Geometric computer vision v.2021.1: Homework assignment #1

Leshchev Dmitrii

28 March 2021

Github repo: LINK

1 Solution

1.1

First of all we needed to **write your code to constrict a world-frame point cloud from a depth image, using known intrinsic and extrinsic camera parameters. Hints: use the class 'RaycastingImaging' to transform image to points in camera frame, use the class 'CameraPose' to transform image to points in world frame.**

1. As an extrinsic matrix represent the location of the camera in the 3-D scene, it is definitely the one we need to use for initialization of CameraPose object (inversion is done inside initialization).

   $pose\_i = CameraPose(extrinsics[i])$

2. RaycastingImaging class need resolution_image and resolution_3d as an input, and they are contained in intrinsics_dict for every element i.

   $imaging\_i = RaycastingImaging(intrinsics\_dict[i]['resolution\_image'], intrinsics\_dict[i]['resolution\_3d'])$

3. Finally, we need to transform our image firstly to points in camera frame, using RaycastingImaging object and then to points in world frame, using CameraPose object.

   $points\_i = pose\_i.camera\_to\_world(imaging\_i.image\_to\_points(image\_i))$

1.2

The next task was the following: **Reproject points from view_j to view_i, to be able to interpolate in view_i. We are using parallel projection so this explicitly computes (u, v) coordinates for reprojected points (in image plane of view_i). TODO: your code here: use functions**

from CameraPose class to transform ‘points_j‘ into coordinate frame of ‘view_i‘

points_j were given in the world frame, CameraPose object of view_i was also given, so to reproject points from view_j to view_i we just needed to use method world_to_camera of CameraPose object.

$reprojected\_j = pose\_i.world\_to\_camera(points\_j)$

1.3

Next task: **For each reprojected point, find K nearest points in view_i, that are source points/pixels to interpolate from. We do this using imaging_i.rays_origins because these define (u, v) coordinates of points_i in the pixel grid of view_i. TODO: your code here: use cKDTree to find k=‘nn_set_size‘ indexes of nearest points for each of points from ‘reprojected_j‘**

1. u and v are the first two coordintates of rays_origin, so we take them first.

   $uv\_i = imaging\_i.rays\_origins[:, :2]$

2. Then we need to first built kd-tree for quick nearest-neighbor lookup on *uv_i* and then find k (=nn_set_size) nearest neighbors from *uv_i* to all the points reproject points from view_j to view_i (reprojected_j). We again take first two coordinates of that points, for calculating the distance.

   $\_, nn\_indexes\_in\_i = cKDTree(uv\_i).query(reprojected\_j[:, :2], k = nn\_set\_size)$

1.4

Build an [n, 3] array of XYZ coordinates for each reprojected point by taking UV values from pixel grid and Z value from depth image. TODO: your code here: use ‘point_nn_indexes‘ found previously and distance values from ‘image_i‘ indexed by the same ‘point_nn_indexes‘

To make an array on nearest neighbours of every reprojected point, we need to concatenate corresponding uv_i coordinates with corresponding values from depth image (as it is stated in the hint). As our image is 2D array and cKDTree.query returns indexes of any array, making it 1D, we need to use flattened image (image and distances are flattenef on the 71 and 72 rows of the file respectively:($image\_i\_flatten = image\_i.flatten()$; $distances\_i\_flatten = distances\_i.flatten()$). To concatenate two vectors columns I used hstack.

$point\_from\_j\_nns = np.hstack((uv\_i[point\_nn\_indexes], image\_i\_flatten[point\_nn\_indexes].reshape(-1, 1)))$

1.5

TODO: compute a flag indicating the possibility to interpolate by checking distance between ‘point_from_j‘ and its ‘point_from_j_nns‘ against the value of ‘distance_interpolation_threshold‘ I used a Euclidean distance between projected point and its neighbours and then checked, if all distances (because there are k neighbours) are less, then the threshold.

$distances\_to\_nearest = np.linalg.norm(poin\_from\_j - point\_from\_j\_nns, axis = 1)$
$interp\_mask[idx] = np.all(distances\_to\_nearest < distance\_interpolation\_threshold)$

1.6

The last task was: **TODO: your code here: use ‘interpolate.interp2d‘ to construct a bilinear interpolator from distances predicted in ‘view_i‘ (i.e. ‘distances_i‘) into the point in ‘view_j‘. Use the interpolator to compute an interpolated distance value.**

1. First we define an interpolation function $z = f(x, y)$, used first two coordinates of obtained nearest neighbours of point from j ($point\_from\_j\_nns$ as x and y and distances of $i - th$ view (corresponding to the neighbours indexes) ($distances\_i$) as z.

   $interpolator = interpolate.interp2d(point\_from\_j\_nns[:, 0], point\_from\_j\_nns[:, 1], distances\_i\_flatten[point\_nn\_indexes])$

2. And then we compute distances for points, projected form j to i, itself, using defined above interpolator.

   $distances\_j\_interp[idx] = interpolator(point\_from\_j[0], point\_from\_j[1])$

2 Results

As a result, I got the following objects for 5 initial files from med_res.