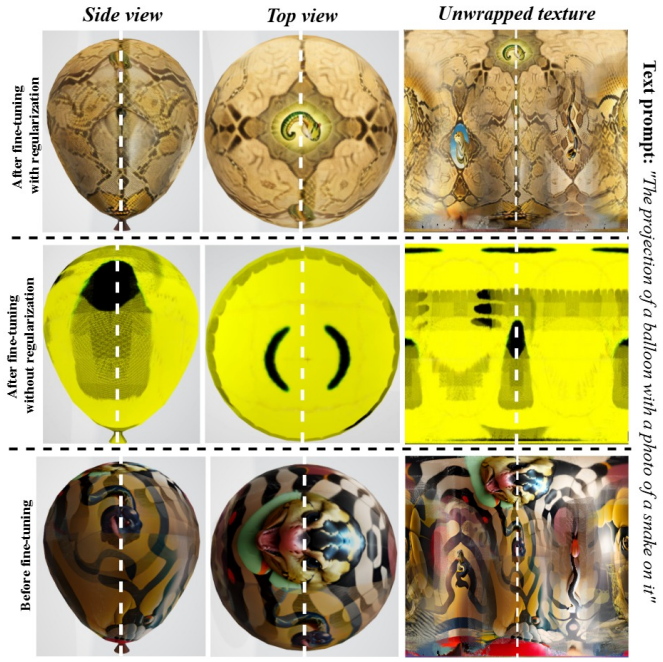
We observe two recurring failure modes under challenging inputs (see Fig. 17). First, the *symmetry-aware* reward can introduce artifacts on non-extrinsically-symmetric or geometrically complex meshes. Our symmetry reward fits a dominant symmetry plane via Principal Component Analysis (PCA) on vertex coordinates, which implicitly assumes the vertex distribution is aligned with a clear axis/-plane. When this assumption fails (e.g., complex geometry or uneven vertex distributions), PCA can return an incorrect plane and the symmetry reward then drives inconsistent, non-symmetric texture patterns (e.g., in the top two rows of Fig. 17, the missing eye on one side of the rabbit's face while present on the other, or asymmetric texture patterns on the face of the 3D female bust). Second, the *camera-pose-aware* reward, designed to remove manual viewpoint tuning by learning camera azimuth/elevation, can fail if camera initialization is extremely poor (e.g., cameras placed too far from the object or at extreme off-angles). In these cases, the learned viewpoints may not cover the full surface and parts of the mesh remain untextured (black regions), as shown in Fig. 17 (bottom row).

Table 4. A quantitative comparison on the bunny model, using various texture patterns, evaluates our fine-tuned method against InTeX [33] (before fine-tuning) for geometry–texture alignment. Each column reports the number of aligned vector pairs (i.e., minimum curvature directions versus texture gradients) in UV space. Our approach yields significantly higher alignment counts, demonstrating that the generated textures respect the underlying geometric structure, whereas the InTeX [33] baseline fails to achieve such alignment.

Texture pattern	After fine-tune	Before fine-tune
Tiger stripe	**630/4487 (14.04%)**	379/4487 (8.45%)
Black & white line	**826/4487 (18.41%)**	372/4487 (8.29%)
Checkerboard	**485/4487 (10.81%)**	321/4487 (7.15%)
Brick	**477/4487 (10.63%)**	302/4487 (6.73%)
Feather	**556/4487 (12.39%)**	343/4487 (7.64%)
Weave	**472/4487 (10.52%)**	299/4487 (6.66%)



(a) Ablation on colorfulness regularization. Each row shows (top to bottom): (1) result after fine-tuning with colorfulness regularization, (2) result after fine-tuning without colorfulness regularization, and (3) the input / before fine-tuning baseline. Columns show a set of canonical views (side / back / side2 / front / top) and the unwrapped texture. The colorfulness regularizer reduces excessive saturation and preserves perceptually balanced color distributions in the texture image while keeping geometric detail.



(b) Ablation on LoRA scaling regularization. Rows are arranged as in Fig. 8a: (1) after fine-tuning with LoRA regularization, (2) after fine-tuning without LoRA regularization, and (3) before fine-tuning. Columns show side, top and the unwrapped UV texture. Using a mild-to-high LoRA regularization term helps keep the fine-tuned model's distribution close to the pre-trained one while allowing parameter updates to satisfy the reward objective. This yields texture images that both exhibit natural patterns and meet specific objectives (e.g. symmetry patterns here).

Figure 8. Ablation studies on regularization strategies. (Top) colorfulness regularization; (Bottom) LoRA sacling regularization. Each subfigure compares the same mesh and texture image before fine-tuning, after fine-tuning without the regularizer, and after fine-tuning with the regularizer (see subcaptions for details).

Figure 9. **Left:** Qualitative results of the camera-pose-aware texture generation experiment on three different 3D mesh objects (female head, dragon, and male head). Our method produces textures with more aesthetic patterns that are more realistic and align more closely with the mesh's semantic parts. **Middle:** Qualitative results for the symmetry-aware experiment on a 3D balloon mesh. For each viewpoint, we present the rendered mesh with a vertical dashed line indicating the symmetry axis. Compared to InTeX [33], TEXTure [29], Text2Tex [5], and Paint3D [40], our method produces textures that exhibit consistent patterns across symmetric regions of the mesh, confirming the reward's ability to enforce symmetry. In contrast, without symmetry supervision, the textures on opposite sides often diverge significantly. **Right:** Qualitative results of the texture features emphasis experiment on a rabbit (bunny) object. We show the rendered 3D bunny from multiple viewpoints. Our method enhances texture features, such as edges and mortar, in proportion to mean curvature, a capability that all four baselines lack, often resulting in pattern-less (white) areas, particularly on the back and head of the rabbit.

Figure 10. Qualitative results of the camera-pose-aware texture generation experiment for different text prompts on different 3D mesh objects. The objective of this experiment is to learn optimal camera viewpoints such that, when the object is rendered and textured from these views, the resulting texture maximizes the average aesthetic reward. Consequently, by maximizing this reward, the model becomes invariant to the initial camera positions: regardless of where the cameras start, the training will adjust their azimuth and elevation to surround the 3D object in a way that yields high-quality, aesthetically pleasing textures. We compare our fine-tuning approach against InTeX [33] and TEXTure [29]. All methods share identical initial camera parameters; once initialized, each algorithm proceeds to paint the object. As shown, our method consistently outperforms both baselines, producing high-fidelity, geometry-aware textures across all shapes. This robustness stems from the aesthetic-reward signal guiding camera adjustment during texture reward learning. In contrast, InTeX [33] and TEXTure [29] remain sensitive to poorly chosen initial viewpoints and thus often fail to generate coherent, high-quality textures under suboptimal camera configurations.

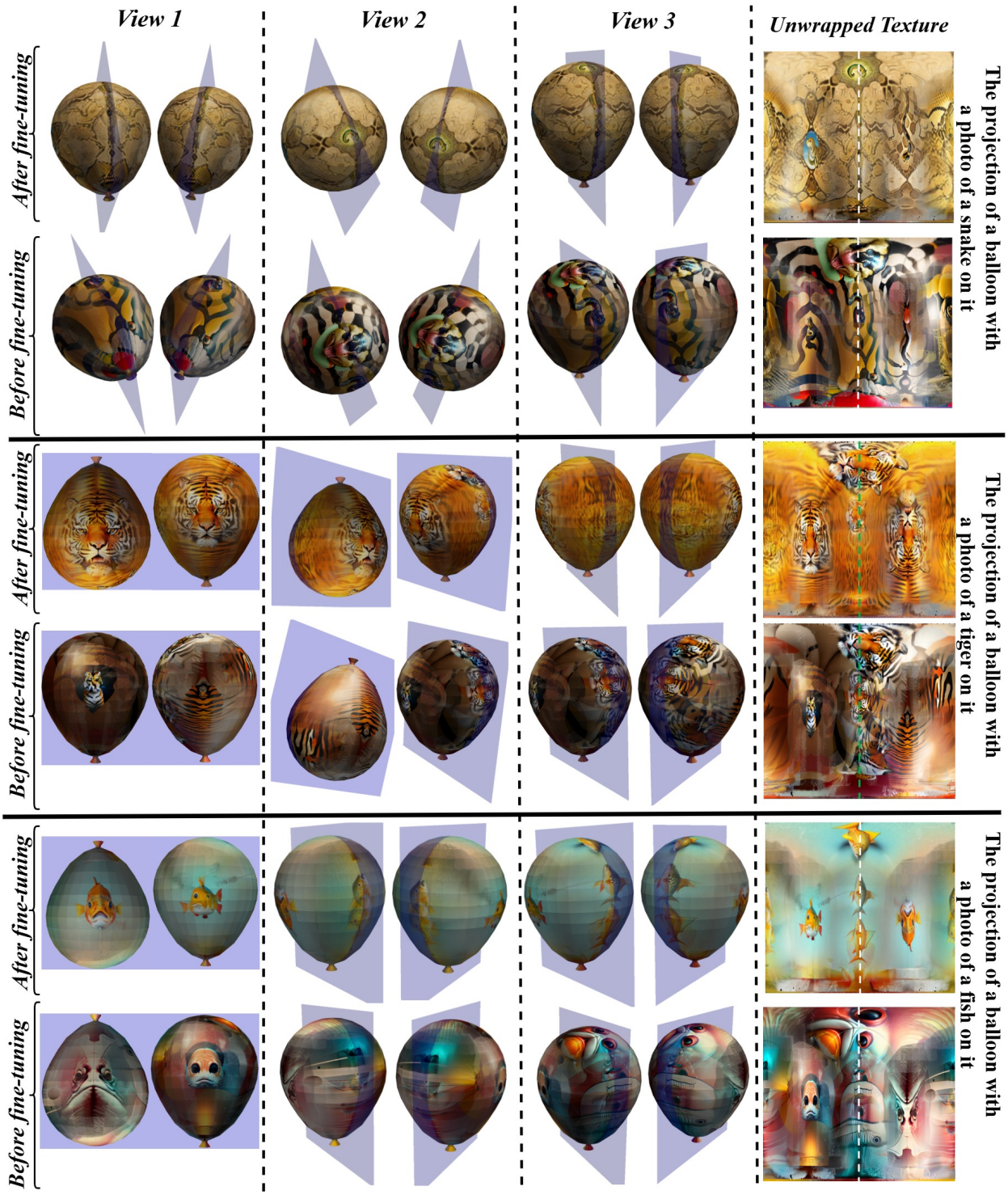


Figure 11. Qualitative results of symmetry-aware experiment for different examples on a balloon mesh object. For each example (each rwo), we show the rendered 3D object from multiple viewpoints, alongside the corresponding texture images (rightmost column), which highlight the symmetric regions. A vertical dashed line marks the symmetry axis in each texture image. The purple plane passing through the center of the balloon in each viewpoint indicates the estimated symmetry plane of the object. As shown, compared to the pre-trained model [33], our method generates textures that are more consistent across symmetric parts of the mesh. Without symmetry supervision, patterns often differ noticeably between sides. In contrast, textures trained with the proposed symmetry reward exhibit visually coherent features across symmetric regions, demonstrating the reward's effectiveness in enforcing symmetry consistency.

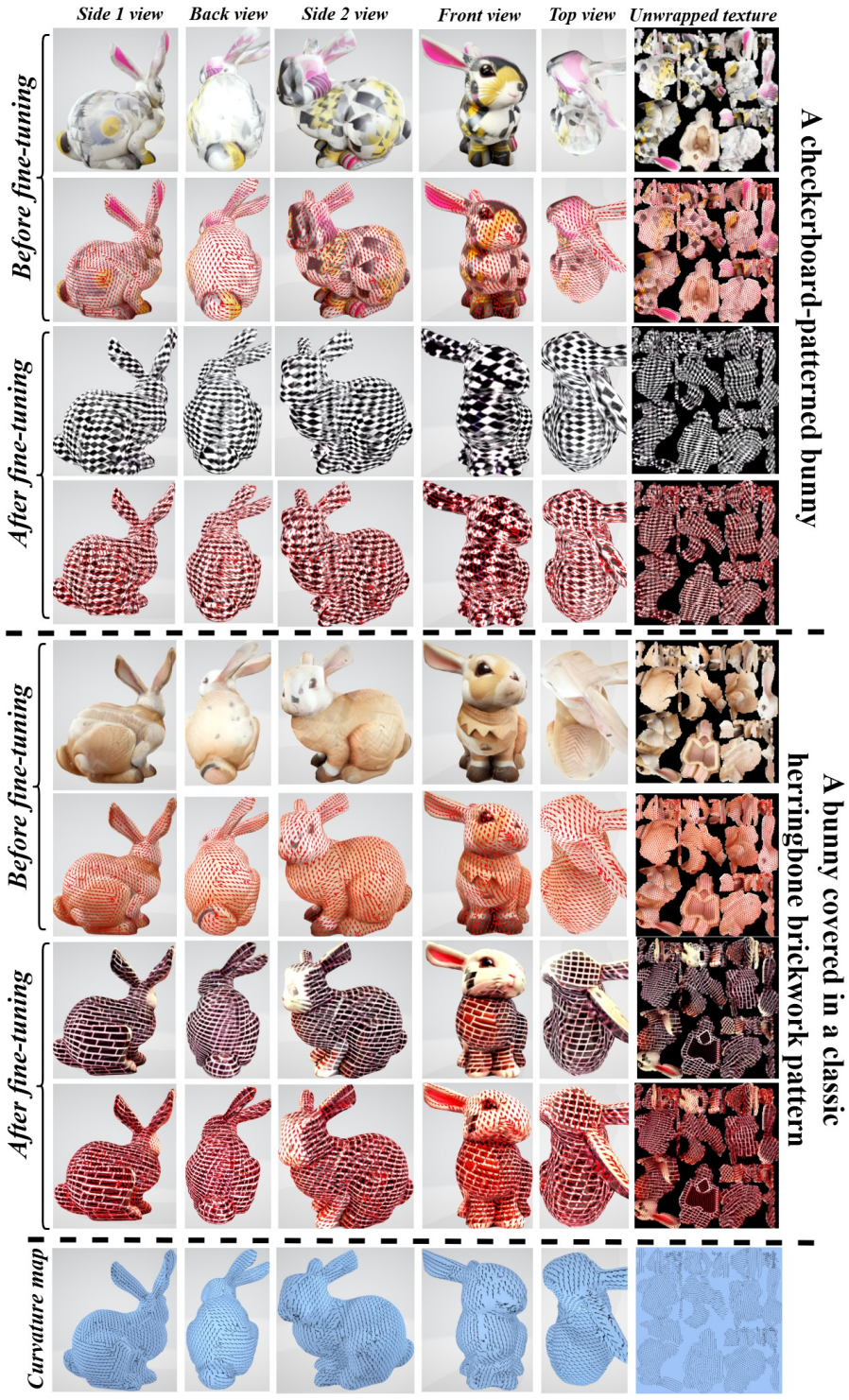


Figure 12. Qualitative results of the geometry-texture alignment experiment on a rabbit (bunny) mesh. For each example, we show the rendered 3D object from multiple viewpoints, with the corresponding texture image in the rightmost column. Minimum curvature vectors, representing the underlying surface geometry, are visualized in the bottom row and overlaid on the textured objects for comparison. As shown, our method produces textures whose patterns align more closely with the mesh's curvature directions, unlike InTex [33]. Moreover, a notable outcome in our results is the emergence of repetitive texture patterns after fine-tuning with the geometry-texture alignment reward. This behavior arises from the differentiable sampling strategy used during reward computation. Specifically, it encourages the model to place edge features at specific UV coordinates which ultimately results in structured and repeated patterns in the texture (see Appendix D).

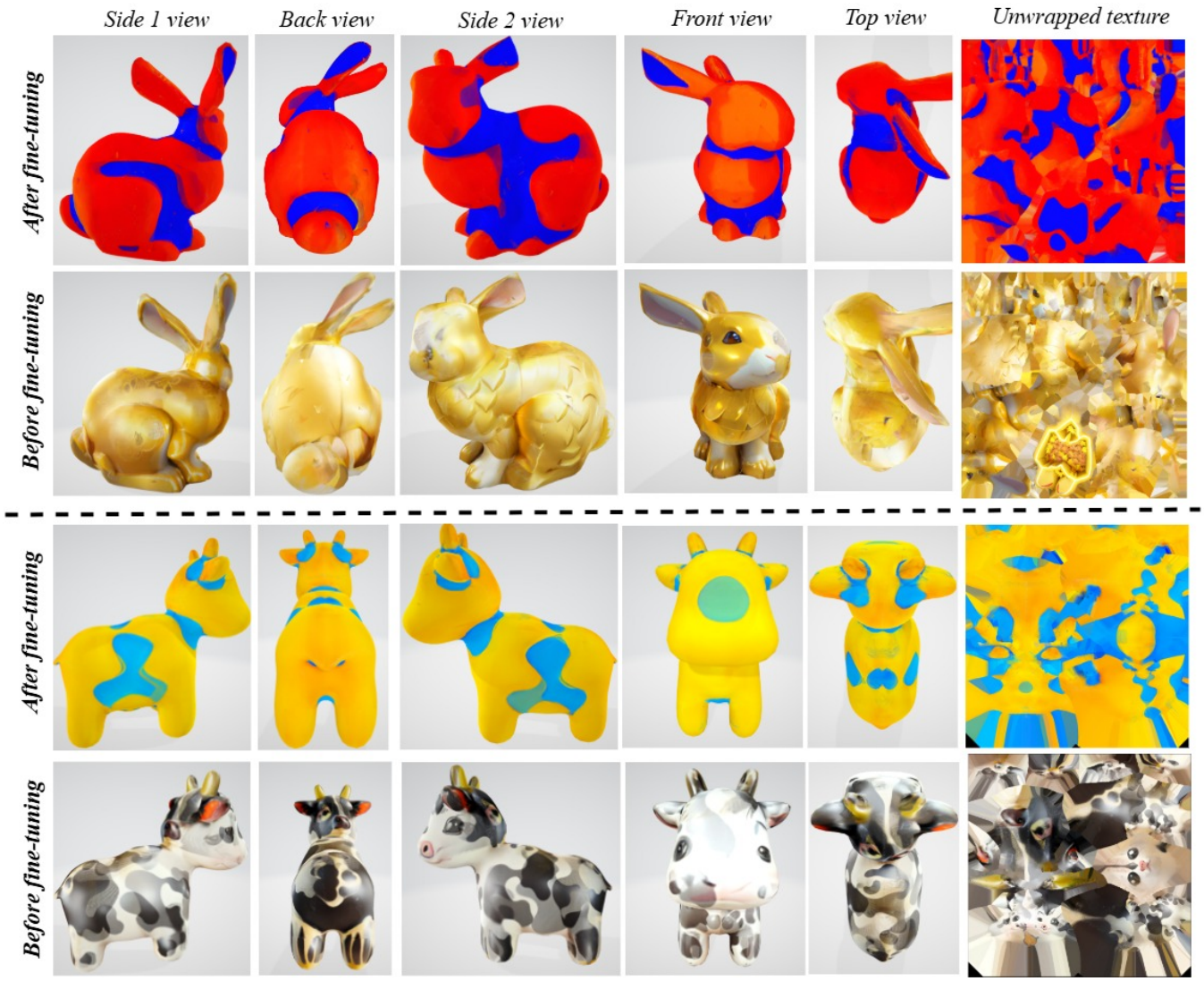


Figure 13. Qualitative results of the geometry-guided texture colorization experiment on rabbit (bunny) a cow mesh objects. For each example, we show the rendered 3D object from multiple viewpoints, with the corresponding texture image in the rightmost column. The goal is to colorize textures based on surface bending intensity, represented by mean curvature, an average of the minimal and maximal curvature directions, on the 3D mesh. Specifically, the model is encouraged to apply warm colors (e.g., red, yellow) in high-curvature regions and cool colors (e.g., blue, green) in low-curvature areas. As illustrated, our method consistently adapts texture colors, regardless of initial patterns, according to local curvature and successfully maps warmth and coolness to geometric variation.

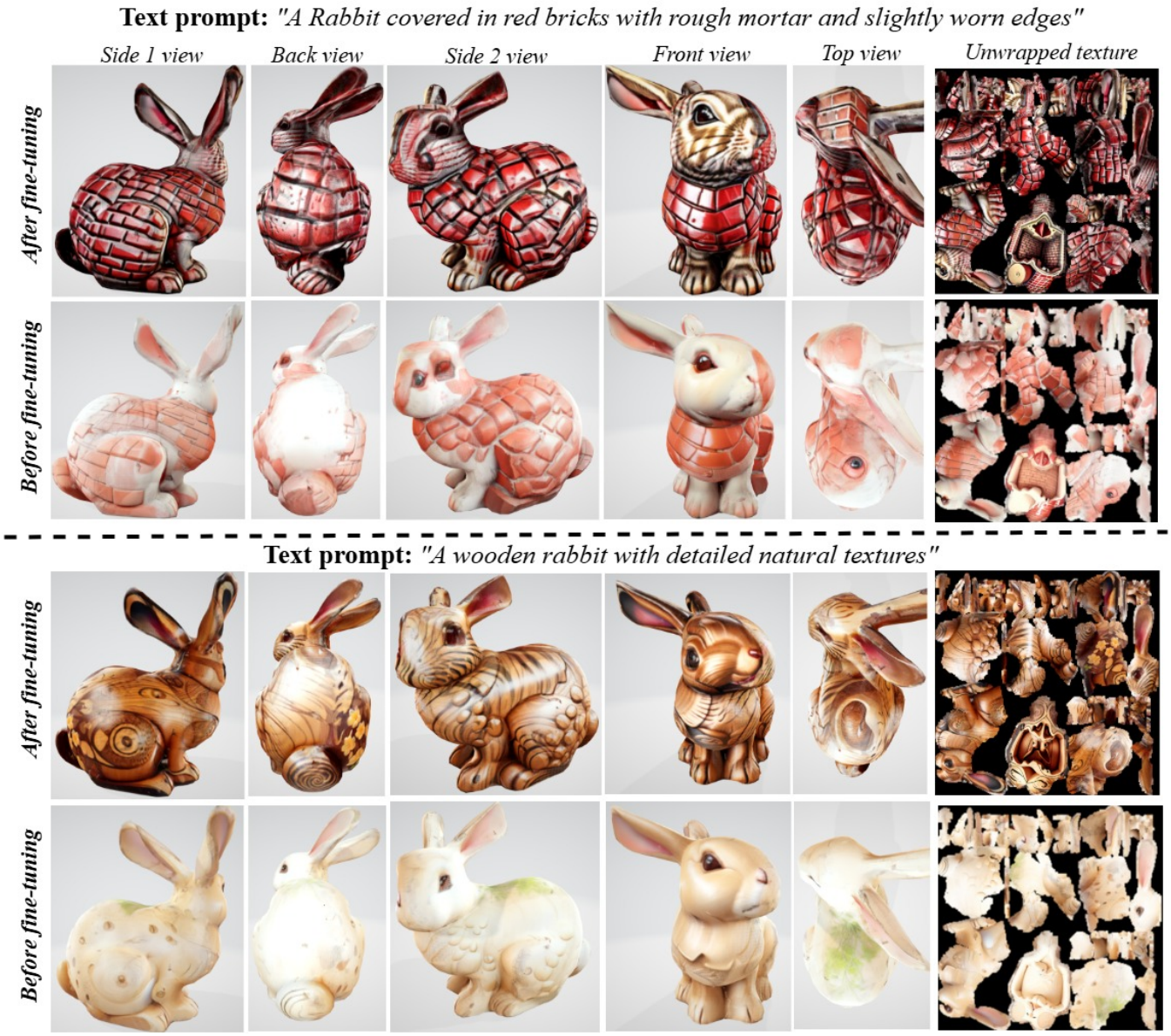


Figure 14. Qualitative results of the texture features emphasis experiment on a rabbit (bunny) objects with different text prompts. For each example, we show the rendered 3D object from multiple viewpoints, with the corresponding texture image in the rightmost column. This goal is to learn texture images with salient features (e.g., edges) emphasized at regions of high surface bending, represented by the magnitude of mean curvature. This encourages texture patterns that highlight 3D surface structure while preserving perceptual richness through color variation. As illustrated, our method enhances texture features, such as edges and mortar, in proportion to local curvature, a capability InTex [33] lacks, often resulting in pattern-less (white) areas, particularly on the back and head of the rabbit.

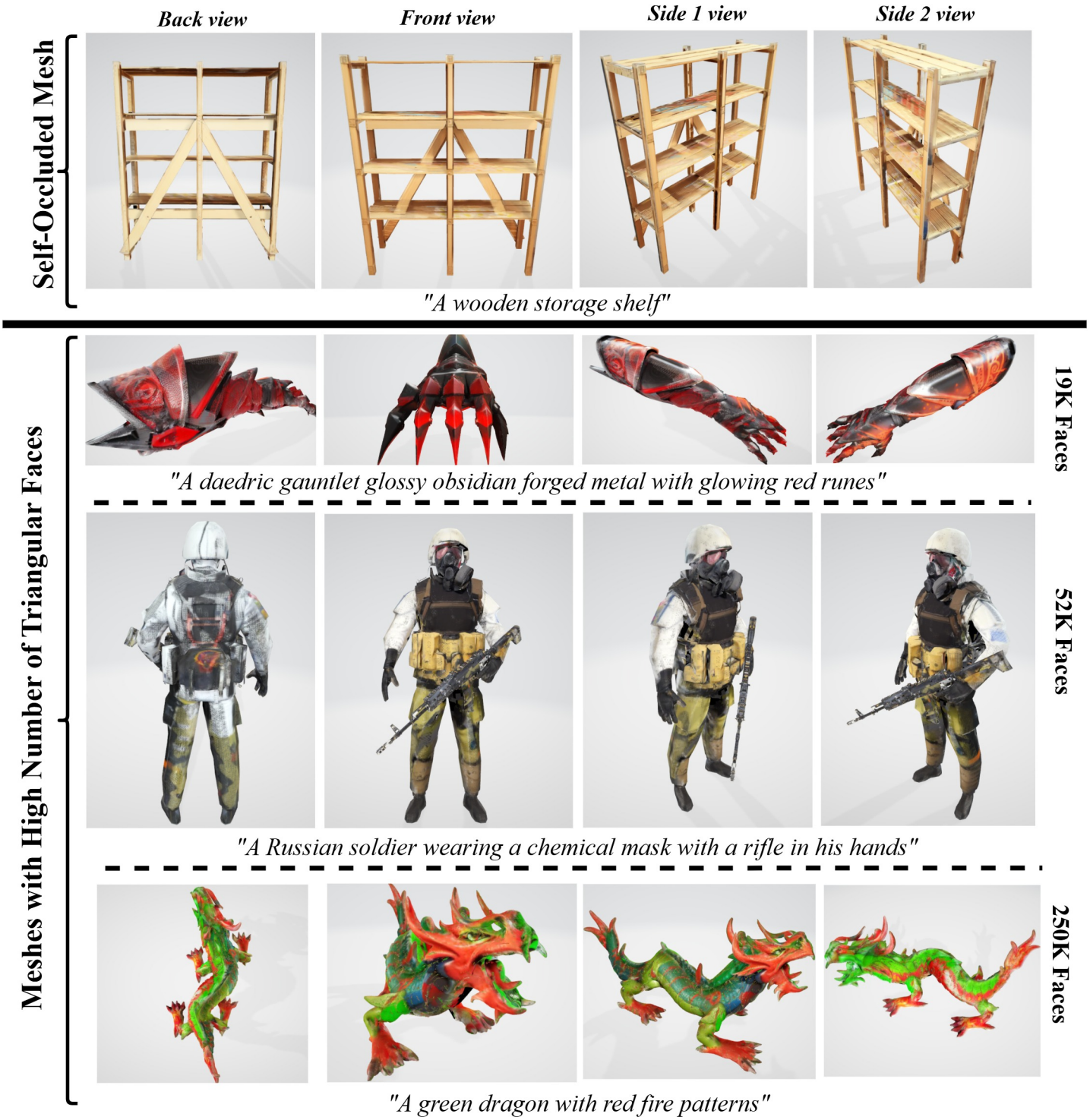


Figure 15. Extended qualitative results for the camera-pose-aware reward. Top row: a self-occluded object; rows 2–4: complex, high-poly meshes. Because our method alters only texture (not geometry), self-occlusion and mesh resolution do not directly change the rendered outputs.

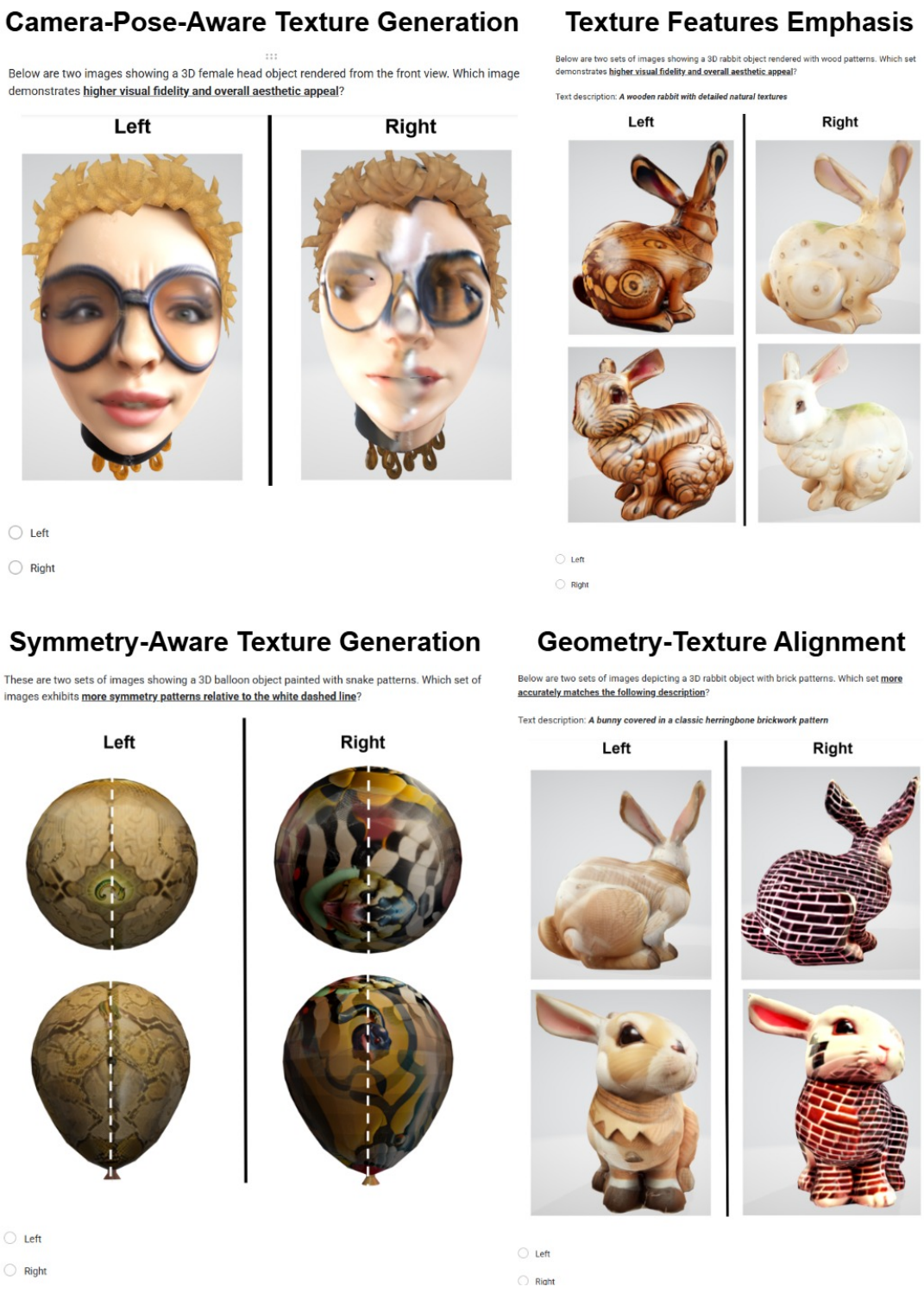


Figure 16. Examples of four of the eleven questions in our user-study questionnaire. We conducted a user-preference study with 40 participants, each of whom completed 11 pairwise comparisons between textured 3D shapes generated by our method and by InTeX [33]. Participants rated visual quality according to four criteria: fidelity, symmetry, overall appearance, and text-and-texture pattern alignment. Specifically, for the three reward variants, cam-pose-aware texture generation, symmetry-aware texture generation, and geometry–texture alignment, we asked three questions apiece, and for the texture-features-emphasis reward we asked two questions. We then applied one-sided binomial tests to each experiment to obtain p-values (see bottom of Tab. 1). As the table shows, the post-reward-learning results are significantly preferred over the pre-learning outputs.

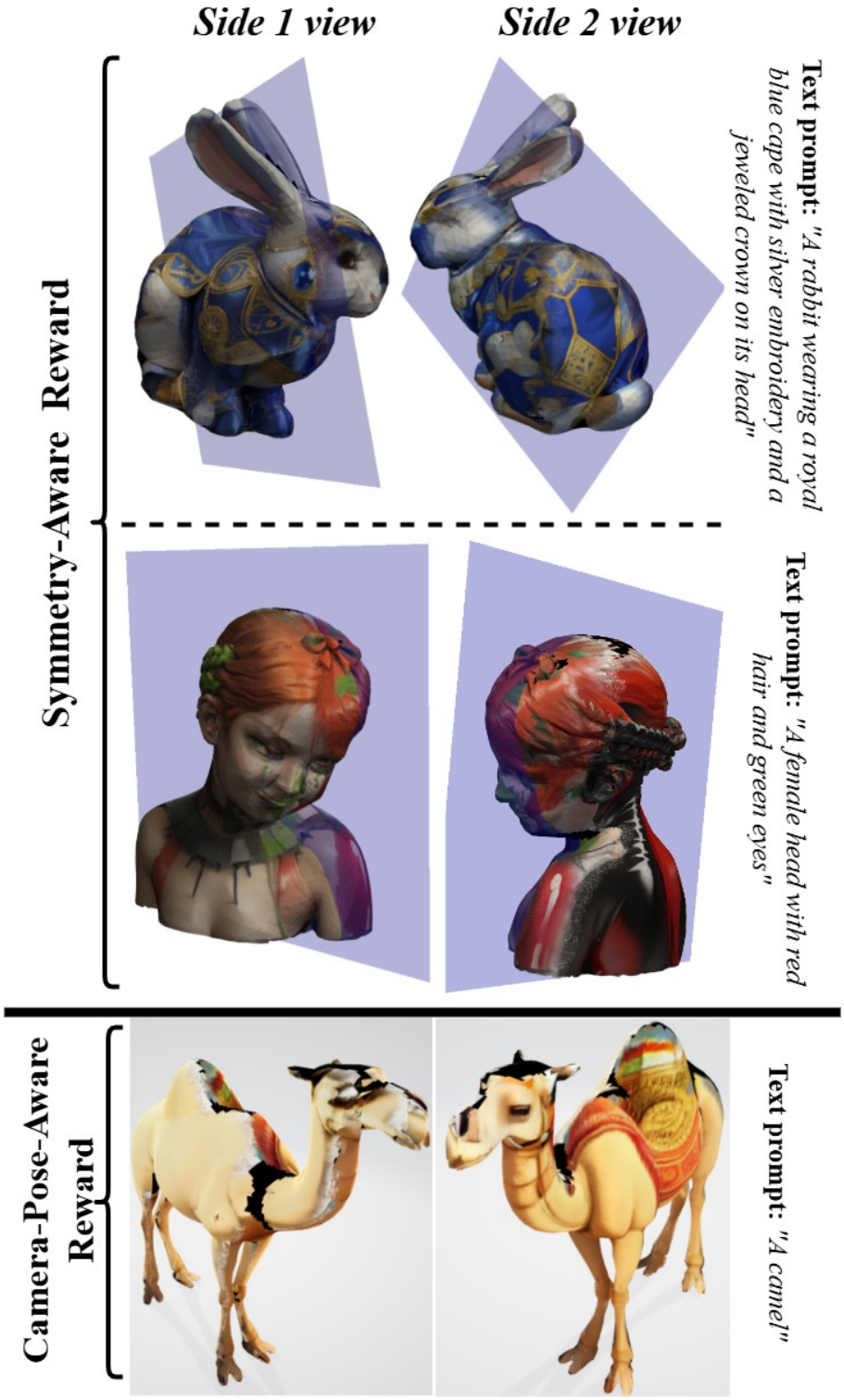


Figure 17. Failure cases. **Top**: artifacts introduced by the symmetry-aware reward on non-extrinsically symmetric (rabbit and bust). Our PCA-based symmetry reward can produce an incorrect symmetry plane when vertex distributions are uneven, causing the symmetry objective to drive inconsistent or non-symmetric textures (e.g., the missing eye on one side of the rabbit's face while present on the other, or asymmetric texture patterns on the face of the 3D female bust). **Bottom**: failure of the camera-pose-aware reward on the Camel. Extremely poor camera initialization (e.g., camera placed too far from the object or at extreme off-angles) can prevent learned viewpoints from covering the full surface, leaving untextured (black) regions.