TP1 - Basic Operations and Structures on Point Clouds

NPM3D - M2 MVA

Tong ZHAO (tong.zhao@eleves.enpc.fr)

1 Point Clouds Manipulations

Q1. The obtained point clouds of original bunny and the transformed bunny are shown as below:

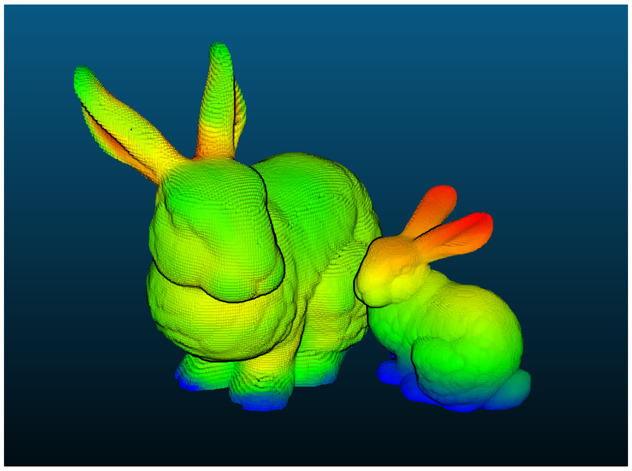


Figure 1: Bunny

2 Subsampling Methods

Q2. We show the decimated point cloud in Fig.2 and the grid subsampled point cloud in Fig.3, both of which with colors.

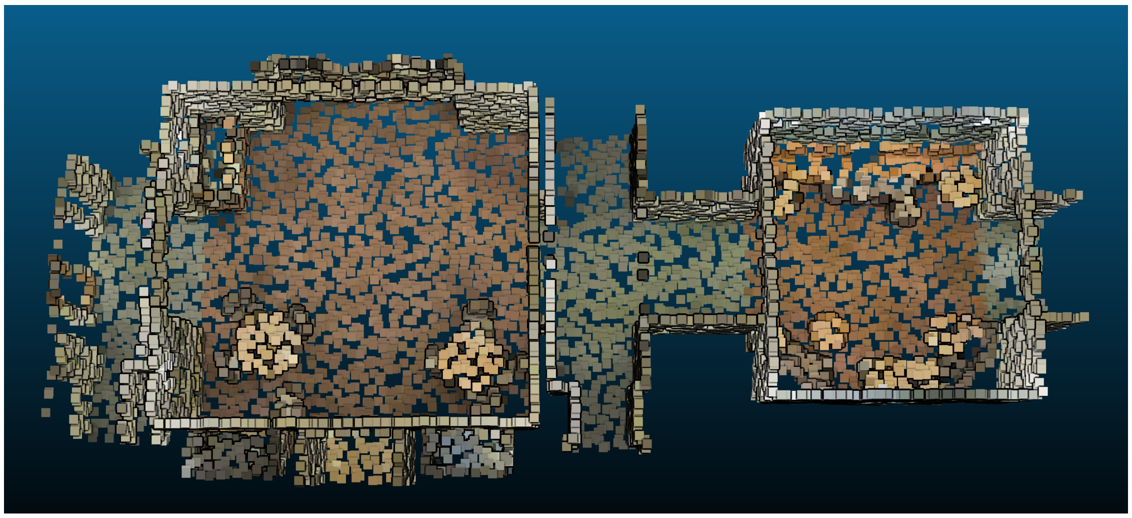


Figure 2: Decimated point cloud

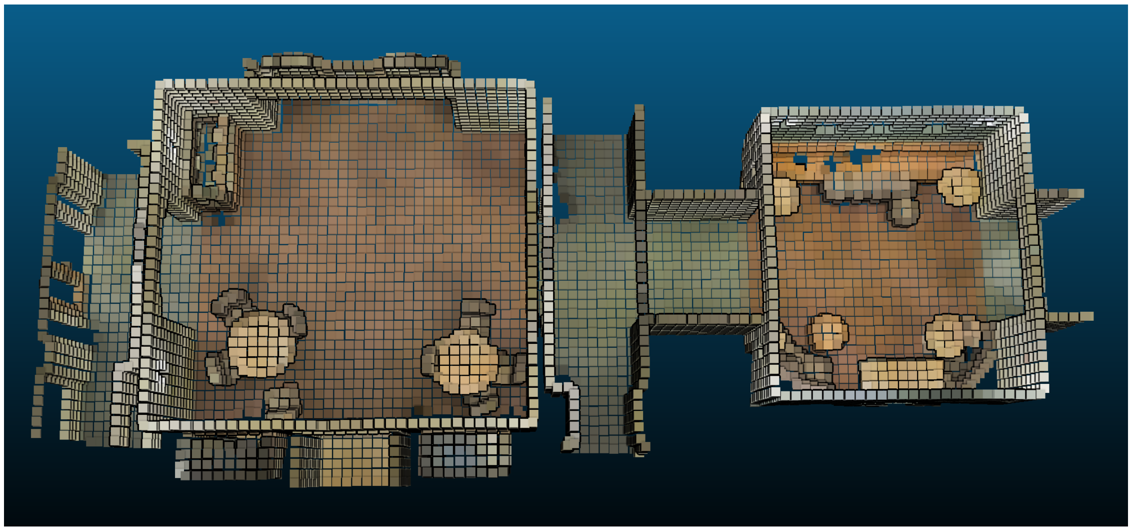


Figure 3: Grid sampled point cloud

We compare the advantages and drawbacks of two subsampling methods by two criterion, namely the execution time and the quality of generated point cloud. The first naïve decimation subsampling takes 0.003s to get sampled point cloud with colors and labels, while the grid subsampling takes 74.205s to get sampled point cloud with colors. Secondly, by comparing the obtained point clouds, we observe that the decimated point cloud has many holes and the point cloud is not uniform at all. On the contary, the grid-sampled one generates a point cloud with much higher quality.

In summary, subsampling with decimatation is quite efficient but it gives rough point cloud. Grid subsampling generates high-quality output while it is quite time-consuming.

Bonus 1. In the following, we show the point clouds with labels sampled by these two methods.

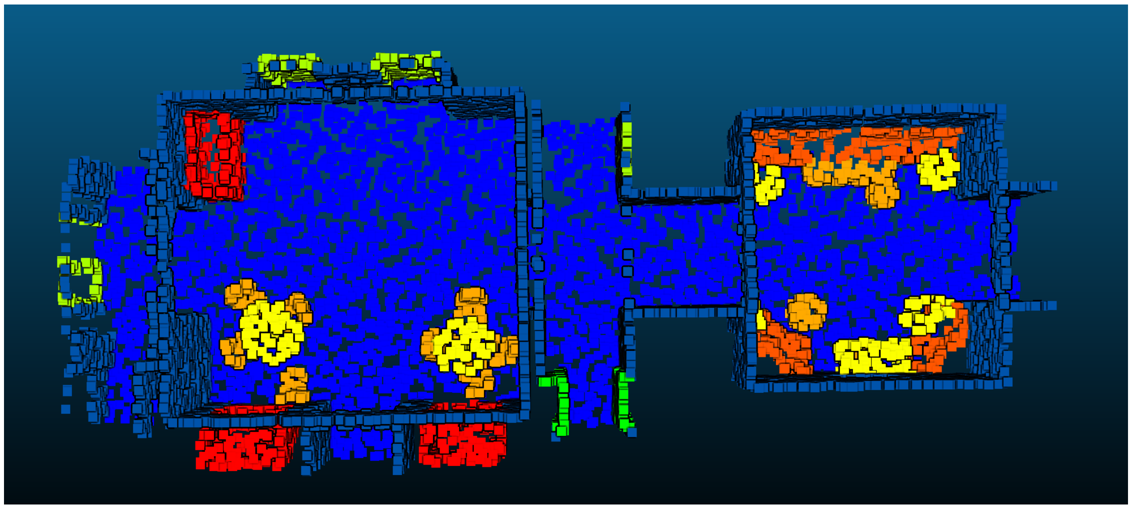


Figure 4: Decimated point ploud with labels

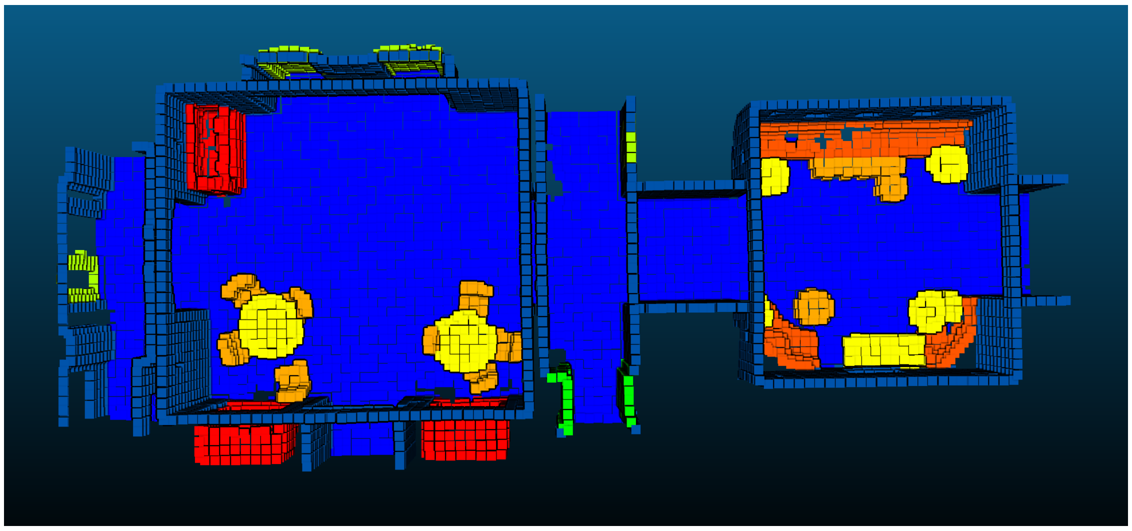
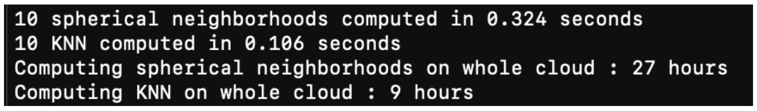


Figure 5: Grid sampled point cloud with labels

3 Structures and Neighborhoods

Q3. We search the neighbors for 10 queries with the brute-force spherical search and the brute-force KNN search, the run time is shown as following:



```
10 spherical neighborhoods computed in 0.324 seconds
10 KNN computed in 0.106 seconds
Computing spherical neighborhoods on whole cloud : 27 hours
Computing KNN on whole cloud : 9 hours
```

Figure 6: Computing time for searching

The experiment shows that two brute-force implementations are very time-consuming and also are not feasible if we need to query all points.

Bonus 2.

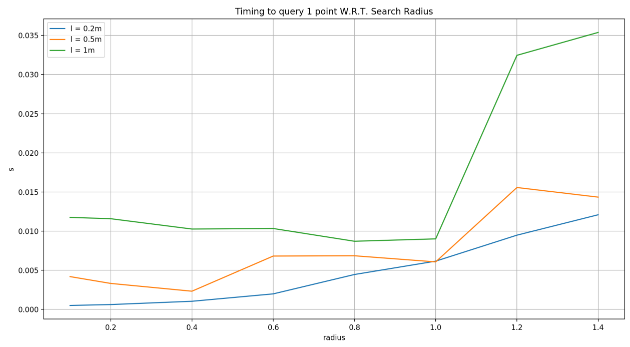


Figure 7: Influence of the ratio between voxel size and search radius

In order to evaluate the influence of the ratio between voxel size and search radius, we construct three kinds of grid whose voxel sizes are 0.20m, 0.50m, 1.00m, respectively. we observe that the optimal voxel size has a strong correlation with the queried radius. When the ratio between the radius and the voxel size is near to 1.00, the algorithm works better. So obviously, the weakness of grid structure is that when we need to query points with different radius, we need to compute another grid to optimize the querying time, which is not efficient.

Q4. We depict the connection between the leaf size and the spherical neighborhoods search as following.

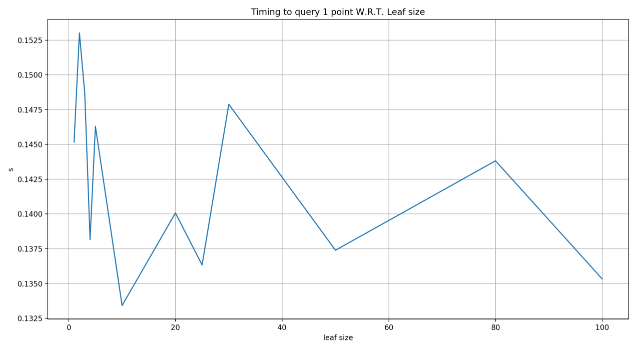


Figure 8: The time to query one point w.r.t. leaf size

According to the curve, the optimal leaf size is 10, which is not 1. In fact, the leaf size decides at which moment we move to brute-force search. If there are only a few points, the brute-force method is faster than search by going down the tree. The timing is a trade-off between search in hierarchical tree and in brute-force, so it takes more time in tree when the leaf size is too small, like 1.

Q5. We plot the timings obtained with KDTree as a function of radius as following.

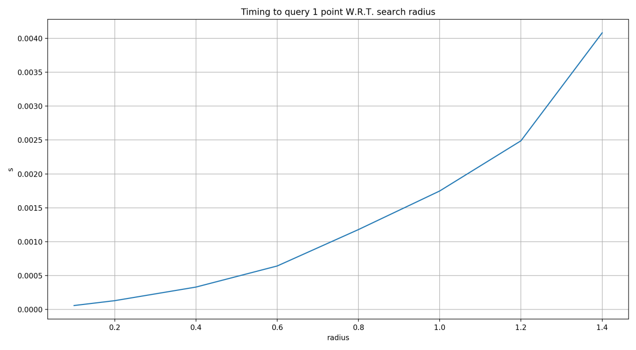


Figure 9: The time to query one point w.r.t. search radius

The querying time increases fast along with the radius increase (roughly polynomial to radius). It takes 456.486s to search for 20cm neighborhoods for all points in the point cloud.

Bonus 3 These three structures apply to different application scenarios.

The grid works well when the ratio between the voxel size and the search radius is approximately 1 and the point clouds have many voxels, meaning that the ratio between the point range and the voxel size is large. However, it is not a generic method due to these strict constraints.

Kd-tree and octree are similar structures, while they still differs in some aspects. Octree can be easily implemented with a good optimization, e.g. we can easily make an octree cache-friendly like storing all 8 children contiguously. Kd-tree is more generic (it works for arbitrary dimensions) but is inherently deeper since it splits each time for one dimension.

In conclusion, I think that octree should be a good choice for 3D data.