Supplement to "3D Neural Embedding Likelihood: Probabilistic Inverse Graphics for Robust 6D Pose Estimation"

Guangyao Zhou* Google DeepMind stannis@google.com
Nishad Gothoskar* MIT nishad@mit.edu
Lirui Wang MIT liruiw@mit.edu
Joshua B. Tenenbaum MIT jbt@mit.edu
Dan Gutfreund MIT-IBM Watson AI Lab dgutfre@us.ibm.com
Miguel Lázaro-Gredilla Google DeepMind lazarogredilla@google.com
Dileep George Google DeepMind dileepgeorge@google.com
Vikash K. Mansinghka MIT vkm@mit.edu

A. Review of SurfEMB

Our noise model on RGB information build on SurfEMB [1] which learns neural embeddings to establish dense 2D-3D correspondences via contrastive learning. For each object class $t \in \{1, \cdots, M\}$, SurfEMB learns a pair of neural networks: the *query model* $f_t : \{0, \cdots, 255\}^{H\times W\times 3} \mapsto \mathbb{R}^{H\times W\times E}$ and the *key model* $g_t : \mathbb{R}^3 \mapsto \mathbb{R}^E$. The *query model* transforms an observed RGB image $\mathbf{I}$ into query embeddings $\mathbf{Q}^t$, while the *key model* transforms a rendered object coordinate image $\tilde{\mathbf{X}}$ into a set of key embeddings.

For a given 2D pixel location on the observed image with query embedding $\mathbf{q}$, SurfEMB specifies a surface correspondence distribution $\mathbb{P}_{\text{RGB}}(g_t(\tilde{x})|\mathbf{q}, t) \propto \exp(\mathbf{q}^T g_t(\tilde{x}))$ for each object class t. To normalize this surface correspondence distribution, for each object class t, we subsample uniformly across the object's surface to get a set Z_t of surface 3D coordinates in object frame. Then, given a pixel with query embedding $\mathbf{q}$, we calculate the probability that this pixel corresponds to a surface point $\tilde{\mathbf{x}} \in Z_t$ on object class t as:

$$\mathbb{P}_{\text{RGB}}(g_t(\tilde{\mathbf{x}})|\mathbf{q}, Z_t, t) = \frac{\exp(\mathbf{q}^T g_t(\tilde{\mathbf{x}}))}{\sum_{\mathbf{x}\in Z_t} \exp(\mathbf{q}^T g_t(\mathbf{x}))} \tag{1}$$

B. More details on the energy-based formulation of 3DNEL

B.1. Existence of the normalization constant for the energy-based formulation

Since we are working with an energy-based formulation (Equation (1)), to make the probability distribution properly defined we need to make sure the normalization constant, i.e. the sum of the energy function over all $\mathbf{I}$ and $\mathbf{C}$

$$\sum_{\mathbf{I}} \int_{\mathbf{C}} \prod_{\mathbf{c}} \left(\epsilon \mathbb{P}_{\text{BG}}(\mathbf{c}; B) + \frac{1-\epsilon}{\tilde{K}} \sum_{\tilde{\mathbf{c}}:\tilde{s}>0} \mathbb{P}_{\text{depth}}(\mathbf{c}|\tilde{\mathbf{c}}; r) \mathbb{P}_{\text{RGB}}(g_{\tilde{s}}(\tilde{\mathbf{x}})|\mathbf{q}^{\tilde{s}}, \tilde{s}) \right)$$

is finite and well-defined. For RGB images of size $H \times W$, since each pixel has only 256 values, there are at most $256^{H\times W\times 3}$ RGB images of size $H \times W$ which is a finite number. Since the value of the energy function is less than 1 for any given $\mathbf{I}$ and $\mathbf{C}$, summing over a finite number of $\mathbf{I}$ and integrating over a bounded region for $\mathbf{C}$ gives us a finite normalization constant, making the probability distribution well-defined.

*Equal contribution

B.2. JAX-based implementation of 3DNEL evaluation given rendering outputs

Given rendering outputs from OpenGL, we use JAX to develop a 3DNEL evaluation implementation that can run efficiently on modern GPUs. The implementation can be easily combined with `jax.vmap` to support 3DNEL evaluation of hundreds of 3D scene descriptions in parallel.

```
# Copyright 2023 DeepMind Technologies Limited
# Copyright 2023 Massachusetts Institute of Technology (M.I.T.)
# SPDX-License-Identifier: Apache-2.0
@functools.partial(jax.jit, static_argnames="filter_shape")
def neural_embedding_likelihood(
    data_xyz: jnp.ndarray,
    query_embeddings: jnp.ndarray,
    log_normalizers: jnp.ndarray,
    model_xyz: jnp.ndarray,
    key_embeddings: jnp.ndarray,
    model_mask: jnp.ndarray,
    obj_ids: jnp.ndarray,
    data_mask: jnp.ndarray,
    r: float,
    p_background: float,
    p_foreground: float,
    filter_shape: Tuple[int, int],
):
    """
    Args:
        data_xyz: Array of shape (H, W, 3). Observed point cloud organized as an image.
        query_embeddings: Array of shape (H, W, n_objs, d).
            Query embeddings for each observed pixel using models from different objects.
        log_normalizers: Array of shape (H, W, n_objs).
            The log normalizers for each pixel given each object model
        model_xyz: Array of shape (H, W, 3). Rendered point cloud organized as an image.
        key_embeddings: Array of shape (H, W, d). Key embeddings organized as an image.
        model_mask: Array of shape (H, W). Mask indicating relevant pixels from rendering.
        obj_ids: Array of shape (H, W). The object id of each pixel.
        data_mask: Array of shape (H, W). Mask indicating the relevant set of pixels.
        r: Radius of the ball.
        p_background: background probability.
        p_foreground: foreground probability.
        filter_shape: used to restrict likelihood evaluation to a 2D neighborhood.
    """
    obj_ids = jnp.round(obj_ids).astype(jnp.int32)
    padding = [
        (filter_shape[ii] // 2, filter_shape[ii] - filter_shape[ii] // 2 - 1)
        for ii in range(len(filter_shape))
    ]
    model_xyz_padded = jnp.pad(model_xyz, pad_width=padding + [(0, 0)])
    key_embeddings_padded = jnp.pad(key_embeddings, pad_width=padding + [(0, 0)])
    model_mask_padded = jnp.pad(model_mask, pad_width=padding)
    obj_ids_padded = jnp.pad(obj_ids, pad_width=padding)

    @functools.partial(
        jnp.vectorize,
        signature='(m),(n),(o,d),(o)->()',
    )
    def log_likelihood_for_pixel(
        ij: jnp.ndarray,
        data_xyz_for_pixel: jnp.ndarray,
        query_embeddings_for_pixel: jnp.ndarray,
        log_normalizers_for_pixel: jnp.ndarray,
    ):
        """
        Args:
            ij: Array of shape (2,). The i, j index of the pixel.
        """
        model_xyz_patch = jax.lax.dynamic_slice(
            model_xyz_padded,
            jnp.array([ij[0], ij[1], 0]),
            (filter_shape[0], filter_shape[1], 3),
        )
        key_embeddings_patch = jax.lax.dynamic_slice(
            key_embeddings_padded,
            jnp.array([ij[0], ij[1], 0]),
            (filter_shape[0], filter_shape[1], key_embeddings.shape[-1]),
        )
        model_mask_patch = jax.lax.dynamic_slice(model_mask_padded, ij, filter_shape)
        obj_ids_patch = jax.lax.dynamic_slice(obj_ids_padded, ij, filter_shape)
        log_prob_correspondence = (
            jnp.sum(
                query_embeddings_for_pixel[obj_ids_patch] * key_embeddings_patch,
                axis=-1,
            ) - log_normalizers_for_pixel[obj_ids_patch]
        ).ravel()
        distance = jnp.linalg.norm(
            data_xyz_for_pixel - model_xyz_patch, axis=-1
        ).ravel()
        a = jnp.concatenate([jnp.zeros(1), log_prob_correspondence])
        b = jnp.concatenate(
            [
                jnp.array([p_background]),
                jnp.where(
                    jnp.logical_and(distance <= r, model_mask_patch.ravel() > 0),
                    3 * p_foreground / (4 * jnp.pi * r**3),
                    0.0,
                ),
            ]
        )
        log_mixture_prob = logsumexp(a=a, b=b)
        return log_mixture_prob

    log_mixture_prob = log_likelihood_for_pixel(
        jnp.moveaxis(jnp.mgrid[: data_xyz.shape[0], : data_xyz.shape[1]], 0, -1),
        data_xyz,
        query_embeddings,
        log_normalizers,
    )
    return jnp.sum(jnp.where(data_mask, log_mixture_prob, 0.0))
```

C. Details on coarse enumerative pose hypotheses generation

C.1. Formal description of the coarse enumerative pose hypotheses generation process

We develop a novel spherical voting procedure and a heuristic scoring using the query embeddings and observed point cloud image $\mathbf{C}$ defined in Section 3.2, and use them in an enumerative procedure to efficiently generate pose hypotheses. We use the object center and n_k points sampled using farthest point sampling from the object surface as our keypoints, and discretize the camera frame space into a $L_x \times L_y \times L_z$ voxel grid .

For a given keypoint, our spherical voting procedure aggregates information from the entire image to score how likely the keypoint is present at different voxel locations, and stores the scores in a voxel grid. Figure 1(a) visualizes spherical voting for the center x^* of the mug object: for pixel location (i,j) with camera frame coordinate $\mathbf{c} \in \mathbb{R}^3$ and query embedding $\mathbf{q} \in \mathbb{R}^E$, we identify its most likely corresponding point on the mug surface $x = \arg\max_{\tilde{x}} \mathbb{P}_{\text{RGB}}(g_t(\tilde{x})|\mathbf{q}, t)$, calculate the distance $r_x = ||x - x^*||_2$ from x to x^* in the object frame, and cast votes [3] with weight $p_{i,j} = \max_{\tilde{x}} \mathbb{P}_{\text{RGB}}(g_t(\tilde{x})|\mathbf{q}, t)$ towards all points on a sphere of radius r_x centered at $\mathbf{c}$. Figure 1(b) visualizes the 20 top-scoring voxels from the voxel grid for the mug center, and Figure 1(c) visualizes the 20 top-scoring voxels from the voxel grids for the n_k keypoints on the mug surface.

We coarsely discretize the object pose space. We reuse the same camera frame space discretization into a voxel grid, and use the $L_x \times L_y \times L_z$ voxel centers to discretize the location space. We use a customized procedure (Appendix C.3) to generate n_r rotations and discretize the rotation space. We use the voxel grid for the object center to identify top-scoring object locations, and score all n_r rotations at these locations. We score a given object pose with the sum of the scores of the voxels the corresponding n_k keypoints fall into. Figure 1(d) visualizes 3 example top pose hypotheses from the enumerative

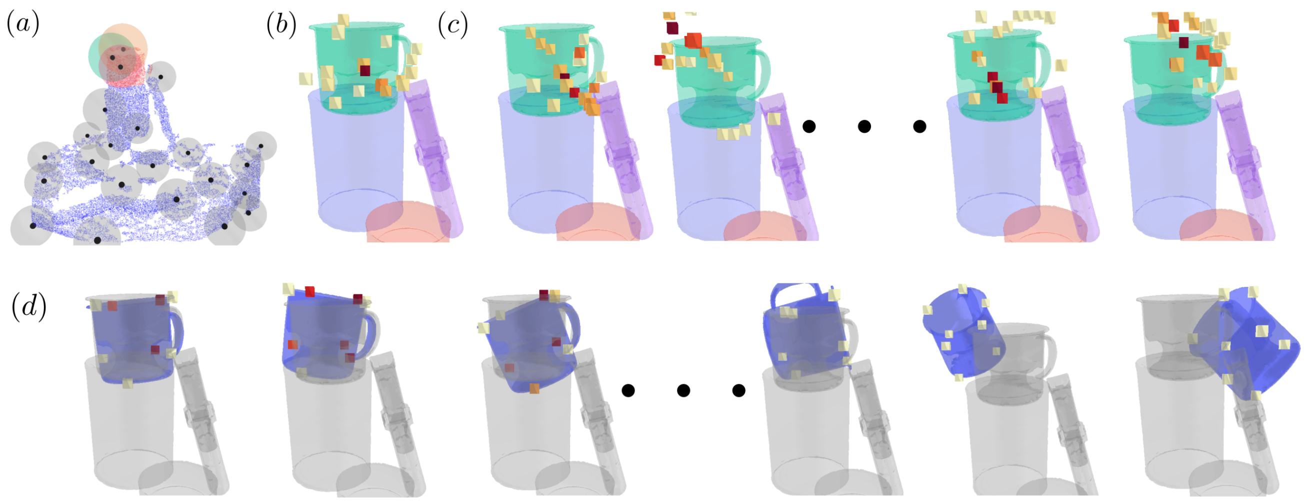

Figure 1. **Coarse Enumerative Pose Hypotheses Generation** We visualize our coarse enumerative pose hypotheses generation process using a mug object as an example. **Figure 1(a)** visualizes spherical voting for identifying the mug center. The red points in the observed point cloud represent points associated with the mug. The 3 colored spheres illustrate how votes from points on the mug combine to identify the mug center. The same procedure can also be used to identify the n_k keypoints on the mug surface. **Figure 1(b)** visualizes 20 top-scoring voxels from the voxel grid associated with the mug center. **Figure 1(c)** visualizes 20 top-scoring voxels from the voxel grids associated with the n_k keypoints. **Figure 1(d)** show how we score pose hypotheses by summing the scores of the voxels the corresponding n_k keypoints fall into. The scores of the poses hypotheses decrease as we go from left to right. In **Figures 1(b) to (d)**, light yellow represents low voxel scores, while dark red represents high voxel scores.

procedure. Figure 1(e) visualizes how poses far away from ground truth get low scores from our heuristic scoring.

In Algorithm 1 we present a detailed description of our coarse enumerative pose hypotheses generation process. In practice we also optionally apply non-max suppression in the `TopPositions` function in the algorithm to make sure the promising positions we identify are well spread out to cover different parts of the image and better represent uncertainty.

C.2. Taichi-based spherical voting

We use Taichi [2] to develop a spherical voting implementation that can run efficiently on modern GPUs.

```
# Copyright 2023 DeepMind Technologies Limited
# Copyright 2023 Massachusetts Institute of Technology (M.I.T.)
# SPDX-License-Identifier: Apache-2.0

@ti.kernel
def taichi_spherical_vote(
    centers: ti.types.ndarray(element_dim=1),
    radiuses: ti.types.ndarray(),
    weights: ti.types.ndarray(),
    voxel_grid: ti.types.ndarray(),
    voxel_grid_start: ti.types.ndarray(element_dim=1),
    voxel_diameter: float,
    multipliers: ti.types.ndarray(),
):
    """
    Args:
        centers: Array of shape (batch_size, n_centers, 3,). Coordinates of the centers of the spheres.
        radiusss: Array of shape (batch_size, n_centers). Radiuses of the spheres.
        weights: Array of shape (batch_size, n_centers,). Weights of votes from the spheres.
        voxel_grid: Array of shape voxel_grid_shape
        voxel_grid_start: Array of shape (3,). Coordinate of the center of voxel (0, 0, 0)
        voxel_diameter: float. Diameter of a voxel.
        multipliers: fixed length-2 1D array with elements 1.0, -1.0
    """
    for voxel in ti.grouped(voxel_grid):
        voxel_grid[voxel] = 0.0

```

Algorithm 1: Coarse Enumerative Pose Hypotheses Generation

```
/* Basic setups                                                                  */
Object-specific: A set of surface points $Z_t$ for object class $t$, query model $f_t$, key model $g_t$, $n_k$ keypoints with object
    frame coordinates $x_1^*, \cdots, x_{n_k}^* \in \mathbb{R}^3$.
Spatial discretization: Camera frame coordinates of the center of the boundary voxel $y \in \mathbb{R}^3$, size of the voxel grid
    $(L_x, L_y, L_z)$, diameter of the voxels $d > 0$.
Orientation discretization: $n_r$ representative orientations $\mathbf{R}_1, \cdots, \mathbf{R}_{n_r} \in \mathbb{SO}(3)$.
Parameters: Number of pose hypotheses to generate $n_p$, number of top positions $n_t$.
────────────────────────────────────────────────────────────────────────────────
/* Coarse Enumerative pose hypotheses generation                                 */
Input: RGB image $\mathbf{I} \in \mathbb{R}^{H\times W\times 3}$, observed point cloud $\mathbf{C} \in \mathbb{R}^{H\times W\times 3}$.
Output: Top scoring pose hypotheses $\mathbf{P}_1^t, \cdots, \mathbf{P}_{n_p}^t \in \mathbb{SE}(3)$.
$\mathbf{Q} \leftarrow f_t(\mathbf{I})$;            // Get query embeddings $\mathbf{Q} \in \mathbb{R}^{H\times W\times E}$ from the RGB image $\mathbf{I}$
$\mathbf{V}^0 \leftarrow$ Voting $(\mathbf{Q}, \mathbf{C}, (0,0,0))$;            // Aggregation for object center $(0,0,0)$
for $i \leftarrow 1$ to $n_k$ do            // Aggregation for $n_k$ keypoints.
    $\mathbf{V}^i \leftarrow$ Voting $(\mathbf{Q}, \mathbf{C}, x_i^*)$;            // $\mathbf{V}^i \in \mathbb{R}^{L_x\times L_y\times L_z}$
/* Identify top positions based on $\mathbf{V}^0$'s largest entires.                */
$l_1, \cdots, l_{n_t} \leftarrow$ TopPositions $(\mathbf{V}^0)$;            // $l_1, \cdots, l_{n_t} \in \mathbb{R}^3$
for $i \leftarrow 1$ to $n_t$, $j \leftarrow 1$ to $n_r$ do
    $s_{i,j} \leftarrow$ Scoring $(l_i, \mathbf{R}_j, \mathbf{V}^1, \cdots, \mathbf{V}^{n_k})$;            // Heuristic pose scoring
$\mathbf{P}_1^t, \cdots, \mathbf{P}_{n_p}^t \leftarrow$ RankByScore $(l_1, \cdots, l_{n_t}, s_{i,j}, i = 1, \cdots, n_t, j = 1, \cdots, n_r)$;
return $\mathbf{P}_1^t, \cdots, \mathbf{P}_{n_p}^t$;
────────────────────────────────────────────────────────────────────────────────
/* Voting and heuristic scoring                                                  */
def Voting $(\mathbf{Q}, \mathbf{C}, x^*)$            // Voting-based evidence aggregation
    $\mathbf{V} \leftarrow \mathbf{0}$;            // Initialize $V \in \mathbb{R}^{L_x\times L_y\times L_z}$ to all 0 array
    for $i \leftarrow 1$ to $H$, $j \leftarrow 1$ to $W$ do
        $x \leftarrow \arg\max_{\tilde{x}} \mathbb{P}_{RGB}(\tilde{x}|Q_{i,j}, Z_t, t)$, $p_{i,j} \leftarrow \max_{\tilde{x}} \mathbb{P}_{RGB}(\tilde{x}|Q_{i,j}, Z_t, t)$;
        for $u \leftarrow 1$ to $L_x$, $v \leftarrow 1$ to $L_y$, $w \leftarrow 1$ to $L_z$ do
            $c \leftarrow (y_1 + (u-1)d, y_2 + (v-1)d, y_3 + (w-1)d)$;
            if $||\mathbf{C}_{i,j} - c||_2 \approx ||x - x^*||_2$ then
                $\mathbf{V}_{u,v,w} = \mathbf{V}_{u,v,w} + p_{i,j}$
    return $\mathbf{V}$
def Scoring $(l, \mathbf{R}, \mathbf{V}^1, \cdots, \mathbf{V}^{n_k})$            // Heuristic pose scoring
    $s \leftarrow 0$;            // Initialize score to 0
    for $i \leftarrow 1$ to $n_k$ do
        $x \leftarrow \mathbf{R}x_i^* + l$;            // Location of $x_i^*$ in world frame for pose $l, \mathbf{R}$
        $(u, v, w) \leftarrow Round[(x - y)/d]$;            // Identify corresponding voxel of $x$
        $s = s + \mathbf{V}_{u,v,w}^i$;            // Heuristic scoring
    return $s$
```

```python
    for ii, jj in centers:
        center_on_voxel_grid = (
            centers[ii, jj] - voxel_grid_start[None]
        ) / voxel_diameter
        center_on_voxel_grid = ti.round(center_on_voxel_grid)
        radius_in_voxels = radiuses[ii, jj] / voxel_diameter + 0.5
        for x in range(ti.ceil(radius_in_voxels)):
            for y in range(ti.ceil(ti.sqrt(radius_in_voxels**2 - x**2))):
                z_range = (
                    ti.ceil(
                        ti.sqrt(
                            ti.max(
                                0.0,
                                (radiuses[ii, jj] / voxel_diameter - 0.5) ** 2
                                - x**2
                                - y**2,
                            )
                        )
                    ),
                    ti.ceil(ti.sqrt(radius_in_voxels**2 - x**2 - y**2)),
                )
                for z in range(z_range[0], z_range[1]):
                    for xx in range(2):
                        if x == 0 and multipliers[xx] < 0:
                            continue

                        x_coord = ti.cast(
                            center_on_voxel_grid[0] + multipliers[xx] * x,
                            ti.i32,
                        )
                        if x_coord < 0 or x_coord >= voxel_grid.shape[1]:
                            continue

                        for yy in range(2):
                            if y == 0 and multipliers[yy] < 0:
                                continue

                            y_coord = ti.cast(
                                center_on_voxel_grid[1] + multipliers[yy] * y,
                                ti.i32,
                            )
                            if y_coord < 0 or y_coord >= voxel_grid.shape[2]:
                                continue

                            for zz in range(2):
                                if z == 0 and multipliers[zz] < 0:
                                    continue

                                z_coord = ti.cast(
                                    center_on_voxel_grid[2] + multipliers[zz] * z,
                                    ti.i32,
                                )
                                if z_coord < 0 or z_coord >= voxel_grid.shape[3]:
                                    continue

                                ti.atomic_add(
                                    voxel_grid[ii, x_coord, y_coord, z_coord],
                                    weights[ii, jj],
                                )
```

C.3. Discretizing the rotation space

We discretize $\mathbb{SO}(3)$ into 6400 representative orientations. We generate these orientations by first picking 200 points roughly uniformly on the unit sphere using the Fibonacci sphere. The 6400 representative orientations are genearted by first rotating the axis $(0, 0, 1.0)$ to point to one of the 200 points, followed by one of 32 in-plane rotations around the axis.

D. Details on inference pipeline implementation

D.1. Using additional 2D detection and mask predcition from SurfEMB

In the best performing setup, we leverage the same 2D detector used in SurfEMB as part of the pose hypotheses generation process. Although our spherical voting procedure can robustly aggregate information from the entire image to generate pose hypotheses, as demonstrated by the competitive performance of *3DNEL MSIGP (No 2D detection)* (Abalations in Table 1), in practice the query embedding images for many objects are very noisy and tend to hurt performance. Empirically, we observe

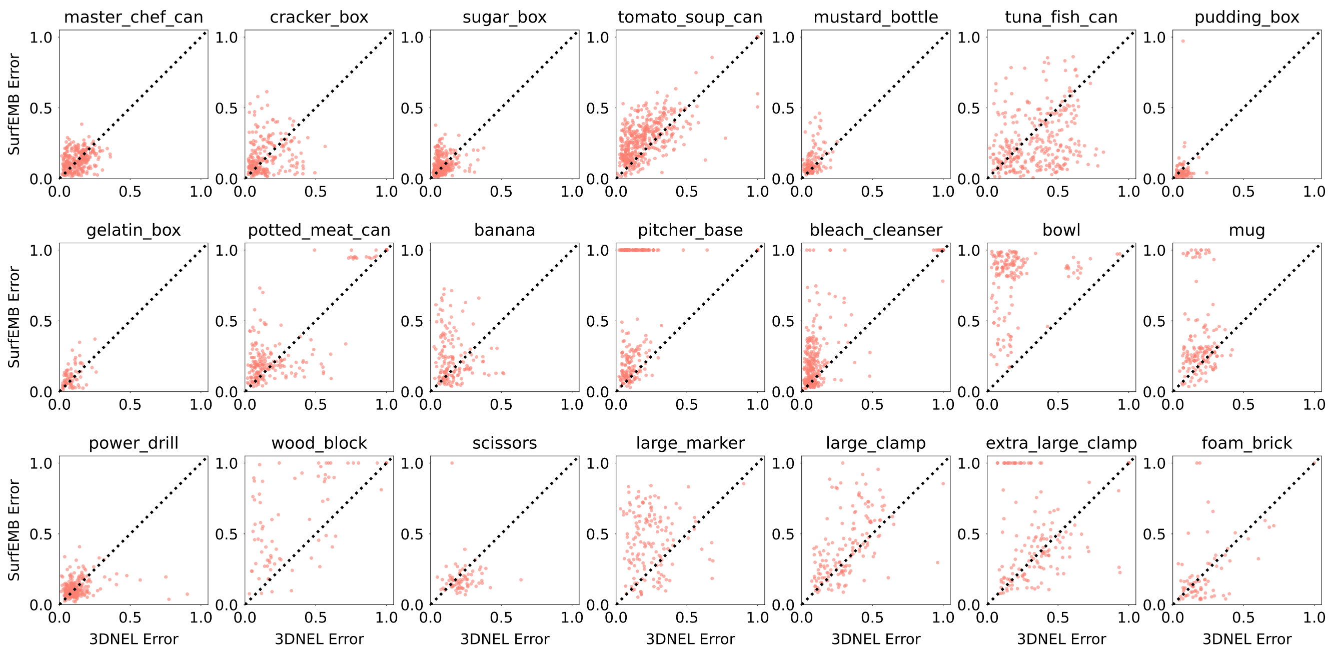

Figure 2. We compare the prediction error (using the VSD error metric) of SurfEMB and 3DNEL MSIGP across all 4123 object instances in the YCB-V test dataset, with each instance represented as a point on this scatter plot. In the main text Figure 3(a), we show the scatter plot across all objects. Here, we show the results per object.

that by additionally using 2D detections, we can focus the spherical voting process on regions of the observed image that is likely relevant for the objects, and further improve performance.

For each object class in the scene, there can be multiple 2D detections. For each 2D detection, we do spherical voting just within the detector crop and generate 80 pose hypotheses per detector crop. When there is a missing 2D detection, we obtain the query embeddings by upsampling the input RGB image by 1.5x, and do spherical voting on the whole image. In such cases, when we identify top positions, we additionally do non-max suppression with a filter size of 10 to spread the top-scoring positions out. For each such top-scoring position we identify top 2 orientation, and we consider all top-scoring positions and generate in total 30 pose hypotheses.

D.2. Implementation details and hyperparameters

We use OpenGL for rendering, use Taichi for spherical voting, and use JAX for 3DNEL evaluation. We refer the readers to the attached Python source code for a complete implementation of our pipeline.

As we describe in the main text, we pick hyperparameters by visually inspecting detection results on a small number of real training images that are outside the test set.

We select $n_k = 8$ keypoints from the surface of each object class, and use $y = (-350.0, -210.0, 530.0), L_x = 129, L_y = 87, L_z = 168$ and $d = 5.0$ to make the voxel grids large enough to cover all the keypoints that can be present in the camera frame. Here the units for the values in y and d are mm.

We use $r = 5.0$ in evaluating 3DNEL for all our experiments. When identifying points in the rendered point cloud $\tilde{\mathbf{C}}$ (organized as an $H \times W \times 3$ image) that is within distance r from a point (i, j) in the observed point cloud $\mathbf{C}$, to further speed up 3DNEL evaluation, we only look at the points from the patch of size $(10, 10)$ centered at (i, j).

In Figure 3 we include additional visualizations of SurfEMB and 3DNEL MSIGP predictions on YCB-V test images.

E. Additional robustness results

In Figure 2 we include additional robustness results.

F. Additional visualizations of SurfEMB and 3DNEL MSIGP predictions on YCB-V test images

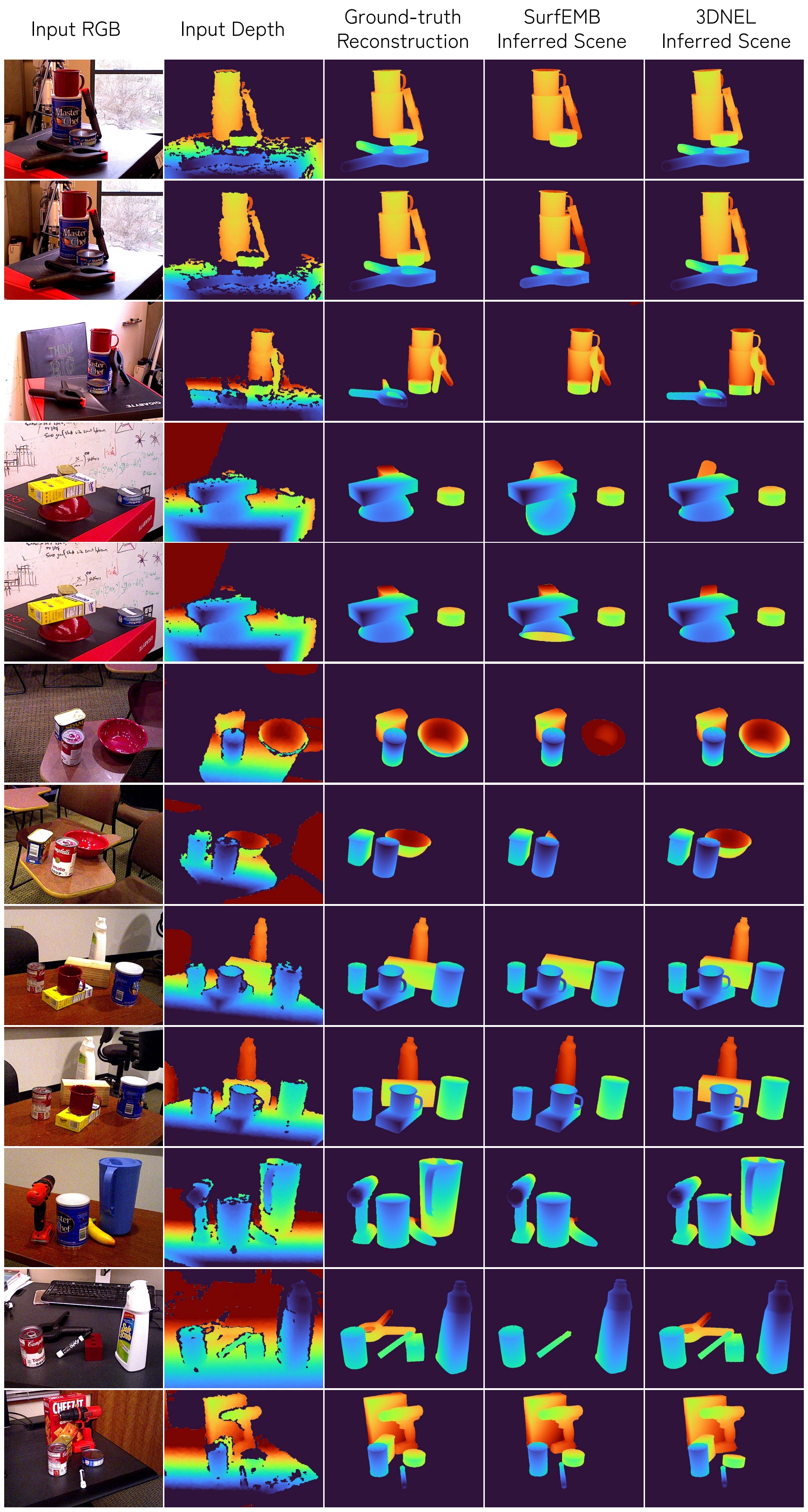

Figure 3. Additional visualizations of SurfEMB and 3DNEL MSIGP predictions on YCB-V test images

References

[1] Rasmus Laurvig Haugaard and Anders Glent Buch. Surfemb: Dense and continuous correspondence distributions for object pose estimation with learnt surface embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6749–6758, 2022. 1

[2] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):201, 2019. 4

[3] Charles R Qi, Or Litany, Kaiming He, and Leonidas J Guibas. Deep hough voting for 3d object detection in point clouds. In *proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9277–9286, 2019. 3