Computer Vision Course — A.A. 2020/2021

Lab 4: Point Clouds & Meshes

Andrea Montagner
andrea.montagner@unitn.it

What's up today

- Point Cloud introduction
- Point Cloud manipulations (pt1): affine transformations
- Point Cloud manipulations (pt2): down-sampling, normal estimation, cropping, differencing, filtering
- Surface reconstruction: Ball-Pivoting algorithm

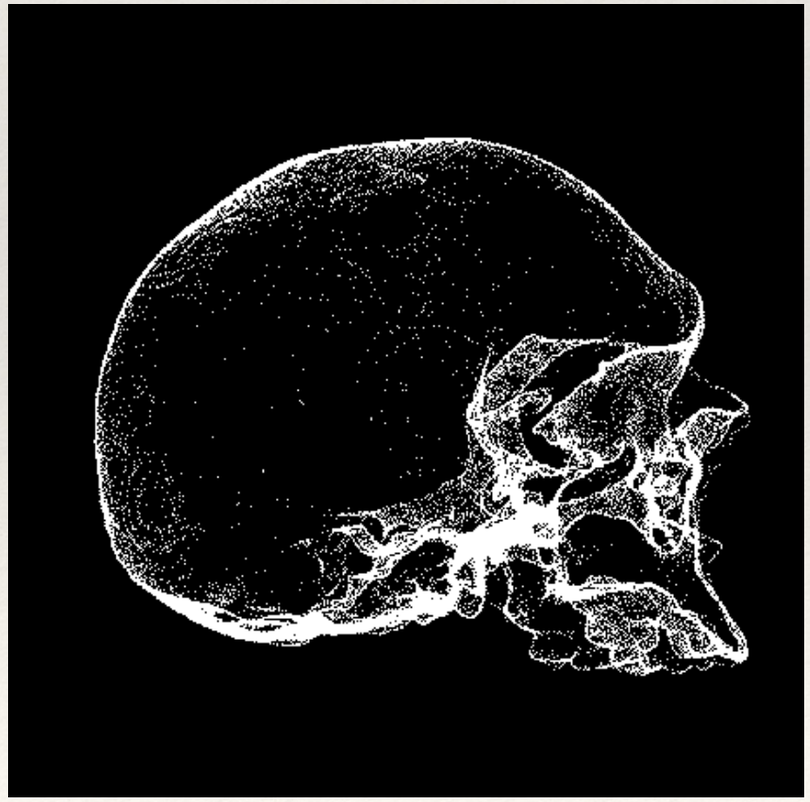
Point Clouds introduction

A "cloud" P of nD points (usually $n = 3$) is a set of points which represent a shape or an object.

- Each point is generally defined by three coordinates (x ,y, z)
- It is a structured intended mainly to represent external surfaces
- It can embed also additional information, such as color
- It is an *UNORGANIZED STRUCTURE*

$$p = (x_i, y_i, z_i)$$

$$P = (p_1, p_2, \ldots p_i, \ldots, p_n)$$



Point Clouds introduction

For this laboratory we will use Open3D, an open-source library for 3D data manipulation.

- Python API <— our case
- C++ API

It should be already installed in the VM, in case you want to install it in your own machine

- Can be installed with a package manager like *pip* (for ubuntu) or *brew* (for MacOS)
- Can be build from source, instructions here:

Manipulation pt1: Read point clouds

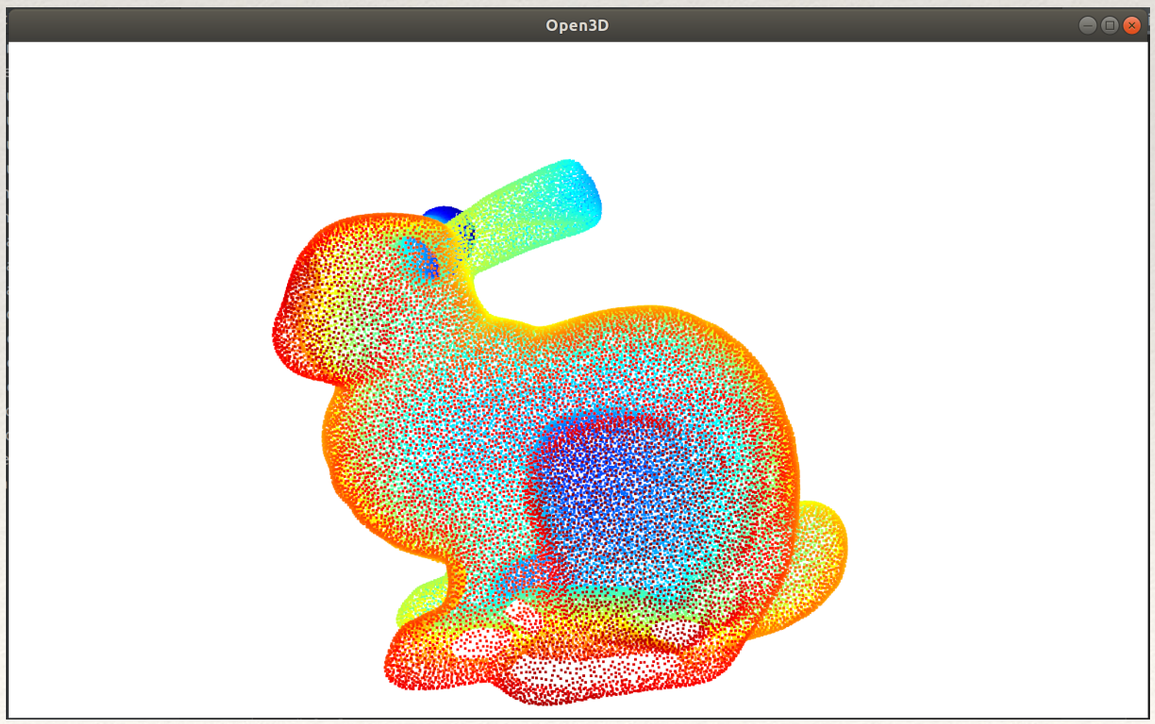
- Open3D offers a very simple and straightforward way to read a point cloud:

```
o3d.io.read_point_cloud(“path/to/point_cloud.ply“)
```

- Then it’s time to visualize the opened point cloud:

```
o3d.visualization.draw_geometries([pc])
```

which takes as input the list of objects to display



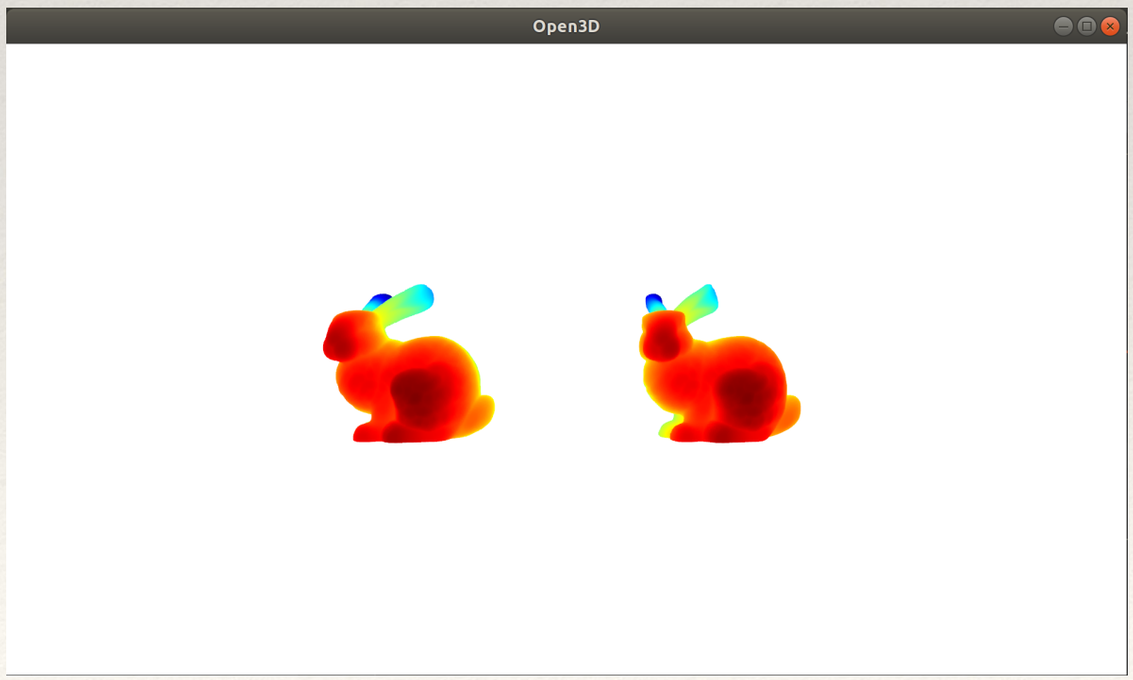
Manipulation pt1: Translation

A point cloud can be subject to any rigid transformation. The easiest is translation.

- To translate a point cloud means apply the translation vector to all its points
- Given a translation vector **t** and a point cloud **P**, we have that:

$$P + t = (p_1 + t, p_2 + t, \ldots, p_i + t, \ldots p_n + t) \quad \text{where} \quad p_i + t = (p_x + t_x, p_y + t_y, p_z + t_z)$$

- `PointCloud.translate(translate_vector)`
  - `translate_vector` (`numpy.ndarray[float64[3,1]]`) the translation vector



Manipulation pt1: Rotation

Rotation follows the same principles of translation.

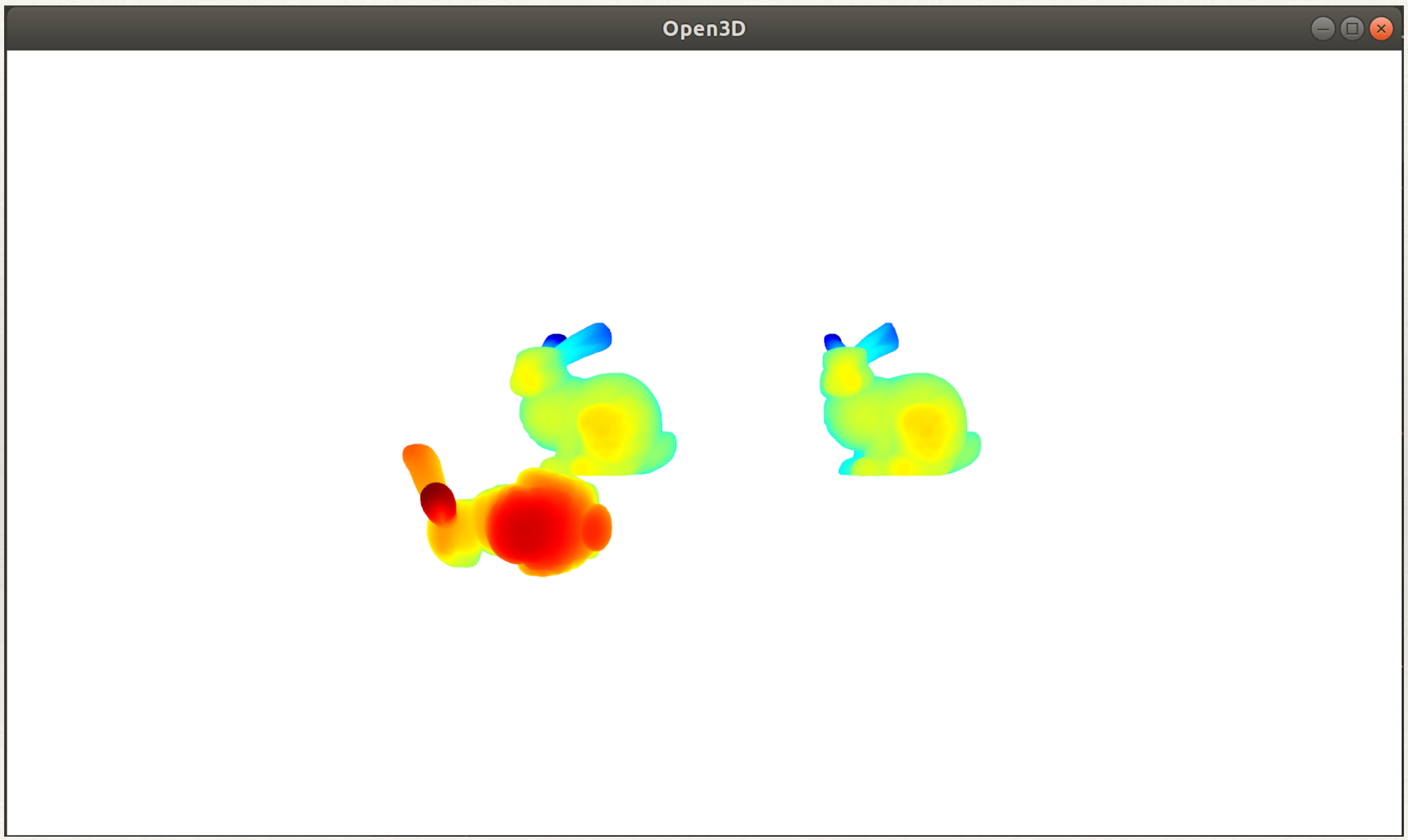
- To rotate a point cloud means apply the rotation matrix to all its points wrt a given center

$$R \times P = (R \times p_1, R \times p_2, \dots R \times p_i, \dots R \times p_n)$$

$$\text{where } R \times p_i = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{33} & r_{33} \end{pmatrix} \times \begin{pmatrix} p_x^i \\ p_y^i \\ p_z^i \end{pmatrix} = P_i^T$$

- `PointCloud.rotate(R, center)`
  - *R* (*numpy.ndarray[float64[3,3]]*): the rotation matrix
  - *center (numpy.ndarray[float64[3,1]])*: the rotation center

Manipulation pt1: Rotation



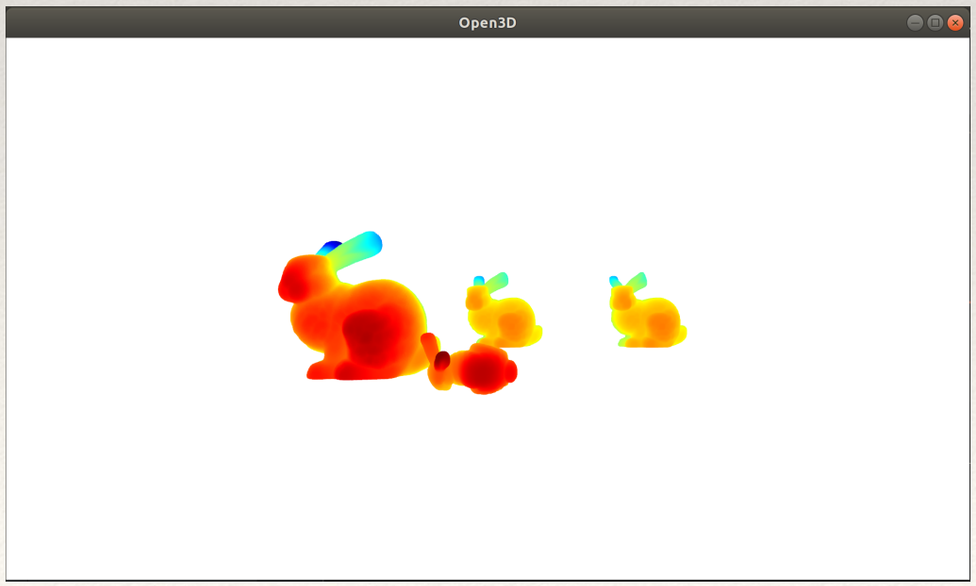
Manipulation pt1: Scale

Same considerations for the scaling operation.

- To scale a point cloud means apply the scaling factor to all its points
- If we need to double the size of a point cloud **P**, we have:

$$sP = (sP_1, sP_2, \ldots sP_i, \ldots sP_n)$$

- `PointCloud.scale(scale, center)`
  - `scale` (*`float`*): the scaling factor
  - *`center (numpy.ndarray[float64[3,1]])`*: the scaling center



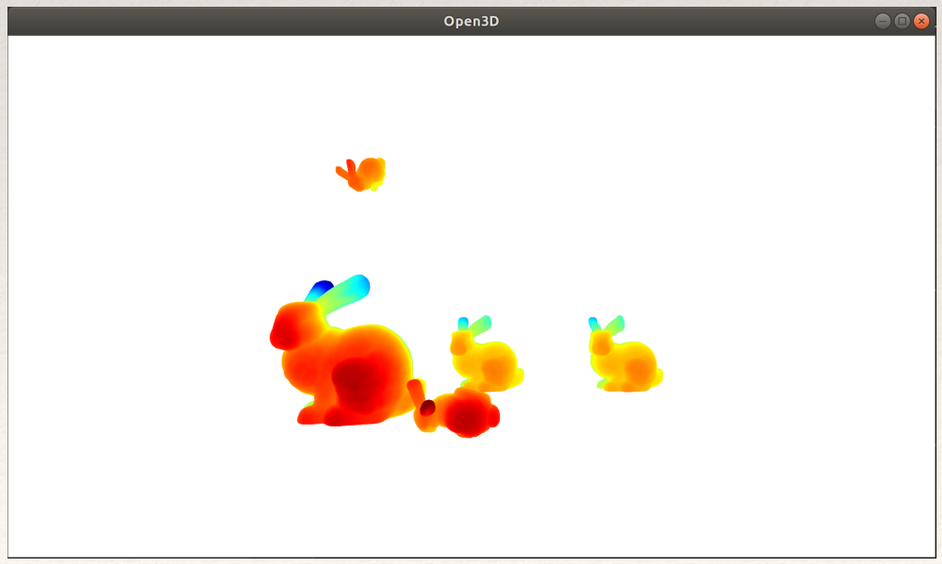
Manipulation pt1: Affine transformation

As you already know, it is possible to combine these three transformation in a single matrix.

$$A = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{33} & r_{33} & t_z \\ 0 & 0 & 0 & s \end{pmatrix}$$

which can be applied also to a point cloud

- `PointCloud.transform(T)`
  - `T` (*`numpy.ndarray[float64[4, 4]]`*): the general affine matrix to apply



Manipulation pt2: Downsampling

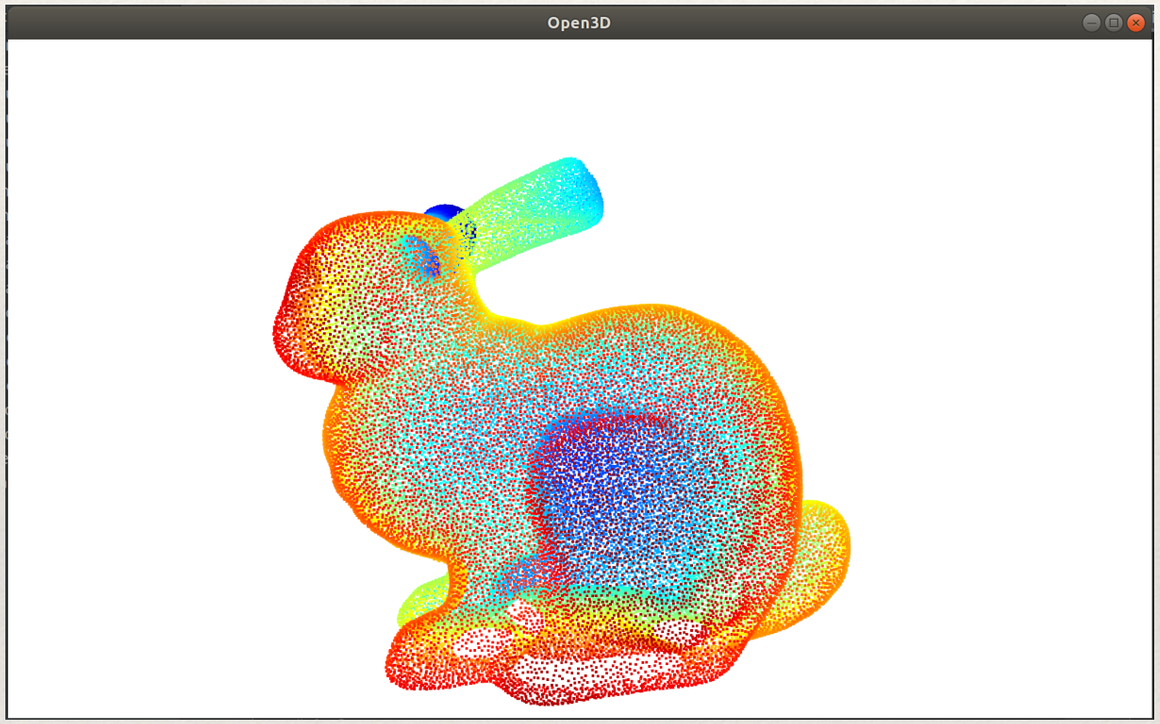
Point clouds can be very big, with a huge number of points, sometimes too many for what it is required. Reducing the density (aka the number of points) of a point cloud is called *downsampling*.

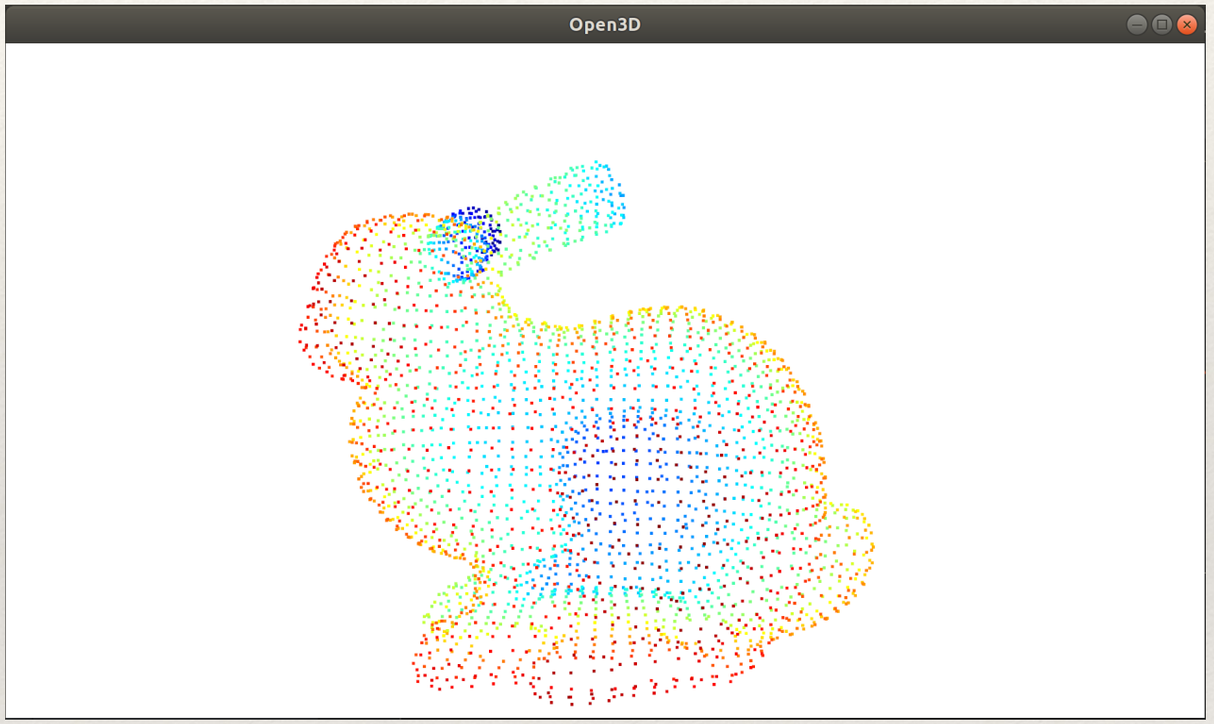
- Two main ways of downsampling:
  - ***Uniform***: uniformly reduce the number of points, eliminating a point every k points. Possible high risk to lose important information.
  - ***Voxel based***: reduce the number of points by creating a voxel (volumetric pixel) grid and approximating all points inside the voxel with its centroid.

- `PointCloud.uniform_down_sample(every_k_points)`
  - `every_k_point` (*`int`*): the sample rate, selected point indices are [0, k, 2k, …]

- `PointCloud.voxel_down_sample(voxel_size)`
  - `voxel_size` (*`float`*): voxel size to downsample into

Manipulation pt2: Downsampling





Manipulation pt2: Normals

Normals estimation can be an arbitrary complex problem to solve. The simplest reduction can be formulated as follows:

> *"The computation of the normal of a point on the surface is approximated to the problem of computing the normal of a tangent plane on the surface on that point"*

FYI: The solution is a complex mathematical problem which involves analysis of the PCA (Principal Component Analysis) of a covariance matrix created from the nearest neighbours of the point (not the target of this lab).

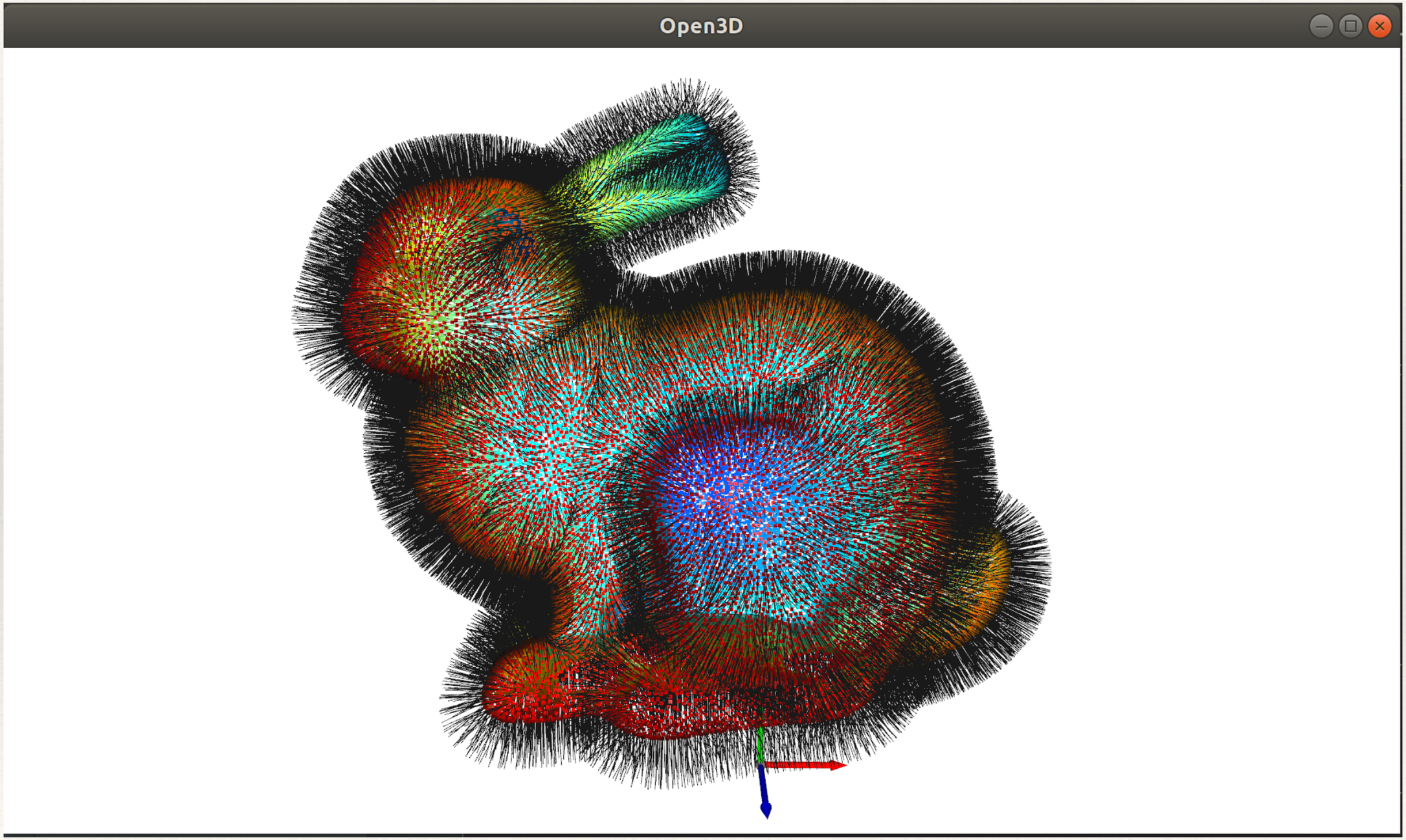
Two main problems with normals:

- ***Sign***: is the problem of finding the right orientation of the normals
- ***Scale factor***: is the problem of choosing the right number of *k* nearest points or the radius *r* for determining surface where the normal will be computed.

Manipulation pt2: Normals

- `PointCloud.estimate_normals(search_param)`
  - `search_param (`*`KDTreeSearchParamHybrid`*`)`: search parameters for neighbourhood search
- `PointCloud.normalize_normals()`
- `PointCloud.orient_normals_towards_tangent_plane(k)`
  - `k (`*`int`*`)`: Number of k nearest neighbours used in constructing the graph used to propogate normal orientation.
- `PointCloud.orient_normals_to_align_with_direction(orientation)`
  - `orientation (`*`numpy.ndarray[float64[3, 1]]`*`)`: Normals are oriented with respect to *orientation*.

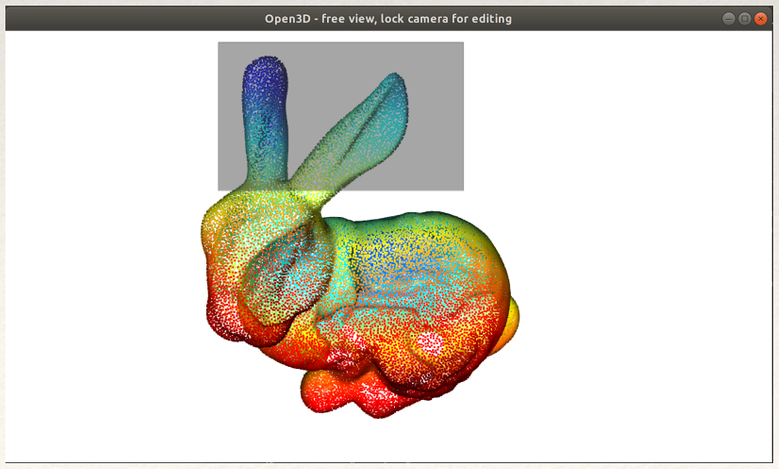
Manipulation pt2: Normals

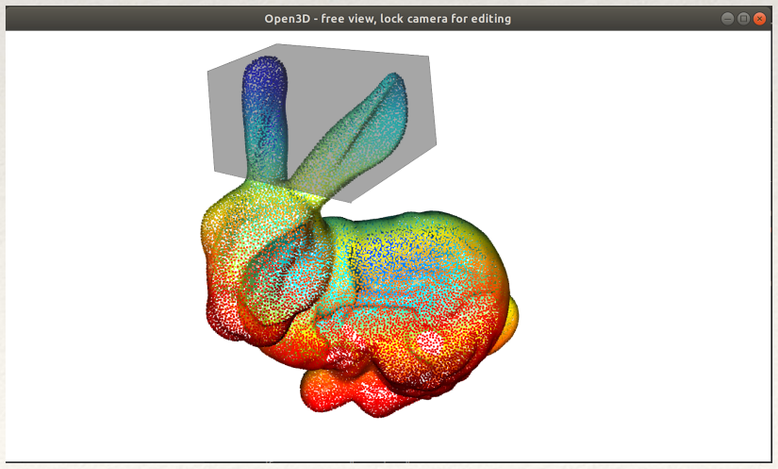


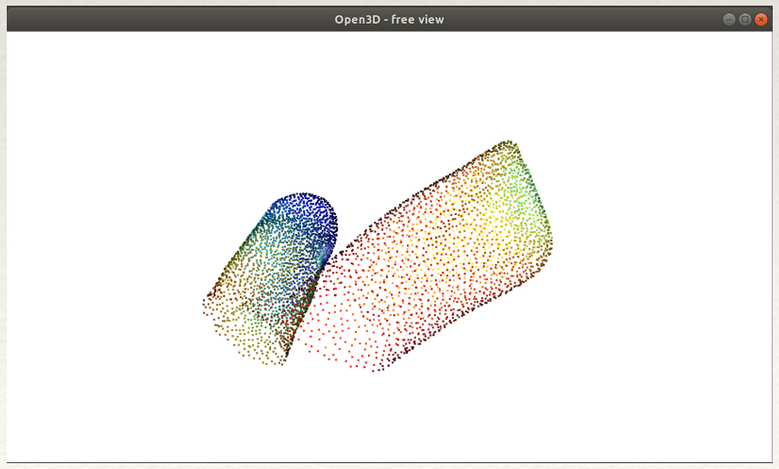
Manipulation pt2: Cropping

Sometimes we need to extract objects of interest from large acquisitions. To do this operation, we need to *crop* the point cloud.

- Open3D offers a fast interactive way to crop point clouds and save the cropped parts with the correct file type.
- `o3d.visualization.draw_geometries_with_editing([pc_to_crop])`
  - Find right view, then press *k*
  - Select part to crop:
    - Rectangular selection: `drag and drop with mouse`
    - Polygonal selection: `ctrl (hold) + mouse clicks`
  - Press `c` to save the points within the selection into a new point cloud







Manipulation pt2: Difference

Once we have a crop and the whole point cloud, it is also possible to compute the difference between them.

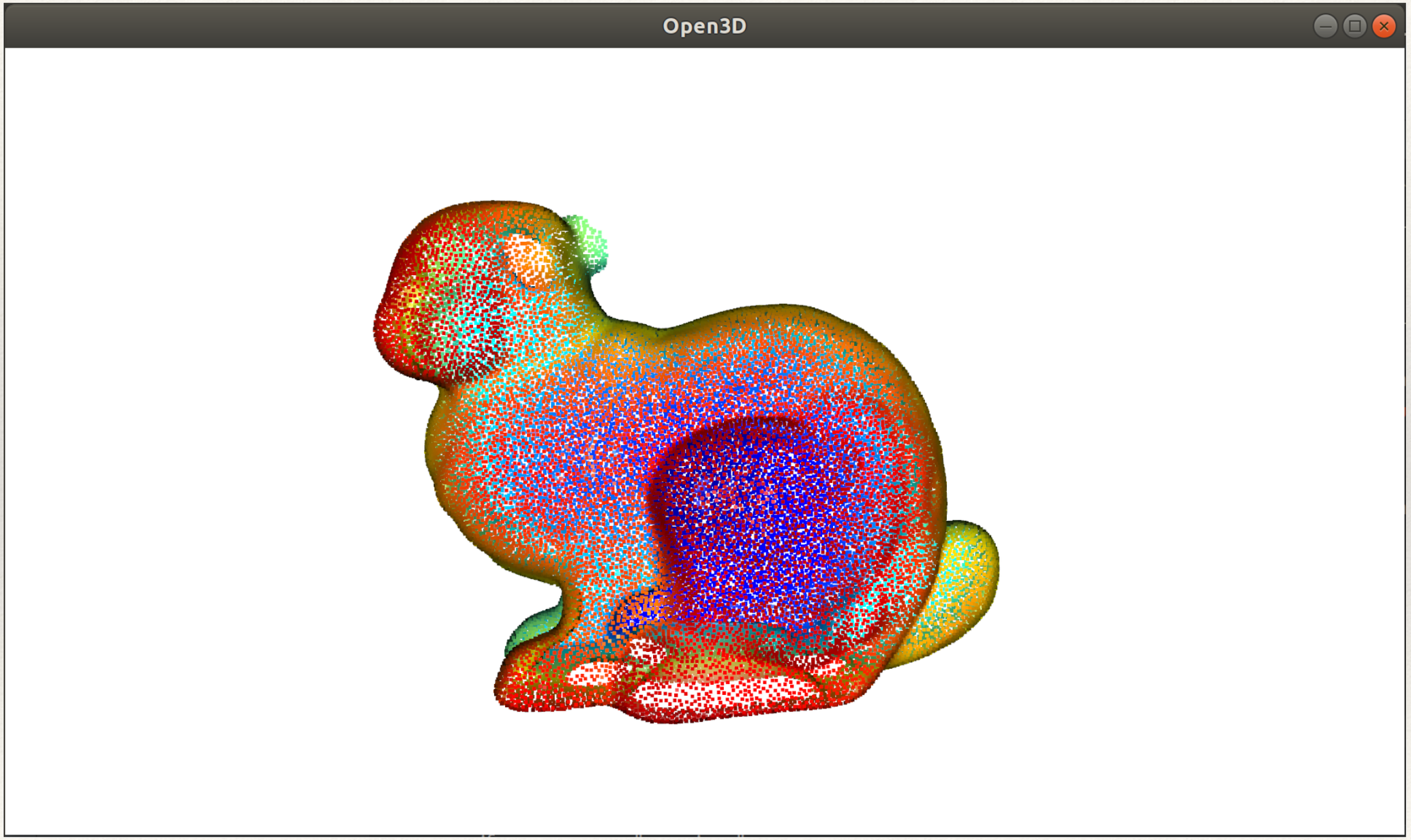
- This is done computing the distance between to point clouds and filtering points with a distance lower than a threshold

- `PointCloud.compute_point_cloud_distance(target)`
  - `target` (*`open3d.cpu.pybind.geometry.PointCloud`*): the target point cloud
  - Returns a double vector with the distances of each points to the target point cloud

- Then we filter the points according to the distance value (using `numpy.where`) getting the indices of those points with a distance greater then a threshold

- `PointCloud.select_by_index(ind)`
  - `ind` (*`List[int]`*): Indices of points to be selected.

Manipulation pt2: Difference



Manipulation pt2: Outlier removal

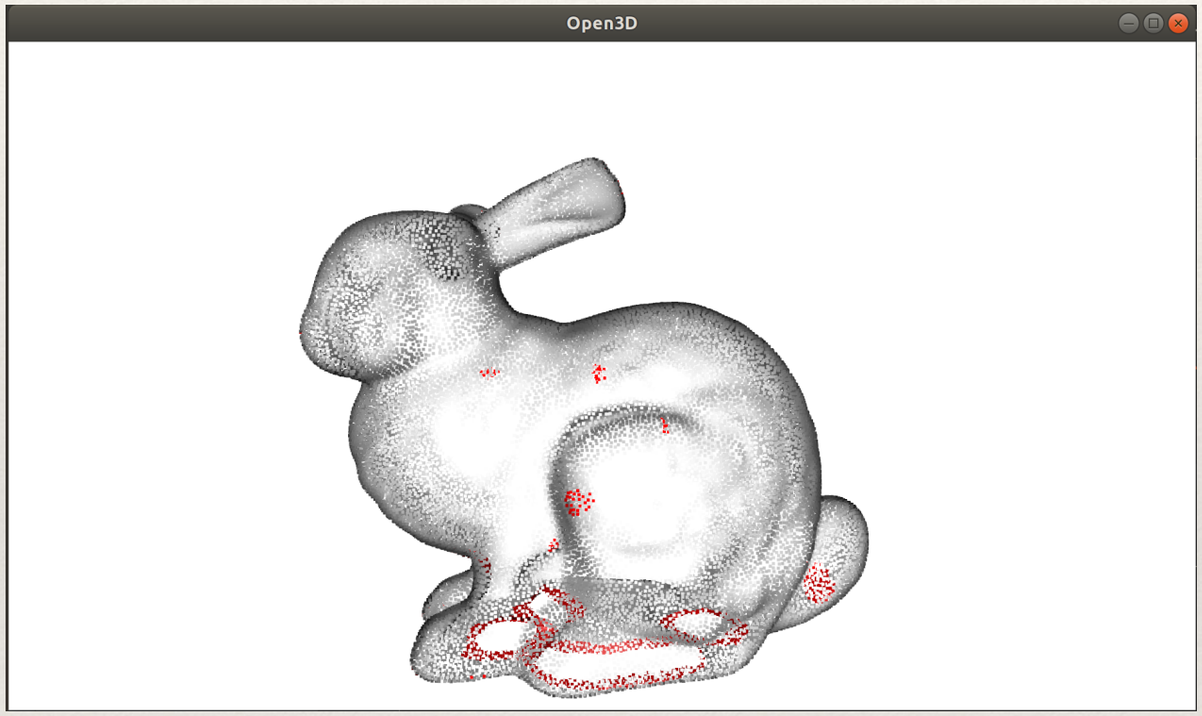
Acquired point clouds tend to contain noise and artifacts, which we might want to remove.

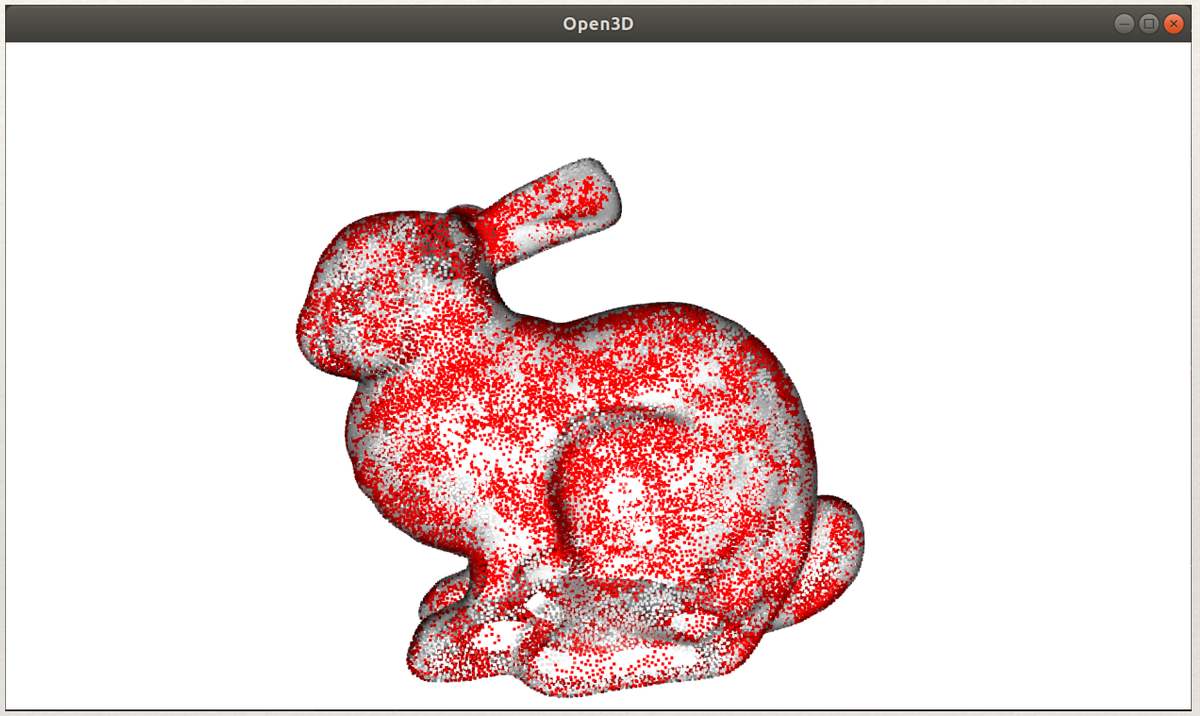
- There are two main types of outlier removal methods:
  - ***Statistical based***: removes points that are further away from their neighbours compared to the average for the point cloud.
  - ***Radius based***: removes points that have few neighbours in a given sphere around them

- `PointCloud.statistical_outlier_removal(neighbours, std)`
  - `neighbours (`*`int`*`)`: Number of neighbors around the target point
  - `std (`*`float`*`)`: Standard deviation ratio.

- `PointCloud.radius_outlier_removal(points, radius)`
  - `points (`*`int`*`)`: Number of points within the radius.
  - `radius (float)`: Radius of the sphere.

Manipulation pt2: Outlier removal



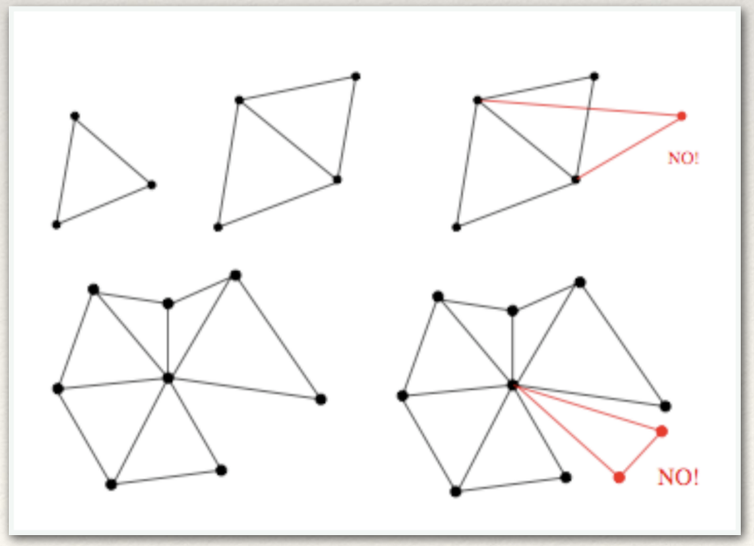


Surface reconstruction: Meshes

Point clouds are unorganised structures. They lack a series of information, such as global connectivity which become very hard to compute.

- To get these information we need to move to organised structures, such as *meshes*.
- ***Triangular mesh***:
  - (*Easy*) Set of triangles where one edge can belong to maximum two triangles,
  - (*Medium*) A mesh M is a tuple (V, A) where $V = \{v_i \in R^3 \mid i = 1,...,n\}$ is the set of vertices and A is the adjacency matrix of each vertex,
  - (*Theoretical, but hard*) A triangular mesh is a pure simplicity 2-complex and also a bidimensionale manifold with boundary. <— *This is not the target of this class*

- Some properties of meshes:
  - Mesh **boundary**: a closed sequence of edges, also called **loop**
  - If there are no boundary edges, the mesh is **closed**
  - Mesh **orientation** is given by the cyclic order of its vertices.
    The convention is that faces where vertices are anti-clock wise is the front, clock wise is the back

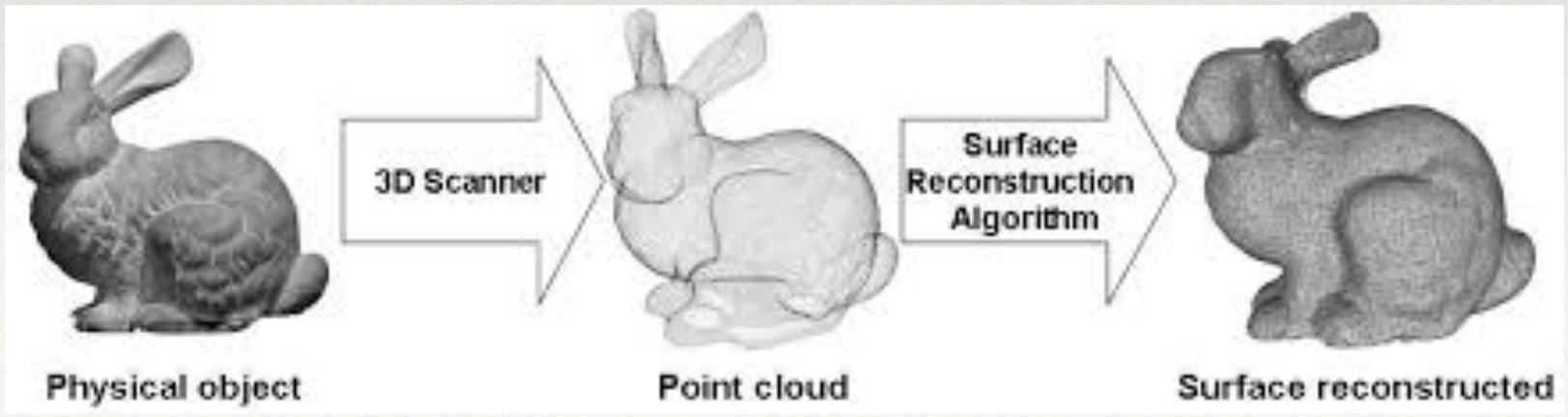


Surface reconstruction: Ball-Pivoting

The operation of creating a mesh from a set of points is called *surface reconstruction*.
Typically, there are three main steps to follow:

- ***Pre-processing***: phase where erroneous data are eliminated or point cloud is filtered/ sampled to reduce computation time
- ***Triangulation***: core part of the reconstruction, phase where the points are actually converted into a consistent polygonal shape
- ***Post-processing***: after the model is created, it is usually edited for refinement of the polygonal surface

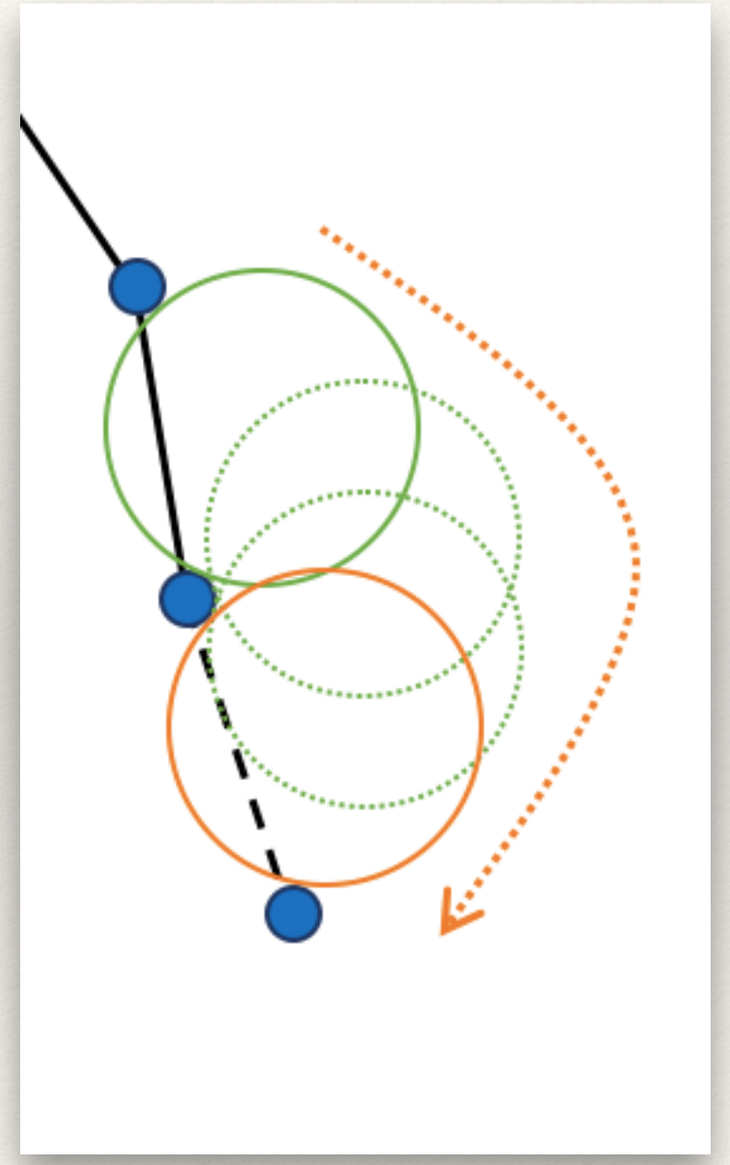
There are different triangulation techniques, today we will see one of the most common, called **Ball-Pivoting algorithm**.



Surface reconstruction: Ball-Pivoting

The name derives from the use of a virtual ball to help the reconstruction.

- Main idea:
  - Obtain a point cloud *P,*
  - Assume P is dense enough for a ρ-ball (ρ is the radius) not to pass without touching any point
  - Start by placing the ball in contact with three points
  - Then, keeping the ball touching two of these points, pivot the ball until it touches another point
  - Repeat until the ball does not find any valid point

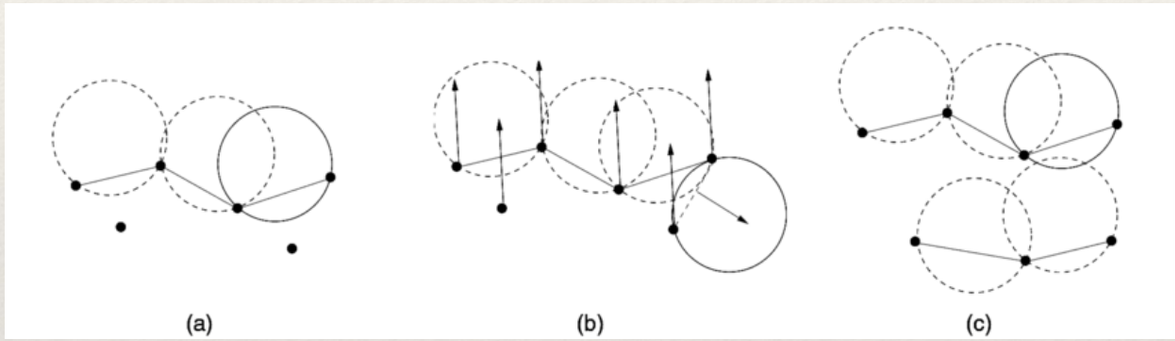


- Triplets of point touched by the ball form new triangles.

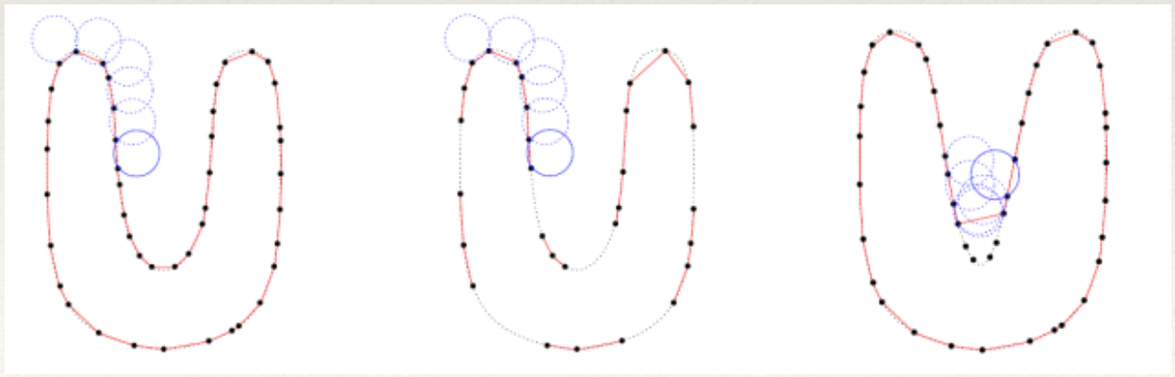
- Sets of triangles formed while the ball “walks” constitutes the mesh.

Surface reconstruction: Ball-Pivoting

- Noisy point clouds can lead to unwanted behaviour of the ball pivoting algorithm:
  a) Surface samples lying "below" surface level are discarded by the algorithm
  b) In case of pivoting around an edge, check if the data points normals are consistently oriented to decide if generate or not that triangle
  c) Presence of spurious components, that can be eventually filtered in post-processing



- Another crucial factor is the ***sample density***:
  a) If points are correctly sampled, the ball will be able to visit all points
  b) If density is too low, some edges will not be created leaving holes
  c) If the curvature is larger then $1/\rho$, the ball will not reach some samples and some feature will be missing



Surface reconstruction: Ball-Pivoting

- If the number of points of the cloud is very high, ball pivoting is usually run multiple times with a set of different radii of the ball. The starting value will be the smallest one, iteratively increasing until all values are used.

- When working with multiple radii, there are some empirical conventions (called *scale factors*) for scaling the ρ value:
  - Factor 2: $\{0.5\rho, 1\rho, 2\rho\}$
  - Factor 10: $\{0.1\rho, 1\rho, 10\rho\}$

- `create_from_point_cloud_ball_pivoting(pcd, radii)`
  - `pcd` (*`open3d.cpu.pybind.geometry.PointCloud`*): PointCloud from which the TriangleMesh surface is reconstructed. Has to contain normals.
  - `radii` (*`open3d.cpu.pybind.utility.DoubleVector`*): he radii of the ball that are used for the surface reconstruction.

Exercise

Let's try to face a real world scenario:

- Open the point cloud *panoptic.ply, t*his is a real acquisition from the CMU Panoptic Studio dataset
- *Main goal*: reconstruct the mesh of the person
- *How to achieve it*: try to use all previously explained tools to manipulate the point cloud in order to isolate only the human figure and then use BPA to create the mesh.

TIP: in order to correctly align the mesh with the right axis you can visualise a coordinate frame to help you.

- `o3d.geometry.TriangleMesh.create_coordinate_frame(size, origin)`