TP 1 : Basic operations and structures on point clouds

NPM3*D - January 16*th, 2020
[*mva.npm3d@gmail.com*](mailto:mva.npm3d@gmail.com)

Objectives

- *Load and visualize point clouds on CloudCompare*
- *Understand basic operations like subsampling and neighborhood search.*
- *Understand how optimized structures help to compute basic operations faster.*

The report should be a pdf containing the answers to the Questions *and named "TPX_LASTNAME_Firstname.pdf". Your code should be in a zip file named "TPX_LASTNAME_Firstname.zip". You can do the report as a pair, just state both your names inside the report.*
Send your code along with the report to the email address above. The object of the mail must be "[MVA_NPM3D] TPX LASTNAME Firstname".

A. Point clouds manipulations

In the first part of the practical session, you will use CloudCompare software to visualize various point clouds, and you will write a python script applying a simple transformation to a point cloud.

1) *In CloudCompare, open the PLY file : bunny.ply*
   Drag and drop the file or "File > Open > Choose file"

2) *Following the CloudCompare Cheat Sheet, play with the visualization :*
   a) *Change point size*
   b) *Test orthographic projection and perspective view*
   c) *Activate/Remove EDL (Eye-Dome Lighting)*
   d) *Show RGB or Scalar field*

3) *In transformation.py, complete the code to apply the transformations described by the following steps:*

a) Center the cloud on its centroid.
b) Divide its scale by a factor 2.
c) Apply a -90° rotation around z-axis
d) Recenter the cloud at the original position
e) Apply a -10cm translation of along y-axis

Apply this transformation to bunny and save the result.
Tip 1 : You might find the following functions useful : np.mean, [np.dot](np.dot)
Tip 2 : In "utils/ply.py" we provided the functions read_ply and write_ply with a help in their definition

<u>Question 1 (2 point):</u> Show a screenshot of the original bunny and the transformed bunny together. Pay attention to the appearance of the point cloud (activating EDL with perspective projection generally gives better visualizations). You should obtain something looking like figure 1.

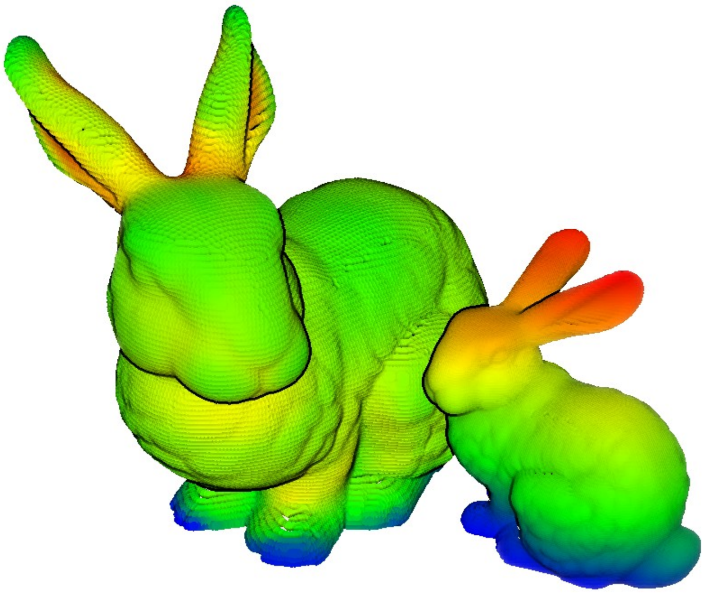

Figure 1 : original and transformed bunnies

B. Structures and neighborhoods

We will introduce here the concept of point neighborhoods. You will have to understand the concept of point cloud structures to implement a fast neighborhood computation.

a. *The concept of neighborhoods*

In the case of 3D points, the two most commonly used neighborhood definitions are the **spherical neighborhood** *and the* **k-nearest neighbors** *(KNN). For a chosen point* P*, the spherical neighborhood comprises the points situated less than a fixed radius from* P*, and*
the k-nearest neighbors comprises a fixed number of closest points to P*.*

Figure 4 : spherical neighborhood (left) and KNN (right)

1) ***In** neighborhoods.py**, implement the functions** brute_force_spherical **and** brute_force_knn **to search the neighbors of some points (the queries) in a point cloud (the supports). Use the indoor scan as support. For the spherical neighborhoods, use a 20cm radius and for the KNN, use*** $k = 1000$.
Tip : Compute the distances from queries to supports one query at a time, or you might consume all of your RAM if you have many queries.

<u>Question 2 (1 points):</u> Try to search the neighborhoods for 10 queries with both methods. Report the time spent. How long would it take to search the neighborhoods for all points in the cloud?

b. *Hierarchical structures*

If you want to search neighborhoods, hierarchical structures are more appropriate. The

two most commonly used structures are **octrees** ***and*** **kd-trees*****. These two structures are very similar, as they are both hierarchical trees defining a partition of the space.***

An octree is specifically designed for three-dimensional space, recursively subdividing it into eight octants. Each node of an octre thus have exactly eight children. Octrees are the three-dimensional analog of quadtrees. A kd-tree is more general and can partition any k-dimensional space.

As opposed to octrees, kd-trees nodes exactly have two children, which are half-spaces separated by an hyperplane. This structure thus recursively subdivide a space into convex sets by hyperplanes, with the condition that the hyperplanes directions follow the space axes successively.

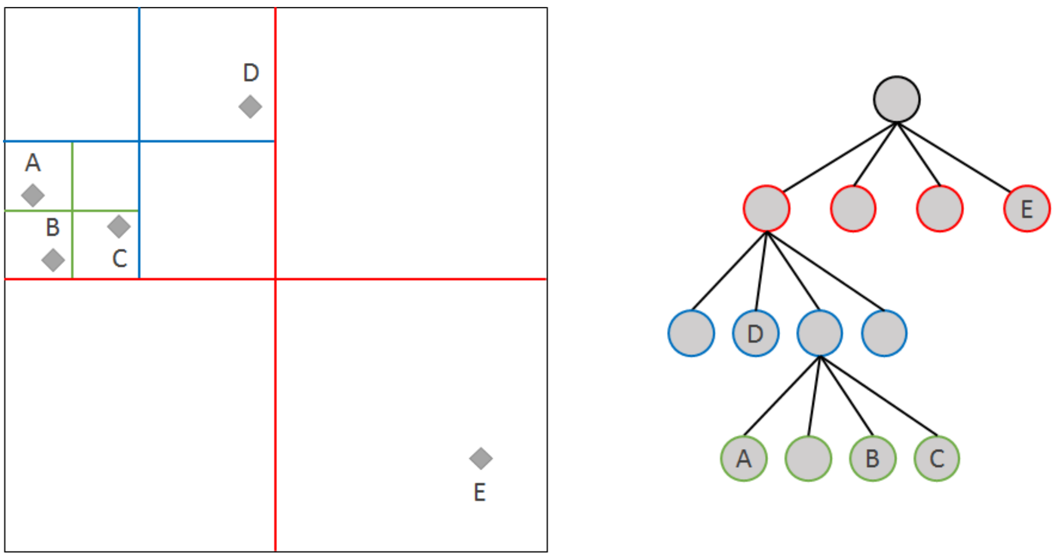

Figure 6 : A quadtree (equivalent of an octree in 2-dimensional space)

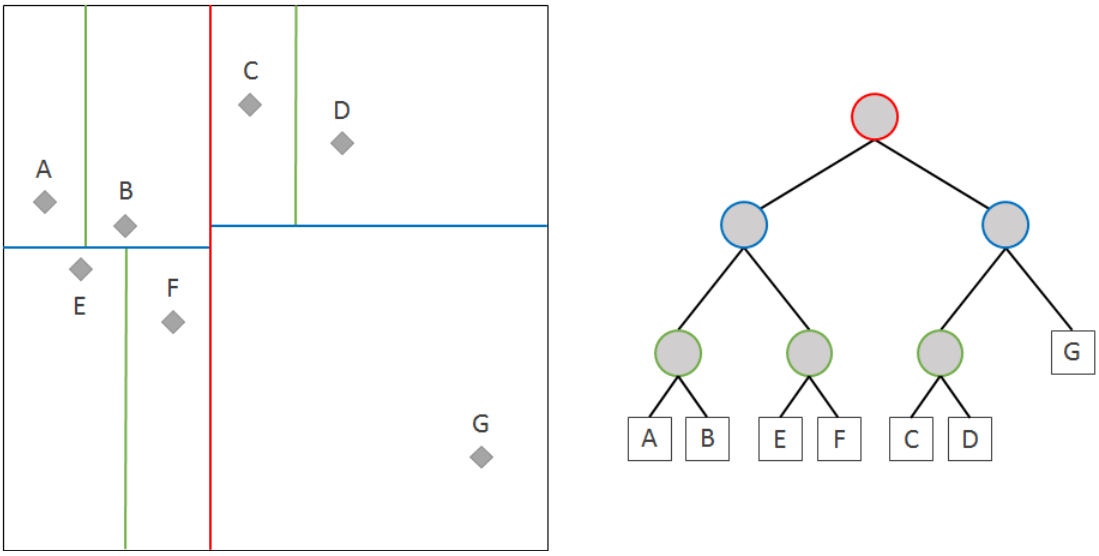

Figure 7 : A kd-tree in 2-dimensional space

1) ***Scikit-learn offers an efficient implementation of [KDTree](). Explore the documentation of this class to implement a spherical neighborhood search. Play with the parameter leaf_size.***

<u>Question 3 (2 points):</u> which leaf size allows the fastest spherical neighborhoods search? In your opinion, why the optimal leaf size is not 1?

2) ***With the optimal leaf_size, time the computation of 1000 random queries with radius values in [10cm, 20cm, 40cm, 60cm, 80cm, 100cm, 120m, 140cm].***

<u>Question 4 (3 points):</u> plot the timings obtained with KDTree as a function of radius. How does timing evolve with radius? How long would it take now to search 20cm neighborhoods for all points in the cloud?

C. *Subsampling methods*

In this part you will experiment two subsampling methods on point clouds: decimation and grid subsampling.

If we define a point cloud C as a $N*3$matrix then the* decimated cloud C_k *is obtained by

keeping one row out of k *of this matrix :*

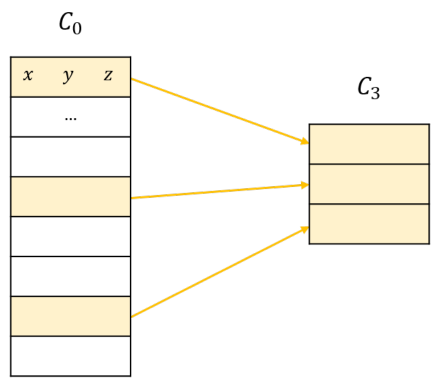

Figure 2 : illustration of decimation

The **grid subsampling** *is based on the division of the 3D space in regular cubic cells called voxels. For each cell of this grid, we only keep one point. This point, the representant of the cell, can be chosen in different ways, for example, it can be the barycenter of the points in that cell.*

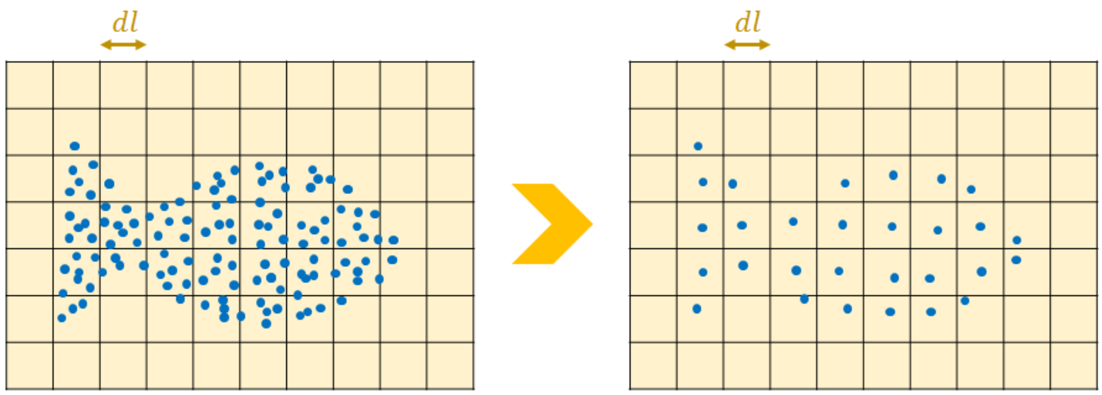

Figure 3 : illustration of grid subsampling in 2D

1) ***In** subsampling.py, complete the function cloud_decimation to decimate the point cloud indoor_scan.ply. You will use a factor* $k=300$*.*
   Tip : Slicing a list in python is pretty simple with the commande `a[start:end:step]`

2) ***In** subsampling.py, complete the function grid_subsampling to subsample the point cloud regularly. You will keep the barycenter of each voxel as new points, and use 20 cm voxels.*
   Tip 1 : You will find some help in the python file where the function is defined.
   Tip 2 : You may use python dictionaries to keep the sum and the number of points in each voxel. This sparse structure is more adapted than full arrays which

will use all your memory on bigger point clouds (see [Mapping Types - dict](#)). Tip 3 : A dictionary cannot take a [i, j, k] vector of coordinates as key if it is a list. However converting it to a tuple (i, j, k) will make it work.

3) ***Add the color to the subsampled cloud in the function grid_subsampling_colors. You can keep the color barycenter in each voxel. Tip : Be careful with the numpy dtype of the sum of colors. The normal color type "uint8" can take values from 0 to 255. Change the type when summing and come back to uint8 after the final division.***

Question 5 (2 points): Show screenshots of the decimated point cloud and the grid subsampled point cloud in color. Comment on the advantages and drawbacks of each method.

(BONUS)

4) ***Add labels to the subsampled cloud. Tip 1 : You cannot use the same method with the barycenter, because it makes no sense with discrete labels. You can keep the predominant label in the voxel. Tip 2 : You might find the function label_binarize of scikit-learn useful .***

Question Bonus 1 : Show a screenshot of the grid subsampled point cloud with labels.