Solution of the homework #1 on Geometric Computer Vision course.

- 1st task:

```
    # 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(images[0].shape, intrinsics_dict[0]
['resolution_3d'])
    points_i = pose_i.camera_to_world(imaging_i.image_to_points(image_i))
```

Here I was to make a world-frame point cloud from a depth image. According to the previously taken labs and well-known `CameraPose` class I initialize an instance of this class as `pose_i` having `extrinsic` camera parameters.
Then I obtain points from depth image and transform them to camera coordinate system.

```
  # 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`
  reprojected_j = pose_i.world_to_camera(points_j)
```

Transformation is done via changing points coordinates according to coordinate points of the `view_j`.

```
  # 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],
nn_set_size)
```

Idk what to comment here, the TODO is pretty self-explanatory, I've just did what I was asked to.

```
  # 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_fl[point_nn_indexes].reshape(-1, 1)), axis=1)
```

Here I just concatenate arrays of coordinates.

```
  # 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)
```

To compute a flag I use `numpy.all` function, pretty simple. Distance is taken as euclidean.

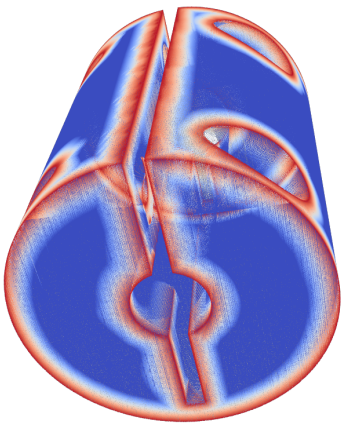
```
  # TODO: your code here: use `interpolate.interp2d`
  #  to construct a bilinear interpolator from distances predicted
  #  in `view_i` (i.e. `distances_i`) into the point in `view_j`.
  #  Use the interpolator to compute an interpolated distance value.
  interpolator = interpolate.interp2d(point_from_j_nns[:, 0],
point_from_j_nns[:, 1], distances_i_fl[point_nn_indexes])
  distances_j_interp[idx] = interpolator(point_from_j[0], point_from_j[1])
```

Default method: we put points x, y and distances and do interpolation.

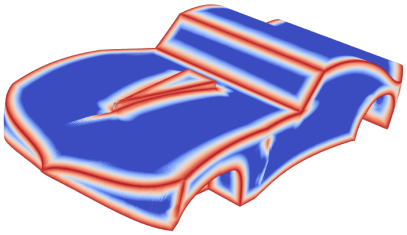
```
  distances_j_interp[idx] = interpolate.bisplev(
  point_from_j[0],
  point_from_j[1],
  interpolate.bisplrep(
      point_from_j_nns[:, 0],
      point_from_j_nns[:, 1],
      distances_i_fl[point_nn_indexes],
      kx=1,
      ky=1
      )
  )
```

Bonus method: same procedure with same logic. The values of `kx` and `ky` equal to 1 (responsible for the degrees of the spline) appeared to be the best choice.

Some of the pictures I have obtained:




z
y
x




z
y
x