16-820: Advanced Computer Vision - HW6 - 2024

Introduction

In this homework we will work with the state-of-the-art foundation model for segmentation, SAM - Paper. Foundation models are large deep learning neural networks that are trained on massive datasets. SAM and other segmentation models use input prompts such as a box around an object to generate a mask of just the object. We will learn how to run off-the-shelf deep learning models and use them in your work. We will use 2D segmentation masks, combined with camera geometry and depth, to arrive at dense 3D point clouds from 2D segmentations. The steps you will implement are:

- Run SAM on a single image from the dataset.
- Project the mask to 3D points in world coordinates.
- For all the unseen views, do:
    1. Project all points in world coordinates to the image frame.
    2. Automatically generate a new input to SAM and run SAM
    3. Project new mask to world coordinates and append to existing coordinates.
- Filter the point cloud using an off-the-shelf filtering approach.

Homework introduction video here. To submit this homework to gradescope, please submit **your code and the output of your code**. E.g., for Q4 show the function you implemented and the visualizations created in the for loop. For Q1.1, give the K matrix you computed.

Instructor: Matthew O'Toole
OUT: November 21st, 2024
DUE: December 6th, 2024
TA's: Nikhil Keetha, Ayush Jain, Yuyao Shi

Definitions

A couple definitions that will hopefully avoid confusion:

These are the existing `frames`:

- Camera frame/OpenCV Camera Frame: This is the reference frame for 3D points with respect to the camera. This is the camera frame discussed in class.
- Blender Camera Frame: This is the camera frame for 3D points used in Blender, the `y` and `z` axes point in opposite direction w.r.t. the OpenCV camera frame.

- World Frame: This is the frame for 3D points with respect to the world origin. This frame differs by a rigid body transformation from any camera frame.
- Image Frame: The **2D points** in the image, e.g., `u`, `v` in range [0,H] and [0,W].

If we refer to a `prompt` we mean the box around an object that is to be segmented, which is the input to SAM.

How to run this homework

We will use deep neural networks which require a cuda-enabled graphics card with at least 12GB of VRAM. The easiest way to get access to this is Google Colab, press the button below to open this homework in Google Colab. You'll find a pretty useful tutorial on how to use Google Colab here.

How to submit this homework

1. You first press run all in the notebook and make sure that all plots come from your code, and are not the plots in the notebook by default.
2. Then export as PDF, for the written version of the homework simply submit this and make sure to select all your results and code for the question.
3. For the code, submit your iPython notebook file (.ipynb). We can use this to check your homework runs and gives the correct output. If we discover your code cannot reproduce the answer submitted in the written part, you will receive zero points for the question.
4. No requirement on filenames.

FAQs

1. Hint: For Q3.1, remember the difference between the image frame (x horizontal), and the coordinates you might get from the mask. E.g., when you retrieve coordinates from the mask, they would not immediate align with the coordinates expected by the intrinsic matrices

2. You should not have to modify the viz_pts_3d function.

3. You should only call filter_points if thresh is not None, e.g. if thresh is not None: (call filter_points)

4. Use another google account, or use Kaggle if you can't connect to Google colab.

5. The function mask2cam should contain an if statement to check if thresh is None. If thresh is None, do not run filter_points().

6. You should not have to change any of the given plotting functions, if you do there is probably an error in your code.
7. In Q4, you should not have to change any code in the main for loop. Only the functions we have separated, i.e., cam2img, keep_dist, filter_for_box, prompt_points_to_box.
8. Filter_for_box should not take in K as input, the input of the function is already in world coordinates
9. You should not add a thresh argument in mask2cam in the for loop in Q4, this is by design.
10. Filtering in filter_for_box should happen in world frame.
11. If you are getting unexpected results in Q4, you might be doing the correction for the Blender frame wrong. Remember, we have OpenCV Frame <-> Blender Frame <-> World frame. The 'transforms' given are Blender Frame <-> World frame, make corrections accordingly.
12. Another common source of error in Q4 is not getting the correct inverse of the transform. Hint: Google 'inverse of rigid body transforms'.
13. To get a pdf for this homework, please follow the following steps:
    - Download the python notebook
    - Open it in jupyter notebook
    - Download as html
    - Save as pdf

Environment Set-up without Google Colab

If you're not running on Google Colab, use the prep_no_colab.sh script to install the right libraries, pull the model checkpoint and download the data. This script was tested on Ubuntu Linux only. After running the script your folder should look something like this:

```
├── images
│   ├── dataset
│   │   ├── train
│   │   ├── test
│   │   ├── val
│   │   ├── transforms_train.json
│   │   ├── transforms_test.json
│   │   ├── transforms_val.json
```

```
├── ckpts
│   ├── sam_vit_h_4b8939.pth
```

Our recommended method for loading iPython notebooks on your local computer is to use a Visual Studio Code plugin, here is a short tutorial on how to do that. Are you having issues setting up your system? Problems with Cuda versions? Use Colab instead, or ask a TA if you really want to use your own compute.

Set-up

Necessary imports and helper functions for displaying points, boxes, and masks.

In [1]:
```
%matplotlib inline

import numpy as np
import torch
import matplotlib.pyplot as plt
import cv2
import sys
import os

print("numpy version: ", np.__version__)
```

```
numpy version:  1.26.4
```

In [2]:
```
def show_mask(mask, ax, random_color=False):
    # This function is used to visualize the mask on the image in a matplotl
    # bool mask: (H, W). True for each pixel that belongs to the object.
    # ax: matplotlib axis
    # random_color: if True, use a random color for the mask. Otherwise, use

    if random_color:
        color = np.concatenate([np.random.random(3), np.array([0.6])], axis=
    else:
        color = np.array([30/255, 144/255, 255/255, 0.6])
    h, w = mask.shape[-2:]
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    ax.imshow(mask_image)

def show_box(box, ax):
    # This function is used to visualize the bounding box on the image in a
    # box: (4,) array. [x0, y0, x1, y1]
    # (x0, y0): top-left corner
    # (x1, y1): bottom-right corner
    # ax: matplotlib axis

    x0, y0 = box[0], box[1]
    w, h = box[2] - box[0], box[3] - box[1]
    ax.add_patch(plt.Rectangle((x0, y0), w, h, edgecolor='green', facecolor=
```

Q1: Loading and Understanding the Dataset [2 pts]

The dataset you'll be working with is a synthetic dataset generated specifically for this homework. We used the free and open-source 3D graphics software tool Blender to render images from 100 different poses. You will have access to the following data:

- The intrinsics parameters, constant for all images. The dataset was rendered using a pinhole camera model without distortion. Therefore, the intrinsics can be captured using camera matrix K.
- The extrinsics, as a [100 x 4 x 4] array. Each [4 x 4] matrix gives the cam2world transformation.
- File path to the 100 images, each of shape [800 x 800 x 3].
- 100 depth images, each of shape [800 x 800 x 3]. Each channel has the depth in meters, so 2 channels are redundant.

We will now load and visualize the dataset.

Q1.1 Compute the camera matrix K [2 pts]

In [3]:
```
import json

# dataset class provided to load extrinsics, intrinsics and image paths.
class Dataset:
    def __init__(self, json):
        self.json = json # the json file containing the extrinsics, intrinsi
        self.load_extrinsics()
        self.load_intrinsics()
        self.compute_intrinsics()

    def load_extrinsics(self):
        # This function loads the extrinsics parameters from the json file.

        with open(self.json) as f:
            self.data = json.load(f)
        self.frames = self.data['frames'] # 'frames' in the json contains th
        self.transforms = np.array([frame['transform_matrix'] for frame in s
        self.file_paths = np.array([frame['file_path'] for frame in self.fra

    def load_intrinsics(self):
        # This function loads the intrinsics parameters from the json file.
        self.f_x = self.data['fl_x'] # focal length in x
        self.f_y = self.data['fl_y'] # focal length in y
        self.w = self.data['w'] # image width
        self.h = self.data['h'] # image height
        self.cx = self.data['cx'] # principal point in x
        self.cy = self.data['cy'] # principal point in y

    def compute_intrinsics(self):
        self.K = None # K: the intrinsic matrix, shape (3, 3)
        # compute the K matrix form the intrinsic parameters computed in loa
        self.K = np.array([[self.f_x, 0, self.cx], [0, self.f_y, self.cy], [
```

```
dataset = Dataset('images/dataset/transforms_train.json') # load the dataset
np.set_printoptions(precision=3, suppress=True) # do NOT remove this line wh

print('Shape of extrinsic matrices: {}'.format(dataset.transforms.shape)) #
#TODO: print the intrinsic matrix and add to your gradescope submission.
print('K matrix {}'.format(dataset.K)) # The intrinsic matrix K you computed
```

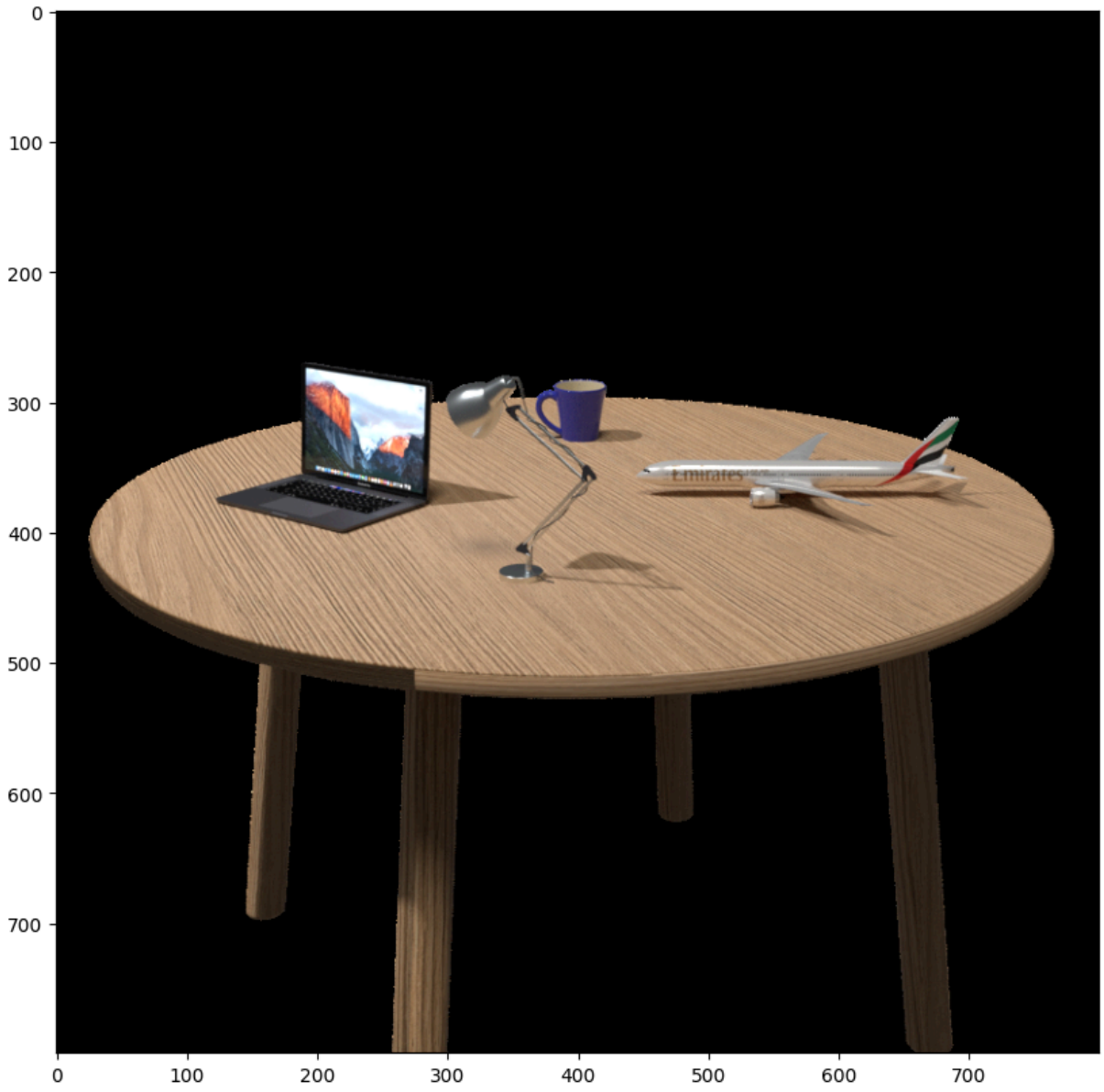
```
Shape of extrinsic matrices: (100, 4, 4)
K matrix [[1111.111    0.     400.   ]
 [   0.    1111.111  400.   ]
 [   0.       0.       1.   ]]
```

Visualize the dataset [0 pts]

Here we show the RGB and depth data that are part of the dataset. Reasoning about algorithm design is often easier when you understand the data.

In [4]:
```
image = cv2.imread(os.path.join('images/dataset',dataset.file_paths[0]))
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

In [5]:
```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.axis('on')
plt.show()
```

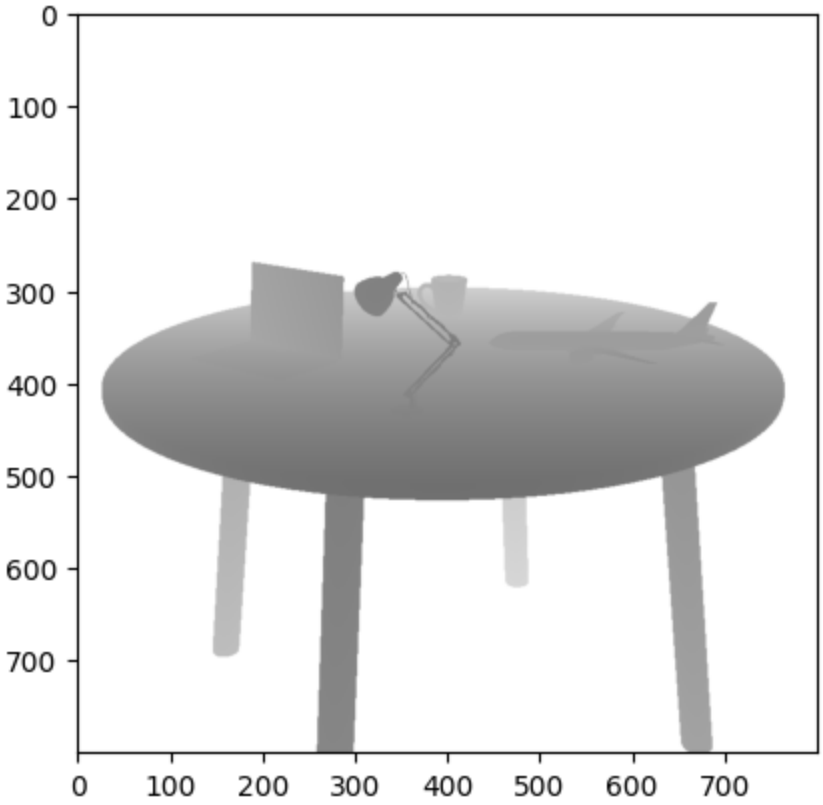


In [6]:
```
def depthmap_viz(depth,min_d=0.0,max_d=3.5):
    # depth: (H,W,3) - depth map, every channel contains the same depth valu

    # min_d: minimum depth value to visualize
    # max_d: maximum depth value to visualize

    depth = np.clip(depth,min_d,max_d)

    depth = (depth-min_d)/(max_d - min_d)

    image = depth

    plt.clf()
    plt.imshow(depth,cmap='magma', vmin=min_d,vmax=max_d)
```

In [7]:
```
depth_location = 'images/dataset/train/depth.npy' # location of ground truth
depths = np.load(depth_location) # load the depth maps

depthmap_viz(depths[0]) # visualize the first depth map
```

```
print('Shape of depth maps: {}'.format(depths.shape))
plt.show() # show the plot
```

```
Shape of depth maps: (100, 800, 800, 3)
```



Loading SAM [0 pts]

The Segment Anything Model (SAM) produces high quality object masks from input prompts such as points or boxes, and it can be used to generate masks for all objects in an image. It has been trained on a dataset of 11 million images and 1.1 billion masks, and has strong zero-shot performance on a variety of segmentation tasks.

Here we load the SAM model and predictor. Running on CUDA and using the default model are recommended for best results. Do not change any settings, as it will complicate our grading, you may not receive full credit if you alter the SAM settings.

In [8]:
```
import sys
from segment_anything import sam_model_registry, SamPredictor

sam_checkpoint = "ckpts/sam_vit_h_4b8939.pth" # the checkpoint loaded in the
model_type = "vit_h"

# device = "cuda" # loading to GPU.
device = "mps"

sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)

predictor = SamPredictor(sam)
```

```
/Users/abhi/Documents/CMU/2024-25/Sem 1/Courses/CV/Assignments/hw6/hw6_env/lib/python3.11/site-packages/segment_anything/build_sam.py:105: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  state_dict = torch.load(f)
```

Q2: Designing our prompt [2 point]

SAM and other segmentation models use input prompts such as a box around an object to generate a mask of the object. From now on if we refer to a `prompt` we mean the box around an object that is to be segmented, which is the input to SAM.

In this homework we will start by acquiring a single user-generated `prompt` in one image: a box around the coffee mug. We will then use depth and camera geometry to propogate the mask to other frames. It is therefore important for the one user-specified prompt to be high-quality. Set the `input_box` parameter such that we can get a high-quality segmentation.

Now we load in the first example image into the model.

In [9]:
```
#TODO: YOUR CODE HERE
# add this plot to gradescope submission.
input_box =  np.array([360, 275, 425, 335]) # [x0, y0, x1, y1]

plt.figure(figsize=(10, 10))
plt.imshow(image)
show_box(input_box, plt.gca())
plt.axis('off')
plt.show()
```

In [10]:
```
# Here we're running SAM on the image with the bounding box.
predictor.set_image(image) # loading the image to the predictor.
masks, _, _ = predictor.predict(
    point_coords=None,
    point_labels=None,
    box=input_box[None, :],
    multimask_output=False,
)
# Calling the predictor with the bounding box.
# You will not need to change any of the other arguments in this homework.
```

Visualizing the mask.

In [11]:
```
mask = masks[0]
h, w = mask.shape[-2:]
mask_image = mask.reshape(h, w, 1)

plt.figure(figsize=(10, 10))
```

```
plt.imshow(image)
show_mask(mask, plt.gca())
show_box(input_box, plt.gca())
plt.axis('off')
plt.show()
```

Q3: Project to 3D [20 points]

As discussed before, the aim of this homework is to use the image mask in one image and propagate it to novel views. In this section we will use the mask generated in the previous question, and project the pixels in the mask to 3D coordinates using depth and camera geometry. Remember we can project points in image coordinates to 3D points by using the P matrix, with P = K[R|t].

Q3.1: Image frame to camera frame [10 pts]

In this part of the question you will be asked to project the points from image frame, so pixel coordinates, to a point cloud in the camera frame. For this you will only need the intrinsic matrix K and the depth. You will not need dataset.transforms in Q2.1.

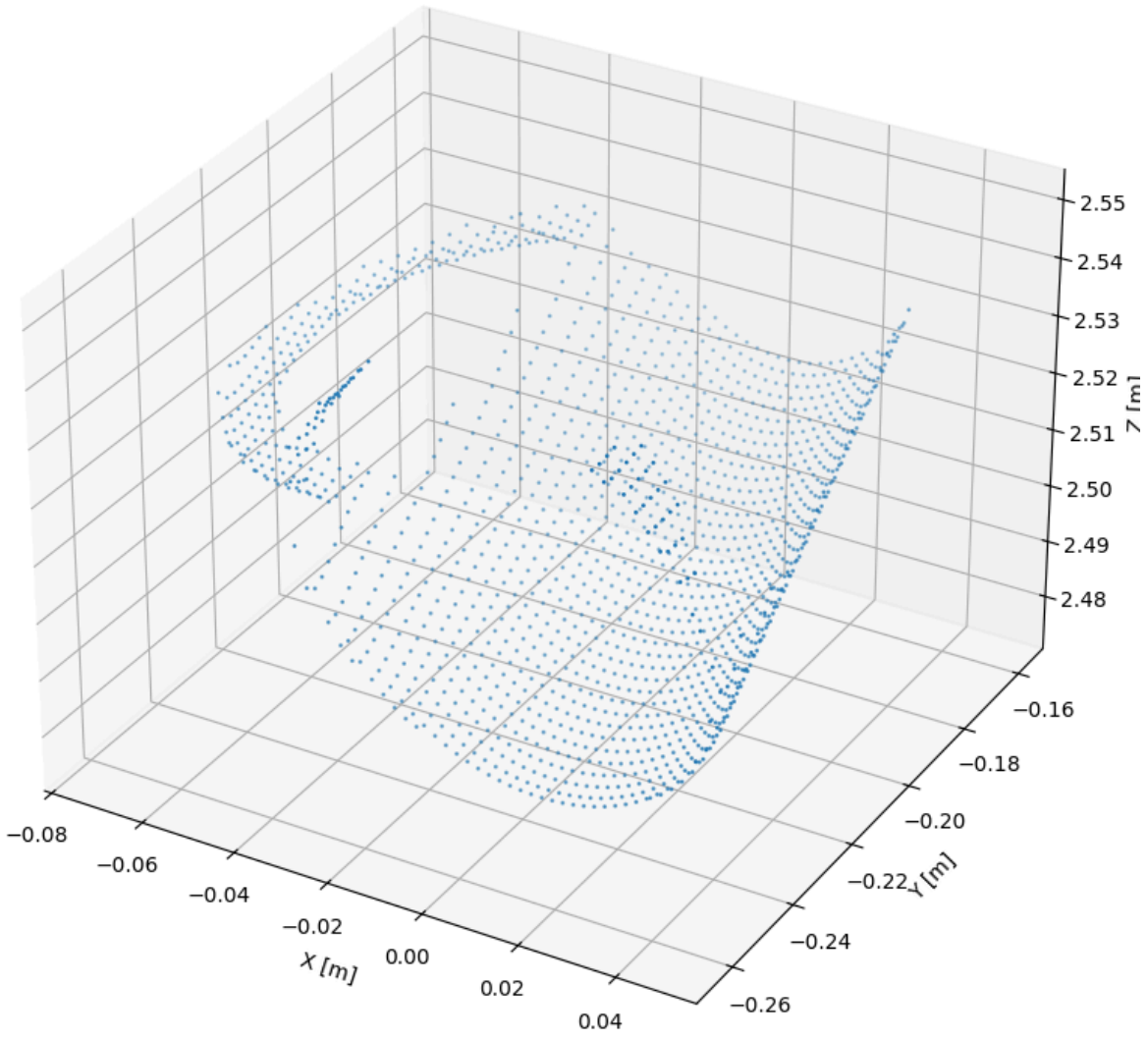
Hint: For Q3.1, remember the difference between the image frame (x horizontal), and the coordinates you might get from the mask. E.g., when you retrieve coordinates from the mask, they would not immediate align with the coordinates expected by the intrinsic matrices

In [12]:
```python
def img2cam(points, K, depths=None):
    # project the points from image coordinates to camera coordinates [5 pts
    cam_3d = None

    # TODO: YOUR CODE HERE
    # (1) Use the intrinsic matrix K to convert the points from image coordi
    # (2) Normalize the points to a plane with z=1.
    # (3) Use depths to scale the points to be at the correct distance from
    points[:, [1, 0]] = points[
        :, [0, 1]
    ]  # matrix coordinates xy = image coordinates yx
    pointsT = np.vstack((points.T, np.ones(len(points))))
    points3dT = np.matmul(np.linalg.inv(K), pointsT)
    points3dT[-1, :] = depths.T
    points3dT[:2, :] = points3dT[:2, :] * points3dT[-1, :]
    cam_3d = points3dT

    return cam_3d


def filter_points(coords, depths, thresh=2.55):
    # filter out points that are too far away in the first mask, this first
    # return filtered coords and depths
    # don't make it too complicated, this should be a one-liner.
    # TODO: YOUR CODE HERE

    in_coords, _ = np.where(depths <= thresh)
    coords_f = coords[in_coords]
    depths_f = depths[in_coords]

    return coords_f, depths_f


def mask2cam(mask, K, depths, thresh=None):
    # project mask points to camera frame [3 pts]
    # steps todo:
    # (1) get all coordinates where the mask is True, this should be N x 2
    # (2) get the depth values for these coordinates, Nx1
    # (3) call filter_points to filter out points that are too far away, wit
    # Here far away means the depth is above a certain threshold.
    # (4) call img2cam to convert the points to camera frame using intrinsic
    # TODO: YOUR CODE HERE
    # Xs, Ys, _ = np.where(mask == True)
    # coords = np.hstack((Xs.reshape(-1, 1), Ys.reshape(-1, 1)))
```

```
    # l = len(coords)
    # Ds = np.zeros((l, 1))
    # for i in range(l):
    #     Ds[i] = depths[coords[i][0], coords[i][1]][0]



    # coords_f, Ds_f = filter_points(coords, Ds, thresh)
    # points3d = img2cam(coords_f, K, Ds_f)

    X, Y, _ = np.where(mask)
    coords = np.stack((X, Y), axis=1)

    Ds = depths[X, Y]
    # keep only the first column of Ds
    Ds = Ds[:, 0]
    # reshape Ds to be a column vector
    Ds = Ds.reshape(-1, 1)

    if thresh is not None:
        coords, Ds = filter_points(coords, Ds, thresh)

    points3d = img2cam(coords, K, Ds)

    return points3d
```

In [13]:
```
cam_pnts_3d = mask2cam(mask_image,dataset.K,depths[0],thresh=2.55) # convert
```

In [14]:
```
def viz_pts_3d(pts,xrange=None,yrange=None,zrange=None,title=None):
    # viz the 3D points
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(pts[0,:],pts[1,:],pts[2,:],s=1)
    ax.set_xlabel('X [m]')
    ax.set_ylabel('Y [m]')
    ax.set_zlabel('Z [m]')

    if xrange is not None:
        ax.set_xlim(xrange)
    if yrange is not None:
        ax.set_ylim(yrange)
    if zrange is not None:
        ax.set_zlim(zrange)

    if title is not None:
        ax.set_title(title)
    plt.show()

np.save('cam_pnts_3d.npy',cam_pnts_3d)

#TODO: add this plot to gradescope submission
viz_pts_3d(cam_pnts_3d)
```



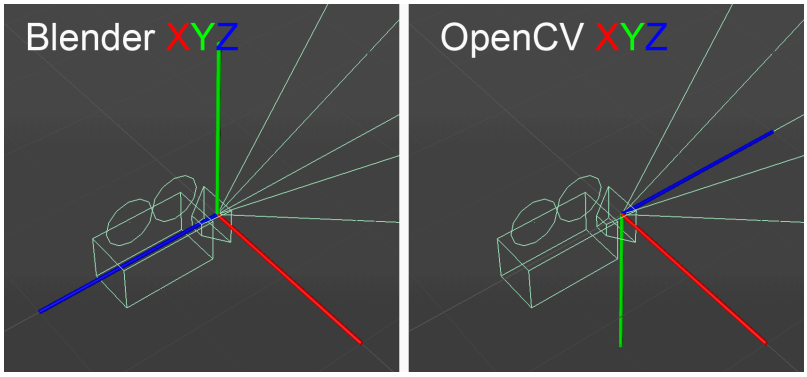
Q3.2: Camera frame to world frame [10 points]

We now project the points in the camera frame to the world frame, keep in mind that the transforms provided are between the world frame and the Blender camera frame (pictured below). You will need to take this difference into account when using the equations of projective geometry. For example, the intrinsics matrix `K` will expect 3D points in the OpenCV camera frame. Summarizing, these are the existing `frames`:

- Camera frame/OpenCV Camera Frame: This is the reference frame for 3D points with respect to the camera, as seen in the class up to now.
- Blender Camera Frame: This is the camera frame for 3D points used in Blender, the `y` and `z` axes are pointing in opposite directions w.r.t. OpenCV frame.
- World Frame: This is the frame for 3D points with respect to the world origin.
- Image Frame: The **2D points** in the image, e.g., `u`, `v` in range [0,H] and [0,W].

In [15]:
```
from IPython.display import Image
img_size = 400
Image(filename="images/cam_frames.png", width=img_size)
```

Out[15]:

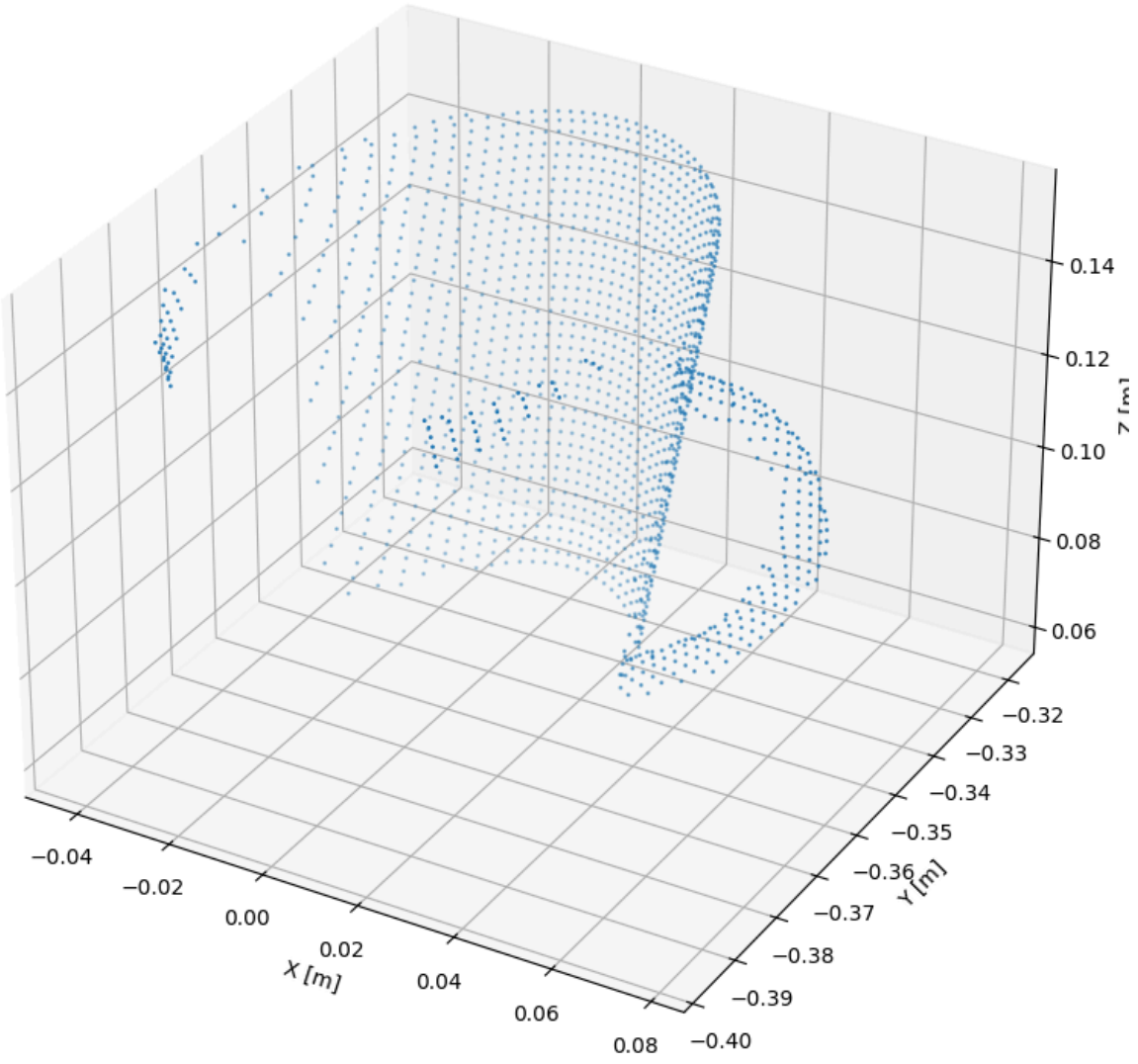


In [16]:
```
def cam2world(points,transform):
    # project camera coordinates to world coordinates [5 pts]
    # NOTE: transform is the transformation from the blender camera frame to
    # TODO: YOUR CODE HERE

    # print(transform.shape)
    # print(points.shape)

    # camera to blender coordinates
    points[1,] = -points[1,]
    points[2,] = -points[2,]

    # camera to world coordinates
    points = np.concatenate([points, np.ones((1, points.shape[1]))], axis=0)
    world_points = transform @ points

    # de homogenize
    world_points /= world_points[3, :]
    return world_points[:3, :]

def world2cam(points,transform):
    # project world coordinates to camera coordinates [5 pts]
    # NOTE: do not use np.linalg.inv to compute the inverse of transform, we
    # There is an intuitive and elegent way to compute the inverse of transf
    # NOTE: do not forget about blender coordinates!

    # TODO: YOUR CODE HERE

    # compute the inverse of transform without using np.linalg.inv
    rot = transform[:3, :3]
    trans = transform[:3, 3]
    inv_rot = rot.T
    inv_transform = np.zeros_like(transform)
    inv_transform[:3, :3] = inv_rot
    inv_transform[:3, 3] = -inv_rot @ trans
    inv_transform[3, 3] = 1

    # project points to blender camera coordinates
    points = np.concatenate([points, np.ones((1, points.shape[1]))], axis=0)
    camera_points = inv_transform @ points
```

```
    # convert to camera coordinates
    camera_points[1,] = -camera_points[1,]
    camera_points[2,] = -camera_points[2,]

    # de homogenize
    camera_points /= camera_points[3, :]

    return camera_points[:3, :]
```

In [17]:
```
def show_mask(mask, ax, random_color=False):
    # This function is used to visualize the mask on the image in a matplotl
    # bool mask: (H, W). True for each pixel that belongs to the object.
    # ax: matplotlib axis
    # random_color: if True, use a random color for the mask. Otherwise, use

    if random_color:
        color = np.concatenate([np.random.random(3), np.array([0.6])], axis=
    else:
        color = np.array([30/255, 144/255, 255/255, 0.6])
    h, w = mask.shape[-2:]
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    ax.imshow(mask_image)

def show_box(box, ax):
    # This function is used to visualize the bounding box on the image in a
    # box: (4,) array. [x0, y0, x1, y1]
    # ax: matplotlib axis

    x0, y0 = box[0], box[1]
    w, h = box[2] - box[0], box[3] - box[1]
    ax.add_patch(plt.Rectangle((x0, y0), w, h, edgecolor='green', facecolor=
```

In [18]:
```
transform_0 = dataset.transforms[0]

world_pts = cam2world(cam_pnts_3d,transform_0)

np.save('world_pts.npy',world_pts)
#TODO: add this plot to gradescope submission
viz_pts_3d(world_pts)
```



Q4: Casting masks to new frames [10 pts]

Now we will look at new viewpoints and extract their point clouds. To do this we will loop through new viewpoints one by one, to run SAM on each new image. Each iteration of the loop we add the new 3D points to the previous 3D points, which are then used in the next iteration to create a new mask using projective geometry and depth.

A simple approach might be to deploy the projective geometry we have developed, and find the bounding box at each iteration based on the coordinate range of the projected point cloud. That is, we project all 3D points to the novel view, an array of [N,2] with each row an x and y coordinate in the image frame. The bounding box could be simply [min_x,min_y,max_x,max_y]. What would be the problem with an approach like this?

Occlussions and noise! Noise can be from some pixels that weren't segmented correctly, occlussions can cause entirely errorneous masks. You will be implementing several

functions to improve the results:
(1) Filter out masks that have a confidence that's too low
(2) Filter out points that are too far away from anything else we've seen so far.
(3) Carefully choose the bounding box dimensions to avoid noise to have a big effect on the result.

In [19]:
```python
def cam2img(points, K):
    # project camera coordinates to image coordinates [3 pts]
    # output should be pixel coordinates in the correct range with shape (2,

    img_pts = K @ points
    img_pts /= img_pts[2, :]
    img_pts = img_pts[:2, :]
    return img_pts
```

In [20]:
```python
# (1) Filter out masks that have a confidence that's too low. [0 pts]

score_thresh = 0.85
def keep_score(score):
    return score > score_thresh
```

In [21]:
```python
# (2) Filter out points that are too far away from anything else we've seen

from hmac import new


dist_thresh = 0.13 # decide on a good distance threshold [2 pts]

def keep_dist(new_pts,existing_pts):
    # TODO: YOUR CODE HERE
    # compute median of all_world_pts
    # compute distance between tmp_world_pts and median of all_world_pts
    # reject outliers that are too far away from all other points

    median = np.median(existing_pts, axis=1)
    dist = np.linalg.norm(new_pts - median[:, None], axis=0)

    return new_pts[:, dist < dist_thresh]
```

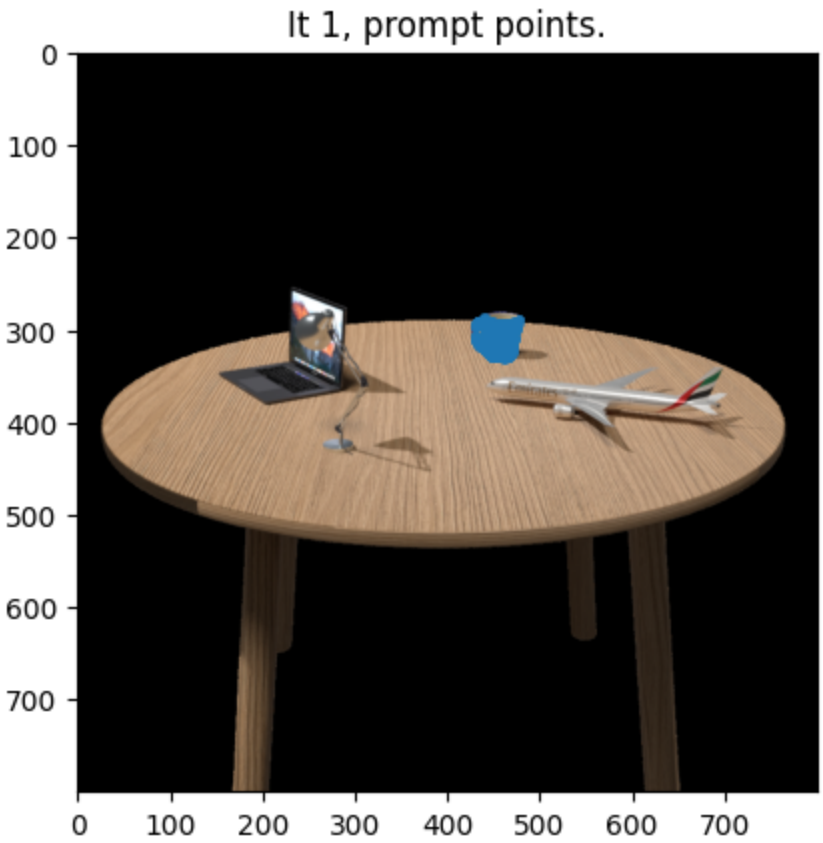
In [22]:
```python
# (3) Carefully choose the bounding box dimensions to avoid noise to have a

# filter out points n_std away from mean of all points [3 pts]
from sympy import im


def filter_for_box(world_points,transform, n_std=2):
    # TODO: YOUR CODE HERE
    # compute mean and std of all points [2 pts]
    # filter out points that are n_std away from mean of all points [3 pts]
    # transform to cam frame [1 pt]
    # you will need intrinscis matrix K here, simply call dataset.K

    mean_pt = np.mean(world_points, axis=1)
```

```
    std_pt = np.std(world_points, axis=1)
    condition1 = np.abs(world_points[0, :] - mean_pt[0]) < n_std * std_pt[0]
    condition2 = np.abs(world_points[1, :] - mean_pt[1]) < n_std * std_pt[1]
    condition3 = np.abs(world_points[2, :] - mean_pt[2]) < n_std * std_pt[2]
    condition = np.logical_and(np.logical_and(condition1, condition2), condi
    world_points_f = world_points[:, condition]
    cam_points = world2cam(world_points_f, transform)
    img_points = cam2img(cam_points, dataset.K)

    return img_points

# based on the filtered points, compute the bounding box [2 pts]
def prompt_points_to_box(prompt):
    # TODO: YOUR CODE HERE
    # output: np.array([x0, y0, x1, y1]])
    # (x0, y0): top-left corner
    # (x1, y1): bottom-right corner

    x0 = np.min(prompt[0])
    y0 = np.min(prompt[1])
    x1 = np.max(prompt[0])
    y1 = np.max(prompt[1])

    return np.array([x0, y0, x1, y1])
```

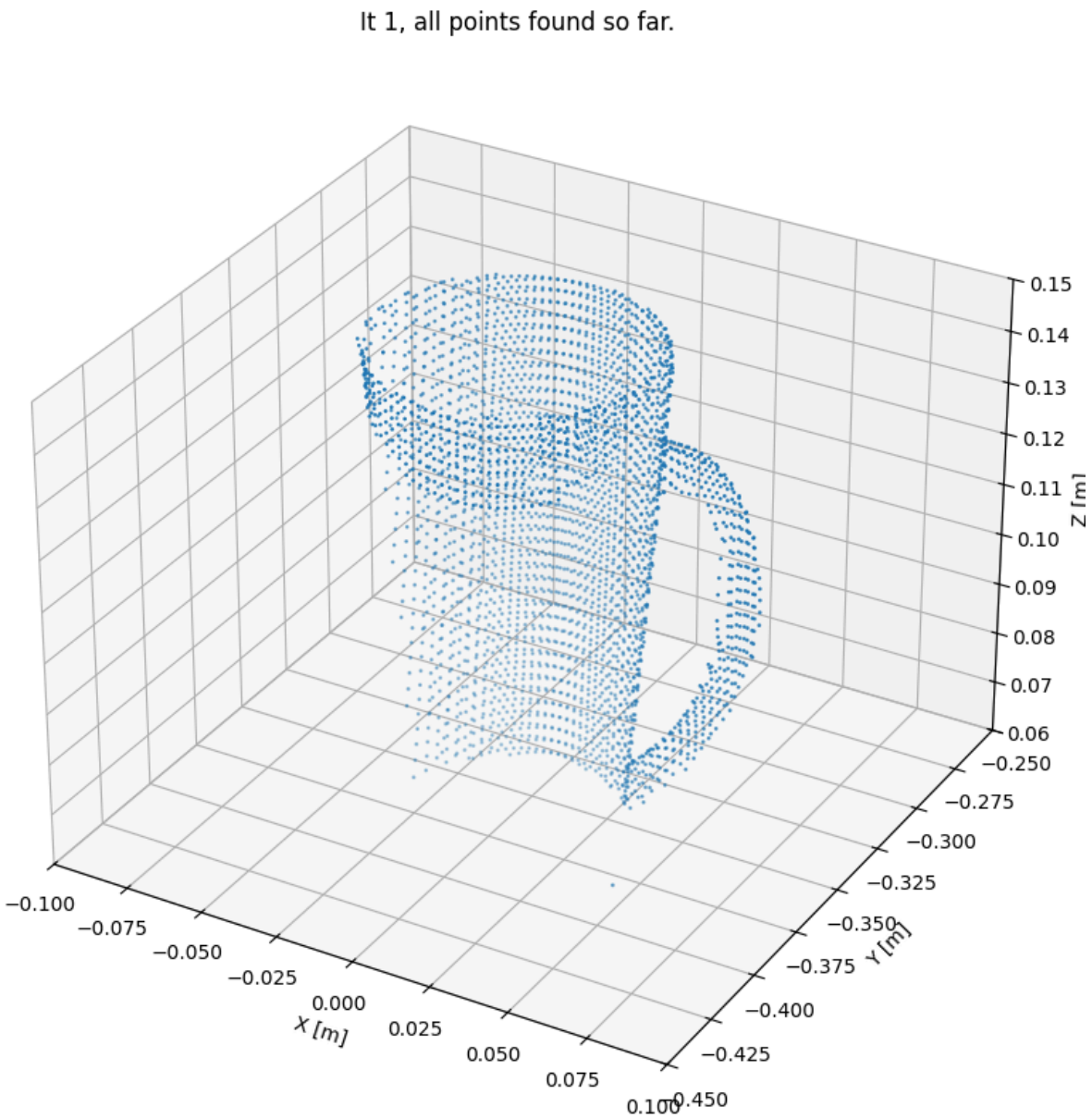
In [23]:

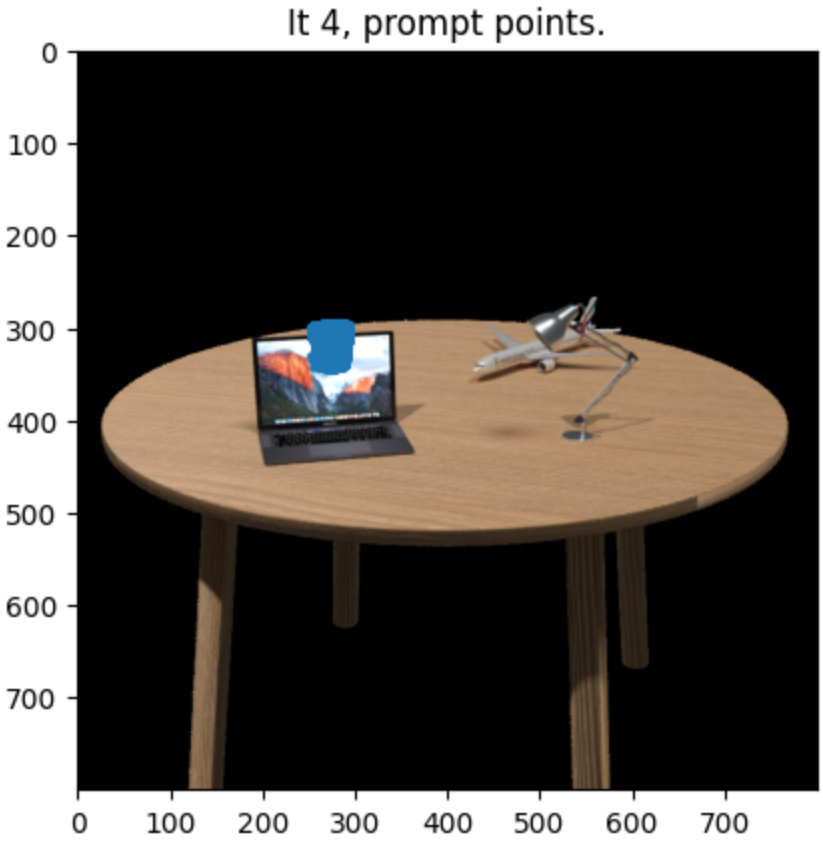
```
import os
import copy
from sklearn.cluster import KMeans
all_world_pts = copy.deepcopy(world_pts)

it = 1

# show_its = [1,4,10]
show_its = [1,4,10,50,75,99]

end_idx = -1 # set this to a different number, e.g. 10, for faster debugging

#TODO: add all plots generated to gradescope submission
for transform, file_path, depth in zip(dataset.transforms[1:end_idx], datase
    # compute 3d points
    image = cv2.imread(os.path.join('images/dataset/', file_path))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    prompt_points = filter_for_box(all_world_pts,transform) # (3) Carefully

    if it in show_its:
        plt.imshow(image)
        # scatter plot the img_pts
        plt.scatter(prompt_points[0,:],prompt_points[1,:],s=5)
        plt.title('It {}, prompt points.'.format(it))
        plt.show()

    predictor.set_image(image)

    prompt_box = prompt_points_to_box(prompt_points) # (3) Choose the boundi

    masks, scores, _ = predictor.predict(
```

```
        box=prompt_box,
        point_labels=[1],
        multimask_output=False,
    )

    if it in show_its:
        plt.figure(figsize=(10,10))
        plt.imshow(image)
        show_mask(masks, plt.gca())
        show_box(prompt_box, plt.gca())
        plt.axis('off')
        plt.title('It {}, mask.'.format(it))
        plt.show()

    mask = masks.reshape((h, w, 1))
    cam_pnts_3d = mask2cam(mask,dataset.K,depth, thresh=2.55)
    tmp_world_pts = cam2world(cam_pnts_3d,transform)

    if keep_score(scores): # (1) Filter out masks that have a score that's t
        tmp_world_pts = keep_dist(tmp_world_pts,all_world_pts) # (2) Filter
        all_world_pts = np.hstack([all_world_pts,tmp_world_pts])

    if it in show_its:
        viz_pts_3d(all_world_pts,title='It {}, all points found so far.'.for

    it+=1
```

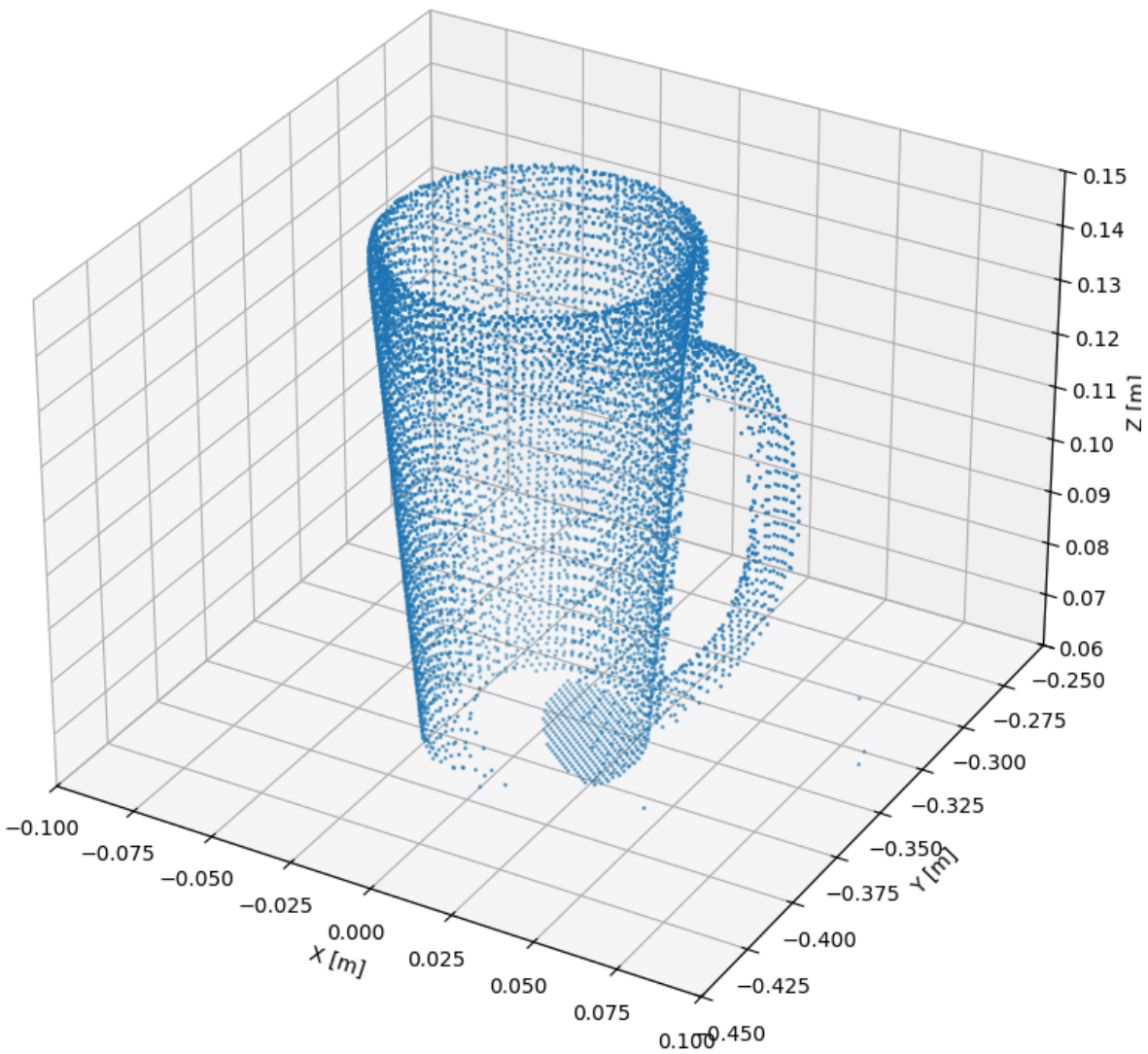


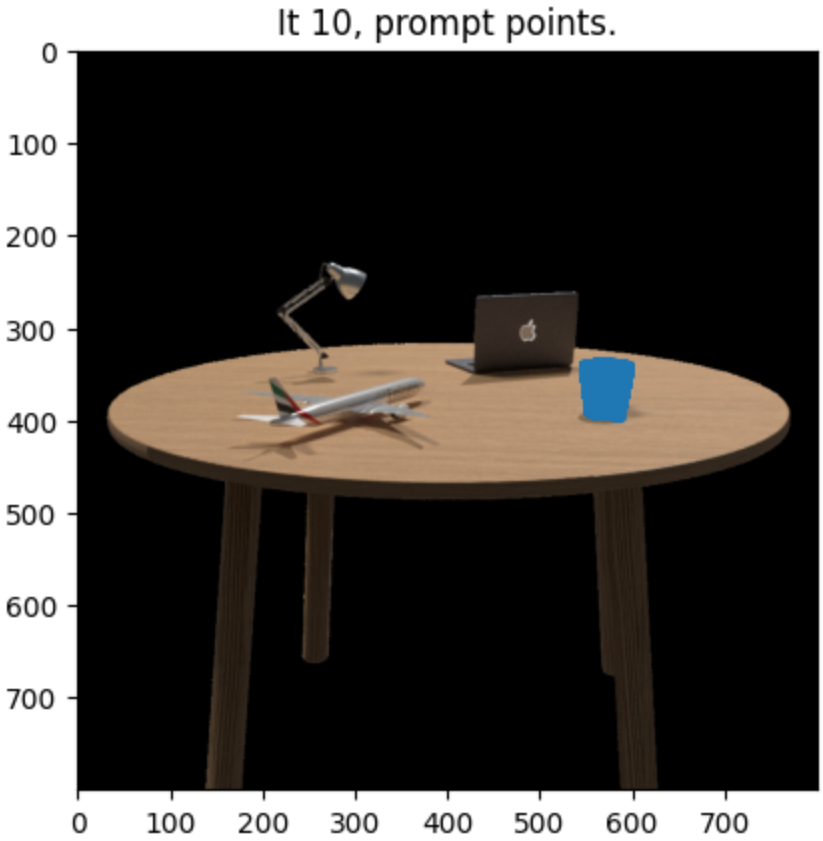

It 1, mask.


It 1, all points found so far.
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06
Z [m]
−0.250
−0.275
−0.300
−0.325
−0.350
−0.375
−0.400
−0.425
−0.450
Y [m]
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100
X [m]

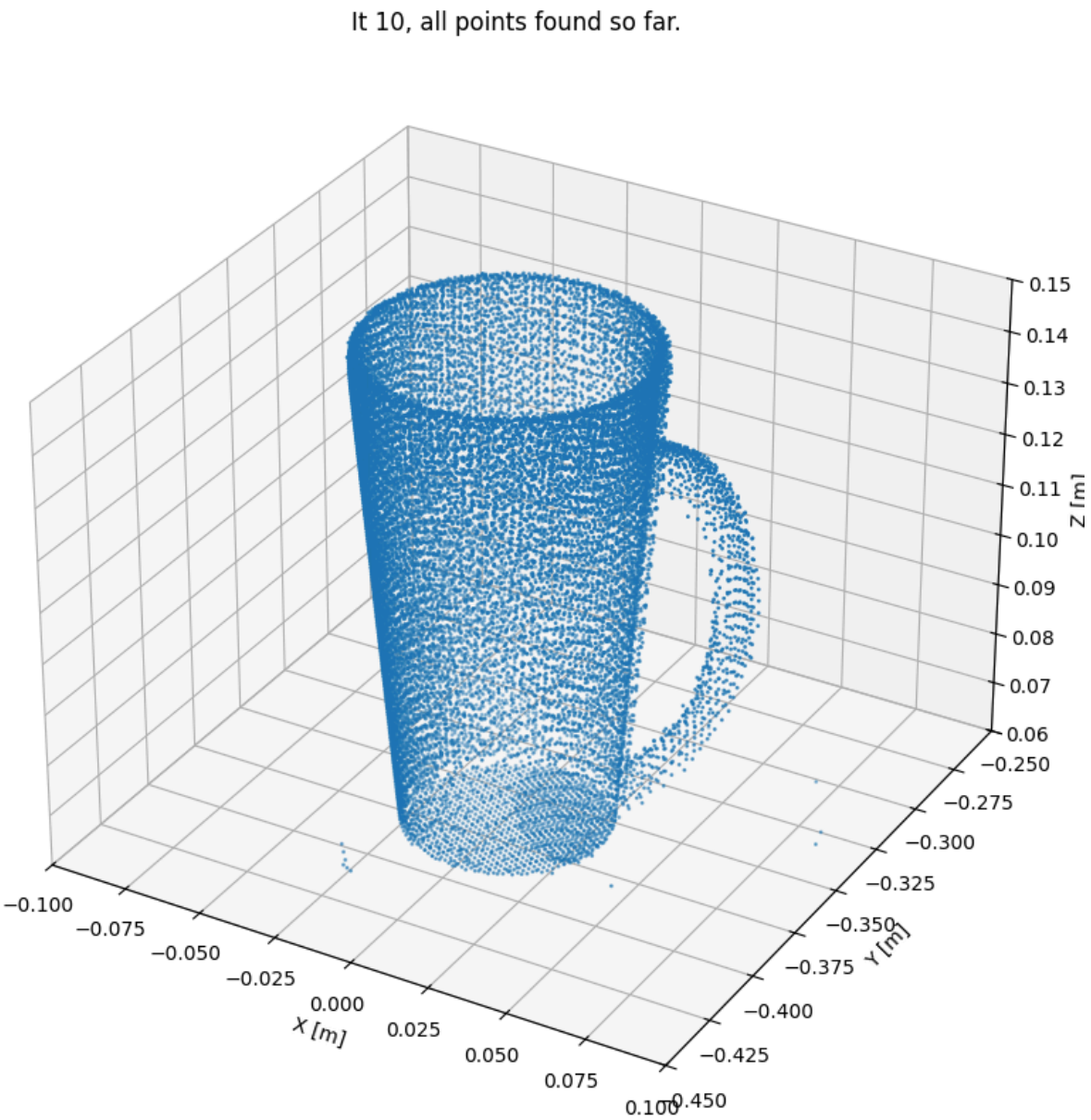

It 4, prompt points.
0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700

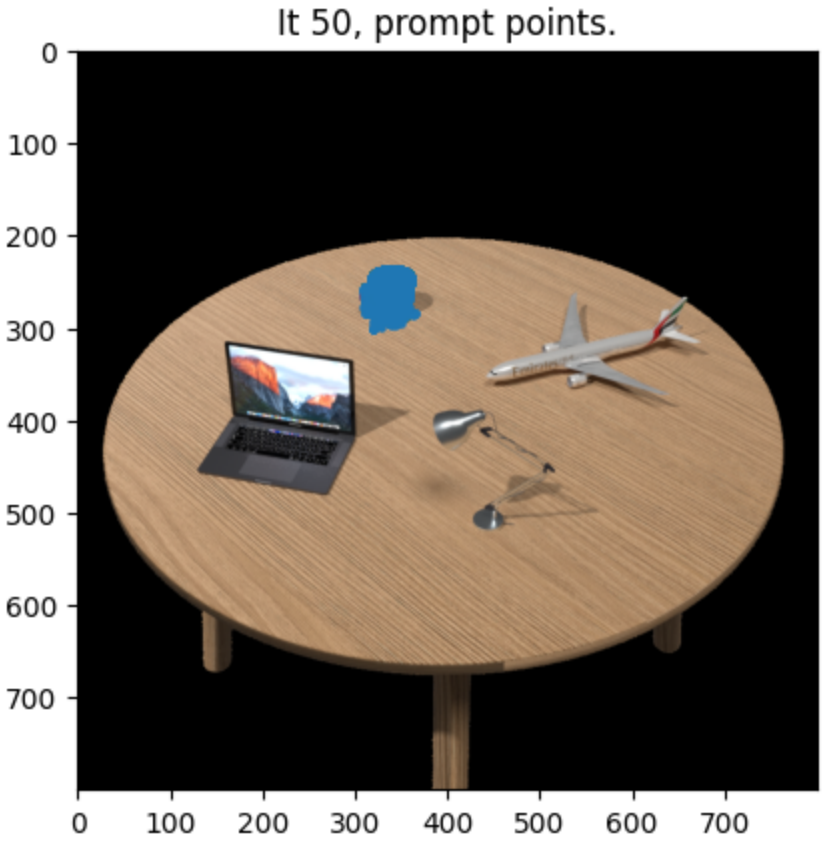
It 4, mask.


It 4, all points found so far.
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06
Z [m]
−0.250
−0.275
−0.300
−0.325
−0.350
−0.375
−0.400
−0.425
−0.450
Y [m]
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100
X [m]

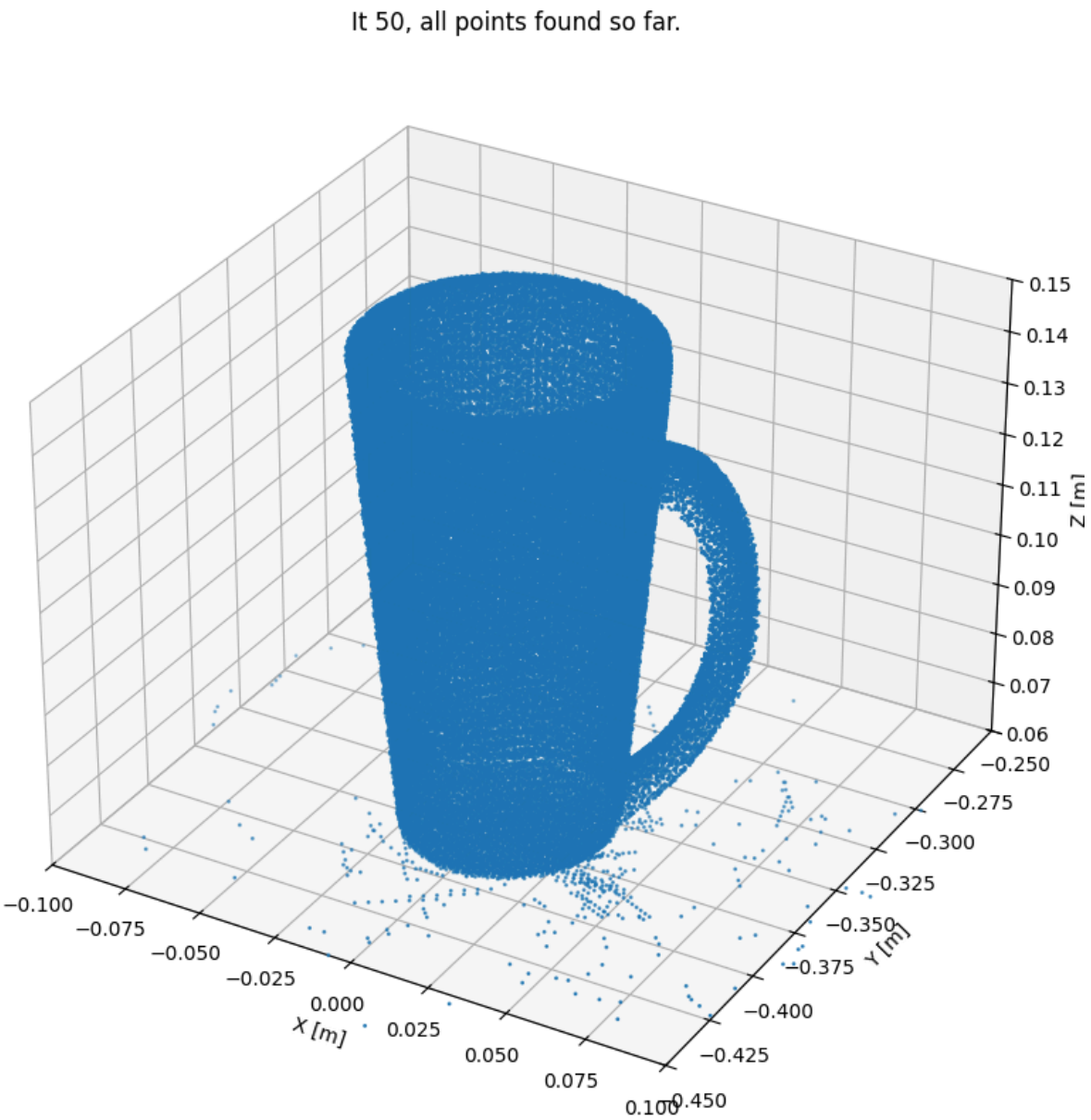

It 10, prompt points.
0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700

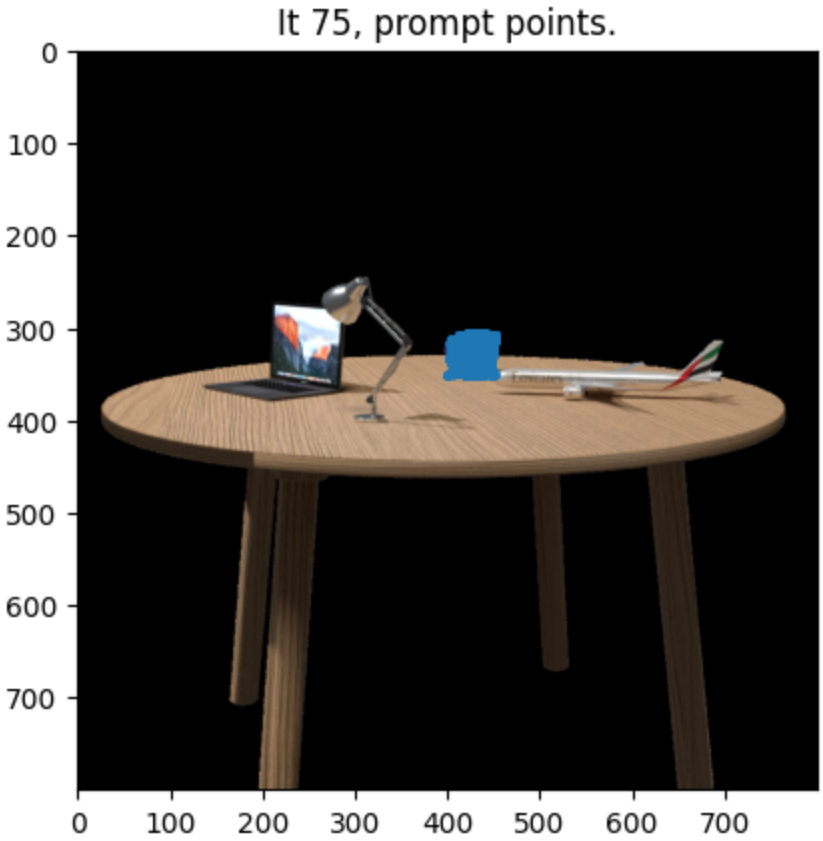
It 10, mask.


It 10, all points found so far.
X [m]
Y [m]
Z [m]
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100
−0.250
−0.275
−0.300
−0.325
−0.350
−0.375
−0.400
−0.425
−0.450
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06

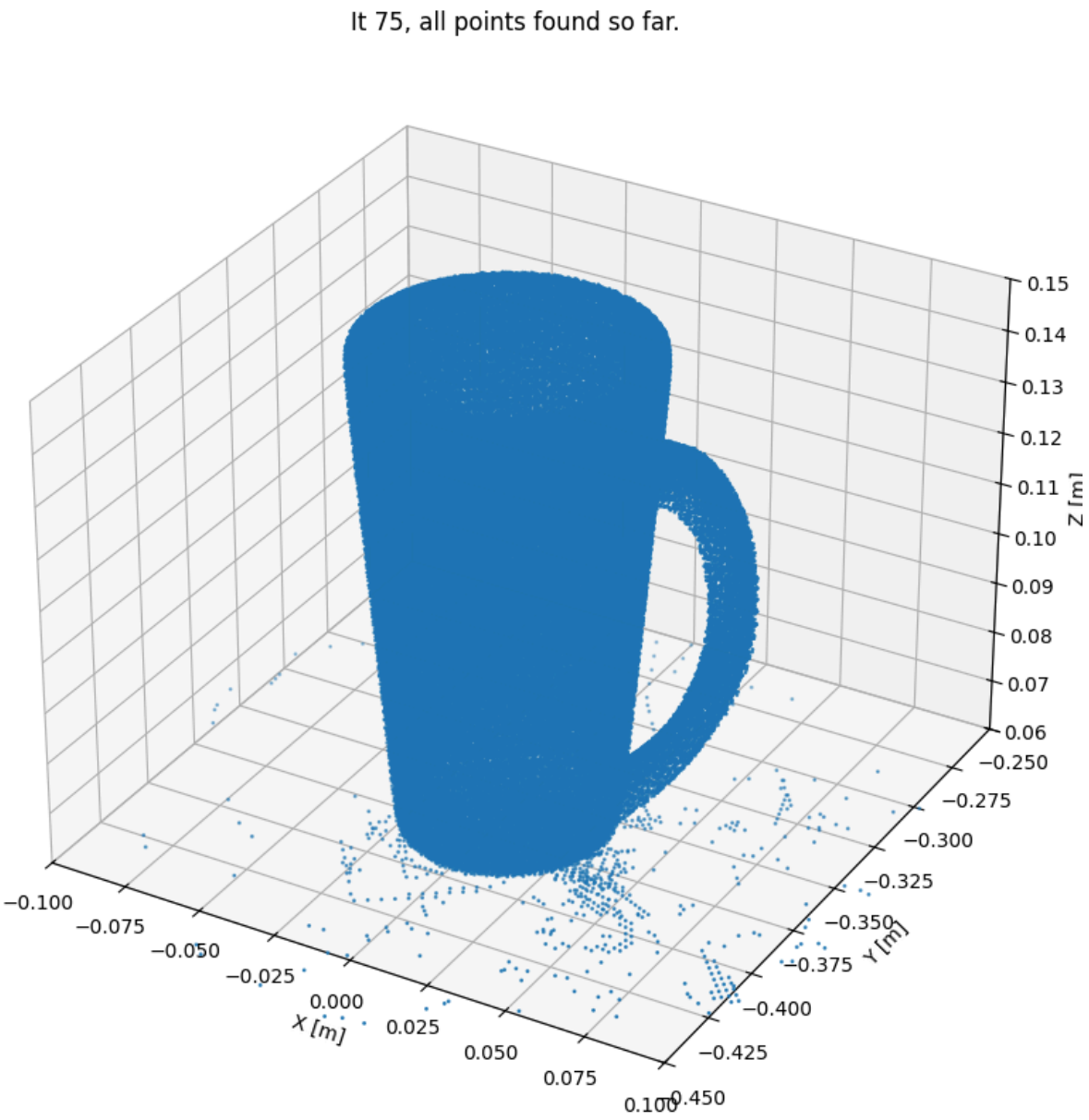

It 50, prompt points.
0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700


It 50, mask.


It 50, all points found so far.
X [m]
Y [m]
Z [m]
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100
−0.250
−0.275
−0.300
−0.325
−0.350
−0.375
−0.400
−0.425
−0.450
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06

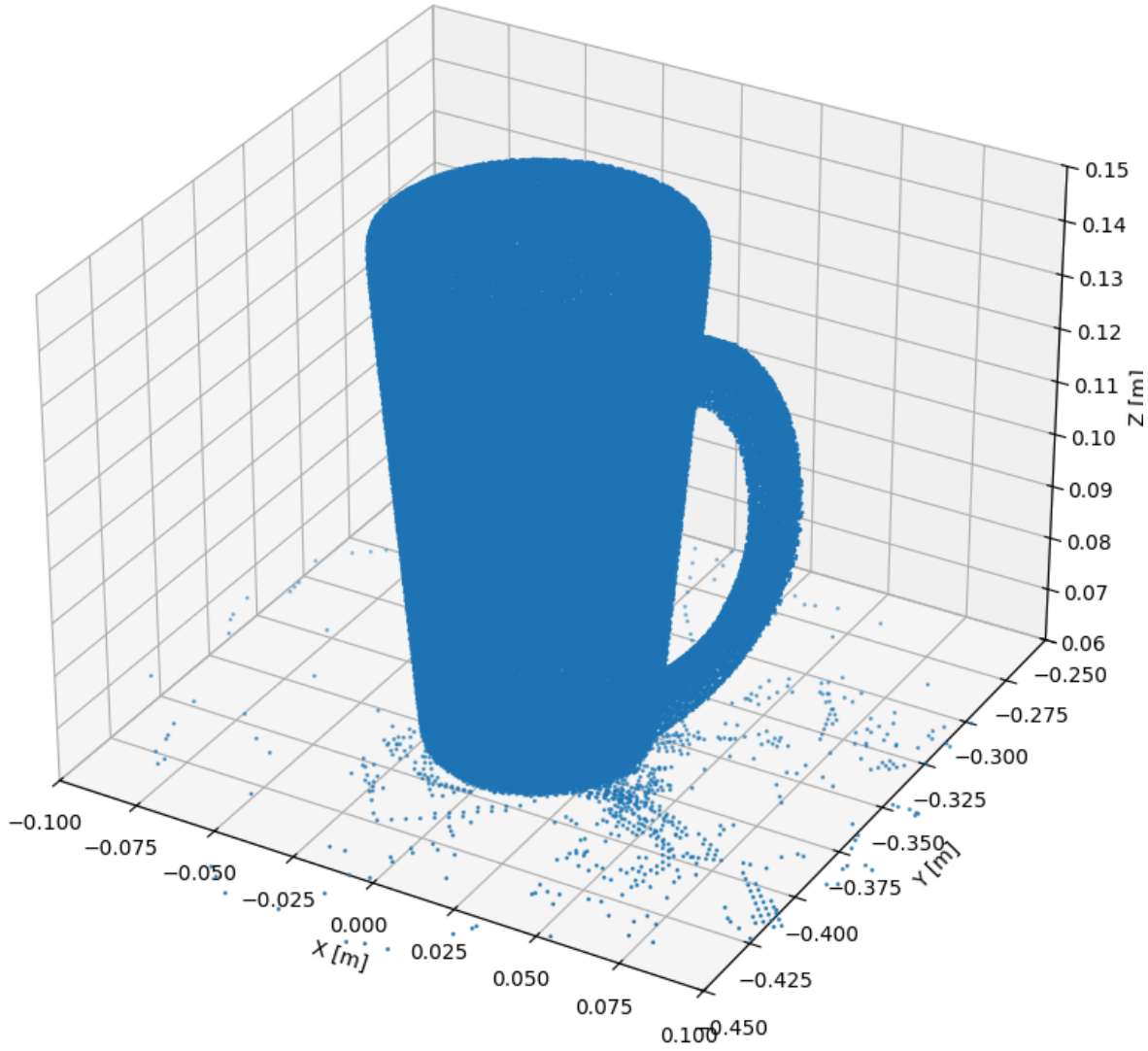

It 75, prompt points.
0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700

It 75, mask.



Visualize all 3D points

In [24]:
```
#TODO: do NOT need to add to gradescope submission
viz_pts_3d(all_world_pts,xrange=[-0.1,0.1],yrange=[-0.45,-0.25],zrange=[0.06
```



Q5: Statistical outlier removal [3 pts]

In [ ]:
```python
import open3d as o3d

print("open3d version: ", o3d.__version__)

def filter_points(points,nb_neighbors=20,std_ratio=2.0):

    # filter points using open3d statistical outlier removal. [3 pts]
    # TODO: YOUR CODE HERE

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points.T)
    cl, ind = pcd.remove_statistical_outlier(nb_neighbors, std_ratio)
    points_f = np.asarray(pcd.points)

    return points_f.T
```

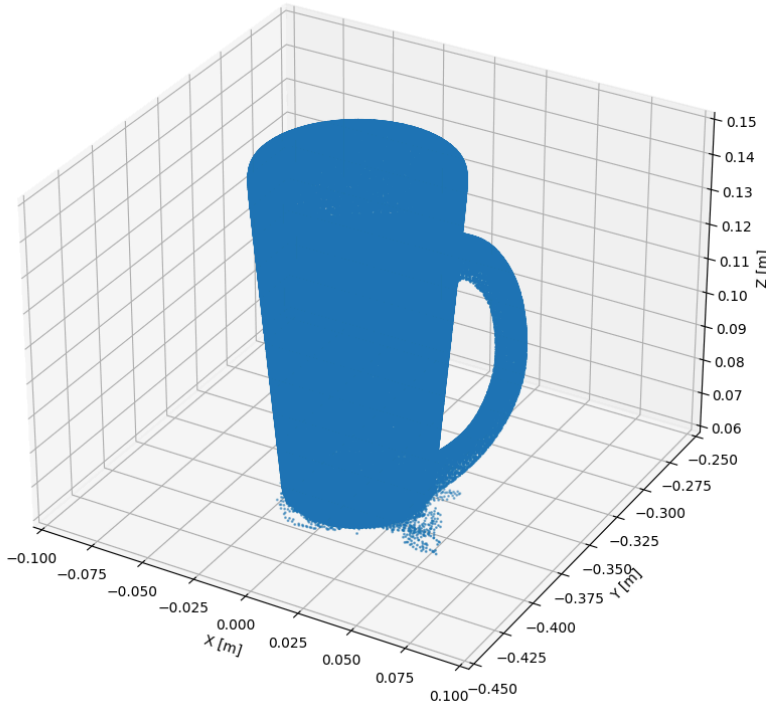
```
all_world_pts_filtered = filter_points(all_world_pts[:3,:])

#TODO: add this plot to gradescope submission
viz_pts_3d(all_world_pts_filtered,xrange=[-0.1,0.1],yrange=[-0.45,-0.25],zra
```

Expected Output

Below is the output we achieved at the end of the homework, you implementation should be similar to receive full credit.

In [7]:
```
Image(filename="images/expected_output.png", width=img_size, height=img_size
```

Out[7]:



Q6 Extra credit: 3D Segmentation without Ground Truth Depth [10 pts Max]

So far we have provided you with ground truth depth from the 3D rendering toolbox. For a maximum of 10 extra points, can you achieve similar accuracy without using ground truth depth? You can use any toolbox/repository you like, as long as they infer dense depth maps: depth for every pixel in the image. Here is one approach we believe would be relatively straightforward:

- The dataset is formatted for Neural Radiance Fields (NeRFs). You should be able to run NeRF on this dataset with little modifications.
- Suggested NeRF pipeline: Torch-NGP. It runs fast using a cuda backend, but all the high-level features are implemented in Torch.

- Some necessary changes: (1) Modify the code to only use the training data, no testing or validation (not provided in dataset). You could also copy training data to a test and validation folder and create the necessary json files. (2) Modify the code to return all *train* depth maps in a `.npy` array, in units [meters].

Any other methods are allowed and encouraged, as long as they infer dense depth maps: depth for every pixel in the image. Keep in mind the difference between z-depth as in Blender (the distance along the cameras principle axis), depth as euclidean distance from the camera center.