A. Implementation and Training Details

A.1. Object Templates Rendering

We use BlenderProc[6] for template generation and use the camera intrinsics from the TUDL dataset. We scale each model and render 42 viewpoints distributed on an icosphere. The radius of this viewsphere is dynamically determined based on the size of the object, ensuring it remains within the camera's field of view. We add directional lighting in such a way that there are shadows that show the geometry of the model. For each rendered view, we generate RGB images and object masks, crop and resizes them to a standardized resolution of 224×224, and calculates bounding boxes around the object. The rendered images, masks, bounding boxes, 3D point clouds of the object, camera intrinsics, and camera poses are then saved into a compressed NumPy archive for each model. Please refer to Figure 7 for a visualization of templates rendered for objaverse [5] objects.

A.2. Variational Autoencoder (VAE) Training Details

We train the VAE on binary masks from the same training dataset as we train OC-DiT and tune the hyperparameters, such that the latent has zero mean and standard deviation 1.0. The training procedure focuses on learning a robust latent representation of object masks using a convolutional variational autoencoder with attention mechanisms. Data is loaded according to the dataset in 4.1. However, we load all templates and generate masks to increase the number of training samples per disk operation. Extensive data augmentation is performed using torchvision's transforms.v2. This includes random horizontal and vertical flips, random affine transformations (rotation, translation, scaling), and random perspective transformations. These augmentations aim to improve the model's generalization capabilities. The latent space has a dimensionality of 64 channels, and the model processes single-channel mask images as both input and output. During the forward pass, augmented mask images are fed into the encoder of the VAE. The encoder produces a latent representation, from which the decoder reconstructs the mask. The training process utilizes the Evidence Lower Bound (ELBO) loss, with a beta parameter of 10^{-5} to control the KL divergence term. This balances reconstruction accuracy and latent space regularization. Optimization is performed using the Adam optimizer with a learning rate of 3×10^{-4} and no weight decay. A learning rate scheduler is employed, featuring a linear warmup followed by cosine annealing. The training proceeds in batches of size 128 for 70 epochs, with the learning rate adjusted by the scheduler.

A.3. OC-DiT Training Details and Hyperparameters

For each sample of the dataset that we load, we have to reduce the number of objects per sample to the desired number. We only load the templates for the objects that we select in the following process. We sample a random point in the image and calculate a weighting scalar dependent on the distance to the sampled point. We use this weighting to randomize the different objects that we keep in the sample. This allows us to bias the sampling process to choose objects that are closer to one another, resulting in better training signal. OC-DiT leverages a pre-trained variational autoencoder (VAE) for latent space operations that has to be pre-trained.

`coarse` is trained on 252×336 images and 288×384 segmentations (18×24 patches), conditioned on multiple objects per sample. The decoder used 10 blocks with a hidden size 1152, and 12 attention heads. Training lasted 150 epochs, with 1000 steps per epoch and batch size 16 on 2 A100 GPUs, using the Adam optimizer with learning rate of $2 \cdot 10^{-4}$), linear warmup for 2000 steps, and cosine annealing after 72%). Data augmentation included random crops and jittered bounding boxes around all conditioning objects.

`refine` is trained on 224×224 images and 256×256 segmentations (16×16 patches), conditioned on single objects. The decoder uses 10 blocks, with a hidden size 1152, and 12 attention heads. Training lasted 200 epochs, with 1000 steps per epoch, and batch size 128 on 2 A100 GPUs, using the Adam optimizer with learning rate $1 \cdot 10^{-4}$, linear warmup for 2000 steps, and cosine annealing after 72%. Training data consists of cropped, jittered object regions.

In this work we use 12 template views due to VRAM constraints, but any number of templates can be used. We train the model on 8 object classes per sample which is a tradeoff between number of objects per sample and computational overhead. The models object capacity is 16. We randomly crop the image with a probability of 0.7 with a scale of $[0.5, 1.1]$ meaning that the image is randomly cropped to a size by this scale. If we do not randomly crop the image, we perform a tight crop around all object instances that we jitter. The Adam optimizer is employed with a learning rate of $2e-4$ and no weight decay. We use a linear warmup for the learning rate for 2000 gradient steps and a cosine annealing after 0.72% of the training is complete. The training uses a batch size of 8 per GPU. We apply two types of data augmentations, RGB augmentations with a probability of 0.9 and background augmentations with a probability of 0.2. The RGB augmentations entail blur, sharpness, contrast, brightness and a basic color augmentation. We use an iterative dataset and each epoch is complete after 1000 training batche with a maximum of 120 epochs. Please refer to Figure 10 and Figure 11 for visualization of the diffusion process of a trained model.

Figure 7. **Templates rendered for objaverse [5] objects.**

A.4. Positional Encodings

We evaluate the following techniques for scaling object queries, addressing the critical challenge of adapting mod-

els trained on a specific number of objects $N_{\mathcal{O}}^{\text{train}}$ to handle a potentially different number of objects at inference ($N_{\mathcal{O}}^{\text{test}}$). We delve into various approaches, each visualized in Figure 8.

(i) *Baseline 1: Sequential Inference after Training on Limited Objects.* The model is trained on $N_{\mathcal{O}}^{\text{train}}$ objects. Subsequently, during inference, the model processes each of the $N_{\mathcal{O}}^{\text{test}}$ object chunks individually, effectively performing a series of independent inferences. This method tests the model's ability to generalize to unseen objects within the fixed training capacity without any explicit scaling mechanisms. A clear downside is that during self-attention, not all queries can attend to one another. This leads to scenarios where we can get multiple for one segment of the image.

(ii) *Baseline 2: Direct Training with Target Object Count.* As another baseline, this method directly trains the model on a dataset that precisely matches the number of objects anticipated at inference, $N_{\mathcal{O}}^{\text{test}}$. This provides a performance ceiling, showcasing the optimal results achievable when the training and inference object counts are perfectly aligned. It allows us to assess the performance loss when object counts are not aligned. However, as we see in the results, due to training dynamics this baseline performs worse than other positional encoding methods.

(iii) *Test-Time Positional Encoding Interpolation: Adapting to Varying Object Counts at Inference.* This technique addresses the mismatch between $N_{\mathcal{O}}^{\text{train}}$ and $N_{\mathcal{O}}^{\text{test}}$ by dynamically adjusting the positional encodings during the inference phase. Specifically, the positional encodings learned during training for $N_{\mathcal{O}}^{\text{train}}$ objects are interpolated to accommodate the $N_{\mathcal{O}}^{\text{test}}$ objects. This allows the model to handle a different number of objects without requiring retraining, offering flexibility in deployment scenarios.

(iv) *Random Interval Training: Enhancing Robustness through Variable Subsets.* This method trains the model with a positional encoding capacity sufficient to handle the maximum number of objects, $N_{\mathcal{O}}^{\text{test}}$. However, during each training iteration, only a random, contiguous interval of length $N_{\mathcal{O}}^{\text{train}}$ is utilized. This approach aims to improve the model's robustness and generalization by exposing it to various subsets of the larger positional encoding space, preventing overfitting to a specific object ordering.

(v) *Random Interpolation Training: Leveraging Interpolation for Flexible Object Handling.* Similar to the previous technique, this method trains the model with a positional encoding capacity designed for $N_{\mathcal{O}}^{\text{test}}$ objects. However, instead of using contiguous intervals, random intervals of lengths greater than $N_{\mathcal{O}}^{\text{train}}$ are selected. Subsequently, the positional encodings within these intervals are interpolated down to $N_{\mathcal{O}}^{\text{train}}$ during training.

(vi) *Random Permutation Training: Promoting Generalization through Diverse Object Orderings.* This method also leverages a positional encoding capacity designed for $N_{\mathcal{O}}^{\text{test}}$ objects. During each training iteration, a random subset of $N_{\mathcal{O}}^{\text{train}}$ indices is selected from the full range of $N_{\mathcal{O}}^{\text{test}}$ indices, effectively permuting the object order. This random permutation of indices aims to improve the model's generalization by exposing it to diverse orderings of the input objects, mitigating biases related to specific object arrangements.

A.5. Common Failure Modes of `coarse`

In Figure 10 and Figure 11 we showcase a few randomly selected samples from the YCBV, HB, and LMO datasets using the noise discretizations $\rho = 5$ and $\rho = 15$. The top row of Figure 10 shows a very common failure mode for YCBV. The clamps, of which there are multiple, only differ in size. To to our template rendering scheme, where we crop closely around the foreground, this change in size is not reflected. Hence, our model has no basis to compare and differentiate the two.

Another common failure mode can be seen in the bottom row of Figure 11 on the LMO dataset. The object are generally quite small, most likely smaller than the average object size in our training data. This leads to some instances of the ensemble to estimate wrong instances which results in multiple instances per object in some cases. These instances deteriorate the performance but since the objective of the coarse model is a high recall, we leave them in and let the refinement model compare which segmentation fits best to the model.

A.6. Training Datasets

To train our diffusion models, we generate two new datasets, one based on the 3D models of Google Scanned Objects [9] and a subset of objects from the Objaverse [5] dataset, picked from the lvis and staff-picked annotations. We use Blenderproc [6] for rendering and base the rendering pipeline closely to the scripts used to generate training datasets for the BOP challenge. For each scene, we create a basic room environment and set up initial lighting from both a large ceiling plane and a point source. We randomly select a varying number of objects from our collection, load them into the virtual scene, and apply randomized material properties, such as roughness and specularity. To ensure realistic arrangements, these objects are enabled for physics simulation, allowing them to settle naturally on surfaces and against each other after an initial random placement. Lighting elements are also randomized in terms of color, strength,

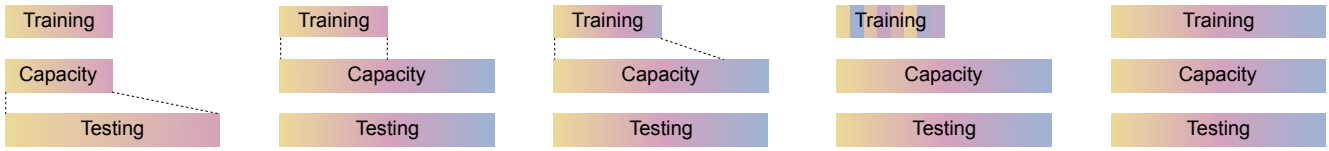
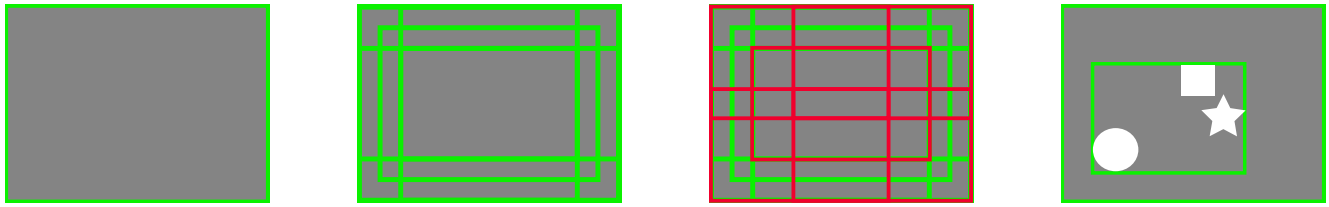


Figure 8. **Positional encodings and adaption methods ablated in this paper.** a) we train a model with a fixed capacity and interpolate the positional embedding during inference to the desired size. b) we train a model with a higher capacity than the number of training objects and randomly select intervals during training. c) same as b) but we sample intervals that are larger than the training number of objects and interpolate the positional encoding to number of training objects d) we train the model with a higher capacity and randomly select positional codes e) we train the model directly on the increased number of objects.



(a) Full image (b) 6 crops (c) 11 crops (d) Full image guided

Figure 9. **Visualization of test-time spatial** augmentations. (a) uses the full image as input, (b) adds 5 spatial crops to the full image, each smaller by a fixed number of pixels and shifted (c) adds another 5 spatial augmentations by adding more crops smaller, and (d) illustrates the guided spatial augmentation. We use the previous model prediction to generate a tight bounding box covering all estimates that we run a diffusion for in a second step.

and position, and various textures are randomly applied to the room's surfaces to add visual diversity. Once the objects are settled, we generate multiple camera viewpoints for the scene, ensuring that the camera has an unobstructed and interesting view of the objects. Finally, for each valid camera pose, it renders both color images and depth maps, along with annotations for the objects' positions and orientations, all of which are then saved in the BOP dataset format. After each scene is rendered and its data recorded, the objects are removed to prepare the environment for the next scene, allowing for the generation of a large number of unique synthetic scenarios.

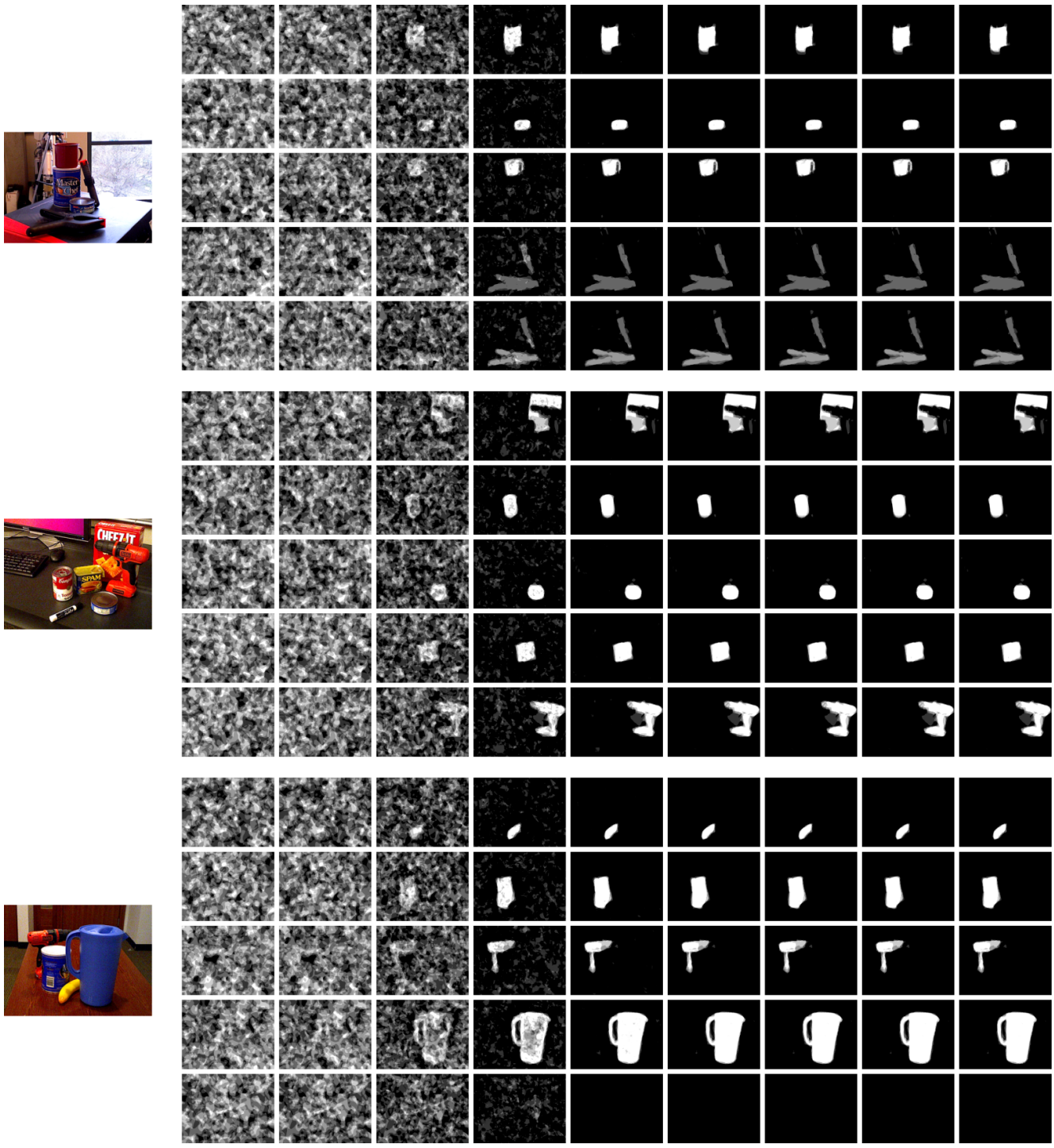


Figure 10. Some examples of the generative process using $\rho = 15$ for noise discretization.

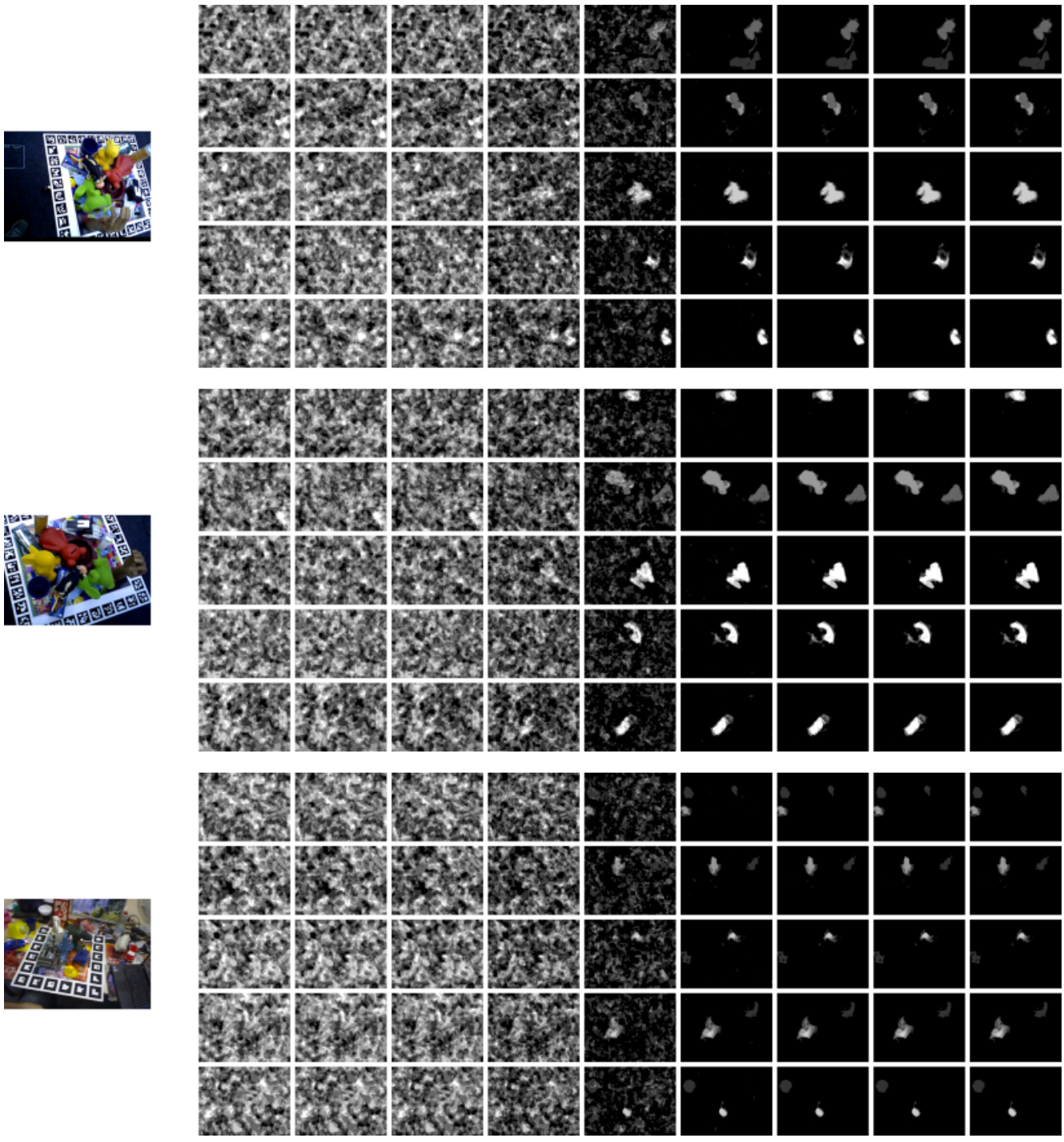


Figure 11. Some examples of the generative process using $\rho = 5$ for noise discretization.