SPARK: Sim-ready Part-level Articulated Reconstruction with VLM Knowledge

Supplementary Material

6. VLM Ablation

Analysis under Inaccurate VLM Guidance. The pipeline can recover from VLM errors, producing high-quality results, as shown in Fig.8. Given an real-world image, we evaluate our method under inaccurate VLM guidance, including: (a) miscounted parts in VLM results, where our method generates the corresponding reduced number of parts; (b) inaccurate or misaligned VLM part images with incorrect textures, where our final results recover the correct geometry with minor texture artifacts; (c) part guidance identical to the input image, where the object can still be generated successfully.

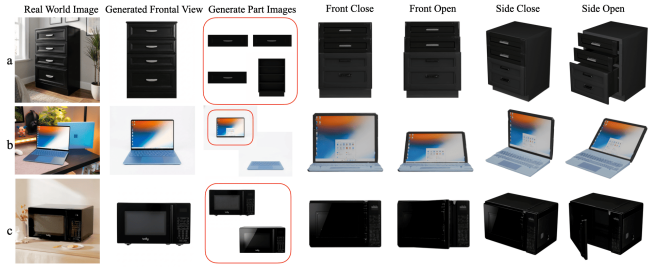

Figure 8. **Robustness Evaluation.** Given a real-world image, **SPARK** generates high-quality results even under inaccurate VLM image guidance, including: (a) miscounted parts; (b) misaligned parts with incorrect textures; (c) parts equal to the input.

VLM Selection and Prediction. We compared joint-capture failures between GPT-4o and GPT-5.2, achieving success rates of 0.76 and 0.81, respectively. Failures mainly occur on secondary joints (e.g., door handles, oven buttons). Fig. 8 shows **SPARK** produces high-quality results despite upstream errors. Part number errors cause over- or under-segmentation (Fig. 8(a)), which can be addressed with easy part splitting or merging. Our method also supports different VLMs and yields high-quality results in Fig. 9.

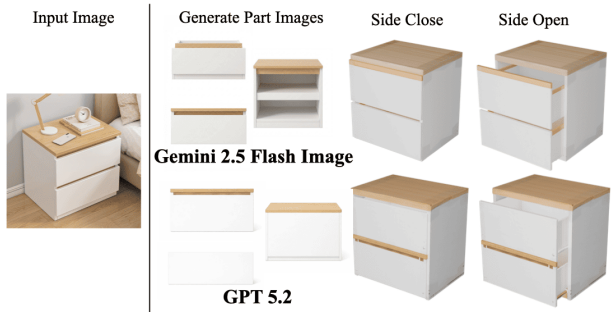

Figure 9. **VLM Selection.**

VLM Re-prediction. Table 5 shows that re-prediction improves discrete parameters (joint type and axis), and URDF optimization refines continuous ones.

Table 5. **Ablation of VLM Re-prediction.**

Variants	TypeErr↓	AxisErr↓
w/o Re-prediction	0.1489	0.2507
Ours	**0.1277**	**0.1605**

7. Joint Origin Optimization Results

As shown in Fig. 10, we present a challenging example to show how the joint origin can be optimized in practice. For the globe, the correct joint origin is difficult to infer because different candidate origins often lead to only slight visual differences in the resulting articulated states. In this example, we optimize only the continuous joint origin parameters while keeping the remaining joint attributes fixed. As shown on the right, the proposed method can still identify and exploit these subtle articulation-dependent cues. From iteration 1 to iteration 20, the optimized result gradually converges to an axis-symmetric configuration aligned with the rotation axis.

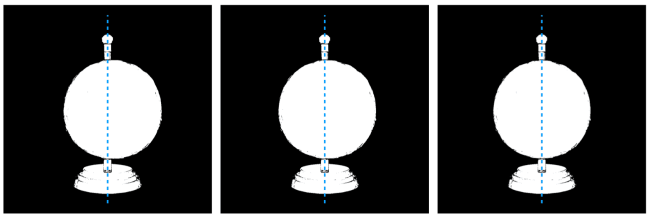

Figure 10. **Joint Optimization Example.** An example of optimizing the joint origin for a globe.

8. Network Architecture Details

In this section, we provide additional details of the network backbone used by SPARK. Our model follows a part-aware Diffusion Transformer (DiT) design with dual-image conditioning and hierarchical attention over kinematic links, built on top of a Variational Autoencoder (VAE).

8.1. Latent Representation and Image Encoders

Given an articulated object with K semantic parts, we represent each part k by a point-based surface encoding and a reference image:

- **Part surfaces**. Each ground-truth part mesh is sampled into P oriented surface points, yielding $\mathbf{S}_k \in \mathbb{R}^{P\times 6}$ (3D position + normal). A shared VAE encoder maps $\mathbf{S}_k$ to a latent sequence

$$\mathbf{z}_{k,0} \in \mathbb{R}^{T\times D_{\text{lat}}},$$

  where T is the number of latent tokens per part and D_{lat} is the latent dimension.
- **Part and global images**. For conditioning, we use a per-part reference image r_k and a single global input image I_0. The global image is replicated K times so each part sees the same holistic context.

Both r_k and I_0 are encoded by a shared DINOv2 image encoder into token sequences

$$E_k^{\text{loc}} \in \mathbb{R}^{L\times d_{\text{img}}}, \qquad E^{\text{glob}} \in \mathbb{R}^{L\times d_{\text{img}}},$$

where L is the number of visual tokens and d_{img} is the image feature dimension. The global embedding E^{glob} is broadcast across parts, while E_k^{loc} carries part-specific appearance cues.

We stack the K per-part latent sequences along the batch axis to obtain

$$\mathbf{Z}_0 = \begin{bmatrix} \mathbf{z}_{1,0} \\ \vdots \\ \mathbf{z}_{K,0} \end{bmatrix} \in \mathbb{R}^{(KT)\times D_{\text{lat}}},$$

which is the clean latent representation used in the diffusion process.

8.2. Diffusion Transformer Backbone

The generative backbone is a DiT with L_{DiT} transformer blocks and hidden dimension D (we use $D \gg D_{\text{lat}}$). Before entering the transformer, we project the VAE latents and inject a timestep token:

$$\mathbf{H} = \text{Proj}(\mathbf{Z}_t) \in \mathbb{R}^{(KT)\times D}, \tag{3}$$

$$\mathbf{h}_{\text{time}} = \phi_{\text{time}}(t) \in \mathbb{R}^{1\times D}, \tag{4}$$

$$\tilde{\mathbf{H}} = \big[\, \mathbf{h}_{\text{time}}; \mathbf{H} \,\big] \in \mathbb{R}^{(KT+1)\times D}, \tag{5}$$

where $\mathbf{Z}_t$ is the noisy latent at time t (defined in Sec. 8.5), ϕ_{time} is a sinusoidal+MLP timestep embedding, and $[\,\cdot;\cdot\,]$ denotes concatenation along the token dimension.

Part and position embeddings. To make the model part-aware and robust to part shuffling, we add two learnable embeddings per part:

- A *relative part embedding* $e^{\text{rel}}(k)$ that encodes the index of part k within its object, $k \in \{0,\dots,K-1\}$.
- An *absolute position embedding* $e^{\text{abs}}(p_k)$ that encodes the canonical identity of the corresponding link (e.g., `link_0`, `link_1`, ...).

During data loading, we maintain an array of absolute indices p_k that is preserved under shuffling, so each semantic link always receives a consistent absolute embedding even when the parts are randomly permuted for augmentation. For each token that belongs to part k, we add both embeddings to the hidden state:

$$\tilde{\mathbf{H}} \leftarrow \tilde{\mathbf{H}} + \text{Broadcast}\big(e^{\text{rel}}(k)\big) + \text{Broadcast}\big(e^{\text{abs}}(p_k)\big), \tag{6}$$

where $\text{Broadcast}(\cdot)$ expands a per-part vector to all tokens of that part.

8.3. Multi-Level Attention and Image Conditioning

We interleave three types of attention to combine part-level geometry, local appearance, and global context:

Self-attention. Each transformer block first applies self-attention with rotary positional encoding over the latent tokens:

$$\mathbf{A}^{\text{self}} = \text{softmax}\big(\mathbf{Q}\mathbf{K}^\top/\sqrt{D}\big), \qquad \mathbf{H}' = \mathbf{H} + \mathbf{A}^{\text{self}}\mathbf{V}, \tag{7}$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are the usual query, key, and value projections of $\tilde{\mathbf{H}}$.

Global cross-attention. In a subset of layers indexed by $\mathcal{L}_{\text{glob}}$, we use cross-attention on the global image embedding E^{glob} to inject holistic object context shared across all parts:

$$\mathbf{A}^{\text{glob}} = \text{softmax}\big(\mathbf{Q}\mathbf{K}^{\text{glob}\,\top}/\sqrt{D}\big), \qquad \mathbf{H}'' = \mathbf{H}' + \mathbf{A}^{\text{glob}}\mathbf{V}^{\text{glob}}, \tag{8}$$

where $\mathbf{K}^{\text{glob}}, \mathbf{V}^{\text{glob}}$ are projections of E^{glob}.

Local cross-attention. In the remaining layers, indexed by $\mathcal{L}_{\text{loc}}$, we instead use cross-attention to the part-specific embeddings E_k^{loc}. Tokens belonging to part k only attend to the visual features of that part:

$$\mathbf{A}_k^{\text{loc}} = \text{softmax}\big(\mathbf{Q}_k\mathbf{K}_k^{\text{loc}\,\top}/\sqrt{D}\big), \qquad \mathbf{H}_k'' = \mathbf{H}_k' + \mathbf{A}_k^{\text{loc}}\mathbf{V}_k^{\text{loc}}. \tag{9}$$

Alternating global and local layers encourages the model to respect both global shape consistency and part-level details driven by the per-part images.

8.4. Hierarchy Attention Implementation

To explicitly encode the predicted kinematic tree, we adopt a hierarchy attention module operating on parent–child link pairs. Let $\pi : \{1,\dots,K\} \to \{-1,1,\dots,K\}$ denote the parent index map over parts, where $\pi(k) = -1$ for the root. We share this map across all tokens of a part.

We first define the sets of parent and child tokens for a given token u:

$$P(u) = \{v \mid \text{token } v \text{ belongs to the parent part of } u\},$$

$$C(u) = \{v \mid \text{token } v \text{ belongs to a child part of } u\}.$$

Child-to-parent attention. Given latent tokens $\mathbf{Z}$, we compute a child-to-parent attention matrix

$$A_{uv}^{c\to p} = \frac{\exp\left(\mathbf{z}_u \mathbf{z}_v^\top / \sqrt{C}\right) \mathbf{1}[v \in P(u)]}{\sum_{v'} \exp\left(\mathbf{z}_u \mathbf{z}_{v'}^\top / \sqrt{C}\right) \mathbf{1}[v' \in P(u)]}, \tag{10}$$

and update the latent tokens via

$$\mathbf{Z}' = \mathbf{Z} + A^{c\to p}\mathbf{Z}. \tag{11}$$

This step lets each child part aggregate structural context from its parent.

Parent-to-child attention. We then allow parents to read back from their children using

$$A_{uv}^{p\to c} = \frac{\exp\left(\mathbf{z}_u \mathbf{z}_v^\top / \sqrt{C}\right) \mathbf{1}[v \in C(u)]}{\sum_{v'} \exp\left(\mathbf{z}_u \mathbf{z}_{v'}^\top / \sqrt{C}\right) \mathbf{1}[v' \in C(u)]}, \tag{12}$$

and obtain the final hierarchy-aware representation

$$\mathbf{Z}'' = \mathbf{Z}' + A^{p\to c}\mathbf{Z}'. \tag{13}$$

In practice, we implement this module as a separate transformer block that takes the current hidden states and the batched parent indices as input. To avoid cross-sample contamination, we offset parent indices per object and explicitly validate that no part attends to parents outside its own object. For efficiency, we compute aggregated parent/child features using scatter-add operations rather than explicit Python loops.

8.5. Rectified Flow Training Objective

We train the DiT backbone using Rectified Flow matching. For each object, we draw an independent base latent per part,

$$\mathbf{z}_{k,1} \sim \mathcal{N}(0, \mathbf{I}),$$

and define the clean and base stacks

$$\mathbf{Z}_0 = \begin{bmatrix} \mathbf{z}_{1,0} \\ \vdots \\ \mathbf{z}_{K,0} \end{bmatrix}, \qquad \mathbf{Z}_1 = \begin{bmatrix} \mathbf{z}_{1,1} \\ \vdots \\ \mathbf{z}_{K,1} \end{bmatrix}.$$

A shared timestep $t \in (0, 1)$ is sampled per object from a non-uniform logit-normal density $\rho(t)$ that emphasizes informative ranges. The interpolated latent is

$$\mathbf{X}_t = (1-t)\mathbf{Z}_0 + t\mathbf{Z}_1, \tag{14}$$

and the target velocity field is time-invariant,

$$\mathbf{U}^\star = \mathbf{Z}_0 - \mathbf{Z}_1. \tag{15}$$

Let C collect all conditioning signals: the global image embedding c_{global}, the per-part image embeddings $\{c_k^{\text{part}}\}$, and the absolute indices $\{p_k\}$. The DiT predicts a velocity field $V_\theta(\mathbf{X}_t, C, t)$ over all tokens. With per-part weights α_k, timestep density $\rho(t)$, and a scalar reweighting function $w(t)$, we optimize

$$\begin{aligned}\mathcal{L}_{\text{RF}} &= \mathbb{E}_{t,\,\mathbf{Z}_0,\,\mathbf{Z}_1,\,C}\Big[w(t)\sum_{k=1}^{K}\alpha_k \big\|v_\theta(\mathbf{x}_k(t), C, t) - \mathbf{u}_k^\star\big\|_2^2\Big] \\ &= \mathbb{E}\Big[w(t)\big\|V_\theta(\mathbf{X}_t, C, t) - \mathbf{U}^\star\big\|_F^2\Big].\end{aligned} \tag{16}$$

Here, we use a reverse-velocity parameterization consistent with our sampler, and share t across all parts of the same object to keep the noise level aligned within an articulated asset.

8.6. Classifier-Free Guidance

We adopt classifier-free guidance at the object level. During training, with probability p_{cfg} we drop all image conditions for an entire object and replace both E_k^{loc} and E^{glob} by learned "null" embeddings, yielding an unconditional branch. The DiT is thus trained on a mixture of conditional and unconditional samples.

At inference time, we evaluate the network twice for each diffusion step: once with all conditions dropped (V_θ^{uncond}) and once with full conditioning (V_θ^{cond}). The guided prediction is

$$V_\theta^{\text{guid}} = V_\theta^{\text{uncond}} + s_{\text{cfg}}\big(V_\theta^{\text{cond}} - V_\theta^{\text{uncond}}\big), \tag{17}$$

where s_{cfg} is the guidance scale. This formulation lets us trade off fidelity to the input image against sample diversity while preserving multi-part consistency.

9. Kinematic-Part Mesh Merging

We unify the raw PartNet-Mobility meshes into a single canonical GLB file per object, where each URDF link corresponds to exactly one mesh. In the original dataset, a single kinematic link may reference multiple OBJ files (e.g., different materials or subcomponents). For our purposes, a *kinematic part* is defined at the link level, so each link must appear as one rigid mesh that moves as a unit under the URDF articulation. We therefore merge all OBJs associated with the same link into a single geometry and export them.

9.1. URDF-Driven Link Grouping

For each object, we start from the original `mobility.urdf`. We parse all `<link>` elements and collect their associated mesh filenames from the `<visual>` blocks:

- We treat every non-base link (i.e., links whose name is not `base`) as a candidate kinematic part.
- For each such link, we traverse all `<visual>` elements and extract the `filename` attribute of the nested `<mesh>` tag.

- The original paths typically resemble `textured_objs/original-50.obj`; we reduce them to basenames (e.g., `original-50.obj`) and assume the corresponding geometry resides in the `textured_objs/` folder.

This yields a mapping

$$\mathcal{G} : \text{link name} \mapsto \{\text{OBJ filenames}\},$$

which defines how raw meshes should be grouped into kinematic parts. If a link has no valid mesh entries in the URDF, it is skipped.

9.2. Per-Link Geometry Cleaning and Merging

We load and merge meshes on a per-link basis using `trimesh`. For each link ℓ with mesh file list $\mathcal{G}(\ell)$:

1. We attempt to load each OBJ file from `textured_objs/`. Files that cannot be loaded (missing or malformed) are skipped with a warning.
2. Each successful load is converted to a pure-geometry `Trimesh`:
   - If the loader returns a single `Trimesh`, we create a new mesh with the same vertices and faces (with `process=False` to avoid automatic repairs) and discard all materials.
   - If the loader returns a `Scene`, we iterate over its geometries and extract each `Trimesh` in the same way.
   - In both cases, we assign a uniform gray face color `[128, 128, 128, 255]` to decouple our geometry pipeline from the original textures. Later, we re-render per-part images using our own lighting and camera setup (Sec. 11), so we do not rely on baked-in materials.
3. We collect all such geometry-only meshes for link ℓ into a list. If the list is empty, the link is effectively dropped.
4. If there is exactly one mesh, we keep it as-is. If there are multiple, we merge them by explicit vertex-face concatenation:

$$V_{\text{all}} = \begin{bmatrix} V_1 \\ \vdots \\ V_m \end{bmatrix}, \tag{18}$$

$$F_{\text{all}} = \begin{bmatrix} F_1 \\ F_2 + |V_1| \\ \vdots \\ F_m + \sum_{i=1}^{m-1} |V_i| \end{bmatrix}, \tag{19}$$

where V_i and F_i are the vertices and faces of the i-th submesh and $|\cdot|$ denotes the number of vertices. We then create a single `Trimesh` from $(V_{\text{all}}, F_{\text{all}})$ with a uniform gray color.

This procedure yields one rigid mesh per URDF link, with all subcomponents fused into the same local frame. We intentionally do not perform heavy processing at this stage (e.g., no automatic repairs or decimation) to preserve the original geometry as much as possible; watertight voxelization and cleanup are deferred to the next stage (Sec. 10).

10. Watertight Part Preprocessing

Our network operates on per-part surface samples extracted from mesh geometry (Sec. 8). In practice, raw CAD / reconstruction data often contain small gaps, self-intersections, or open boundaries, which lead to unstable surface sampling and inconsistent volumes. Before training, we therefore convert every part mesh into a watertight surface via voxelization and marching cubes, combined with thin-part guards and aggressive post-cleaning. Unless otherwise stated, all experiments use a target voxel resolution of R=200 along the largest object axis.

10.1. Per-part Voxelization and Pitch Selection

Given a per-part mesh with axis-aligned bounding box extents

$$\mathbf{e} = (e_x, e_y, e_z) \in \mathbb{R}^3_{>0},$$

we first choose a voxel pitch Δ that balances three goals: (1) roughly R=200 voxels along the largest axis, (2) at least $N_{\min}$ voxels across the thinnest axis to avoid losing doors and sheet-like structures, and (3) a soft memory budget for the voxel grid.

Concretely, we define two candidate pitches

$$\Delta_{\text{res}} = \frac{\max(\mathbf{e})}{R}, \qquad \Delta_{\text{thin}} = \frac{\min(\mathbf{e})}{N_{\min}}, \tag{20}$$

where $N_{\min}$ is a small integer (we use $N_{\min}$=3 in all experiments). The first term enforces the user-specified resolution along the largest axis; the second guarantees a minimum number of cells across the thinnest axis. We adopt the more conservative (higher-resolution) pitch

$$\Delta = \min(\Delta_{\text{res}}, \Delta_{\text{thin}}). \tag{21}$$

Given Δ, the approximate grid shape is

$$\mathbf{n} = (n_x, n_y, n_z) = \left(\left\lceil \frac{e_x}{\Delta} \right\rceil, \left\lceil \frac{e_y}{\Delta} \right\rceil, \left\lceil \frac{e_z}{\Delta} \right\rceil \right), \tag{22}$$

and we estimate memory usage as

$$M_{\text{vox}} \approx n_x n_y n_z \text{ bytes}. \tag{23}$$

If M_{vox} exceeds a rough cap (400 MB in our implementation), we relax the pitch by a global scale factor so that M_{vox} fits into the budget. This procedure yields a per-part voxel grid that is fine enough for thin structures but remains tractable even for large assets.

10.2. Closed Occupancy and Marching Cubes

With the chosen pitch Δ, we voxelize each part mesh into an occupancy grid by calling the `voxelized` interface from `trimesh`. To enforce watertightness, we explicitly convert the occupancy grid into a solid by filling interior and small holes:

1. Voxelize the original mesh at pitch Δ.
2. Apply a fill operation to propagate occupancy to the interior, producing a fully closed solid voxel grid.

We then extract an isosurface using marching cubes on this filled grid. Because the occupancy is explicitly closed before marching cubes, the resulting surface is a watertight shell up to discretization artifacts.

10.3. Axis-wise Rescaling Back to Original Extents

The marching-cubes shell lives in the coordinate frame of the voxel grid, and its bounding box extents can deviate slightly from the original mesh due to discretization. To avoid systematic shrinkage of thin parts, we apply an anisotropic rescaling that exactly matches the original axis-aligned extents.

Let $\mathbf{e}^{\text{orig}}$ denote the original mesh extents and $\mathbf{e}^{\text{shell}}$ the extents of the marching-cubes shell. We compute a per-axis scale

$$\begin{aligned} \mathbf{s} &= (s_x, s_y, s_z), \\ s_i &= \frac{e_i^{\text{orig}}}{\max\left(e_i^{\text{shell}}, \varepsilon\right)}, \quad i \in \{x, y, z\}. \end{aligned} \tag{24}$$

with a small ε to avoid division by zero. We then:

1. Translate the shell so that its bounding-box center is at the origin.
2. Apply the diagonal scale matrix $\mathrm{diag}(\mathbf{s})$.
3. Translate back so that the shell is centered at the original mesh center.

This axis-wise rescale ensures that each reconstructed part exactly matches the original size along all three axes, preserving joint clearances and articulated contact patterns.

10.4. Robust Mesh Cleanup and Early Exit for Watertight Parts

After rescaling, we run a dedicated cleanup routine to remove numerical artifacts introduced by voxelization:

- **Vertex welding.** We merge vertices within a small tolerance (we use a weld radius of 10^{-6} in world units) to eliminate near-degenerate triangles.
- **Degenerate and duplicate faces.** We repeatedly remove duplicate and zero-area faces, then drop unreferenced vertices and fix normals.
- **Largest connected component.** To discard floating fragments, we split the mesh into connected components and retain only the largest one (preferring components with more faces and, when available, larger volume).
- **Optional decimation.** For extremely dense outputs, we optionally apply quadratic decimation toward a target face count. For the experiments reported in this paper, we disable decimation (target faces set to zero) to avoid erasing small but semantically important details.

For parts that are already watertight and reasonably clean in the raw data, we take a conservative path: we skip voxelization and only apply the lightweight cleanup routine above. This preserves the original high-frequency geometry while still enforcing a consistent, watertight representation for problematic parts.

11. Articulation-Aware Data Augmentation

After constructing watertight per-part meshes and canonical surface samples (Sec. 10), we augment the dataset by rendering each articulated object at additional joint configurations while keeping the underlying geometry and part decomposition fixed. Concretely, for every `mobility.urdf` with at least one revolute joint, we synthesize two extra variants:

- a *max* pose, where all revolute joints are placed at their upper joint limits; and
- a *mid* pose, where all revolute joints are placed at half of their maximum angle.

Each variant reuses the same watertight mesh and part-level surface samples but provides a new RGB rendering at a different articulation state. This encourages the model to become robust to joint motion while still reconstructing a consistent canonical geometry for each object.:contentReference[oaicite:0]index=0

11.1. URDF-Based Multi-Joint Forward Kinematics

We rely on the original PartNet-Mobility URDF to recover kinematic structure. For each object we parse:

- all links $\mathcal{L}$, each with one or more `<visual>` blocks that specify a mesh filename, an origin translation $\mathbf{t}^{\text{vis}} \in \mathbb{R}^3$, and an RPY rotation $\mathbf{r}^{\text{vis}} \in \mathbb{R}^3$;
- all joints $\mathcal{J}$, each connecting a parent link p and child link c with a joint origin $(\mathbf{t}^{\text{joint}}, \mathbf{r}^{\text{joint}})$, an axis $\mathbf{a} \in \mathbb{R}^3$, and a type (we only use `revolute` joints for augmentation).

For every link $\ell \in \mathcal{L}$, we choose a representative visual transform

$$T_\ell^{\text{vis}} \in \mathrm{SE}(3)$$

by preferring GLB-based visuals when present and falling back to the first mesh visual otherwise. This defines the transform from the link frame to a canonical visual frame for that link.

We then perform visual-space forward kinematics over the kinematic tree. Root links are detected as links that never appear as a joint child. For each root link r, we initialize its visual transform in world coordinates as

$$T_{\text{vis}}^{\text{world}}(r) = T_r^{\text{vis}}, \qquad T_{\text{link}}^{\text{world}}(r) = I_4,$$

so that its visual frame coincides with its world frame by construction.

For a joint $j \in \mathcal{J}$ with parent link p and child link c, we form:

$$T_j^{\text{joint}} = \text{SE3}\big(R(\mathbf{r}^{\text{joint}}),\, \mathbf{t}^{\text{joint}}\big), \tag{25}$$

$$T_j^{\text{rot}}(\theta_j) = \text{SE3}\big(R_{\text{axis}}(\mathbf{a}, \theta_j),\, \mathbf{0}\big), \tag{26}$$

where $R(\cdot)$ converts RPY to a rotation matrix and $R_{\text{axis}}(\mathbf{a}, \theta_j)$ is the Rodrigues rotation about axis $\mathbf{a}$ with angle θ_j.

We propagate transforms along the kinematic tree in topological order using:

$$T_{\text{link}}^{\text{world}}(c) = T_{\text{vis}}^{\text{world}}(p)\; T_j^{\text{joint}}\; T_j^{\text{rot}}(\theta_j), \tag{27}$$

$$T_{\text{vis}}^{\text{world}}(c) = T_{\text{link}}^{\text{world}}(c)\; T_c^{\text{vis}}. \tag{28}$$

This visual-space formulation matches the single-joint rendering script used during initial preprocessing and extends it to arbitrary joint depth without changing the relative placement of visual frames.

To align the multi-joint scene with the canonical coordinate system used in the main dataset, we pick a reference link $r^\star$ using a simple heuristic (prefer a fixed child of the base link; otherwise use the parent of the first revolute joint). Let $T_{\text{vis}}^{\text{world}}(r^\star)$ denote the computed visual transform and $T_{r^\star}^{\text{vis}}$ be its canonical visual transform. We apply a global similarity transform

$$S = T_{r^\star}^{\text{vis}}\big(T_{\text{vis}}^{\text{world}}(r^\star)\big)^{-1},$$

and left-multiply all link and visual transforms by S. This guarantees that the reference link's visual frame exactly matches the canonical preprocessing pipeline, while preserving all relative joint poses.

Finally, we instantiate a `trimesh.Scene` by loading every mesh referenced in the link visuals and applying the corresponding world transform

$$T^{\text{world}} = T_{\text{link}}^{\text{world}}(\ell)\; T_{\ell,\text{component}}^{\text{vis}}$$

to each visual component. The result is a normalized, articulated scene in the same global coordinate system as the watertight mesh used for sampling.

11.2. Sampling Joint Angles: Reference, Mid, and Max Poses

For each revolute joint $j \in \mathcal{J}$, we read its upper limit $\theta_j^{\max}$ from the URDF `<limit>` tag when available, defaulting to $\theta_j^{\max} = \pi$ (180°) if no limit is specified. We then define:

$$\theta_j^{\text{ref}} = 0, \qquad \theta_j^{\text{mid}} = \frac{1}{2}\,\theta_j^{\max}, \qquad \theta_j^{\text{max}} = \theta_j^{\max}.$$

The original preprocessed dataset already contains the reference pose with all joints set to θ_j^{ref}. Our augmentation script constructs two additional pose families:

Max pose (`_max`). All revolute joints are set to θ_j^{max}, simultaneously driving each joint to its mechanically allowed extreme. This exposes the model to highly opened drawers, doors, and other articulated components.

Mid pose (`_mid`). All revolute joints are set to θ_j^{mid}, producing a configuration between the closed and fully open states. This captures typical everyday articulations without extreme self-occlusion.

Objects without any revolute joints are left unchanged and contribute only their reference pose.

11.3. Consistent Normalization and Rendering Settings

To ensure consistent scale and camera framing across all poses, we normalize each object once using the reference pose. Given the reference scene $\mathcal{S}_{\text{ref}}$ (all joints at θ_j^{ref}), we compute its axis-aligned bounding box centroid $\mathbf{c}$ and extents $\mathbf{e}$. We then define:

$$\mathbf{t}_{\text{norm}} = -\mathbf{c}, \tag{29}$$

$$s_{\text{norm}} = \frac{2}{\max(e_x, e_y, e_z)}. \tag{30}$$

We apply this normalization to all variants (reference, mid, max), so every object is centered at the origin and fits inside a unit cube regardless of articulation. Using a fixed normalization per object avoids small pose-induced scale changes and keeps camera parameters strictly comparable across augmentations.

For rendering, we reuse the same camera and lighting configuration as the main preprocessing pipeline:

- camera placed on a sphere of fixed radius (4 units) around the origin;
- field of view of 40°;
- high-resolution images at 2048×2048 pixels;
- an environment light setup with multiple evenly spaced directional lights (36 directions) and fixed intensity.

The augmentation script calls a shared `render_single_view` routine with these settings, so augmented images are visually indistinguishable from the original dataset except for joint angles.