Multimedia retrieval

Georgios Christopoulos (2789175), David Filipiak (2932091) & Jesse de Wit (6233104)

October 1, 2023

Contents

1 Introduction

The proposed multimedia retrieval application described in this paper aims to help end-users acquire 3D shape representations of real-world objects in an efficient and accurate manner. In short, the application uses databases that store multimedia (i.e., accurately labeled files that represent 3D shapes) which are manipulated using various pre-processing and feature extraction methods (e.g., noise reduction, mesh computation). The challenge for this application to-be, however, is to **(a)** do this in a cost-time efficient manner (taking into account scalability problems), and **(b)** to *correctly* assess whatever it is the end-users are trying to retrieve. Particularly, difficulties arise regarding the semantics of queries issued by end-users. The following contains a short summary of the project.

Firstly, notation and terminology tables are provided to help the reader navigate this document. Secondly, project structure and related (i.e., programming language paradigm and application architecture) considerations are discussed, which are found in section 3.1. Next, we describe which software tools are used to realize our 3D shape retrieval application. This includes adopted programming language, various packages for rendering, user interface packages, as well as dataset considerations (Section 3.2). Lastly, we present the process of rendering meshes within the application including examples in section 3.3.

2 Notation & Terminology

The following list is a collection of terminology and notations used throughout the paper.[1]

Table 1: List of notations and terminology used in this paper.

Term	**Definition**
Shape / Object	3-dimensional (width, length, height) representation of a solid figure occupying space.
Feature	Any combination of properties of a shape's faces, edges and vertices.
Face	Flat surface on the shape.
Edge	Straight lines that define the sides of the polygons that make up each face of the shape.
Vertex / Vertices	Vertices (or corners) are points where at least three edges meet.
Mesh	A surface mesh serves as a representation of each individual surface within a volume mesh, comprising faces typically shaped as triangles along with vertices [13].
Structured mesh	Structured meshes are meshes characterized by a recognizable pattern in the arrangement of cells. Because cells follow a specific order, the mesh possesses a consistent and regular topology. This regularity facilitates the straightforward identification of neighboring cells and points due to the systematic way they are structured. Structured meshes find application in rectangular, elliptical, and spherical coordinate systems, forming grids that adhere to a uniform pattern [13].
Unstructured mesh	Unstructured meshes are less constrained and can take on arbitrary geometric shapes. Unlike structured meshes, they do not adhere to predetermined connectivity patterns, resulting in a lack of uniformity. Nevertheless, unstructured meshes offer more flexibility in terms of their adaptability [13].
Notation	**Definition**
$e = mc^2$	Energy equals mass times the speed of light squared.

[1] NB: This list is incomplete and is updated with every iteration as the project progresses.

3 Step 1: Read and view the data

3.1 Project Structure

3.1.1 Language paradigm

Regarding the project, due to the personal preferences of the team members and to optimize code organization, the Object-Oriented paradigm is adopted, which Python supports. As a bare minimum, utility classes such as Mesh and/or Feature will be implemented. We will create classes for every step of the multimedia retrieval pipeline from the slides to correctly represent said pipeline. Overall, the project structure aims to make the codebase highly modular without unnecessary code repetition. Refactoring and convenience are high priority. Intelligent code completion will be used to help with this process.

3.1.2 System architecture

The following section describes a preliminary architecture of our system step-by-step, which will improve and get more specific as the project progresses. To start, the Main class is created, which is the central point of execution. This class makes the necessary calls to other modules, acting as the orchestrator (pipeline) of our system. In the first step, the system ingests the `.obj` file(s) provided and displays the encoded 3D meshes in an interactive window.

For the further steps of our system, some assumptions are formulated about each step of the process. Regarding cleaning and pre-processing, the assumptions point out that files will need processing to remove noise or information that is not insightful/useful for the feature extraction step. In the feature extraction step, the goal is to extract the most representative features from all the objects by selecting generic but also specific ones (depending on the situation) for more granular querying.

Additionally, the next part of the process involves querying the database. Direct querying can be pretty effective but as the size of features grows in the database, queries become more complex (i.e., scalability problem). For this reason, it is better to use approximation methods to bucket similar items with similar features. For more efficiency, if certain features are very representative, we can enable indexing for faster queries. Afterward, the ranking of the results will take place in terms of similarity with the query term. Lastly, every system has to be evaluated and compared against its counterparts. Some examples of representative measures are precision, recall, F_1 score, among others.

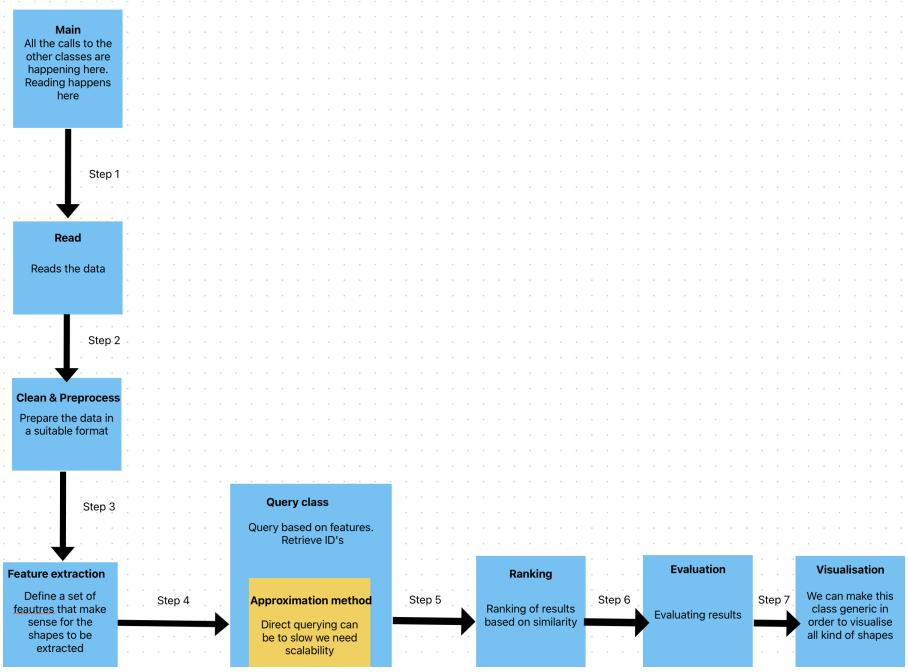


Figure 1: MR system architecture

3.2 Software

3.2.1 General approach

The programming language `Python` (v3.8.0) [5] is used for the development of the 3D shape retrieval application due to its familiarity, cross-platform compatibility, and relative ease of use. Development is fast and functionality remains intact when compared to other programming languages such as Java and C++. The application codebase uses `GitHub` [7] for git version control. A large number of `.obj` mesh files (representing real-world objects) are made available to experiment with, which serves as the application's retrieval database. An additional database will store the processed (e.g., multiple feature filters and/or queries) meshes. This is stored in a `.csv` file.

3.2.2 Packages

A large number of libraries are available for 3D model retrieval. `pymesh` [10] and `pymeshlab` [1] were considered. Consequently, `pymeshlab` was selected due to its ease of use and functionality. It provides many readily available functions for mesh processing. Together with the `polyscope` [11] package, it is possible to render meshes within the application window. PyMeshLab will thus be the 'core' package and foundation of the project.

Table 2: Adopted software used for the application.

Type	**Software**	**Version**
Language	`Python`	`3.8.0`
Package	`pymeshlab`	`2022.2.post4`
Package	`polyscope`	`1.3.4`
Package	`numpy`	`1.24.4`
Package	`scikit-learn`	`1.3.0`
Package	`matplotlib`	`3.7.3`
Package	`pandas`	`2.0.3`

Regarding user interfaces (UI) packages, As `Polyscope` only provides UI for rendering meshes, `tkinter` [6] will be used for buttons, input boxes, plot views, and potential other UI elements. `scikit-learn` [9] and `numpy` [3] are useful libraries for machine learning-related tasks in Python and as such will be utilized in this system. These are often paired with `matplotlib` [4], a package for plotting histograms as well as other data visualization methods. It is highly likely that additional libraries are going to be used during the future development of our 3D shape retrieval application. However, the ones listed here may be regarded as the core or foundation of our application.

3.2.3 Application GUI

The application consists of a simple graphical user interface (GUI). Python's built-in `tkinter` package renders a window and horizontal top-menu. Providing users with a horizontal top-menu, users are promptly able to select files from their computers, which the application displays (seperate window, using polyscope and pymesh for 3D renders), analyzes (provides statistical data on the .obj files such as number of faces, vertices, etc.) and pre-processes this, based on whichever action the user chooses to perform.

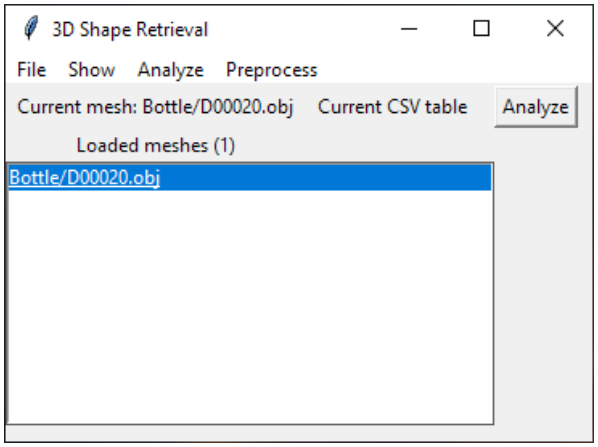


Figure 2: Application GUI

3.2.4 3D Shape Dataset

The 3D shape dataset used for the application contains 2483 `.obj` files and consists of 69 shapes in total. Figure 3 shows that the distribution of data samples is relatively equal among the shape classes, however, outliers can be spotted for the car, humanoid and jet shapes. In the final phase of the development, we plan to test the application on other shape datasets, such as Princeton Shape Benchmark [12] or PBS Dataset [8]. We believe that testing the application using different datasets will paint a better picture of the usability of our application.

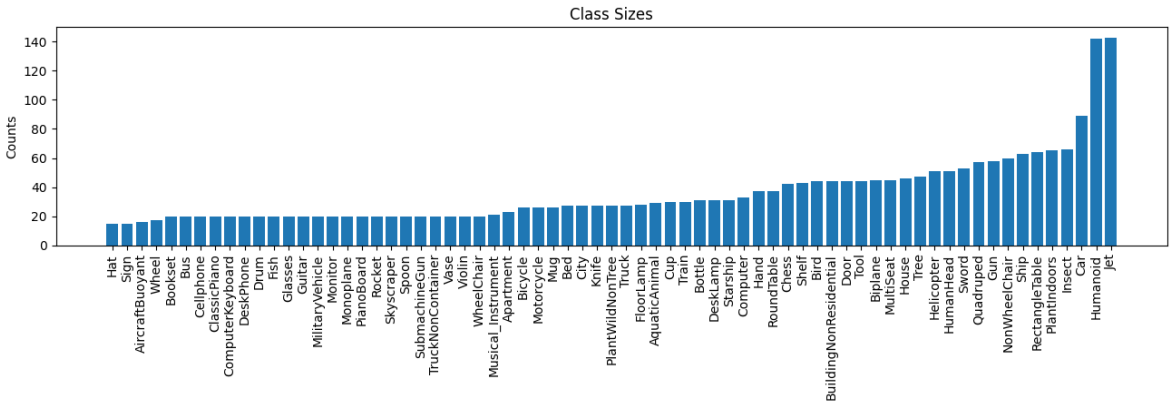


Figure 3: Number of `.obj` files for each shape in the database

3.3 Mesh rendering

Rendering a mesh starts with the selection of an object file from the database. As seen in Figure 2, `tkinter` renders a window with a horizontal top-menu which includes four actions for users: File, Show, Analyze and Pre-process. Clicking the File menu item gives the user the option to load a Mesh `.obj` file through a dialog where users can browse their computers. The rendering of the 3D objects is done using the `pymesh` package. Once the `.obj` file is loaded, tkinter launches a seperate window where 3D rendering takes place. The renders **without** and **with** visible cell edges are displayed in Figures 5 and 4, respectively.

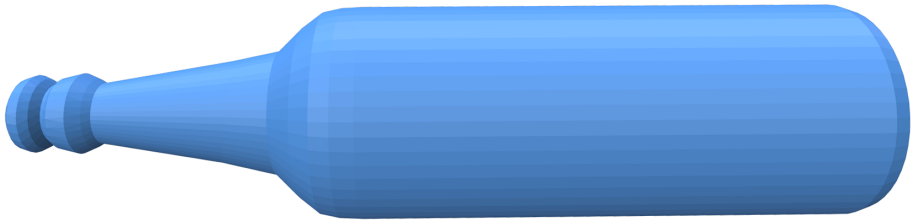


Figure 4: Render of a 3D bottle object.

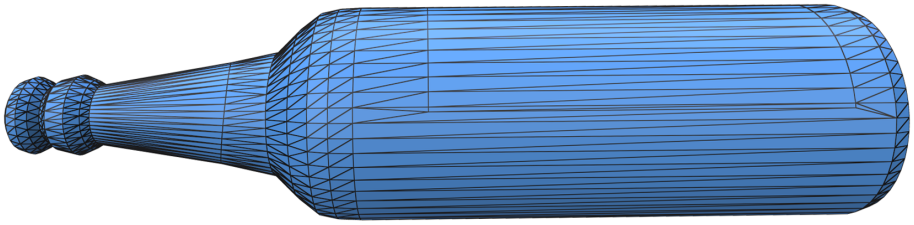


Figure 5: Render of a 3D bottle object with visible wireframes enabled.

4 Step 2: Pre-processing and cleaning

This section describes analysis done by the application on single shapes as well as the entire database of shapes.

4.1 Step 2.2: Statistics over the whole database

We have performed extensive analysis of the mesh database. We implement a tool that is able to analyze all loaded meshes, given they have been loaded from a .csv file. We can obtain the following information for any feature from the entire dataset:

- Histogram of values across the dataset
- Minimum value and the mesh with this value
- Maximum value and the mesh with this value
- Average value and the mesh closest to this value

The features we have analyzed thus far are the number of vertices, number of faces (triangular, quadrangular and all), size of the bounding box in X, Y and Z dimensions. The following figures present the results of this tool for selected features.

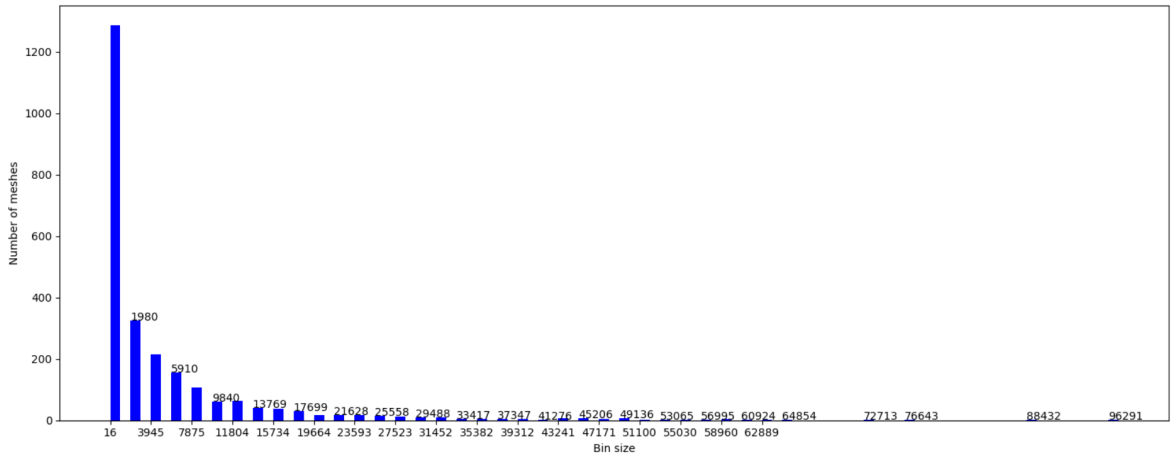


Figure 6: Distribution of the number of vertices across all meshes

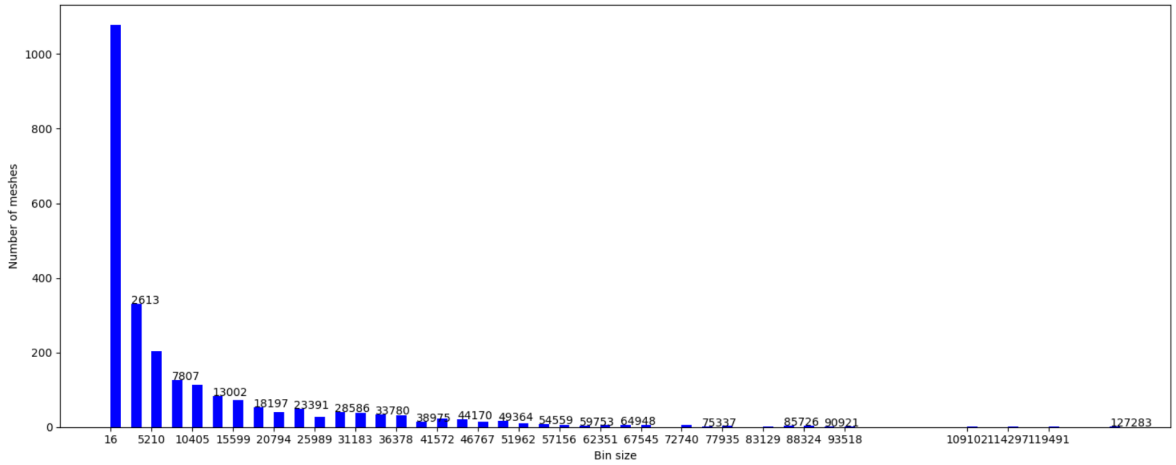
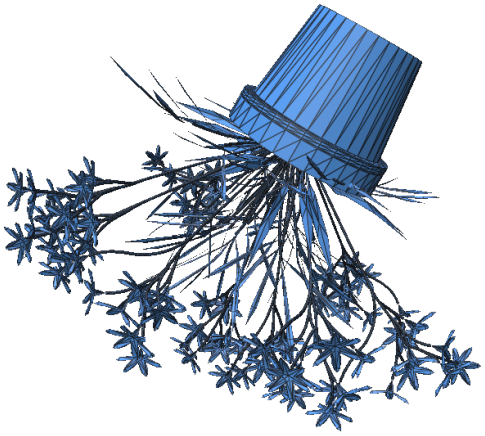
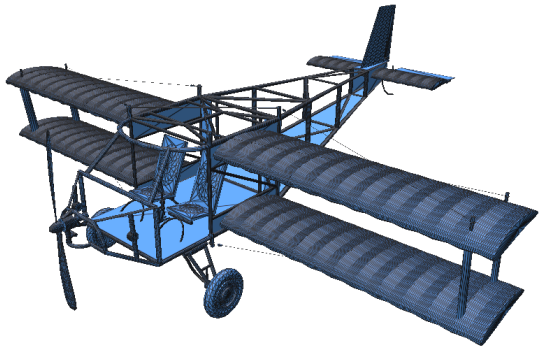


Figure 7: Distribution of the number of faces across all meshes

From figures 6 and 7, we can see a very similar distribution. The average number of vertices in the database is 5609 and the average number of faces is 10691. The majority of meshes has quite low number of vertices and faces, that is between 16 and around 2000 for each. However, we can clearly see outliers with disproportionately big values in each feature. The mesh with the maximum number of vertices is PlantIndoors\D00159.obj with 98,256 vertices, and the mesh with the maximum number of faces is Biplane\m1120.obj with 129,881 faces. These meshes can be seen in figures 8a and 8b respectively. From the available distributions, we can notice that they are very spread out, which is undesirable for the database querying as well as for feature extraction in the future.



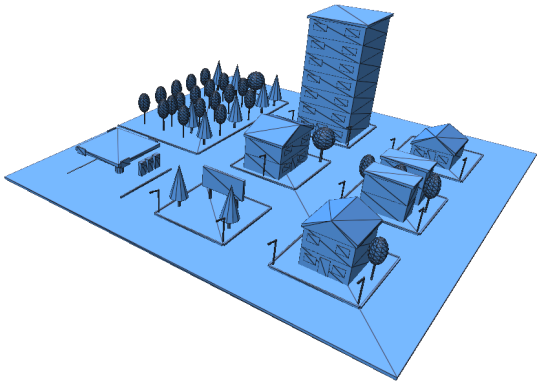
(a) PlantIndoors with 98,256 vertices



(b) Biplane with 129,881 faces

Figure 8: Meshes with the maximum amount of vertices and faces in the dataset, respectively

(a) Helicopter with 5,567 vertices



(b) City with 10,692 faces

Figure 9: Meshes with the average amount of vertices and faces in the dataset, respectively

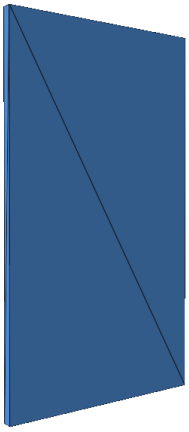


Figure 10: The Door with the lowest amount of vertices (16) and faces (16)

When it comes to the axis-aligned bounding box of the meshes, it is quite small in every dimension for all meshes. However, there is one noticeable example, whose bounding box exceeds 95 million, 193 million and 462 million units in dimensions X, Y and Z respectively. This is obviously very large, and the mesh needs to be scaled down in order to be used in further steps. The distribution of sizes of the bounding boxes along dimension X can be viewed in figure **??** where it is compared with a scaled distribution.

4.2 Step 2.3: Resampling outliers

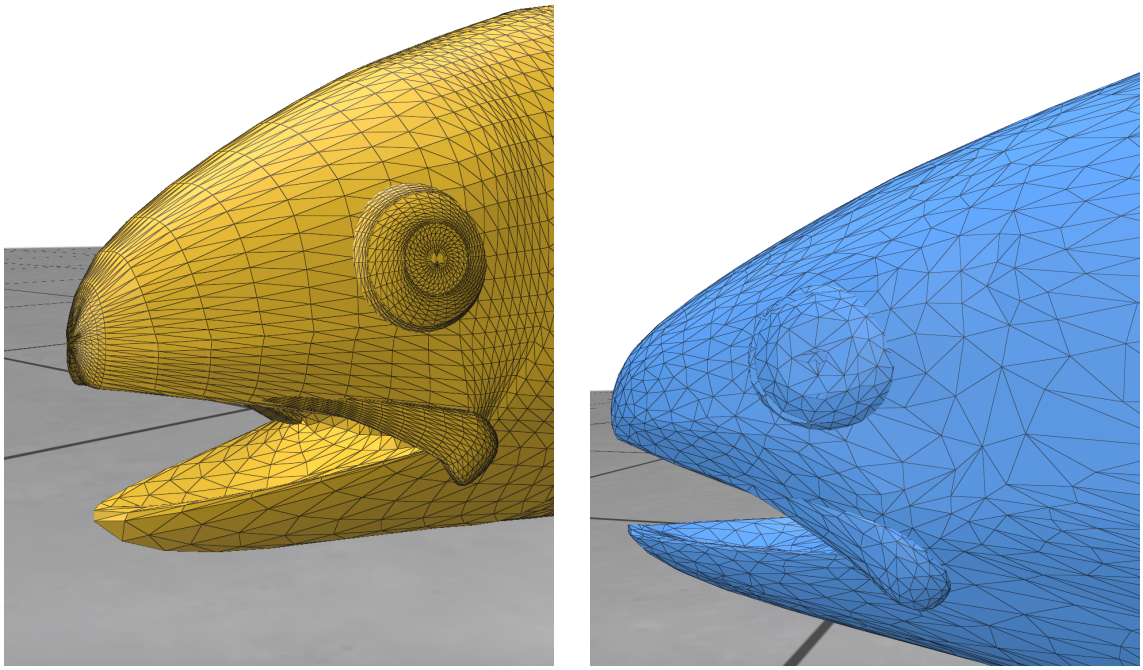
In this stage of the preprocessing, there is an effort to resample poorly-sampled shapes by supersampling them, meaning that the faces and the vertices are broken down into smaller ones, for a more detailed model. On the other hand, very detailed shapes need to be downsampled so that they are computationally inexpensive, while still retaining the information.

In our experiments, the target value for downsampling and supersampling is set to 10 thousand vertices. Of course, it is pretty difficult to achieve exactly 10 thousand vertices or if not possible a value very close to the target. For supersampling, the meshing surface subdivision butterfly algorithm was initially used which in essence refines simple surfaces by splitting edges, adjusting vertex positions, and creating new faces to achieve a smoother and more detailed result. This algorithm worked pretty well until it could not deal with non-manifoldness. Non-manifoldness was addressed by using meshing repair non-manifold edges which is a library provided by `pymeshlab` [1]. Still, it took considerable time to address the shapes that had non-manifold edges.

4.2.1 Isotropic Explicit Remeshing

Afterward, the isotropic explicit remeshing algorithm was used, since it was more robust and it could handle non-manifoldness automatically. The target length of the remeshed shape was adjusted to 0.001. This was crucial given that some classes contained really simple shapes that were impossible to refine in order to reach the target value. In general, to refine simple shapes we decided to overshoot(refine a model to more than 10 thousand vertices) and then simplify as the next step. The reasoning behind this idea was that when the mesh was simplified the decimation algorithm contained an attribute for the desired face number. Using Euler's formula which states "the number of vertices V, faces F, and edges E in a convex 3-dimensional polyhedron, satisfy $V + F - E = 2$. [2] By decimating the mesh to 20000 faces, it will produce a mesh of 10000 vertices, but sometimes it can also fall under 9999. The idea is to simplify the mesh to a face number slightly above 20000, and then slowly refine the solution until the mesh reaches exactly 10,000 vertices. The results of the remeshing algorithm subsampling are illustrated below in figures 11a and 11b. In 11a the aquatic animal is displayed with a lot of detail, not only around the body but also in its eye. Whereas the subsampled mesh keeps much of the detail but illustrates it in a simpler manner. On the other hand, in figures 12a and 12b the transition from a simpler model towards a more refined one is displayed. In the aforementioned figures, an iterative subdivision happens until the model splits the large faces into smaller surfaces. It should be noted,

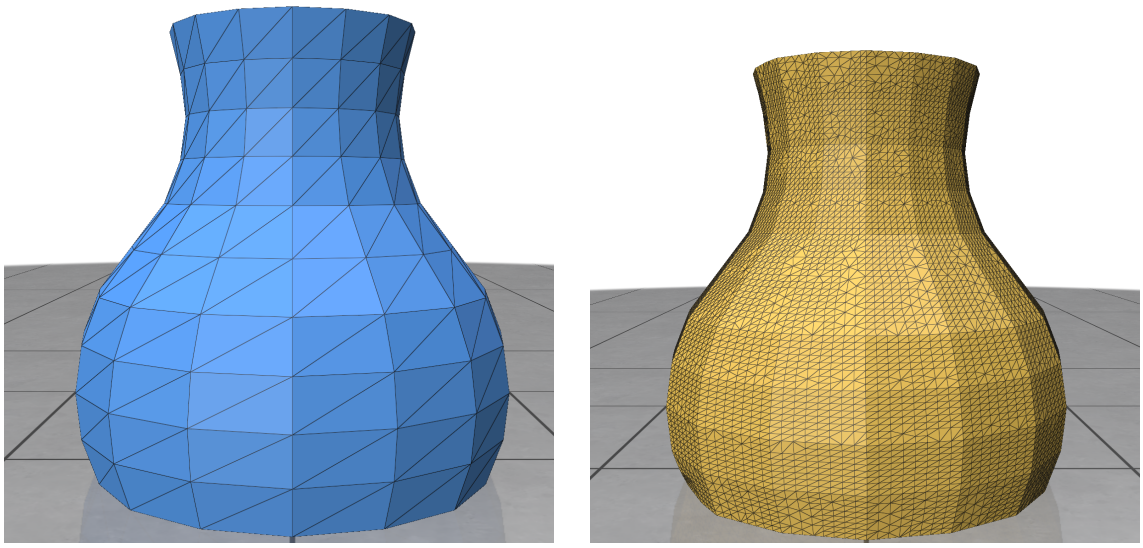
that 5 files were removed from the dataset because they were crashing the application due to data integrity issues.



(a) Aquatic animal with 18,409 vertices

(b) Aquatic animal with 10,000 vertices

Figure 11: A refined mesh(left) subsampled to a simpler mesh(right)



(a) Vase with 176 vertices

(b) Vase with 10,000 vertices

Figure 12: A simple mesh(left) supersampled to a more refined mesh(right)

4.3 Step 2.4: Checking the resampling

In figure 13 is displayed the refinement results are described in section 4.2. The refinement algorithm has managed to upscale or downscale the number of vertices close to the desired target which is 10,000 vertices. Overall, the process is effective with a caveat on the outliers presented in the distribution. The process could not refine all the shapes to reach the desired target, due to their simplicity and low amount of vertices. Furthermore, the results can be better highlighted when someone compares figure 13 to figure 6. Comparing the 2 distributions, it is clear that a lot of the shapes that were spread across the distribution have now been squashed closer to the target value. This fact proves that remeshing the shapes has worked quite well.

In the next steps, there may be a cut-off point in the resampled distribution so that feature extraction could perform better.

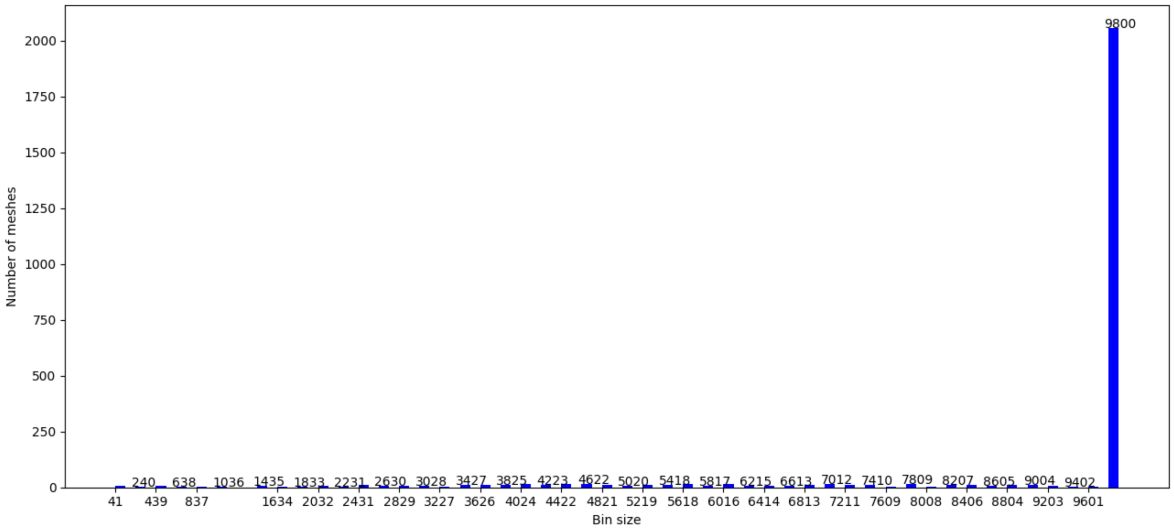


Figure 13: Distribution of the number of vertices across all resampled meshes

4.4 Step 2.5: Normalizing shapes

Normalization is a very important procedure, which allows us to globally eliminate certain features, such as size or orientation, which would only make searching unnecessarily difficult. Since a car is a car regardless of its size, orientation, or position, by normalizing we stop considering certain characteristics that do not provide any valuable information about a given mesh. Thus far, we have implemented two normalizing steps: centering and scaling.

4.4.1 Barycenter alignment

We center every mesh, so that it's barycenter coincides with the origin of the scene, or point (0,0,0). We get the barycenter as a simple average of positions of all vertices. Then, all vertices are shifted by the difference between the origin and the barycenter. Translation to origin is necessary, because it greatly simplifies following computations by allowing the assumption that meshes are centered. In cases, where the location of the barycenter needs to be known, it is known by default to be (0,0,0). The translation of a mesh is simple, and we do it as follows:
1. The coordinates of the barycenter (G_x, G_y, G_z) are calculated:

$$G_x = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$G_y = \frac{1}{n}\sum_{i=1}^{n} y_i$$

$$G_z = \frac{1}{n}\sum_{i=1}^{n} z_i$$

2. The translation vector (T_x, T_y, T_z) is given by:

$$T_x = -G_x$$

$$T_y = -G_y$$

$$T_z = -G_z$$

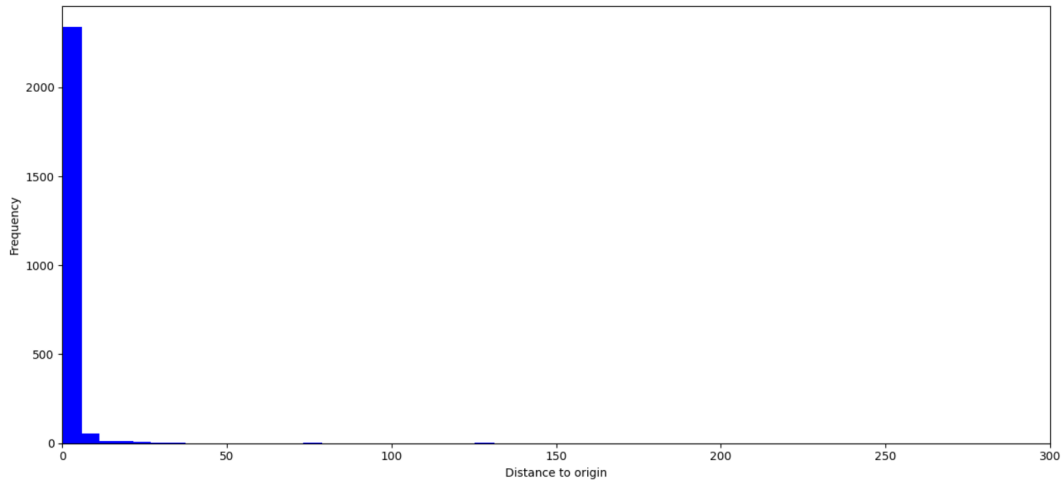
3. To center the mesh at the origin, we add the translation vector to the coordinates of all vertices:

$$\text{New}_{xi} = x_i + T_x$$

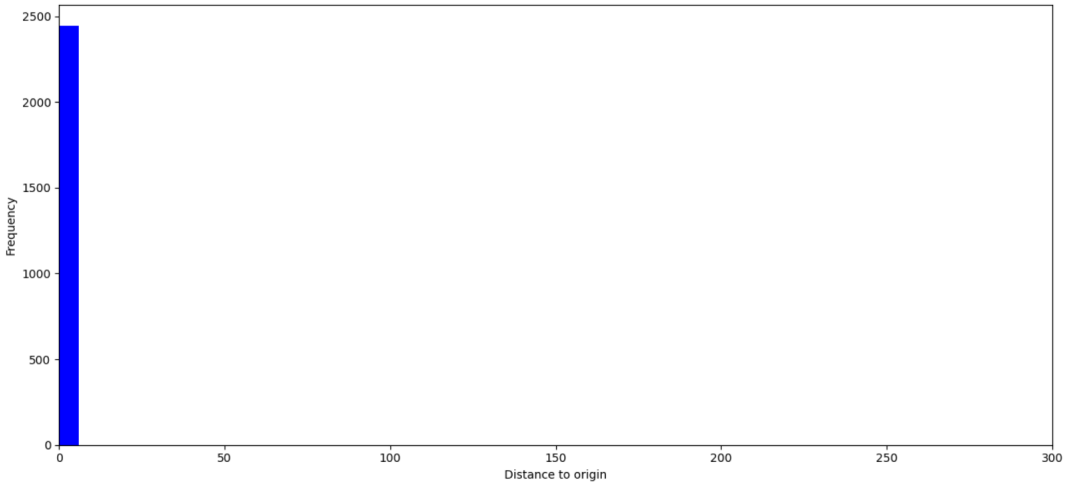
$$\text{New}_{yi} = y_i + T_y$$

$$\text{New}_{zi} = z_i + T_z$$

We prove that translation worked well by calculating the distance between the origin and the barycenters of meshes. The distribution of these distances is shown in the figure 14. Before normalization the distances vary, although the majority of meshes was already near the centre. We can also see that there are cases whose barycenters are further distant. After normalization, we see that all barycenters are located at (0,0,0).



(a) Distribution before normalization



(b) Distribution after normalization

Figure 14: Distribution of distances of barycenters to the origin of scene, the point $(0, 0, 0)$

4.4.2 Scaling

Scaling is another simple normalizing step, in which we scale every mesh to fit inside a unit cube, meaning that the longest side of the bounding box along any axis will have a length equal to 1. This not only eliminates the size variable from querying, but also allows us to estimate a good value for target edge length when performing remeshing.
Scaling can be interpreted using the following equations. [14] find the extent of the axis-aligned bounding box of the shape

$$(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$$

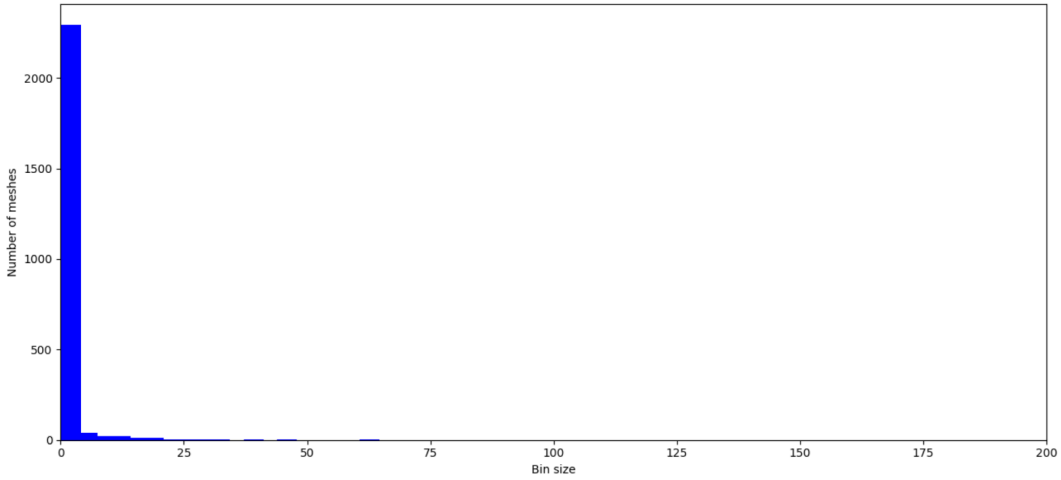
find the largest dimension:

$$s = \max(x_{max} - x_{min}, y_{max} - y_{min}, z_{max} - z_{min})$$

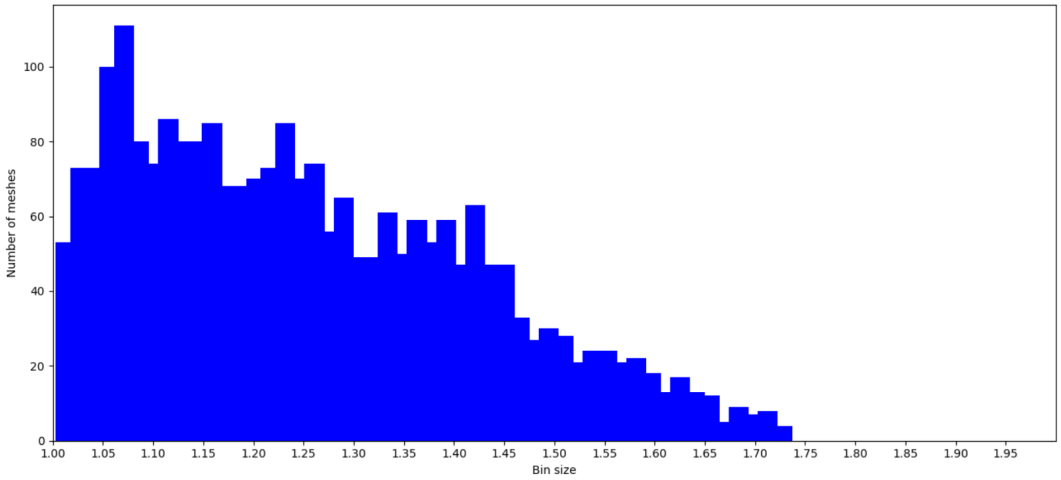
Then update every vertex's v coordinates. Here, we take advantage of previous preprocessing step (translation to origin), because the mesh should be scaled with respect to its barycenter. We do not have to consider this, since the barycenter is implicitly at (0,0,0):

$$v = v * \frac{1}{s}$$

We can check the diagonal length of the bounding boxes before and after scaling to find out if it worked well. In figure 15 we see the distribution of these diagonal lengths. After scaling, no diagonal should be longer than $\sqrt{3} \approx 1.73$, which is the length of a diagonal of a unit cube. Before scaling, we notice that some meshes have diagonals around 60 units. Due to better visualization, we excluded some high extremes from the histograms. The maximum length of any diagonal is $5.1 * 10^8$, which would heavily skew the visual representation. After scaling has been done, we clearly observe that no diagonal exceeds 1.73, thus scaling has been successful.



(a) Distribution before normalization



(b) Distribution after normalization

Figure 15: Distribution of lengths of diagonals of bounding boxes

The figure 16 clearly compares a mesh of a bird before and after normalization. The blue bird is the original, while the yellow one is translated to the origin of the scene and also scaled down into a unit cube.

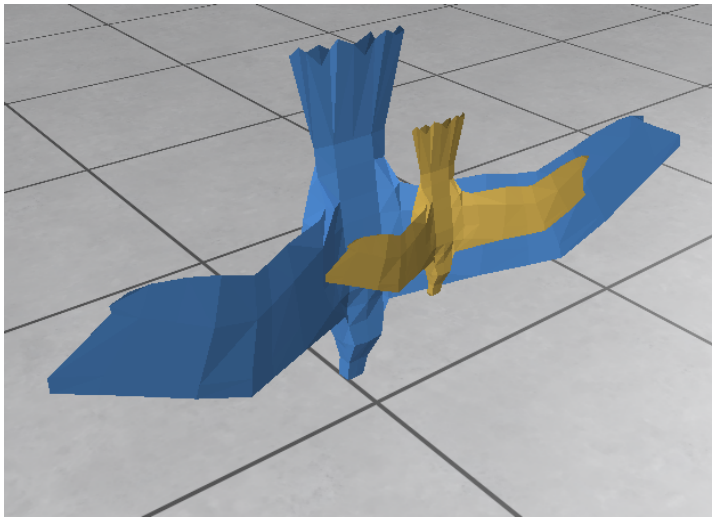


Figure 16: A comparison of the original and normalized mesh

5 Step 3

5.1 Full normalization

5.1.1 Normalization steps

The order in which the normalization in our project is executed is the following:

- Translation
- Scaling
- Hole stitching
- Remeshing
- Pose(alignment)
- Flipping

In the translation step, every mesh is centered so that its barycenter coincides with the origin of the axes which is (0,0,0). For a more detailed analysis, translation is explained in section 4.4.1. Next, scaling is used to fit every mesh inside a unit cube described in more detail 4.4.2. Translation and scaling can be in an arbitrary sequence as one does not affect the other. Additionally, scaling is important to be placed in the beginning, not only because it increases computational efficiency but also because we can set a relatively good target edge length for the remesh step when we are aware of the mesh sizes we work with.

5.1.2 Hole stitching

In hole stitching, there is an effort to close any hole or cavities that do not make the shape water-tight(meaning if we fill the shape with water, the water would spill out). This is an important step of the process for the correct calculation of shape volume and compactness, but it can be omitted by translating the shape to the origin. This gives an identical volume estimation by connecting the border points of the openings to the origin.

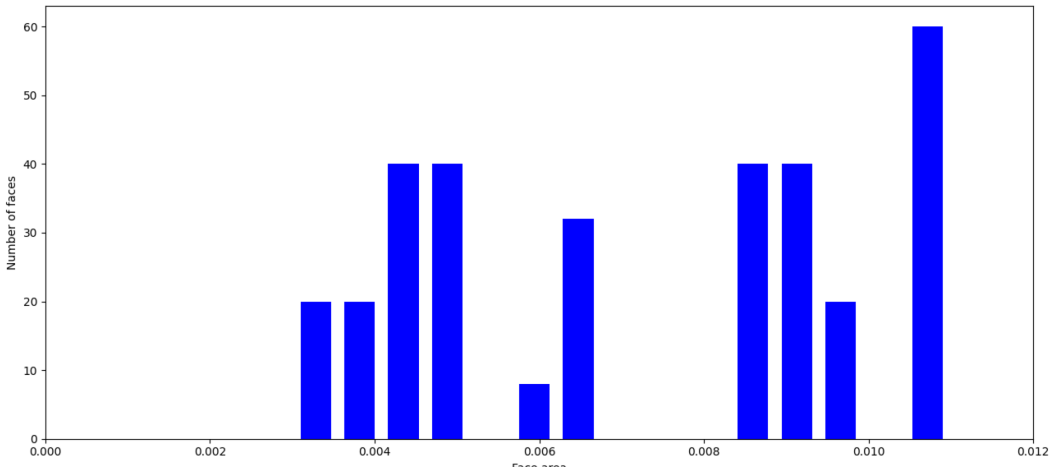
5.1.3 Remeshing

Remeshing can have a drastic effect on the alignment step. One reason is the insufficient variability in the data due to fewer points, on which the Principal component analysis (PCA) relies, for computing the principal axes, which is described in more detail in section 5.1.5. If the low-resolution mesh lacks

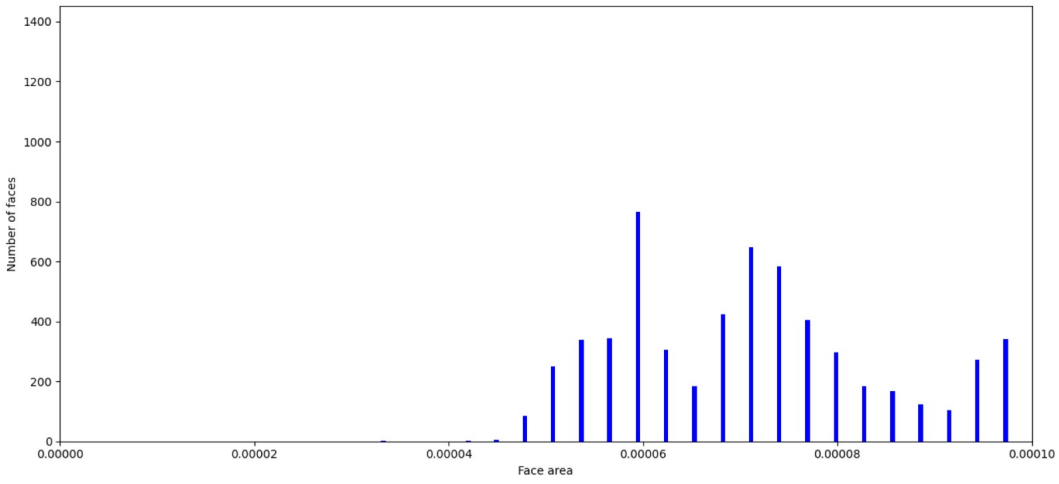
variation or contains only minimal differences between data points, PCA probably will not identify meaningful principal components. This is because PCA aims to capture the directions of maximum variance in the data, and if the data doesn't exhibit much variance, the extracted components may not be informative. Another reason is the loss of detail. For example, low-resolution meshes typically have reduced detail and precision compared to their high-resolution counterparts.

5.1.4 Remeshing distributions

In order to verify that our remeshing algorithm has performed its task well, there was a need for visualization of the distributions that occur before and after remeshing. The expected outcome is to have a relatively normal distribution with smaller variance and uniformity across the face areas. This testament is displayed in figure 17a in which the mesh is exhibiting a higher variance in the number of surface areas which has a range from around 0.004 to 0.011. In contrast, in figure 17b the variance has been contracted to a range between 0.00004 and 0.0010. Of course, visual inspections verified the aforementioned claim but proving it statistically is more solid.



(a) Distribution before normalization



(b) Distribution after normalization

Figure 17: Distribution of average face area before(a) and after remeshing(b)

5.1.5 Pose Alignment

As second to last step of our preprocessing pipeline, we do the pose alignment. The aim is to align the shape with the x, y, and z axes. By doing so, we eliminate the rotation of meshes from the pool of querying descriptors, as well as we significantly help future computations, for example regarding

bounding boxes, since it becomes aligned with the axes after this step.
In order to align a shape, we use Principal Component Analysis. In our case, the major principal component is the axis or line, along which the variance of vertices is the greatest. The goal is to align it with the x-axis. Medium and minor principal components are aligned with y and z axes respectively. First, we have to find the shape covariance matrix of the positions of all vertices. It is calculated as follows, where $\overline{x}$ is the average of x and C is the covariance matrix:

$$\sigma_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$$

$$C = \begin{pmatrix} \sigma_x^2 & \sigma(x,y) & \sigma(x,z) \\ \sigma(y,x) & \sigma_y^2 & \sigma(y,z) \\ \sigma(z,x) & \sigma(z,y) & \sigma_z^2 \end{pmatrix}$$

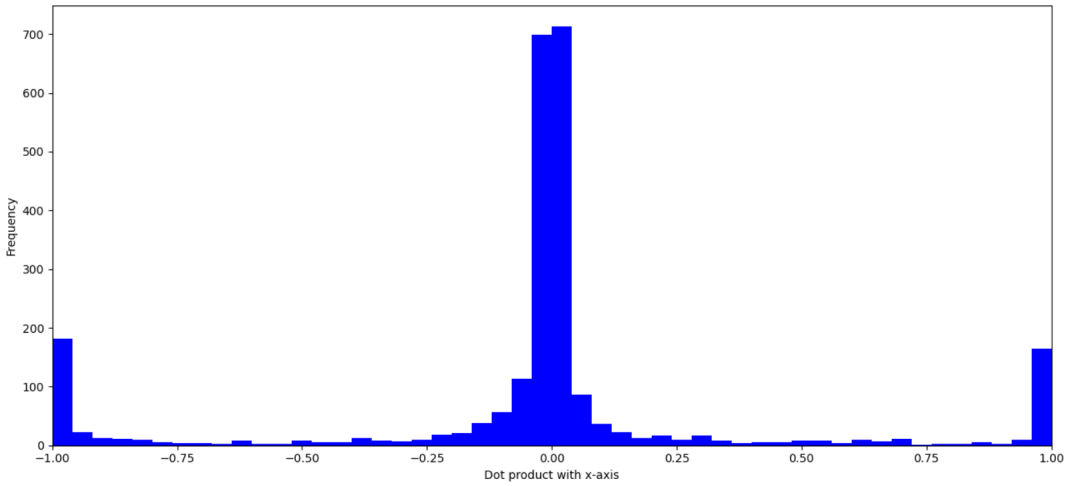
We then solve for normalized eigenvectors e_1, e_2 and e_3 and their respective eigenvalues. These eigenvectors are the principal components we were looking for. The major eigenvector is the one associated with the highest eigenvalue. The next aim is to rotate the shape, so that these eigenvectors are aligned with x, y, z axes. Therefore, we update the coordinates of vertices across the entire mesh. We use the following formulas to update every vertex. We assume that c is the barycenter vector of the mesh and p_i is a 1 by 3 vector of x, y, z coordinates of vertex i:

$$x_i^{updated} = (p_i - c) \cdot e_1$$

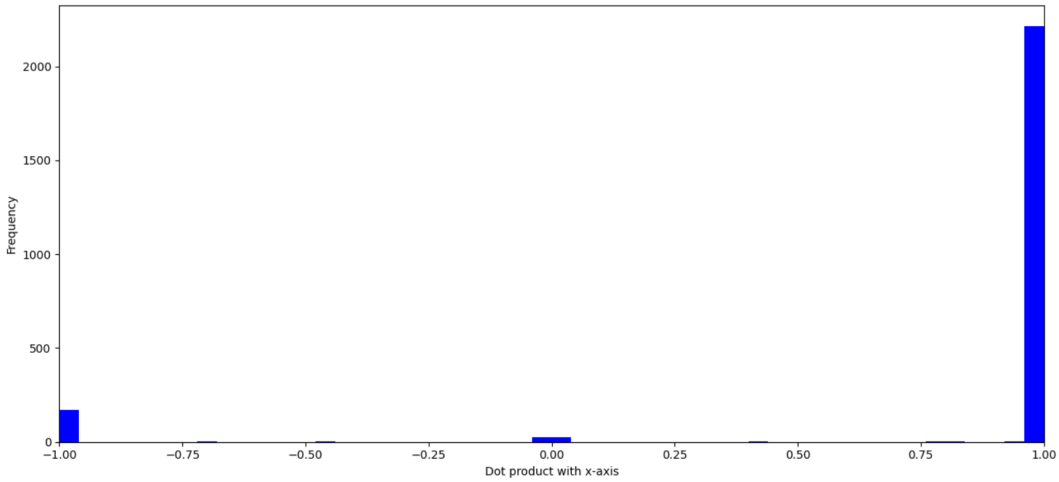
$$y_i^{updated} = (p_i - c) \cdot e_2$$

$$z_i^{updated} = (p_i - c) \times (e_1 \cdot e_2)$$

We can quantify success of the alignment for example by computing the dot product of a new major eigenvector computed after the alignment, and the x axis. Since the dot product of two vectors is equal to the cosine of the angle between them, we can see if the pose alignment worked well by checking if the dot product is 1. When two vectors point to the same direction, the angle between them is 0, and $cos(0) = 1$, hence the dot product we aim for is 1. Indeed, we can show that the pose alignment is successful with the following histograms. The figure 18a shows the distribution of dot product of the major eigenvector and the x axis in the original dataset, and the figure 18b shows the distribution after pose alignment. The distribution in the original dataset clearly varies, although some meshes seem to be already aligned. The second histogram proves that the meshes have been aligned, as the dot product is 1 in the majority of cases. The cases that have dot product -1 can be interpreted as the eigenvector pointing towards negative x axis, which is still means that the mesh is aligned with the x axis. However, there are a few cases, where the dot product is 0. We suspect that this means that these meshes have equal variance along multiple axes and the choice of which eigenvector is the major one is somewhat arbitrary.



(a) Distribution before normalization



(b) Distribution after normalization

Figure 18: Distribution of dot products between the major eigenvector and the x axis

5.1.6 Flipping

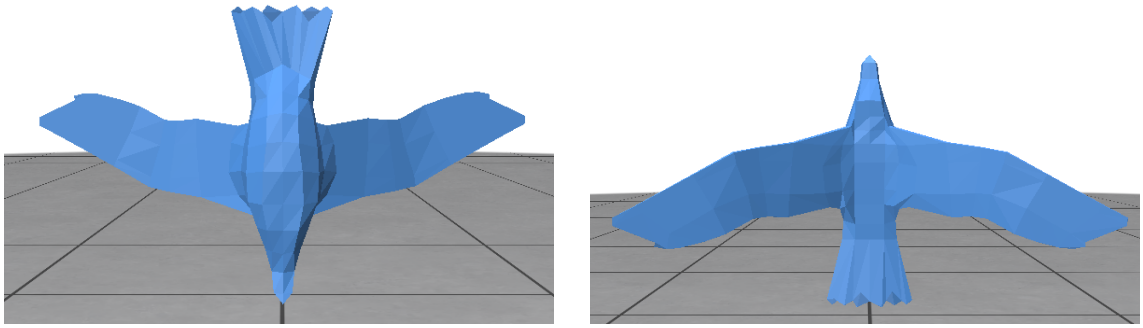
Flipping comes as the last step of our implementation because there is a need to maintain a consistent flip for every mesh processed. For easiness and consistency, we maintained the center of mass towards the positive direction of each of the axes. Flipping is similar to scaling but instead, it uses a scaling factor equal to -1 if the mesh needs to be flipped around a certain axis. The sign function returns 1 if the input is positive and a -1 if the input is negative, otherwise 0. The flipping test computes a value f_i along each axis (that is, i=0 means x, i=1 means y, i=2 means z) as: where the summation goes over all triangles t in the mesh and C_t, i is the i-th coordinate of the center of triangle t.

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

Then, the update formula is very simple: mirror the mesh using the following scaling factors:

$$\begin{aligned} x_i^{\text{updated}} &= x_i \cdot \text{sign}(f_0) \\ y_i^{\text{updated}} &= y_i \cdot \text{sign}(f_1) \\ z_i^{\text{updated}} &= z_i \cdot \text{sign}(f_2) \end{aligned}$$

The figure 19 shows the visual difference of a mesh before and after flipping. The majority of the mass is in the lower half in figure 19a, so for the sake of consistency, it is flipped upside down, the majority of the mass on the side towards the positive axis.

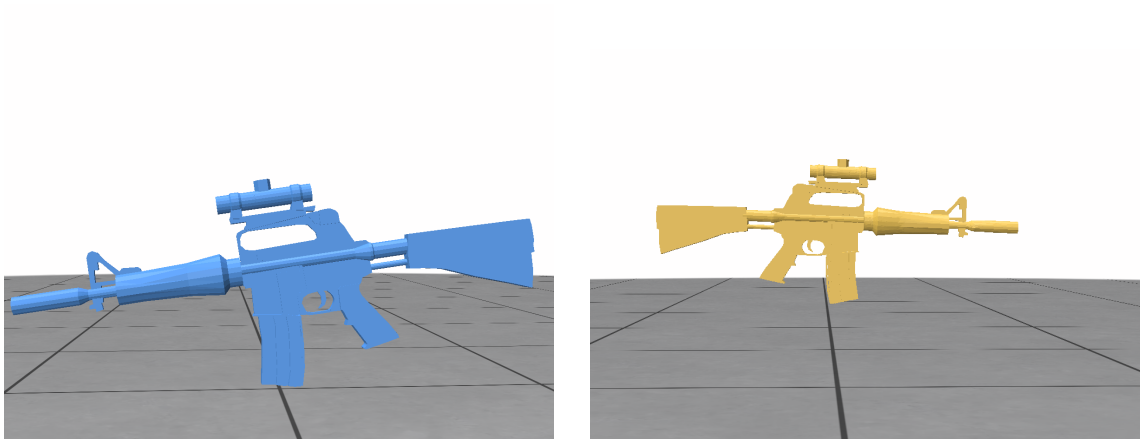


(a) Before normalization (b) After normalizatino

Figure 19: Visualization difference of the flipping test

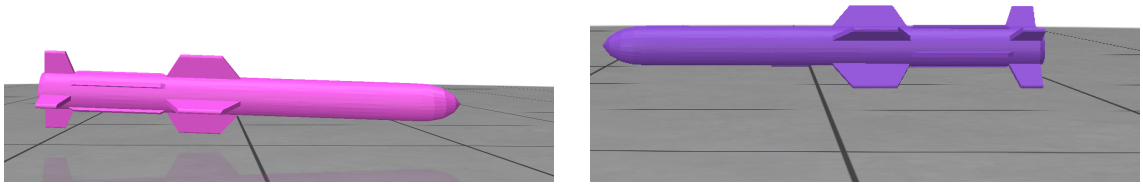
5.1.7 Full Normalization

The full normalization process can be seen in the following figures 20 and 21. For the moment we have not figured out a way to draw the axis on the rendering window, to prove our claim visually.



(a) Submachine gun before normalization (b) Submachine gun after normalization

Figure 20: Visualization difference before and after Normalization



(a) Rocket before normalization (b) Rocket after normalization

Figure 21: Visualization difference before and after Normalization

5.2 Elementary descriptors

In this section, we present the descriptors we have chosen to use. This is the first set of features to be compared during database querying.

5.2.1 Surface area

The surface area (S) of a 3D mesh can be calculated as the sum of the areas of all its triangular faces. In mathematical notation

$$S = \sum_{i=1}^{n} A_i$$

Where A_i represents the area of the i-th triangular face.

5.2.2 Volume

The volume (V) of a 3D mesh is computed based on its vertex matrix (V_matrix) and face matrix (F_matrix) by summing the signed volume of tetrahedra formed by each triangular face.

$$V = \frac{1}{6} \left| \sum_{t_i} \mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) \right|$$

5.2.3 Compactness

Compactness is a measure that relates the volume and surface area of the mesh. It's often expressed as the ratio of the cube root of the volume to the surface area:

$$C = \frac{\sqrt[3]{36\pi V^2}}{S}$$

5.2.4 Axis-Aligned Bounding Box (AABB) Volume

The volume of an AABB can be calculated by multiplying its dimensions along the x, y, z axes:

$$AABB_{\text{Volume}} = \text{dim}_x \cdot \text{dim}_y \cdot \text{dim}_z$$

5.2.5 Eccentricity

Eccentricity (E) measures how elongated or stretched an object is. It's calculated as the square root of 1 minus the square of the ratio of the minimum dimension to the maximum dimension as follows:

$$E = \sqrt{1 - \left(\frac{\min(\text{dim}_x, \text{dim}_y, \text{dim}_z)}{\max(\text{dim}_x, \text{dim}_y, \text{dim}_z)} \right)^2}$$

5.2.6 Rectangularity

Rectangularity compares the volume of the 3D mesh to the volume of its AABB. It's calculated as the ratio of the AABB volume to the mesh volume:

$$R = \frac{AABB_{\text{Volume}}}{V}$$

5.2.7 Convexivity

Convexivity (K) measures how much a shape deviates from being convex. It's calculated as the ratio of the mesh volume to the AABB volume as follows:

$$K = \frac{V}{AABB_{\text{Volume}}}$$

5.2.8 Diameter

The diameter (D) of a 3D mesh is as follows. Let's assume V_i represents the i-th vertex of the 3D mesh, and N is the total number of vertices in the mesh. The diameter (D) of the mesh is computed as the maximum Euclidean distance between any two vertices:

$$D = \max_{i,j} \|V_i - V_j\|$$

Where V_i and V_j are individual vertices of the mesh. In our case, since the mesh is centered in the origin $V_j = o$, where o signifies the origin point. Then:

$$D = \max_{i} \|V_i - o\|$$

References

[1] P. C. A, Muntoni. Pymeshlab. https://pymeshlab.readthedocs.io/en/latest, 2022. Accessed: 16-09-2023.

[2] Cuemath. What is mesh and what are the types of meshing. https://www.cuemath.com/eulers-formula/. Accessed: 24-09-2023.

[3] N. development team. Numpy. https://numpy.org, 2023. Accessed: 16-09-2023.

[4] T. M. development team. Matplotlib. https://matplotlib.org, 2023. Accessed: 16-09-2023.

[5] P. S. Foundation. Python. https://www.python.org, 2023. Accessed: 16-09-2023.

[6] P. S. Foundation. tkinter. https://docs.python.org/3/library/tkinter.html, 2023. Accessed: 16-09-2023.

[7] G. Inc. Github. https://github.com, 2023. Accessed: 16-09-2023.

[8] E. Kalogerakis, A. Hertzmann, and K. Singh. Learning 3D Mesh Segmentation and Labeling. *ACM Transactions on Graphics*, 29(3), 2010.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Z. Qingnan. Pymesh. https://pymesh.readthedocs.io/en/latest, 2018. Accessed: 16-09-2023.

[11] N. Sharp. Polyscope. https://polyscope.run, 2019. Accessed: 16-09-2023.

[12] P. Shilane, P. Min, M. Kazhdan, and T. Funkhouser. The princeton shape benchmark. In *Proceedings Shape Modeling Applications, 2004.*, pages 167–178. IEEE, 2004.

[13] P. technologies. What is mesh and what are the types of meshing. https://www.pre-scient.com/knowledge-center/product-development-by-reverse-engineering/mesh.html#redirect2_1. Accessed: 17-09-2023.

[14] A. Telea. Multimedia retrieval features. https://webspace.science.uu.nl/~telea001/MR/Slides, 2023. Accessed: 24-09-2023.