Table 1: Glossary and notation

(A)TT3D	(Amortized) Text-to-3D
NeRF	Neural Radiance Field [6]
DDM	Denoising Diffusion Model
MLP	Multi-layer Perceptron
DF27, DF411	DreamFusion's [1] 27 main text prompts & the extended 411 prompts
$n, m \in \mathbb{N}$	The size of different objects
$x, y, z, \dots \in \mathbb{R}$	Scalar coordinates
$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \dots \in \mathbb{R}^n$	Vectors
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \dots$	The domain of $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \dots$
$\boldsymbol{x} = [x, y, z] \in \mathcal{X}$	A point
$\boldsymbol{r} = [\sigma, r, g, b] \in \mathcal{R}$	The density and color values
$\boldsymbol{w} \in \mathcal{W}$	The parameters of the point encoder function
$\boldsymbol{\gamma_w} : \mathcal{X} \to \boldsymbol{\Gamma}$	The point encoder function
$\boldsymbol{\nu} : \boldsymbol{\Gamma} \to \mathcal{R}$	The final MLP mapping point encodings to radiance
$\boldsymbol{z}$	The problem context for amortization
$\boldsymbol{c} \in \mathcal{C}$	A text embedding used to condition the DDM and as problem context
$\boldsymbol{m} : \mathcal{C} \to \mathcal{W}$	The mapping network from problem context to modulations
$\boldsymbol{v} \in \mathbb{R}^n$	The intermediary vector-embedding of $\boldsymbol{c}$ in $\boldsymbol{m}$
$\mathcal{N}, \mathcal{U}$, Dir, Bern	Normal, uniform, Dirichlet, and Bernoulli distributions respectively
$\boldsymbol{\epsilon}, \hat{\boldsymbol{\epsilon}}$	Noise added to rendered frames, or as predicted by the DDM
$\kappa \in \mathbb{R}^+$	The concentration parameter of the Dirichlet distribution
$\alpha \in [0, 1]$	An interpolation coefficient, sampled from Dir(κ) in training
$\mathcal{L}$	A loss function
ω	A guidance weight
β_1, β_2	Parameters of the Adam optimizer [26]

A. Glossary

B. Experimental Setup

B.1. Implementation Details

We replicate DreamFusion [1] and Magic3D's [2] setup where possible and list key details here. We recommend reading these papers for additional context.

B.1.1 Point-encoder γ

We followed Instant NGP [7] to parameterize our NeRF, consisting of dense, multi-resolution voxel grids and dictionaries. We only use dense voxel layers unless specified, which trained faster with negligible quality drop. For our multi-resolution voxel grid, we use resolutions of $[9, 14, 22, 36, 58]$, with 4 features per level. When active, we use a further three levels of hash grid parameters. Each level's features are linearly interpolated according to spatial location and concatenated, leading to a final output feature size of 20 with dense voxel grids and 32 with the full INGP.

B.1.2 Final NeRF MLP ν

We select a minimal final MLP to maintain evaluation speed, with a single hidden layer with 32 units and a SiLU activation [72]. The majority of our model's capacity comes from the point-encoder. We use a softplus activation for the density output and sigmoid activations on the color.

B.1.3 Mapping Network m

The mapping network computes a fixed-size vector representation $\boldsymbol{v}$ of the task from the text embedding. We only use the CLIP embedding for feature grid modulation because it was sufficient and including the T5 embedding increases network size. We apply spectral normalization to all linear layers. We considered concatenation, hypernetwork, and attention approaches for modulation [17]:

Concatenation: The simple strategy of naïvely concatenating (a vector-representation f of) the text to the point-encoding – i.e., $\boldsymbol{\nu}(\boldsymbol{\gamma_w}(\boldsymbol{x}), f(\boldsymbol{c}))$ was prohibitively expensive. This is because we require the cost of the final per-point NeRF MLP $\boldsymbol{\nu}$ to be minimal for cheap rendering. The concatenation approach of $\boldsymbol{\nu}(\boldsymbol{\gamma_w}(\boldsymbol{x}), \boldsymbol{v})$ introduces overhead by increasing per-iteration training time by 37% but doesn't significantly impact quality. In inference, the hypernet method is superior, reducing cost by $\sim 20 - 75\%$, as we only generate the grid parameters $\boldsymbol{w}$ once when rendering multiple views of 1 object, bypassing the use of a single, larger NeRF $\boldsymbol{\nu}$ with concatenation.

Hypernetwork: We first flatten the token and pass it through an MLP to produce a vector-embedding, which is used by a linear layer to output the point-encoder's voxel grid parameters. As the CLIP embedding was already a strong representation, we found that a simple linear layer for the text-embedding to vector-embedding was sufficient.

We converged with deeper hypernetworks – by using spectral normalization on all linear layers – but this offered no quality benefit while taking longer to train. We also found removing the bias on the final linear layer decreased noise by forcing the result to depend on the prompt.

We vary the vector embedding $\boldsymbol{v}$'s size in our experiments, which largely dictates our amortized model's capacity. The mapping network $\boldsymbol{m}$ dominates the model's memory cost, while the text's vector-embedding largely dictates the mapping network size. Our memory cost scales linearly with the vector-embedding size. We use a vector embedding $\boldsymbol{v}$ size of 32 for all experiments except interpolation, where we use 2. We have experiments where the number of text prompts is both smaller (DF27) and larger (DF411, compositional prompts) than the vector-embedding.

Attention: We also investigated using an attention-based mapping network with a series of self-attention layers to process text embeddings before feeding into the hypernetworks for each multi-resolution grid level. Our attention performed with comparable quality but trained more slowly. However, we expect modifications to be necessary on more complex prompt sets.

B.1.4 Environment Mapping Network

In our experiments, we use a background, a function mapping ray directions – and text embeddings – to colors, which we denote as the environment map. Specifically, we encode the ray directions, concatenate them with the vector-embedding $\boldsymbol{v}$ from the mapping network, and feed them into a final MLP. We use a sigmoid activation on the output color and spectral norm on all linear layers. We encode the ray directions with a sinusoidal positional encoding [73] (frequencies $2^0, 2^1, \ldots, 2^{L-1}, L = 8$), and no hidden layers — i.e., a linear layer — for our final MLP.

B.1.5 Spectral Normalization

We found spectral normalization – which can be implemented trivially in PyTorch on linear layers – to be critical for mapping net training, but non-essential on other parts. In the mapping network, we must use spectral normalization on all linear layers for the hypernetwork and attention approaches or we suffer from numerical instability. Using spectral normalization on the linear layers in the environment map, or final NeRF module was unnecessary.

B.1.6 Sampling Text Prompts

We cache the CLIP (and T5) embeddings for all experiments to avoid repeated computation and the memory overhead of the large text encoders. We use multiple text prompts in each batched update.

Interpolations: We sample interpolated embeddings during training in interpolation experiments (Section 4.4). See Section B.1.14 or Figure 18 for more interpolation setup details. When interpolating between prompts with text-embeddings $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, we sample a weight $\alpha \in [0, 1]$ and input $\boldsymbol{c}' = (1 - \alpha)\boldsymbol{c}_1 + \alpha\boldsymbol{c}_2$ to the mapping network.

B.1.7 Sampling Rendering Conditions

As in DreamFusion [1], we randomly sample rendering conditions, including the camera position and lighting conditions. We use a bounding sphere of radius 2 in all experiments. We sample the point light location with distance from $\mathcal{U}(1, 3)$ and angle relative to the random camera position of $\mathcal{U}(0, \pi/4)$. We sample "soft" textureless and albedo-only augmentations to allow varying shades during training. Also, we sample the camera distance from $\mathcal{U}(2, 3)$ and the focal length from $\mathcal{U}(0.7, 1.35)$.

B.1.8 Score Distillation Sampling

For the DDM's sampling, we sample the time-step from $\mathcal{U}(0.002, 1.0)$ and use a guidance weight of 100.

B.1.9 The Objective

The regularizers: The orientation loss [74] (as in DreamFusion [1]) encourages normal vectors of the density field to face the camera when visible, preventing the model from changing colors to be darker during textureless renders by making geometry face "backward" when shaded. Also, DreamFusion regularizes accumulated alpha value along each ray, encouraging not unnecessarily filling space and aiding in foreground/background separation. We do not use these regularizers for all experiments, as we did not observe failure modes they fixed, and they made no significant change in results over the interval $[10^{-3}, 10^{-1}]$. Larger opacity regularization values resulted in empty scenes, while larger orientation values did not change the initialization from a sphere.

The image fidelity: We train with 32 points sampled uniformly along each ray for all experiments except interpolations. For interpolations only, we sample 128 points and reduced batch size to improve quality. Our underlying text-to-image model generates 64×64 images, leading to 4096 rays per rendered frame. At inference time we render with higher points per ray to improve quality for negligible cost.

The initialization: As in DreamFusion, we add an initial spatial density bias to prevent collapsing to an empty scene, shown in Figure 10, left. Our density bias on the NeRF MLP output before the softplus activation takes the form:

$$\text{densityBias}(\boldsymbol{x}) = 10\,(1 - 2\|x\|_2) \tag{7}$$

B.1.10 The Optimization

We use Adam with a learning rate of 1×10^{-1} and $\beta_2 = 0.999$. A wide range momentum β_1 (up to .95) can yield similar qualities if the step size is jointly tuned, while the quickest convergence occurs at 0. We do not use the linear learning rate warmup or cosine decay from DreamFusion.

B.1.11 Memorization Experiments

Our experiments use the same architecture for per-prompt and amortized training settings to ensure a fair comparison. We train models using a batch size of 32 times the number of GPUs used. Amortized training uses 8 GPUs while per-prompt uses a single GPU (due to resource constraints), with more details in Section B.2. The complete set of DreamFusion prompts is located here: `https://dreamfusion3d.github.io/gallery.html`

B.1.12 Generalization Experiments

Our prompt selections are motivated by the compositional experiment in DreamFusion's Figure 4 [1]. Our experiments with pig prompts with the template "*a pig* {*activity*} {*theme*}, where the `activities` and `themes` are any combination of the following:

The `activities`: ["*riding a bicycle*", "*sitting on a chair*", "*playing the guitar*", "*holding a shovel*", "*holding a blue balloon*", "*holding a book*", "*wielding a katana*", "*riding a bike*"]

The `themes`: ["*made out of gold*", "*carved out of wood*", "*wearing a leather jacket*", "*wearing a tophat*", "*wearing a party hat*", "*wearing a sombrero*", "*wearing medieval armor*"]

Our pig holdout, unseen, testing prompts are pairing the i^{th} `activity` and `theme`.

Our experiments with animal prompts with the template "{*animal*} {*activity*} {*theme*} {*hat*}, where the `activities`, `themes` and `hats` are any combination of the following:

The `animals`: ["*a squirrel*", "*a raccoon*", "*a pig*", "*a monkey*", "*a robot*", "*a lion*", "*a rabbit*", "*a tiger*", "*an orangutan*", "*a bear*"]

The `activities`: ["*riding a motorcycle*", "*sitting on a chair*", "*playing the guitar*", "*holding a shovel*", "*holding a blue balloon*", "*holding a book*", "*wielding a katana*"]

The `themes`: ["*wearing a leather jacket*", "*wearing a sweater*", "*wearing a cape*", "*wearing medieval armor*", "*wearing a backpack*", "*wearing a suit*"]

The `hats`: ["*wearing a party hat*", "*wearing a sombrero*", "*wearing a helmet*", "*wearing a tophat*", "*wearing a backpack*", "*wearing a baseball cap*"]

Our holdout, unseen, animal testing prompts are selected homogeneously for each training set size.

B.1.13 Finetuning Experiments

We resume training from an amortized training checkpoint while re-initializing the optimizer state. For the finetuning experiments, in our mapping network, we only finetune an offset to the output and detach all prior weights that only embed text tokens (because we finetune with one prompt). Other training details are kept equal to per-prompt training.

B.1.14 Interpolation Experiments

In interpolations we use 128 ray samples and batch size 16.

Interpolant concentration: We sample the interpolation coefficient $\alpha \sim \text{Dir}(\kappa)$ from a Dirichlet distribution with concentration parameter κ. The Dirichlet distribution allows us to smoothly interpolate from sampling the original text tokens (with concentration $\kappa \approx 0$, to uniformly sampling α (with concentration $\kappa \approx 1$) to focusing on difficult midpoints (with concentration $\kappa > 1$) – see Figure 19. Specifically, in Figures 3 & 20 we use $\kappa = 2.0$ for 5000 steps to stabilize the midpoint, followed by $\kappa = 0.5$ to focus on the original prompts.

Interpolation types: We provide multiple examples of interpolation types to amortize over that provide qualitatively different results – see Figure 18.

A simple strategy is to interpolate over the text embedding used to condition the text-to-image model:

$$\boldsymbol{c}' = (1-\alpha)\boldsymbol{c}_1 + \alpha \boldsymbol{c}_2 \tag{8}$$

Another strategy is to interpolate the loss function used between the two prompts. We could evaluate the loss at both prompts and weight the loss:

$$\mathcal{L}_{\text{final}} = (1-\alpha)\mathcal{L}_{\text{prompt 1}} + \alpha \mathcal{L}_{\text{prompt 2}} \tag{9}$$

Instead, to interpolate in the loss, we sample the loss for each prompt with probability α, which we equate to training with embedding:

$$\boldsymbol{c}' = (1-Z)\boldsymbol{c}_1 + Z\boldsymbol{c}_2 \text{ where } Z \sim \text{Bern}(\alpha) \tag{10}$$

A third strategy, suggested for images in Magic3D [2], interpolates the DDM's guidance weight. Unlike Magic3D, we amortize over guidance weights, reducing cost while providing continuous interpolation (not allowed via retraining on each weight). Specifically, we guide with:

$$\hat{\epsilon} = \epsilon_{\text{uncond.}} + (1-\alpha)\omega_1 \epsilon_{\text{prompt 1}} + \alpha \omega_2 \epsilon_{\text{prompt 2}} \tag{11}$$

Here, the ω_1 and ω_2 are notations for the guidance weights for the predicted noise on the 1^{st} and 2^{nd} prompts respectively, which are fixed and equal in all experiments. This interpolates between using guidance on the first prompt, to guidance on the second prompt.

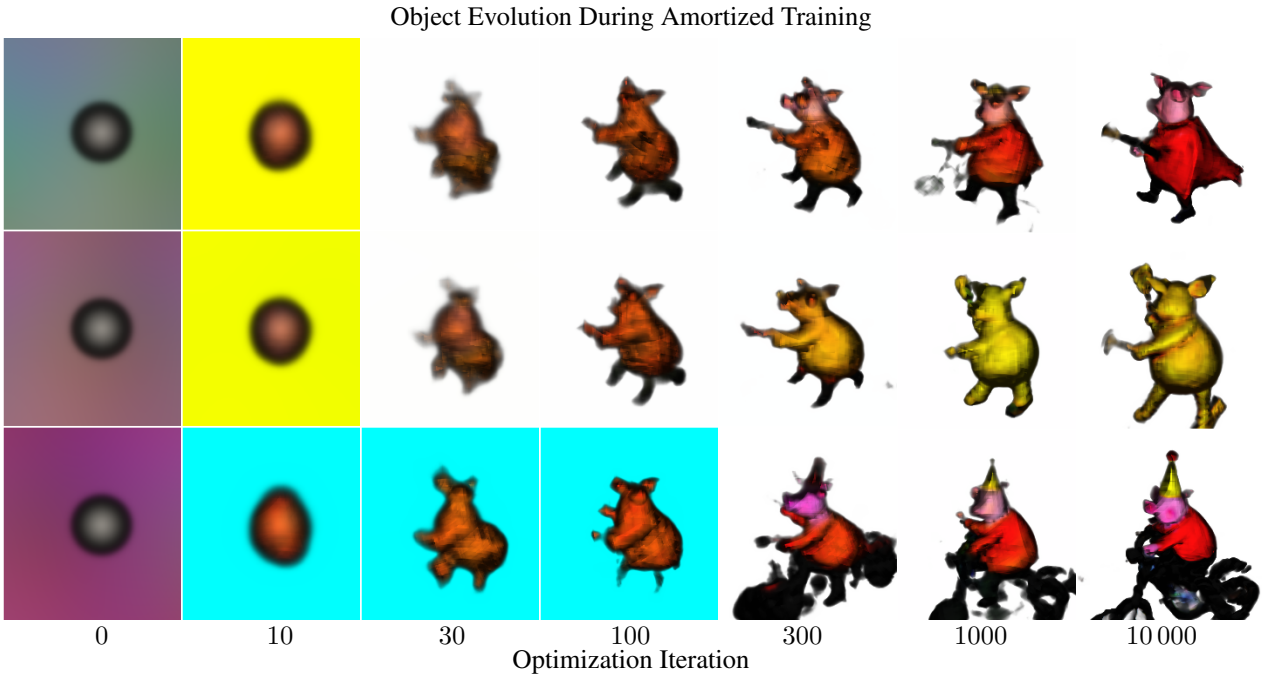


Figure 10: We show assorted training trajectories of the rendered objects during compositional training from Figures 6 and 2. *Left:* We visualize the initialization strategy described in Section B.1.9.

B.2. Compute Requirements

We implement our experiments in PyTorch [70].

B.2.1 Per-prompt Optimization

We do all per-prompt training runs on an NVIDIA A40 GPU, with a batch size of 32 for up to 8000 steps or ~ 4 hours. DF27 (Figure 6, left) use 27 runs, while the compositional prompts (Figure 8) use 50 or 300 subsampled runs respectively. Each training step costs $\sim$1 second.

B.2.2 Amortized Training

Memorization & generalization: When amortizing many prompts, we use multiple GPUs to train with a larger batch size, causing amortized and per-prompt training to have different update costs. So we report the total rendered frames to compare compute accurately. Updates are roughly 1 second in each setup.

We perform the DF27 (Figures 6, 11) and DF411 (Figures 9, 14) runs on 8 NVIDIA A40 GPUs, each with a batch size of 32. We train DF27 for 13 000 steps ($\sim$ 4 hours) and DF411 for 100 000 steps (about a day).

The compositional runs (Figures 2, 6, 8) were performed on 4 NVIDIA A100 GPUs, with a batch size of 32 per GPU, for 40 000 steps or about 10 hours.

Interpolations (Figure 3): We use a single NVIDIA A40 GPU as in per-prompt training.
Finetuning (Figure 17): We use a single NVIDIA A40 GPU as in per-prompt training.

B.2.3 Inference

At inference – delineated from training in Figure 4 – we generate grid parameters in < 1 second and render frames in real-time due to our small final NeRF ν and efficient point-encoding γ. We use more ray samples at inference than training due to negligible cost and enhanced fidelity. Modulation generation occurs once and is reused for each view & location query, creating negligible overhead with many views or high-resolution images. During training with 1 view and image size 64 (batch size 8), hypernet modulations introduced an overhead of 24% more time per iteration, which could be avoided if our weights do not need to be generated. With 1 view and image size 256 (batch size 1) in inference, the modulation introduced an 11% overhead in rendering time, dropping to $< 1\%$ with 30 views.

C. Results

C.1. Additional Experiments & Visualizations

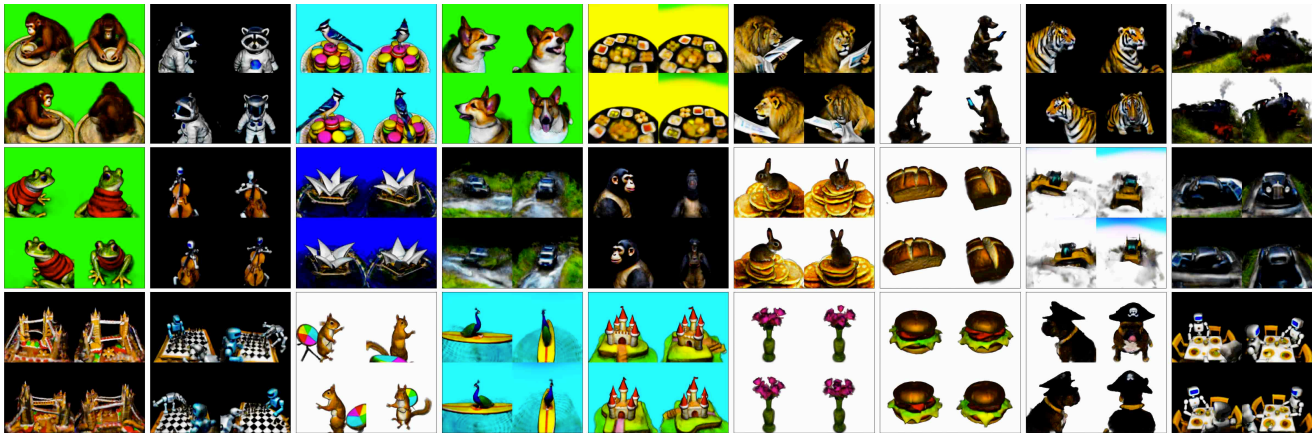


Figure 11: Our method, ATT3D, uses a single model to produce 3D scenes with varying geometric and texture details from the set of 27 prompts in the main DreamFusion paper [1]. The quality is comparable to existing single prompt training and requires far fewer training resources (Fig. 6).

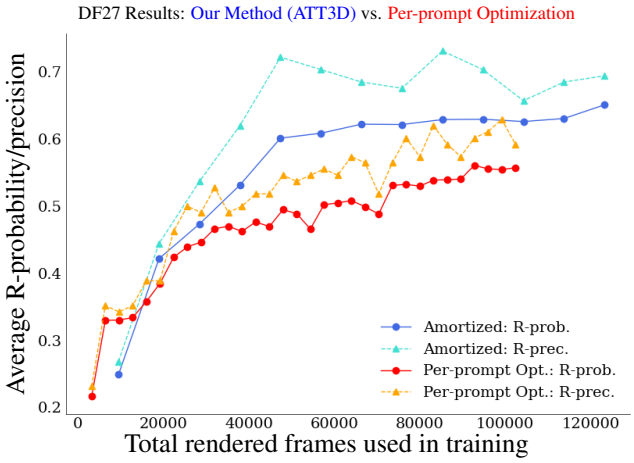


Figure 12: We show the same plot as Figure 6 with the addition of R-precision. **Takeaway:** Results with R-precision are similar to — but noisier than – R-probability when we have few prompts.

Figure 13: We qualitatively compare the unseen "testing" results from the various training strategies in Figures 6 and 2, with our method in blue and baselines in red. Notably, amortized requires no test time optimization, while fine-tuning uses a small amount, and per-prompt uses a large amount to tune from scratch.

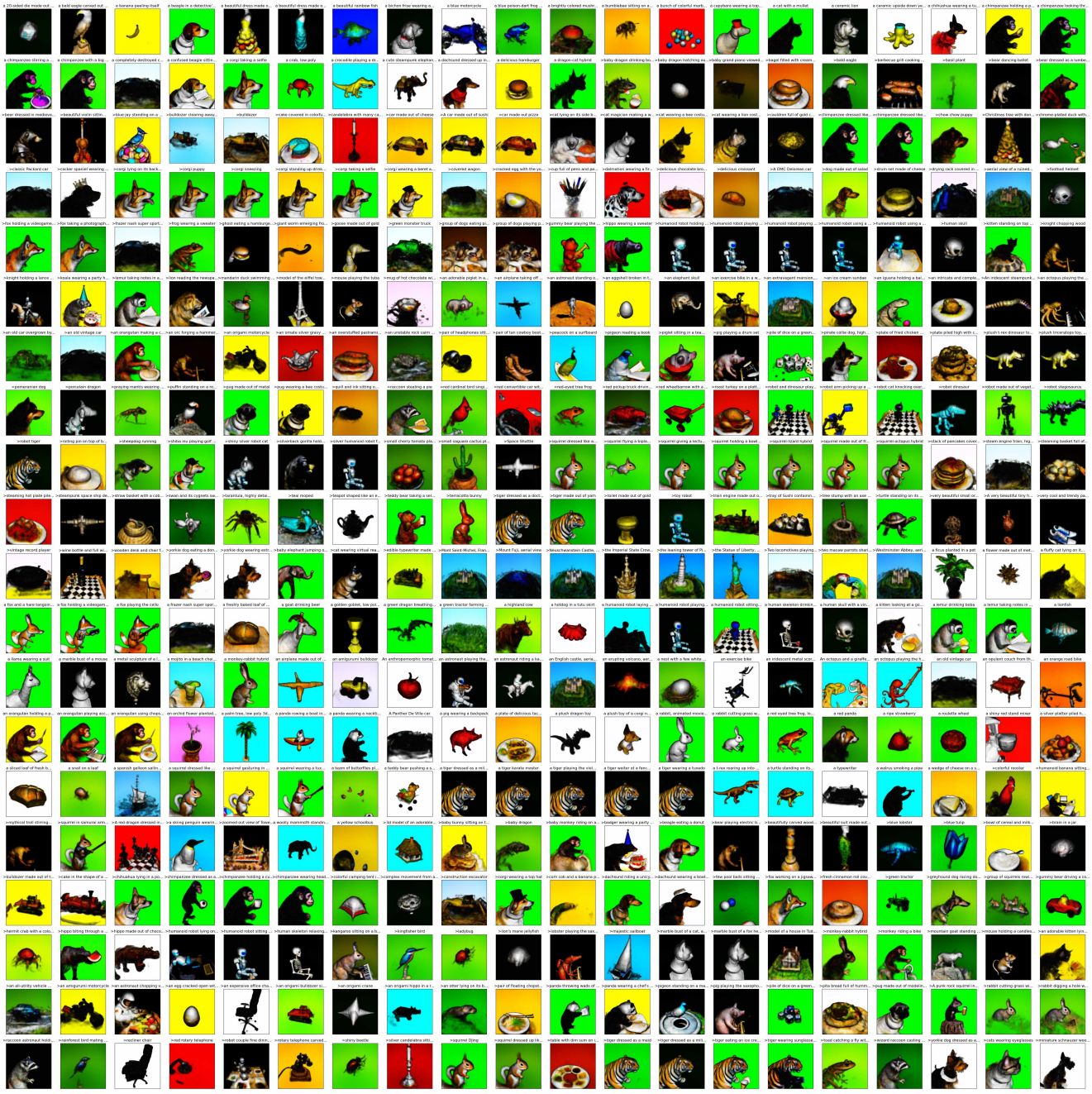


Figure 14: We show full results from our method on the DF411 prompt set, which we truncate for Figure 9. There are various examples of the model re-using object components across prompts – see Figure 15.

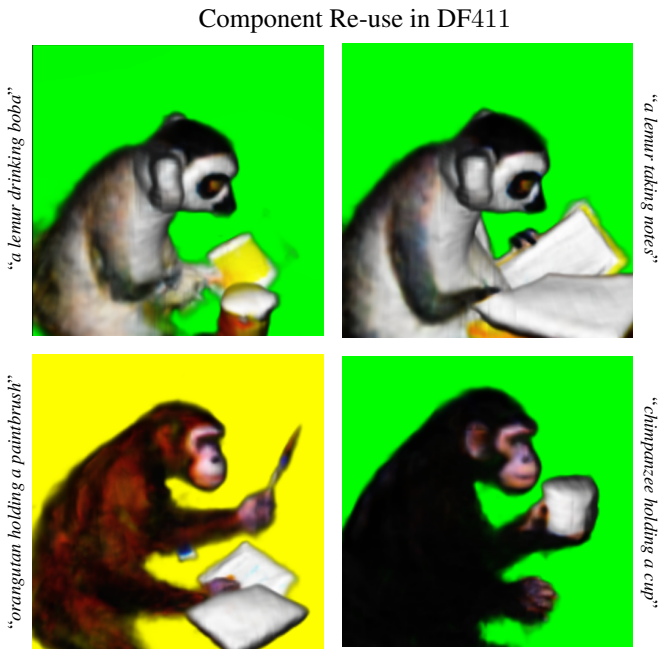
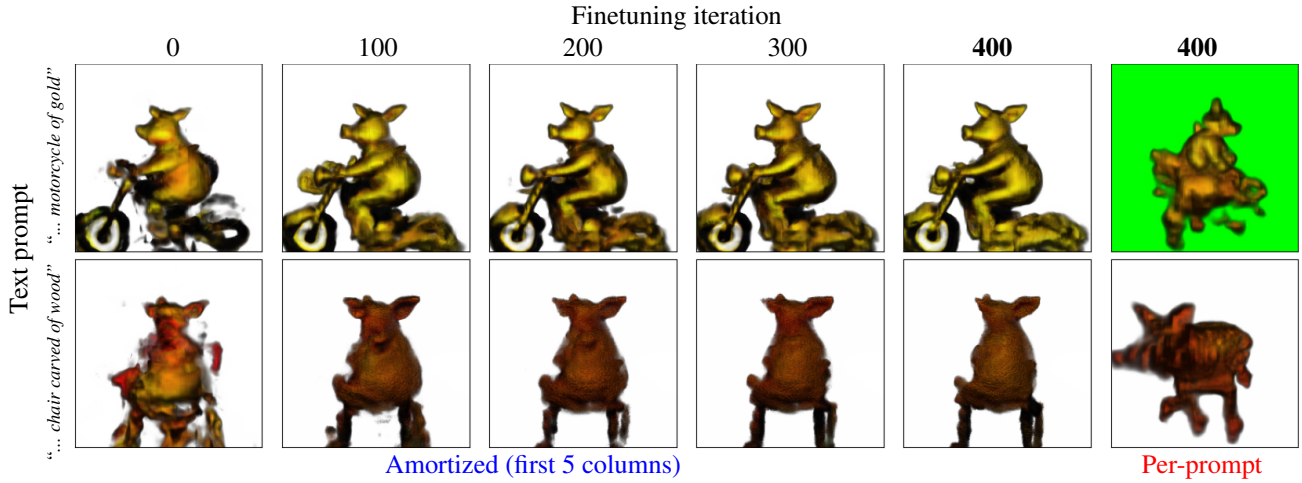


Figure 15: We show examples of prompts in which our model (from the DF411 run in Figures 9 and 14) re-uses components, showing a means by which amortization saves compute. *Top:* The lemur is re-used with different activities. *Bottom:* The orangutan is re-colored to a chimpanzee and given a different activity.

Figure 16: We investigate generalization on the DF411 run (App. Fig. 14). *Left*: Generalization to interpolated embeddings, which produces suboptimal results that we improve by amortizing over interpolants as in Figure 3. *Right*: Generalization to compositional embeddings. **Takeaway**: The generalization is promising, yet could be improved, motivating training on large compositional sets in Figures 6 & 8, and training on interpolants as in Figures 3, 18, 19, & 20.



Various strategies on "*a pig wearing medieval armor holding a blue balloon*"

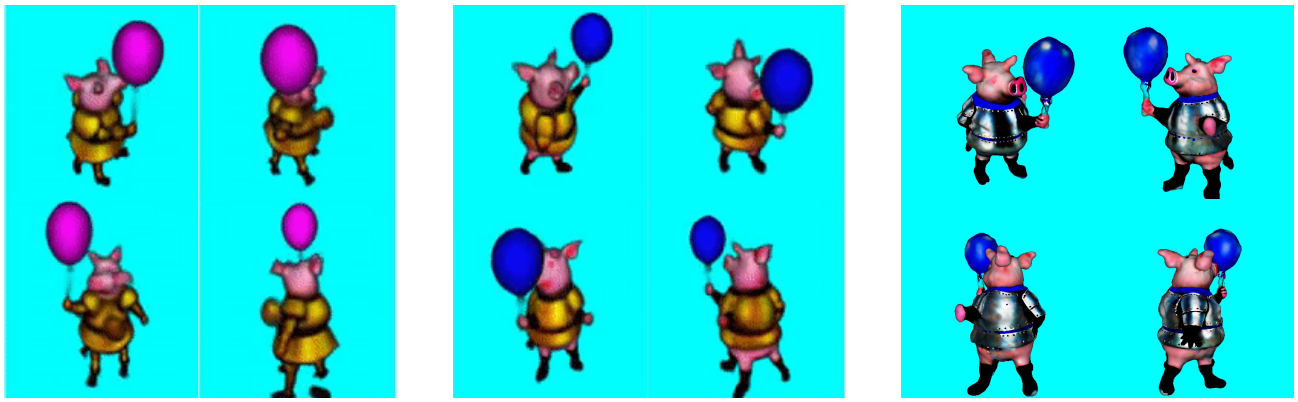


Figure 17: We display the results of finetuning held-out, unseen testing prompts from Fig. 2. *Top:* For amortization, we finetune from the final optimization value, while for per-prompt, we finetune the model from a random initialization. We achieve higher quality with fewer finetuning updates. *Bottom:* Per-prompt optimization fails to recover a blue balloon, and can not be recovered with finetuning. In contrast, amortized optimization recovers the correct balloon and can be fine-tuned using Magic3D's second optimization stage [2].

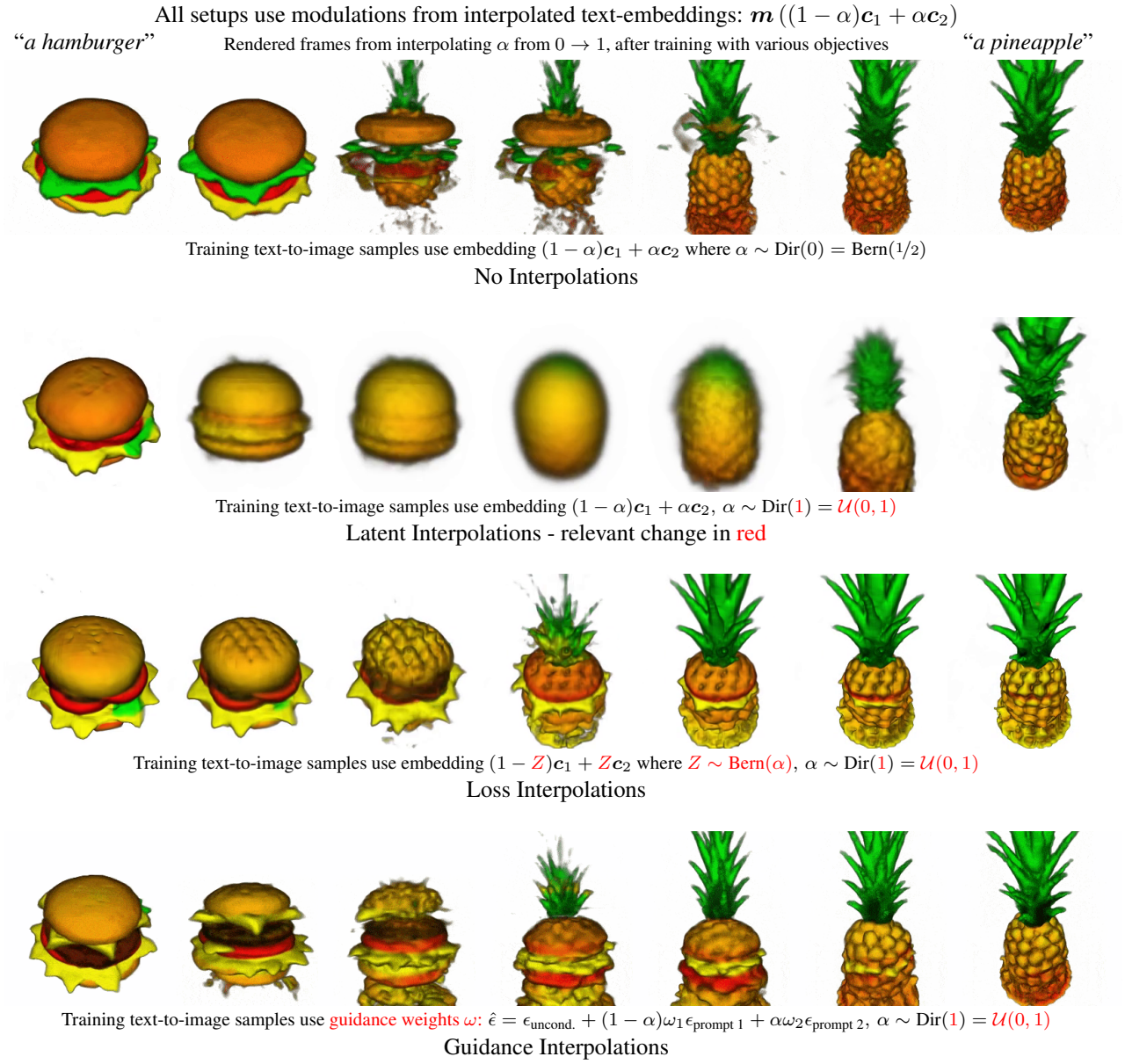


Figure 18: We contrast amortizing over different types of interpolations as described in Section B.1.14. For all examples, we give the mapping network $\boldsymbol{m}$ the interpolated embedding $(1-\alpha)\boldsymbol{c}_1+\alpha\boldsymbol{c}_2$. However, we vary the embedding used by the text-to-image model. **Takeaway:** We can amortize over various training methods to produce qualitatively different results. *Top:* We use no interpolants during training, which can just dissolve between the endpoints. *Latent Interpolation:* We simply interpolate between the latent embeddings used for image sampling. *Loss Interpolation:* We interpolate the loss function used in training between the prompts, producing objects simultaneously solving both losses. *Guidance Interpolation:* We interpolate the guidance weight applied to the prompts, as explored in Magic3D (without amortization) [2].

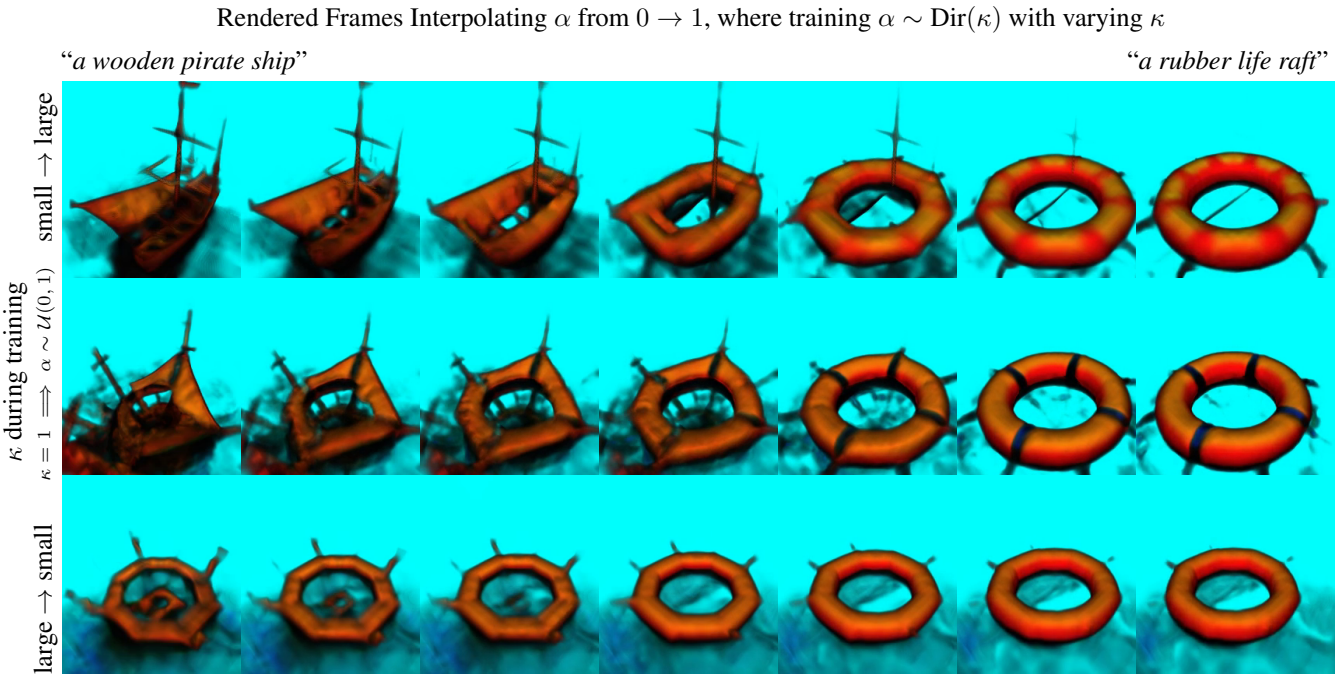


Figure 19: We display the results for differing strategies for changing the concentration parameter κ for the distribution of the interpolation weights α. Note that a concentration of $\kappa = 1$ is simply a uniform distribution: $\text{Dir}(1) = \mathcal{U}(0, 1)$. For both results, we train for 5000 steps with an initial concentration κ, which we then change for the final 5000 steps. **Takeaway:** The initial shapes learned strongly influence subsequent training, and a "large" concentration κ focuses on the midpoint, while a "small" concentration focuses on the endpoints. If we want the original prompts in the interpolation, then we should start with κ small, while if we desire a steering-wheel-life-raft satisfying both losses, we should start with κ large.

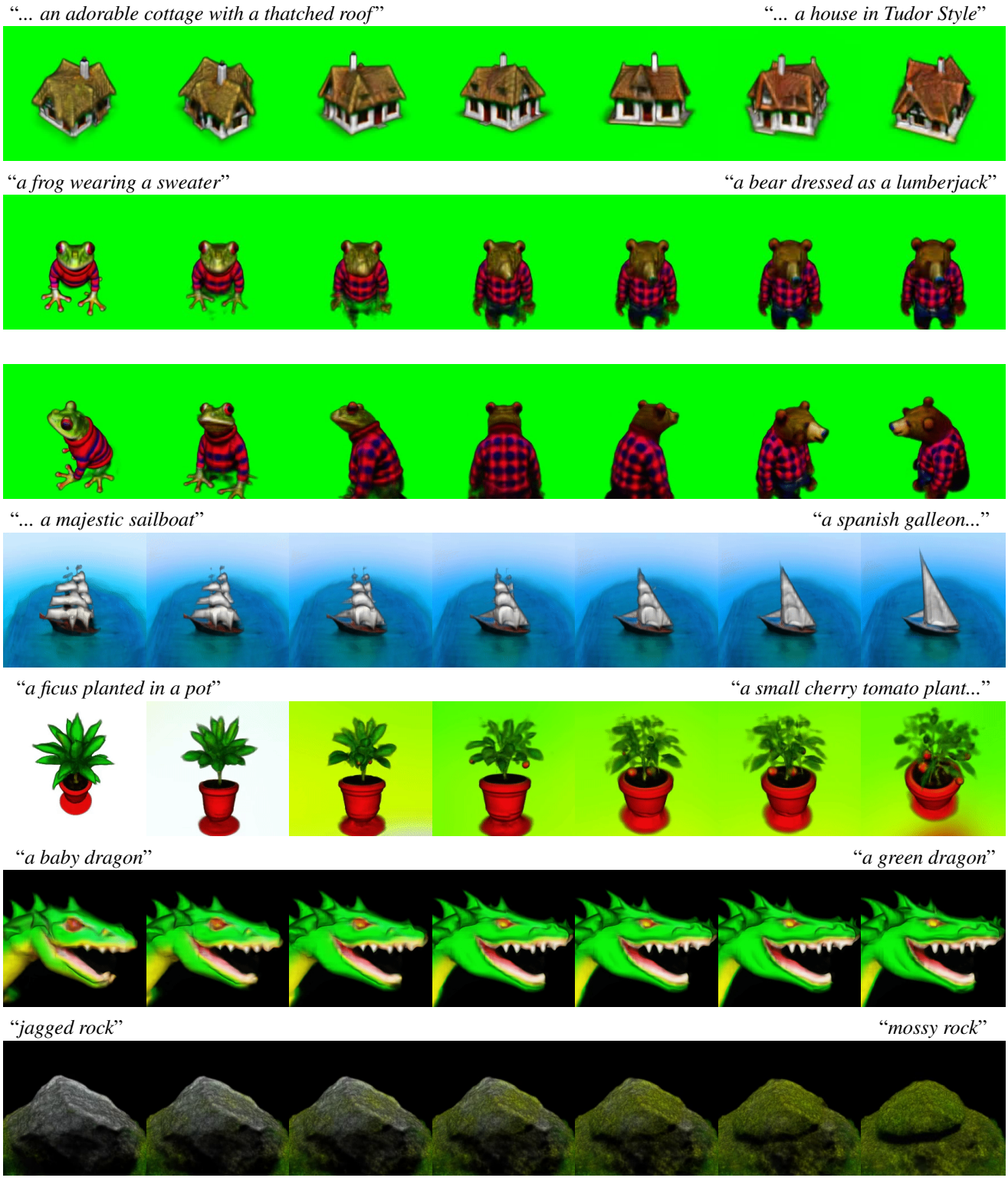


Figure 20: We include additional results for using our method to amortize over (loss) interpolants between prompts. We alternate between a fixed and varied camera view. We show examples of varied buildings, characters, vehicles, plants, landscapes, or a simple animation of "*a baby dragon*" aging into an adult.