Hash3D: Training-free Acceleration for 3D Generation

Supplementary Material

In this document, we provide additional information and analysis for our proposed Hash3D. We begin by describing how the feature is extraction from diffusion model in Section A. Following that, we delve into further analysis for Hash3D, including ablation studies in Section B, and provide visualizations in Section C. More implementation details are disclosed in Section D, which also includes the pseudo-code for our hash table data structure and the feature hashing process in Section E. For additional information, please refer to the source code available in the uploaded files.

A. Details for Feature Extraction

As Hash3D involves the extraction of features from U-Net, we here introduce how we define and indexing those features. As illustrated in Figure 10, we adopt the definition that, the indices for the downsampling layers are arranged in decreasing order, whereas for the upsampling layers, the indices follow an increasing order. With in total l upsample layers and l down-sample layers, the skip connection merges high-level features from U_{i+1} with low-level features from D_i, as expressed by the equation:

$$\mathbf{v}_{i+1}^{(U)} = \text{concat}(D_i(\mathbf{v}_{i-1}^{(D)}), U_{i+1}(\mathbf{v}_i^{(U)})) \tag{10}$$

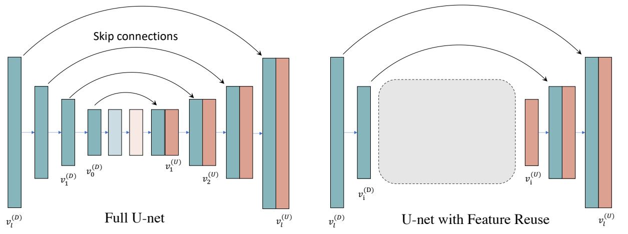

Figure 10. Structure of the U-Net and our feature extraction setup.

If we would like to reuse the feature $\mathbf{v}_i^{(U)}$ from the U-Net, upon retrieval, the model only requires the forwarding of layers D_l to D_i and of U_{i+1} to U_l. This approach allows us to bypass all intermediate computational blocks, enhancing efficiency.

B. Analysis and Ablation Study

B.1. Key-based hashing & Content-based Aggregation

In fact, Hash3D utilizes a hierarchical process for feature reuse, involving a *key-based* hashing stage and a *content-based* feature aggregation stage. In the first stage of key-based hashing, Hash3D computes a hash code corresponding to a bucket according to the camera pose and time step. This efficiently retrieves a set of candidate features. Subsequently, Hash3D performs a content-based refinement within the retrieved bucket. Features are aggregated based on the similarity (distance) between their input latents.

This section investigates the effectiveness of the two-stage hashing.

Experimental setup. To assess the contribution of each hashing stage, we conducted two experiments:

- **Ablation 1: Removing Key-based Hashing.** In this experiment, we removed the key-based hashing stage. Instead, the query feature's latent vector was directly compared against the entire pre-extracted feature pool (no hashing at all). To achieve this, we established a queue with maximum length of 1000 to store all previously extracted features.
- **Ablation 2: Removing Content-based Aggregation.** Here, we omitted the content-based aggregation stage. As replacement, within each bucket, only the features with closest hash key (camera pose and timestep) will be returned.

We test it on Zero-123 (NeRF) and compare the visual fidelity.

Results. Our study presents visualization for various retrieval strategies, as shown in Figure 11. We refer to our first variation as "Ours without key hashing" and the second as "Ours without feature aggregation".

It is observed that our complete solution achieves the highest visual fidelity. Interestingly, the exclusion of feature aggregation leads to the emergence of moiré patterns, exemplified by the `eye of the robot`. This phenomenon occurs because multiple hash keys can map to the same cached feature, resulting in overlapping patterns in the generated images. On the contrary, the omission of the key-based hashing stage produces images that are overly smooth and lack detail. By first filtering features within a grid and subsequently aggregating them based on latent similarity, our method ensures clearer boundaries of the generated objects.

B.2. Hashing Feature *vs.* Hashing Noise

Beyond the quantitative results presented in Table 9 of the main paper, we offer visual comparisons between hashing features and hashing denoising predictions in Figure 12. We implement Hash3D on top of Zero-123 (NeRF) and visualize the multi-view images of the reconstructed objects.

Hashing noise leads to the generation of saturated 3D objects, occasionally exhibiting mosaic patterns. Although

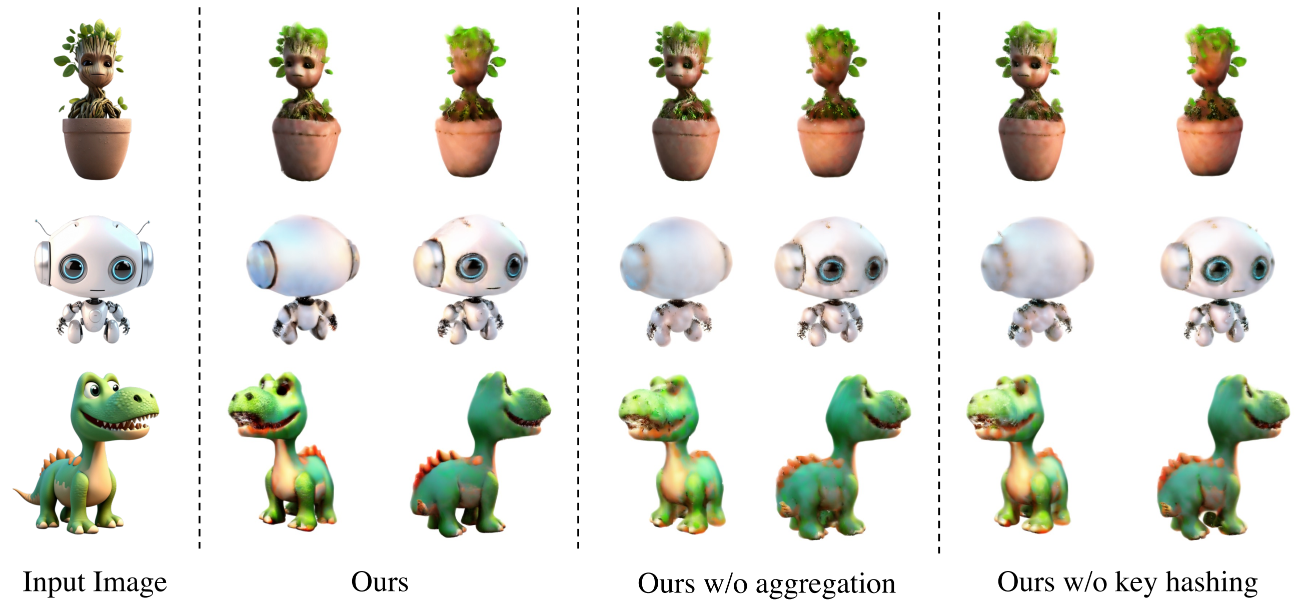

Figure 11. Results with different hashing strategy. "Our w/o aggregation" is short for "Ours without feature aggregation" and "Ours w/o key hashing" is for "Ours without key hashing".

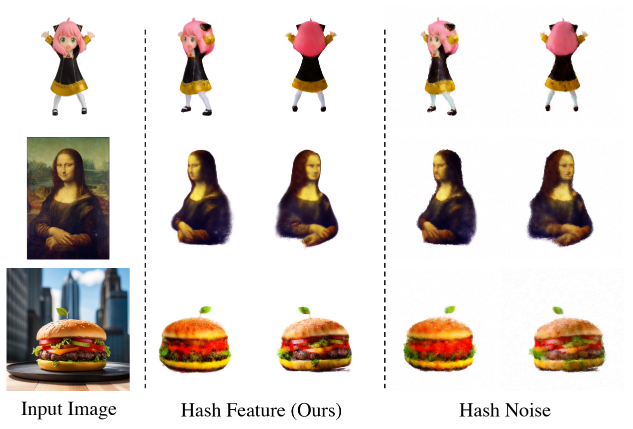

Figure 12. Results when hashing features or hashing the denoising predictions.

this method proves to be slightly faster, it compromises visual quality, aligning with our quantitative findings. Consequently, we advocate for the use of feature hashing in our study, as it maintains higher fidelity in the visual results.

B.3. Optimal Layer for Feature Extraction

In caching and retrieving features within diffusion models, a critical question arises: *which layer's features should be extracted?* Ideally, extracting features from deeper layers, closer to the output, can significantly reduce computational overhead but might result in a slight loss of fidelity in the predicted images. On the other hand, hashing features from earlier, low-level layers retains higher performance at the cost of increased inference overhead. This presents a trade-off between computational efficiency and output quality. We in this section valid our selection.

For example, the Zero123 U-Net contains 10 skip connections, each associated with a down-sampling layer and a up-sampling layers. We test 10 positions for feature extraction, and show the results.

Figure 13 illustrates that, generally, a larger layer index i—indicating proximity to the output—results in reduced optimization time but slightly diminished visual quality. However, given the minimal impact on fidelity, we opt for using $i = 10$, the layer before the last upsampling, for feature extraction in our experiments. This choice effectively balances computational efficiency with the maintenance of high visual quality.

C. Additional Results

This section presents further visualizations demonstrating the effectiveness of our method. Specifically, we compare our Hash3D+Zero123 approach with the original Zero-123 method in the context of image-to-3D reconstruction, as illustrated in Figure 14. Additionally, we evaluate our method against Gaussian-Dreamer for text-to-3D generation, as shown in Figure 15. Our results showcase superior visual quality: we achieve this in 7 minutes compared to Zero-123's 20 minutes, and in 10 minutes against Gaussian-Dreamer's 15 minutes.

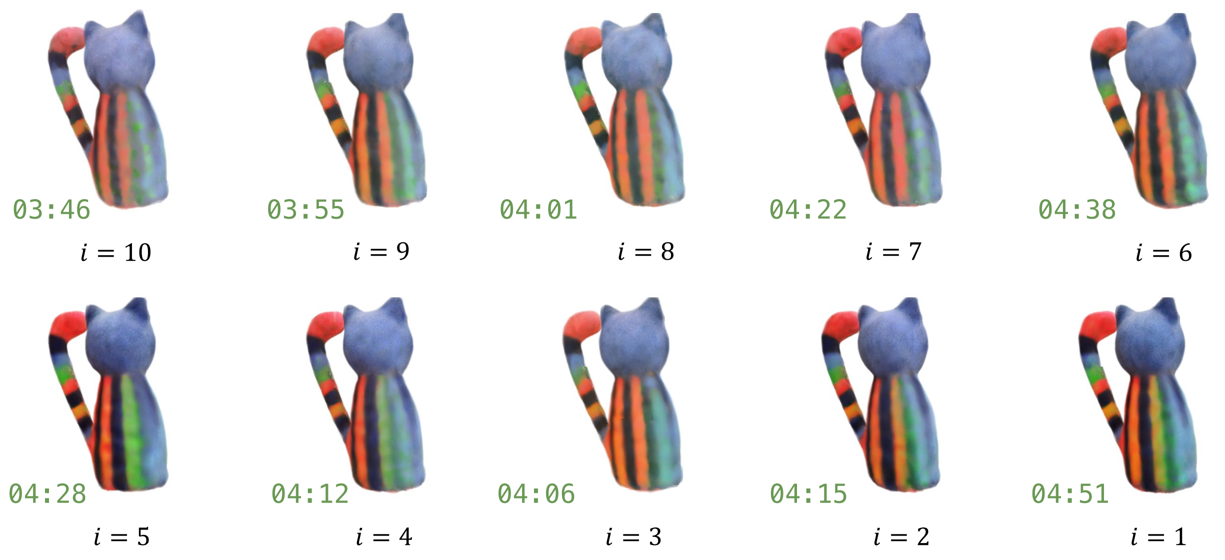

Figure 13. Impact of feature hashing at various layers on optimization time and visual fidelity. Note that, *larger layer index indicating closer to the output, with smaller computation.*

D. Implementation Details

We use the official implementation for Dream-Gaussian and Gaussian-Dreamer. For all other methods, we take the `threestudio`'s implementations, with their default experimental configurations.

Image-to-3D:

- **Zero-123 (NeRF)**: We employ NeRF with hash grid encoding for the 3D representation. We leverage `stable-zero123` as the diffusion model to optimize this representation using the SDS loss. A classifier-free guidance of 3.0 is used, and the Adam optimizer updates the parameters for 1,000 steps with learning rate of 0.01. We use a batch size of 1.
- **Zero-123 (GS)**: We employ Gaussian Splatting for the 3D representation. For other details, we follow the setup for Zero-123 (NeRF). We use the implantation from `threestudio-3dgs` [§].
- **Dream-Gaussian**: We use the official implementation [¶]. The initial Gaussians consists of 5,000 randomly colored points on a sphere. In the first stage, we update the parameters for 500 iterations using `stable-zero123` model and the SDS loss. The second stage focuses on refining the mesh for 50 additional steps with the RGB MSE loss. Since this stage doesn't require the SDS loss, we employ Deepcache [30] for acceleration. Deepcache can be considered a simplified version of our Hash3D, focusing solely on temporal reuse.
- **Magic-123**: Following the configurations from `threestudio`, we use `stable- diffusion-v1-5` as the text-to-image diffusion model, and `stable-zero123` as the image-to-3D diffusion model. In the first stage, both models work together to optimize a NeRF as the 3D representation for 10,000 iterations. This NeRF is then converted into an explicit surface mesh representation [48] in the second stage, which also undergoes optimization for another 10,000 iterations. Both stages use the SDS loss, where the loss weights for text-to-image and image-to-3D diffusion are set to 0.025 and 0.1.

§https://github.com/DSaurus/threestudio-3dgs

¶https://github.com/dreamgaussian/dreamgaussian/tree/main

Text-to-3D:

- **Dreamfusion**: We use the `stable-diffusion-2-1-base` to optimize the NeRF representation with hash encoding, using SDS loss. We apply a classifier-free guidance technique, setting its scale to 100. For the optimization process, we use the Adam optimizer with a learning rate of 0.01 and run the process for a total of 10,000 iterations.
- **Latent-NeRF**: We use the same setup as in above Dreamfusion experiment, except that we use a vallina NeRF representation.
- **SDS+GS**: Compared to the Dreamfusion above, the only difference is that we use a 3D Gaussian Splatting to represent the 3D object. The 3D Gaussians are initialized from the shap-e [11] predicted mesh. We use the implementation from `threestudio-3dgs`.
- **Magic3D**: The first stage of Magic3D involves updating an instant-npg like NeRF representation for 10,000 iterations, using the `stable-diffusion-2-1-base` model and SDS loss. Subsequently, this NeRF is converted into an explicit surface mesh, which is then opti-

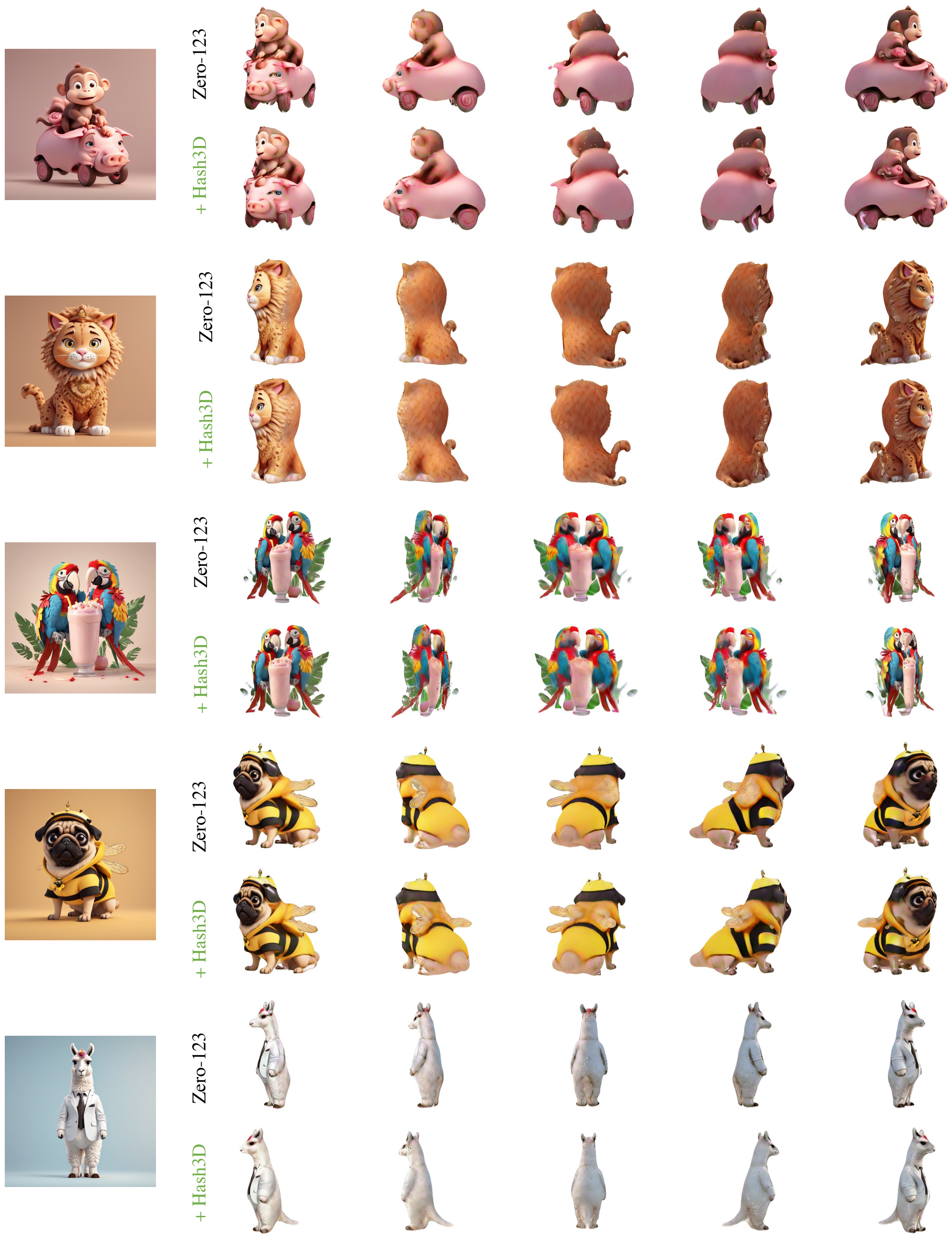

Figure 14. Qualitative Comparison when applying Hash3D on top of Zero-123.

mized for an additional 10,000 iterations.

- **GaussianDreamer**: We take the official implementation [‖] to do the experiments. The Gaussian points are initialized from shap-e [11] predicted mesh. Op-

[‖]https://github.com/hustvl/GaussianDreamer

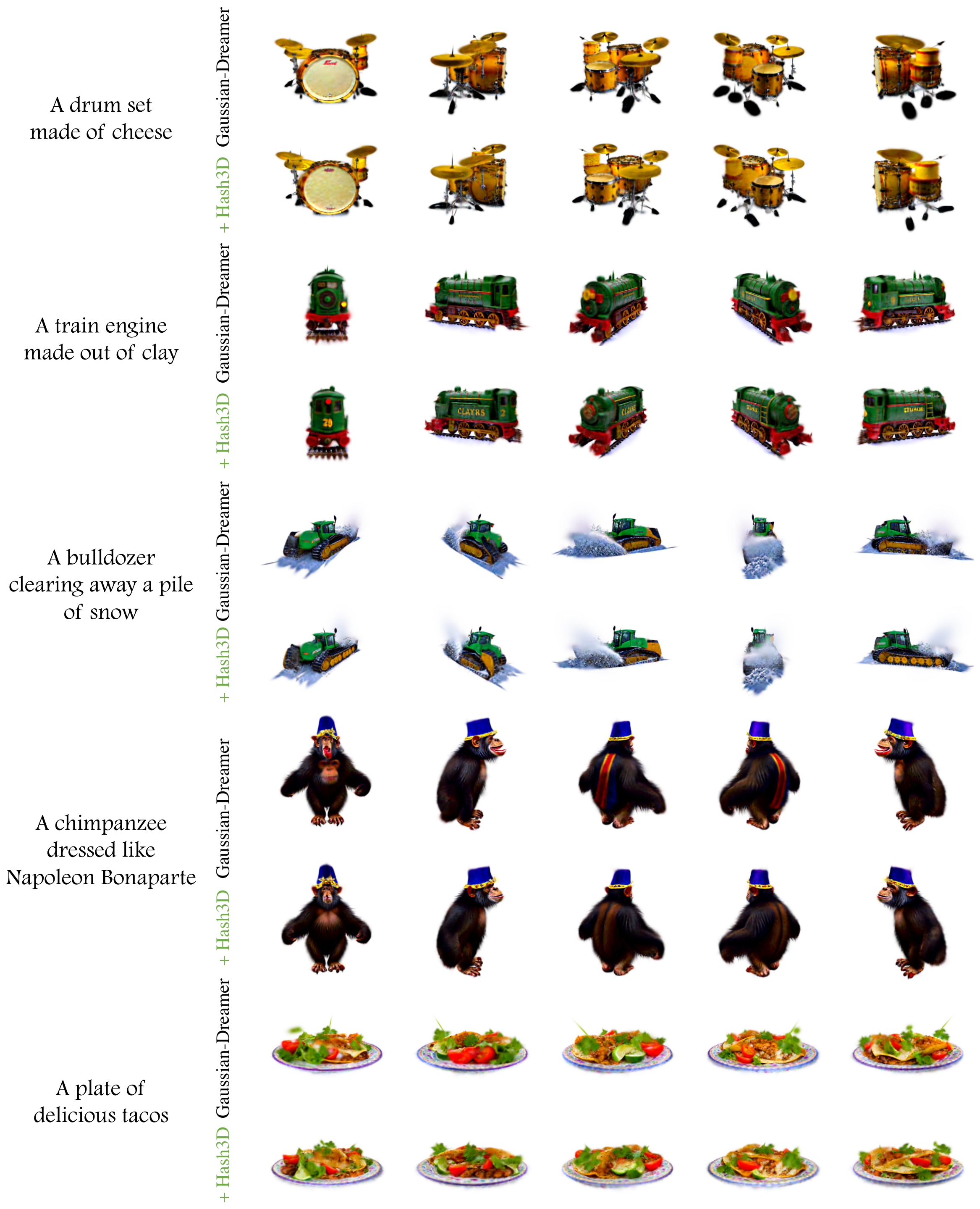
A drum set made of cheese
Gaussian-Dreamer
+ Hash3D
A train engine made out of clay
Gaussian-Dreamer
+ Hash3D
A bulldozer clearing away a pile of snow
Gaussian-Dreamer
+ Hash3D
A chimpanzee dressed like Napoleon Bonaparte
Gaussian-Dreamer
+ Hash3D
A plate of delicious tacos
Gaussian-Dreamer
+ Hash3D

Figure 15. Qualitative Comparison when applying Hash3D on top of Gaussian-Dreamer.

timization is conducted over 1,200 steps using the `stable-diffusion-2- 1-base` model with a classifier-free guidance scale of 100, and Adam optimization at a learning rate of 0.001.

E. Pseudo-code for Hash3D

In our paper, we introduce a core mechanism that utilizes a grid-based hashing table to organize features extracted across various camera poses and time steps. This section provides a detailed overview, including pseudo-code, for two main components: (1) the data structure and associated functions of our grid-based hashing, in Listing 1, and (2) the forwarding process of diffusion model with feature hashing, in Listing 2.

Listing 1. Pseudocode for GridBasedHashTable

```
# GridBasedHashTable Class Definition
Class GridBasedHashTable:
   # Initializes the class with parameters for the hash
        table configuration
   Constructor(delta_c: List, delta_t: Float, N: List,
        max_queue_length: Int, hash_table_size: Int):
      # Spatial and temporal grid sizes and constants
           for hashing
      Store delta_c, delta_t, and N as tensors
      # Maximum queue length for each hash table entry
           and overall size
      Store max_queue_length and hash_table_size
      # Initialize hash table as a list of queues, one
           per hash table entry
      hash_table ← list of deques, each with
           maxlen=max_queue_length

   # Computes a raw hash index based on
        spatial-temporal key
   def compute_hash_index_raw(key: Tensor) -> Int:
      # Applies hashing formula to compute index based
           on key
      i, j, k = floor(key[:3] / self.delta_c)
      l = floor(key[3] / self.delta_t)
      idx = i + self.N[0] * j + self.N[1] * k +
           self.N[2] * l
      return idx

   # Modulo operation to ensure index within hash table
        size
   def compute_hash_index(key: Tensor) -> Int:
      # Modulo hash_table_size to find actual index in
           hash table
      idx = self.compute_hash_index_raw(key)
      return idx % self.hash_table_size

   # Appends feature data to the hash table, associated
        with spatial-temporal key and latent
   def append(key: Tensor, feature: Tensor):
      # Finds hash table index for given key
      idx ← compute_hash_index(key)
      # Appends the key, meta key, and feature as a
           tuple to the specified queue
      hash_table[idx].append((key, feature))

   # Queries the hash table for data matching a
        spatial-temporal key and meta key
   def query(key: Tensor, meta_key: Tensor) -> Tensor
        or None:
      # Finds hash table index for the query key
      idx ← compute_hash_index(key)
      # Retrieves the queue of data at the computed
           index
      queue ← hash_table[idx]

      # If the queue is empty, indicates no data for key
      if queue is empty:
         return None

      # Extracts noisy latent and features from the
           queue for comparison
      Unpack features from queue
      # Computes distances between the query meta_key
           and stored meta_keys
      Compute distances and apply softmax to derive
           weights
      # Aggregates features based on weights to get a
           single output
      Aggregate features using weights and return as
           aggregated output
```

Listing 2. Pseudocode for U-Net Inference with Feature Hashing

```
# function for U-Net forward pass with Feature Hashing
     (Example for Zero-123)
def forward_unet($x_{in}$, $vae_{emb}$, $t$, $t_{in}$, $cc_{emb}$, $polar$,
     $azimuth$, $radius$, $cache$, $cache_{layer\_id}$, $cache_{block\_id}$):
   Initialize $prv\_features$ to None
   # Create a key tensor for caching based on stacking
        input parameters
   $keys$ ← [$t$[:$batch\_size$], $polar$, $azimuth$, $radius$]

   # Conditionally update cache based on a predefined
        probability
   if random.random() < cache probability:
      # Query the cache for each item in the batch
      for each item $k$ in $keys$:
         $prv\_feature$ ← query hash table with key $k$

         # Store retrieved hashed features
         Update $prv\_features$ with hashed features

   # Determine if new features need to be cached
   $append$ ← $prv\_features$ is None

   # Perform U-Net prediction with potential use of
        cached features
   ($noise\_pred$, $prv\_features$) ← unet($prv\_features$, other
        inputs...)

   # Update cache with new features if necessary
   if $append$:
      for each item $f$ in $prv\_features$:
         Cache new features $f$ in the hash table

   return $noise\_pred$
```