Geometric computer vision. Homework assignment 1.

Report by Evgeny Lagutin

```
# TODO: 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.

pose_i = CameraPose(extrinsics[i])
imaging_i = RaycastingImaging(intrinsics_dict[i]['resolution_image'],
intrinsics_dict[i]['resolution_3d'])
points_i = pose_i.camera_to_world(imaging_i.image_to_points(image_i))
```

Camera extrinsics parameters hold transform from the world to the camera and can be used to initialize CameraPose class object

RaycastingImaging class object allows to get points from the depth image with coordinates in the camera's system.

Thus I use camera_to_world to transform these points from the camera to the world coordinate system.

```
# TODO: your code here: use functions from CameraPose class
#  to transform `points_j` into coordinate frame of `view_i`

reprojected_j = pose_i.world_to_camera(points_j)
```

Here I transform points obtained from the view j to the view i by function *world_to_camera*.

```
# 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`
uv_i = imaging_i.rays_origins[:, :2]
_, nn_indexes_in_i = cKDTree(uv_i).query(reprojected_j[:, :2], k=nn_set_size)
```

imaging_i.rays_origins holds (u, v) coordinates points_i in the pixel grid of view i and z coordinates.

uv_i is a dataset of pairs and for each pair in reprojected_j[:, :2] we are to find k nearest points in uv_i

---

```
# 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`
point_from_j_nns = np.concatenate([uv_i[point_nn_indexes], image_i.reshape(-1)
[point_nn_indexes].reshape(-1, 1)], axis=1)
```

depth values are contained in the 3rd coordinate of image_i

```
# 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`
distances_to_nearest = np.linalg.norm(point_from_j[None, :] - point_from_j_nns,
ord=2, axis=1)
interp_mask[idx] = np.all(distances_to_nearest <
distance_interpolation_threshold)
```

here everything seems to be clear

```
# 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.
if method == 'bilin':
    interpolator = interpolate.interp2d(*uv_i[point_nn_indexes].T,
distances_i.reshape(-1)[point_nn_indexes])
    distances_j_interp[idx] = interpolator(*point_from_j[:2])
elif method == 'bispline':
    tck = interpolate.bisplrep(*uv_i[point_nn_indexes].T, distances_i.reshape(-1)
[point_nn_indexes], kx=1, ky=1)
    distances_j_interp[idx] = interpolate.bisplev(*point_from_j[:2], tck)
```

The interpolator should take 3 arrays of coordinates: x, y, z. x and y again come from uv_i - coordinates of all (to the resolution extent) points in the view i and the predicted distances in the view i - values that we want to interpolate.

Then we use interpolator to obtain values (distances) for each of the reprojected points (point_from_j)
