TP 1: Basic operations and structures on point clouds

Objectives

- Load and visualize point clouds on Cloud Compare
- 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.pdf”. Your code should be in a zip file named “TPX_LASTNAME.zip”. You can do the report as a pair, just state both your names inside the report and in the pdf and zip filenames, like “TPX_LASTNAME1_LASTNAME2.pdf”
Send your code along with the report to the email mva.npm3d@gmail.com. The object of the mail must be “[NPM3D] TPX LASTNAME” or “[NPM3D] TPX LASTNAME1 LASTNAME2” if you are a pair working on the report.

A. Point clouds manipulations in Cloud Compare

In the first part of the practical session, you will use Cloud Compare software to visualize various point clouds

1) Cloud Compare installation : http://www.danielgm.net/cc/
   Download and install the latest stable release

2) In Cloud Compare, open the PLY file : bunny.ply
   Drag and drop the file or “File > Open > Choose file”

3) 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

B. Point clouds transformations in Python

You will write a python script to open and apply a simple transformation to a point cloud.

1) Python installation : you need to install python > 3.5 and the following packages:
   a) numpy
   b) scikit-learn
   c) scipy
   d) matplotlib

2) If you never used python, install the IDE pycharm or use notepad++

3) In transformation.py, complete the code to apply the transformations described by the following steps:
   a) Center the cloud on its centroid (centroid = mean of points in x, y, z).
   b) Divide its scale by a factor 2.
   c) Recenter the cloud at the original position.
   d) Apply a -10cm translation along y-axis.

   Apply this transformation to bunny and save the result.
   Tip 1: You might find the following functions useful: numpy.mean, numpy.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:</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 get something looking like Figure 1.

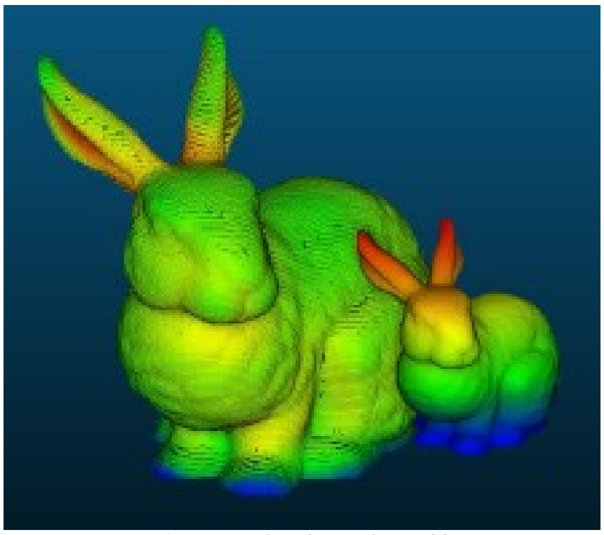
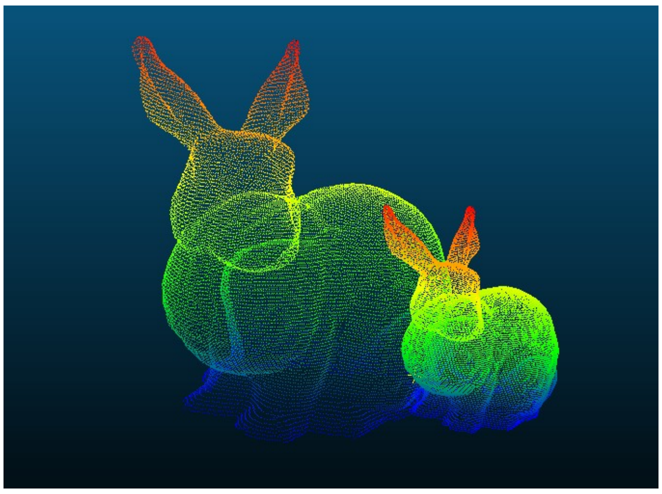


Figure 1 : original and transformed bunnies



EDL is not available on my software (bug on Arch Linux).

C. Subsampling methods

In this part you will test a classical subsampling method on point clouds: decimation.

If we define a point cloud C as a N ∗ 3 matrix then the **decimated cloud** Ck is obtained by keeping one row out of k of this matrix:

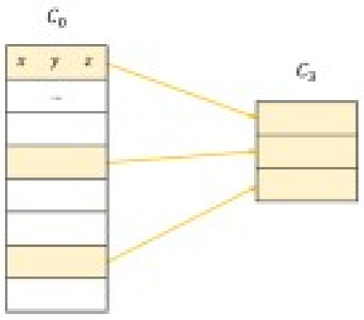
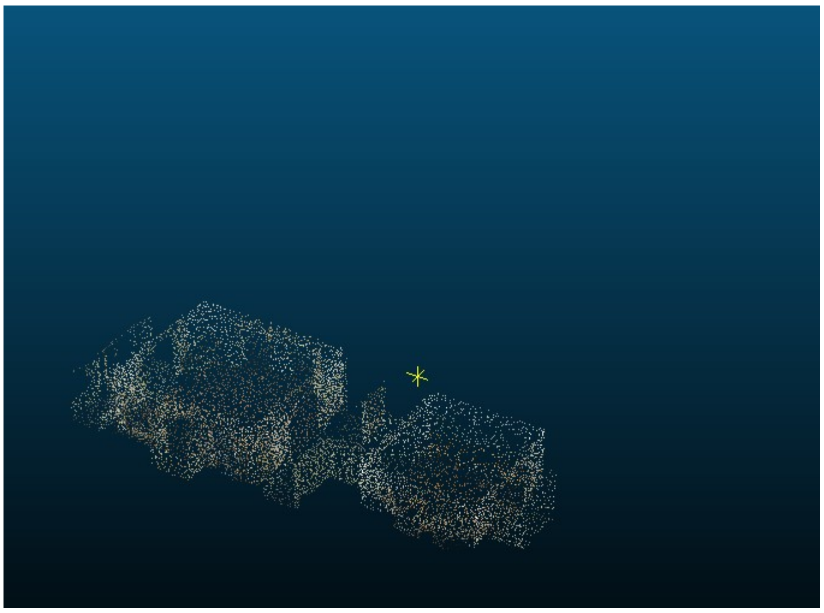


Figure 2 : illustration of decimation

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 command `a[start:end:step]`

<u>Question 2:</u> Show a screenshot of the decimated "indoor_scan" point cloud.



D. Structures and neighborhoods

The last part of the practical session will introduce 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.

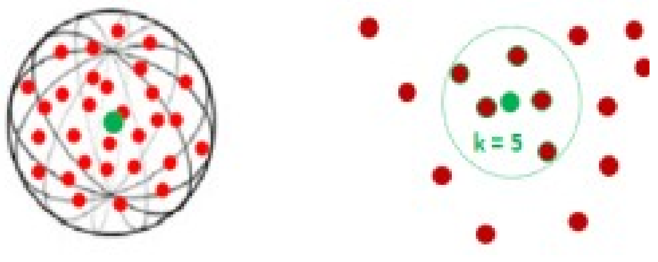


Figure 3: spherical neighborhood (left) and KNN (right)

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 = 100.
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 3:</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?

10 spherical neighborhoods computed in 0.166 seconds

10 KNN computed in 0.271 seconds

Computing spherical neighborhoods on whole cloud : 14 hours

Computing KNN on whole cloud : 23 hours

It would take 14 hours to search for spherical neighborhoods and 23 hours for KNN on the whole cloud.

b. Hierarchical structures

If you want to search neighborhoods with multiple radiuses, hierarchical structures are more appropriate than a fixed grid structure. 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 octree 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.

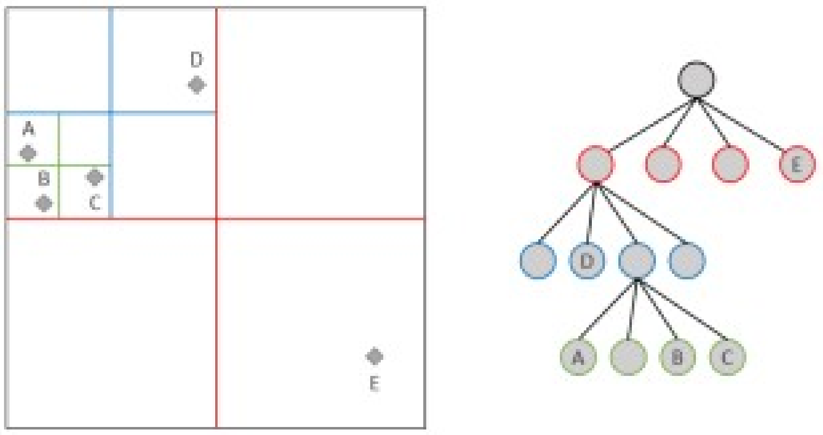


Figure 4: A quadtree (equivalent of an octree in 2-dimensional space)

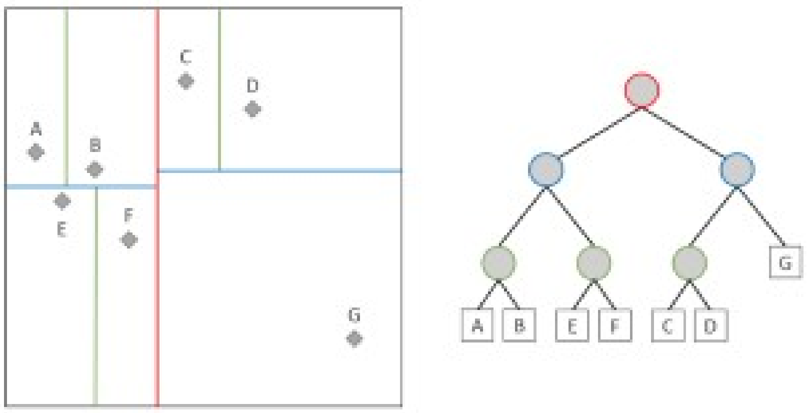


Figure 5: A kd-tree in 2-dimensional space

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 4a:</u> Which leaf size allows the fastest spherical neighborhoods search? In your opinion, why the optimal leaf size is not 1?

The fastest leaf size is such that tree traversal and brute-force calculations are balanced.

The optimal leaf size is not 1 because this increases the number of nodes traversed by query and uses too much memory for large point clouds.

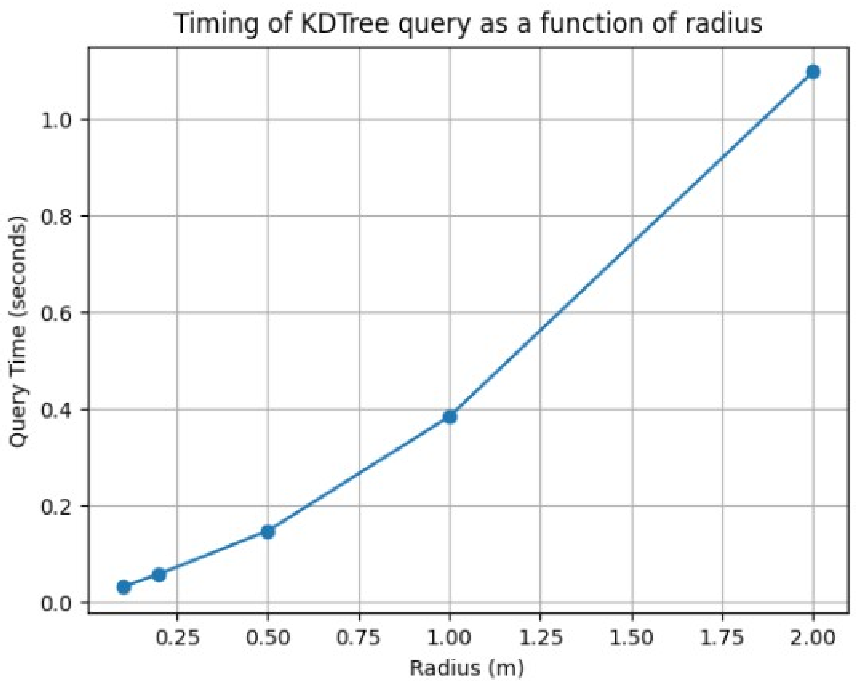
2) With the optimal leaf_size, time the computation of 1000 random queries with the

same radius values as before.

```
Leaf size 1: 0.090 seconds for 1000 queries

Leaf size 5: 0.059 seconds for 1000 queries

Leaf size 10: 0.051 seconds for 1000 queries

Leaf size 15: 0.047 seconds for 1000 queries

Leaf size 20: 0.049 seconds for 1000 queries

Leaf size 25: 0.045 seconds for 1000 queries

Leaf size 30: 0.047 seconds for 1000 queries

Leaf size 35: 0.048 seconds for 1000 queries

Leaf size 40: 0.054 seconds for 1000 queries

Leaf size 100: 0.059 seconds for 1000 queries

Leaf size 1000: 0.093 seconds for 1000 queries

Optimal leaf size: 25

Time with optimal leaf size (25): 0.052 seconds for 1000 queries
```

It takes approx 0.052 seconds with the supposed optimal leaf size of 25. (On several runs it was optimal but not always).

<u>**Question 4b:**</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?**



The timing increases as the radius increases. To search for 20cm neighborhoods of all points in cloud, it would take 0.04 hours (according to our latest run).

E. Going further (BONUS)

Another classical method for subsampling is to use **grid subsampling**. 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 representing of the cell, can be chosen in different ways, for example, it can be the barycenter of the points in that cell.

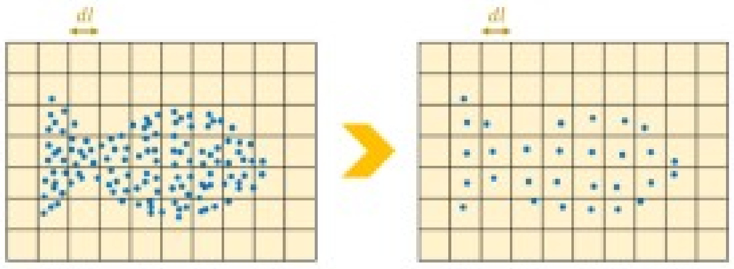


Figure 6: illustration of grid subsampling in 2D

<u>Question Bonus:</u> Implement the grid subsampling in Python and show a screenshot of the decimated indoor_scan point cloud. Comments on the difference with decimated subsampling from Part C.

In Part C, the selected points depend on their order in the data, not their spatial distribution. The density is compeltely different in different parts of the images. On the other hand, in this part, the resulting point cloud has a uniform distribution of points across the 3D space. The image looks less blurry and the structure is better preserved.