3D Mesh Retrieval System

Multimedia Retrieval
Utrecht University

Da Costa Barros, Fabien [0720823]
f.n.dacostabarros@uu.students.nl

Bagheri, Soheil [6208908]
s.bagheri@students.uu.nl

October 11, 2022

Introduction

With the advance in modeling software and 3D scanners, the speed of 3D content creation increased rapidly. Nowadays, much 3D content has been produced, and people are still creating many more objects constantly. These kinds of contents are usually stored in so-called shape databases. But, there is a challenge to browse and find the 3D objects of interest within these numerous amounts of data.

Several techniques have already been presented for finding desired data in these shape databases. For example, searching by keyword, browsing the database along a few predefined categories, or content-based shape retrieval (CBSR). Although the first two options do not need a sample model for searching in a database, they need more effort to label or categorize these contents. On the other hand, there is content-based shape retrieval which can retrieve all the similar samples to the existing query shape.

In this project, we want to categorize the 3D objects of a database based on their shape similarity. This process will be done by normalizing all the objects and extracting some general features from them. Then, we can divide similar objects based on their similarity in their features. Also, there would be a content-based shape retrieval that can find similar 3D objects based on the feature extracted from the given query shape and compared to other objects in the database.

1 Selection and setup the work environment

In this section, four different related packages have been tested and compared. The Comparison of these packages has been done based on several features that might be useful in the future sections of this project. For example, being able to re-mesh objects, scaling them, rotating them, and moving them are the feature that can be useful in the process of normalizing the 3d objects in the database. Also, having some information about the 3d objects, like the number of vertices and faces, is necessary for understanding if these 3d objects are normalized or not. On the other hand, this package must support different formats that are usually used to store 3d objects. In this case, we narrowed down the packages that work with Python language and well-known between the community. So, we tested and analyzed "Pymesh", "PyMeshLab", "TriMesh", and "Open3D". Results can be found in the table 1.

We wanted to be able to open "PLY", "OFF", "OBJ", and "STL" formats. "PLY" and "OFF" formats were discussed in the lecture, while "OBJ" and "STL" are really common. All of them were able to open "PLY", "OFF", "OBJ", and "STL" formats, so that was not really discriminating.

We tried to compare the different re-meshing, repairing and extraction features with our current knowledge and thought that PyMeshLab and Open3D seems to be quite good. The complexity was also one of our requirements as from previous experience less complex library offers less options.

After achieving step 1 with four of those libraries, we try to find a balance between number of dependencies, options proposed for visualizing and analysis. Then we decide to choose "PyMeshLab"[1] Our project has three main dependencies. "PyMeshLab", "Polyscope", and "Numpy", and it runs with Python 3.9.

Features category	Desired Fetures	PyMesh	PyMeshLab	TriMesh	Open3D
3D format supported	PLY	X	X	X	X
	OFF	X	X	X	X
	OBJ	X	X	X	X
	STL	X	X	X	X
Repairing	Isolated Vertices	X	X	X	To implement
	Duplicated Vertices	X	X	X	To implement
	Duplicated Faces	X	X	X	To implement
Remeshing	Uniform sampling	X	X	X	X
Analysis features		Simple	Rich	Sufficient	Rich
Visualisation features	Scale	0	X	X	X
	Pan	0	X	X	X
	Rotate	0	X	X	X
	Screenshot	0	X	0	X
	Multiple mesh	0	X	0	X
Language	Source code	70% C++ 20% Python 10% C	80% C++ 20%Python	100% Python	80% C++ 10% Python 10% Cuda
Number of dependencies		7	2	3	0
Interface of preview windows		None	Advanced	Minimalistic	
Visualisation interface	Shading methods	None	Flat	Flat	Flat
Installation tools	Pip or Anaconda	Docker	Pip	Pip	Pip

Table 1: In-built features for different Python libraries tab:libraries

2 Rendering

We have two scripts for this step called "main.py" and "render.py" Where "main.py" verifies the console arguments and calls the render function from "render.py". The first argument passed through is the path of the file.

In "render.py", there is a function called "render". This function loads the mesh and displays the vertices and the mesh of the 3D object. PyMeshLab has a built-in function ".show_polyscope" that shows a mesh. While we found it more interesting to override it with polyscope functions to have more control if we need to upgrade anything. The result of the mesh visualization script can be found in Figure fig:ant-mesh 1. The 3D mesh visualization windows offered many functionalities.

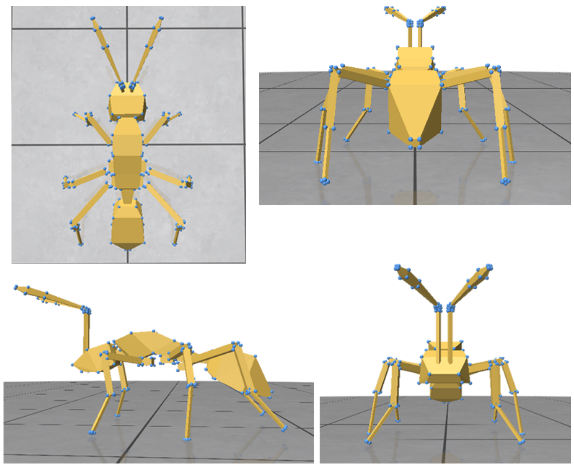

Figure 1: Visualisation of an ant mesh fig:ant-mesh

3 Shape analysis

To avoid too many disparities between the properties of 3D objects in the database, we wanted to normalize some features to extract them easier in the following step. So, we implemented some functions to extract generic information and normalize them from all 3D objects of the database. These features are the number of vertices, the position, the scale, the alignment, the orientation, the location of the barycenter, the size of the bounding box, the vector given by principal component analysis, and the second-order moment of mass.

Also, we had to determine an order for all the different extracting and normalizing steps, as some of them can have an impact on the following one. In general, any transformation T has to be performed before T' if T has a side effect on the properties that T' has to normalize. First of all, we began with the numbers of vertices and how they were sampled to obtain a uniform mesh from the poorly sampled ones. Because this action optimizes some computations that are dependent on the number or position of the vertices. Second, we normalized the positions of 3D objects as it does not impact other properties. As any rotation impacts the orientation of the bounding box, we applied alignment with the axis given by PCA in the next step. Then, we did the flipping in fourth place. In the end, we scaled the 3D objects to the unit cube.

We discuss further details for each feature in the following.

3.1 Vertex numbers

3.2 Position

One way of normalizing the position is to translate the barycenter to the origin of the 3D space. As PyMeshLab has a built-in function for translation, we only needed the barycenters of the 3D objects to be calculated. Usually, we had to compute the average position of the center of all faces weighted by the area of that face because the vertices are non-uniformly distributed over the mesh. However, we ensured in section ?? that vertices spread uniformly around the mesh. With this assumption, we can just compute the average x,y, and z of all the vertices of each 3D object. Here is the formula to compute the barycenter $(i \in \{x,y,z\})$:

$$b_i = \frac{\sum_{v \in V} v_i}{|V|}$$

(with i the i-th coordinate of v $(i \in \{x,y,z\})$and V the set of vertex of the mesh.)

Once the barycenter was computed we translated each vertex by the $\overrightarrow{BO}$ vector (with B the barycenter and O the origin of the 3D reference space). After that, we compared the distance of the barycenter to the origin for each 3D object before and after normalization.

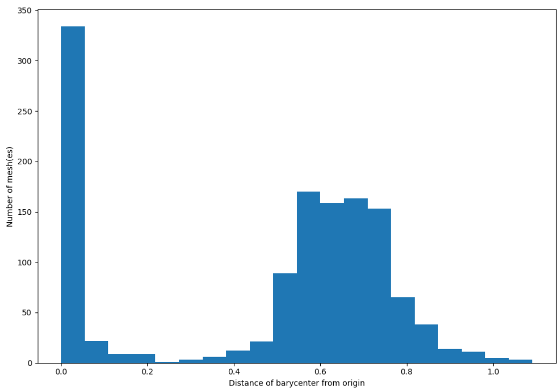

Figure 2: Distances of the barycenters of 3D objects to the origin of 3D space before normalization

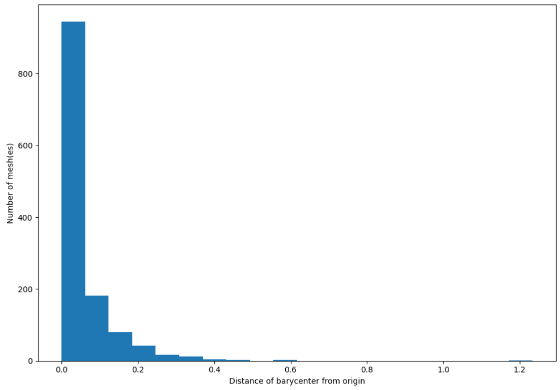

Figure 3: Distances of the barycenters of 3D objects to the origin of 3D space after normalization

As we can see after normalization (see Fig.3), all the 3D objects have their barycenter at or close to the origin.

3.3 Rotation

We used a PyMeshLab built-in function to compute the eigenvector of the 3D mesh and align the shape with that vector. The function computes the eigenvector from the co-variance matrix which can be described with the following formula :

$$C = \begin{pmatrix} \sigma(x,x) & \sigma(x,y) & \sigma(x,z) \\ \sigma(y,x) & \sigma(y,y) & \sigma(y,z) \\ \sigma(z,x) & \sigma(z,y) & \sigma(z,z) \end{pmatrix}$$

where $\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})$ the covariance of x with respect to y

To implement the rotation we could just compute the coordinate $(\overrightarrow{OV}.E_x, \overrightarrow{OV}.E_y, \overrightarrow{OV}.E_z)$ (where O is the origin and V is the vertex that we transfer to the new coordinate)

The effect of rotation normalization can be found in (see Figure 5.

It can be seen that after normalization the axis where the vertices are the most spread is aligned with the x-axis. It means that after normalization the main eigenvector given by the PCA looks like (1,0,0), therefore the 3D object is oriented as we expected.

3.4 Orientation

We now need to normalize the orientation. Determine if flipping along an axis is needed based on the moment test. We compute the second-order moment of the area with respect to a given axis, if it is negative we rotate the mesh of 180°

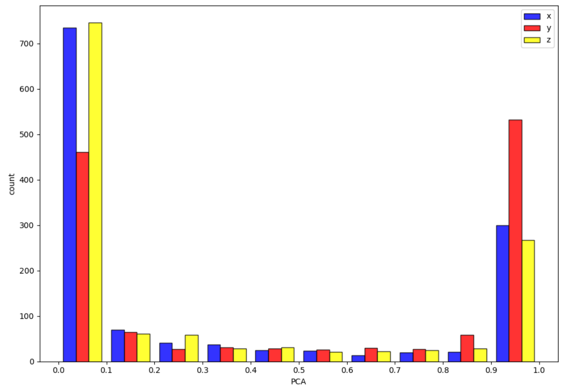

Figure 4: Value of the main eigenvector for the initial 3D mesh database

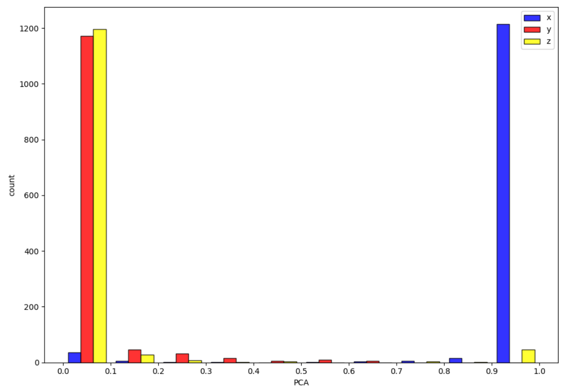

Figure 5: Value of the main eigenvector after normalisation

around this axis to inverse the momentum. The second-order moment of the area with respect to a given axis is computed with this given formula :

$$f_i = \sum_t sign(c_{t,i})(c_{t,i})^2$$

(with $c_{t,i}$ the i-th coordinate of center of triangle T ($i \in \{x, y, z\}$)

3.5 Scaling

The objective is to scale 3D objects to fit in a unit cube. As we wanted all axis to be scaled with a coefficient, we considered that the longest side of the bounding box has to be one, so other sides might be lower or equal to one. We compute two points that will describe the bounding box as follows:

pymeshlab

$$min\_box = (\underset{v\in V}{min}(v_x), \underset{v\in V}{min}(v_y), \underset{v\in V}{min}(v_z))$$

and

polyscope

$$max\_box = (\underset{v\in V}{max}(v_x), \underset{v\in V}{max}(v_y), \underset{v\in V}{max}(v_z))$$

with V the set of vertex of the mesh. We define the length of the bounding box as following:

$$|Bounding\_box| = max(\underset{v\in V}{max\_box_i}) - \underset{v\in V}{max\_box_i}))$$

(with i the i-th coordinate of v ($i \in \{x, y, z\}$)and V the set of vertices of a mesh.) The effect of this scale normalization can be seen in Figure 7.

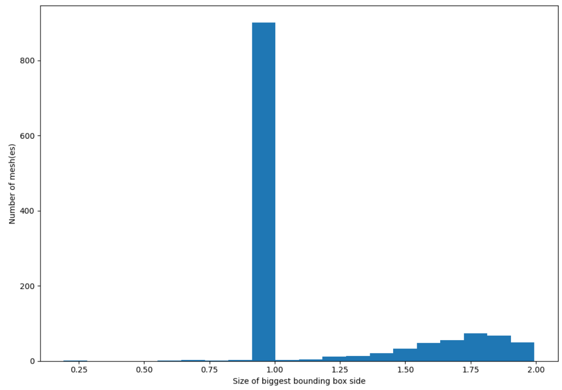

Figure 6: size of biggest bounding box side for the initial 3D mesh database

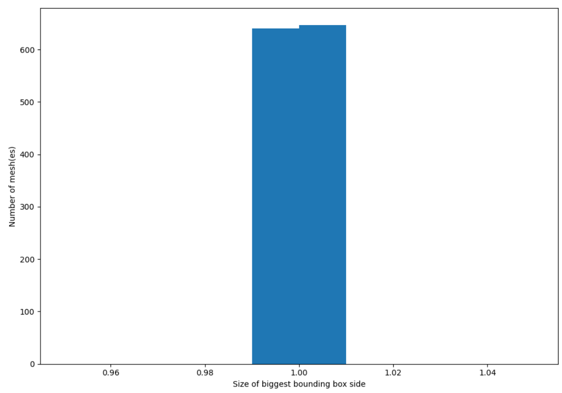

Figure 7: Size of biggest bounding box side after normalisation

In comparison with the initial database (see Figure 6), we can deduce that the size of the bounding boxes has been normalized around one.

References

[1] Alessandro Muntoni and Paolo Cignoni. *PyMeshLab*. Jan. 2021. DOI: 10.5281/zenodo.4438750.

[2] Nicholas Sharp et al. *Polyscope*. www.polyscope.run. 2019.