Content-based 3D shape Retrieval System

Ziv Hochman - 8454434, Stylianos Psara - 2140527 , Panagiotis Andrikopoulos - 1780743

November 15, 2024

1 Introduction

In recent years, the demand for advanced multimedia retrieval systems has surged alongside the exponential growth of digital content. Content-based 3D shape retrieval, in particular, has emerged as a critical technology for numerous applications requiring efficient and accurate shape comparison. Unlike traditional methods that rely on textual or metadata annotations, content-based retrieval utilizes the intrinsic geometry of 3D objects, enabling more precise and scalable solutions for applications necessitating visual or structural matching.

This report details the development of a comprehensive content-based 3D shape retrieval system, which facilitates the retrieval and comparison of 3D objects based on their geometric properties. Implemented in Python, the system integrates key libraries, including Open3D for 3D data handling, visualization, and preprocessing, as well as PyMeshLab and VEDO for advanced mesh processing and remeshing. The proposed framework features a robust preprocessing pipeline for cleaning and normalizing 3D shapes, advanced feature extraction methods to capture both elementary and Shape Property descriptors, and an efficient querying mechanism that ranks shapes by similarity to a user-provided 3D model.

The rest of this report is organized as follows: Section 2 outlines the software and libraries utilized in this project, providing a foundation for the technical implementation. Section 3 introduces the essential notations and terminologies for understanding 3D mesh representation. In Section 4, we describe the data visualization techniques used to explore the 3D shape dataset. Section 5 explains the preprocessing, cleaning, and normalization methods, including calculating mesh statistics (e.g., number of faces and vertices, averages, standard deviation), as well as resampling, translation, alignment, flipping, and scaling to standardize each shape. Section 6 discusses feature extraction, detailing the computation of various descriptors that capture each shape's geometry for effective comparison. Section 7 presents the querying process, describing how the system processes user queries to retrieve the most similar shapes from the database. Section 8 addresses system scalability, introducing Approximate Nearest Neighbor (ANN) search and dimensionality reduction with t-SNE. Section 9 evaluates the system's performance and effectiveness in shape retrieval, using metrics such as accuracy, precision, recall, and F1 score. Finally, Section 10 provides the conclusions of this study.

2 Software and Libraries

Our software is being developed using Python 3.1 as the primary programming language, and we are utilizing Jupyter Notebook with Anaconda as our coding environment. To implement our algorithms and data structures, we are using the following libraries:

- Python (version: 3.11.5) [1]
- numpy (version: 1.26.3) [2]
- open3D (version: 0.18.0) [3]
- pandas (version: 2.1.4) [4]
- pymeshlab (version: 2023.12.post2) [5]
- seaborn (version: 0.12.2) [6]
- matplotlib (version: 3.8.0) [7]
- scikit-learn (version: 1.2.2) [8]
- scipy (version: 1.11.4) [9]

- flask (version: 2.2.5) [10]
- plotly (version: 5.9.0) [11]
- pyngrok (version: 7.2.1) [12]

3 Notations

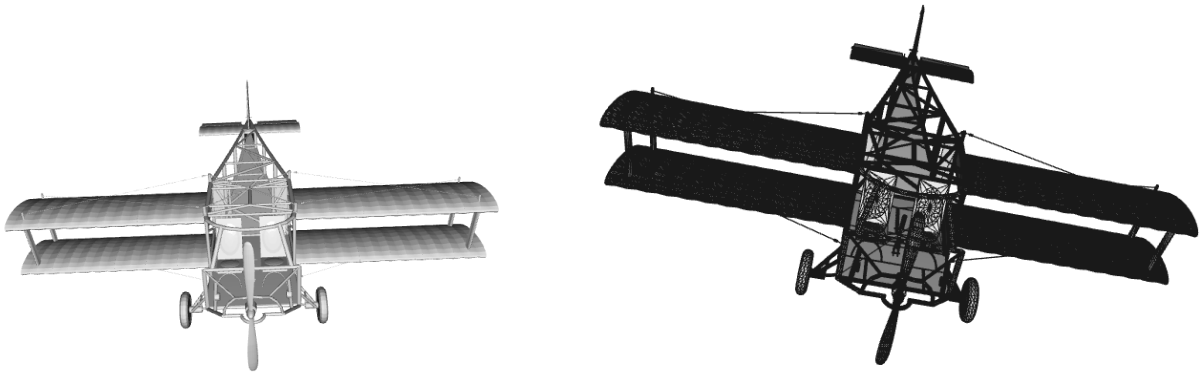
- **Mesh:** mesh is a collection of vertices, edges, and faces that define the shape of a 3D object. It is a polygonal representation of the object's surface, usually made up of triangles or quads.
- **Faces:** A face is a flat surface that connects a set of vertices.
- **Vertices:** A vertex is a point in 3D space, defined by its coordinates (x,y,z). The vertices form the corners of the triangles or polygons that make up the mesh.
- **Planes:** Mathematical surfaces associated with each vertex that approximate its local geometry, used to calculate the error or distance from the ideal mesh surface during simplification.
- **Barycenter:** The barycenter is the point that serves as the center of mass of a set of points, representing their average position in space.
- **Barycenter:** The barycenter is the point that serves as the center of mass of a set of points, representing their average position in space.
- **Eigenvalue:** Is a scalar that represents how much a corresponding eigenvector is stretched or compressed during a linear transformation represented by a matrix.space.
- **Eigenvector:** An eigenvector is a non-zero vector that remains in the same direction after a linear transformation is applied to it.
- **Covariance Matrix:** A covariance matrix is a square matrix giving the covariance between each pair of elements of a given vector.
- **Principal Component Analysis (PCA):** A linear algebra-based statistical technique used to extract the most significant information from a set of data points, reducing and simplifying its dimensionality. It applies an orthonormal transformation that ensures the resulting components are uncorrelated and ranked by variance. [13] By projecting the data onto these components, those with zero or minimal variance can be discarded, preserving only the most informative features.
  For our purposes we denote the primary and secondary eigenvectors as e_1 and e_2, with e_3 defined as their cross product to maintain right-hand consistency in direction.

- **Elementary Descriptors:** Single value features used to describe one entire shape.
- **Shape Property Descriptors:** Descriptors that measures various intrinsic geometry properties of a shape.

4 Step 1: Read and view the data

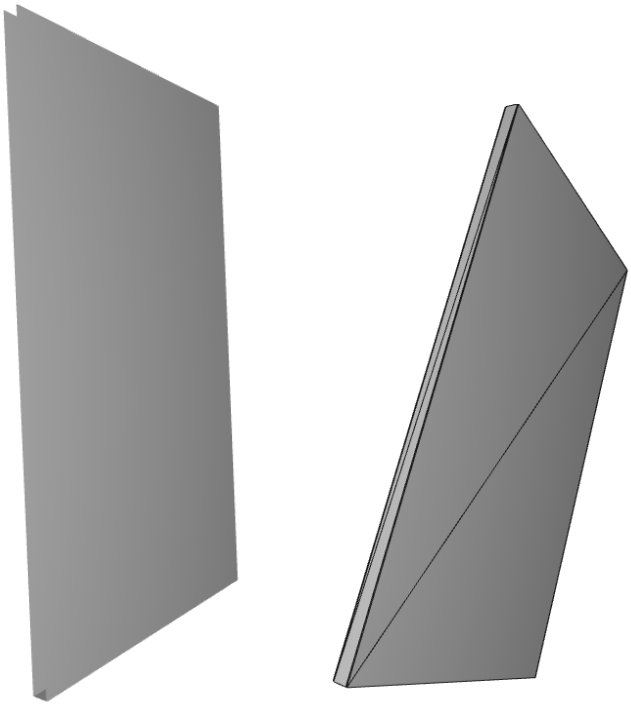
To familiarize ourselves with the 3D shape database, we developed a tool to visualize and process minimal 3D shapes. First, we load the meshes using the Open3D library. Then, we display the shapes in various ways, as shown in Figure 1 and 2, including smooth-shaded views and shaded views with wireframes.

Moreover, to better understand the shapes and how Open3D handles visualization, we chose to display both the shape with the highest number of faces and vertices (see Figure 1) and the one with the fewest (see Figure 2). The shape with the most vertices and faces 65722 and 129881, respectively is the Biplane, which appears highly detailed compared to the shape with only 16 faces and 16 vertices which depicts a door. The contrast in the wireframe density also highlights these differences.



(a) Smooth-shaded Biplane *(b) Shaded with wires Biplane*

Figure 1. *Visualization of the shape with id "m1120.obj" from class "Biplane" with the most faces and vertices*



(a) Smooth-shaded Door *(b) Shaded with wires Door*

Figure 2. *Visualization of the shape with id "D01121.obj" from class "Door" with the less faces and vertices*

Finally, we experimented with Open3D's interactive features, such as rotating the viewpoint around the shapes and zooming in and out, to better observe the details of each shape (see Figure 3).

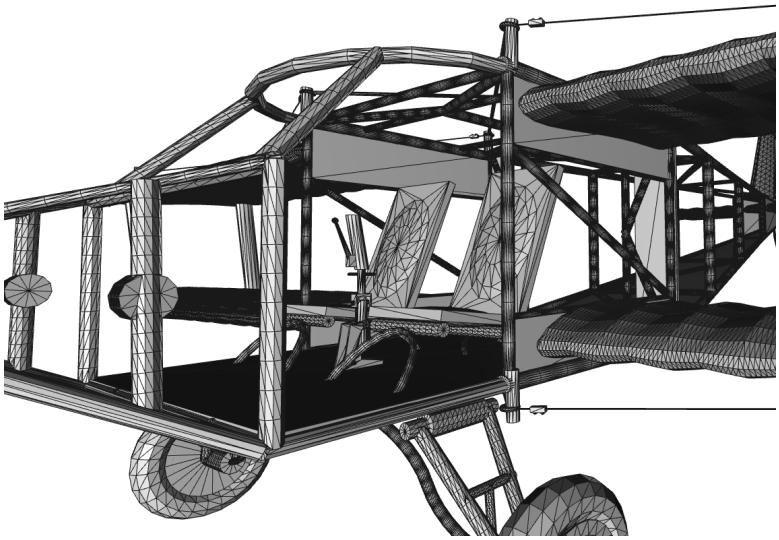


Figure 3. *The inside of the Biplane (m1120.obj) after rotating and zoom in*

5 Step 2: Preprocessing and cleaning

The purpose of this step is to prepare the shape database for the next critical phase of feature extraction. To achieve this, all shapes must be thoroughly examined to ensure they meet specific quality standards. If any shapes fail to meet these requirements, they will undergo preprocessing to bring them into quality-compliance.

This process allows us to better understand the various issues that 3D meshes can present. By becoming familiar with these challenges, we can effectively apply methods to resolve them. Additionally, we will experiment with basic tools for processing 3D meshes and gain familiarity with the OBJ format.

5.1 Step 2.1: Analyzing a single shape

In this step, we developed a simple tool to process all the shapes in the database and extract key information for each one. First, we retrieved the class of each shape. This information is provided to us by grouping shapes of the same class into folders named after the class.

Next, we calculated the number of faces and vertices for each shape. For each mesh, we generated a face matrix, which contains all the faces of the mesh. We then analyzed the number of vertices in each face to determine its type: faces with 3 vertices were classified as triangles, and those with 4 vertices as quads.

To classify the overall mesh, we checked the face types. If all faces were triangles, the mesh was labeled "Only triangles." If all faces were quads, it was labeled "Only quads." If the mesh contained both types, it was labeled as "Mixed triangles and quads."

Finally, using the "get_axis_aligned_bounding_box" command from the Open3D library, we obtained the minimum and maximum bounds of each mesh, allowing us to determine the bounding box for each shape.

An example of the output of the aforementioned tool for a single shape is shown in Figure 4.

Shape Class	Filename	Number of Vertices	Number of Faces	Face Types	Bounding Box Min	Bounding Box Max
AircraftBuoyant	m1337.obj	201	340	Only triangles	[0.025 0.025 0.025]	[0.88923401 0.97500002 0.88923401]

Figure 4. *Analysis of object with id "m1337" from class "AircraftBuoyant"*

5.2 Step 2.2: Statistics over the whole database

After analyzing the key information for each shape, we calculated some statistics for the entire dataset. First, to understand the average characteristics of a shape in the database, we compute the mean number of faces and

vertices. On average, a shape in the dataset has 10691 faces 5609 vertices.

Next, we investigate whether there are any significant outliers among the shapes. To do this, we visualize histograms that show the distribution of shapes across various ranges of face and vertex counts. As seen in Figure 5, although the average number of faces is around 10.6k (indicated by the red line), over 1,000 meshes (represented by the first bar of the histogram) have fewer than 2.5k faces. Each bar represents a range of approximately 2.5k faces. This discrepancy between the average number of faces and the majority of shapes is due to the wide variance in the dataset, with some shapes having as few as 100 faces, while others have close to 120k faces. Moreover, as shown in Figure 5, the standard deviation of the number of faces is 16191, which is a significantly large value. This indicates that the face counts in the dataset are widely dispersed, meaning the average number of faces is not a good representation of the actual data distribution.

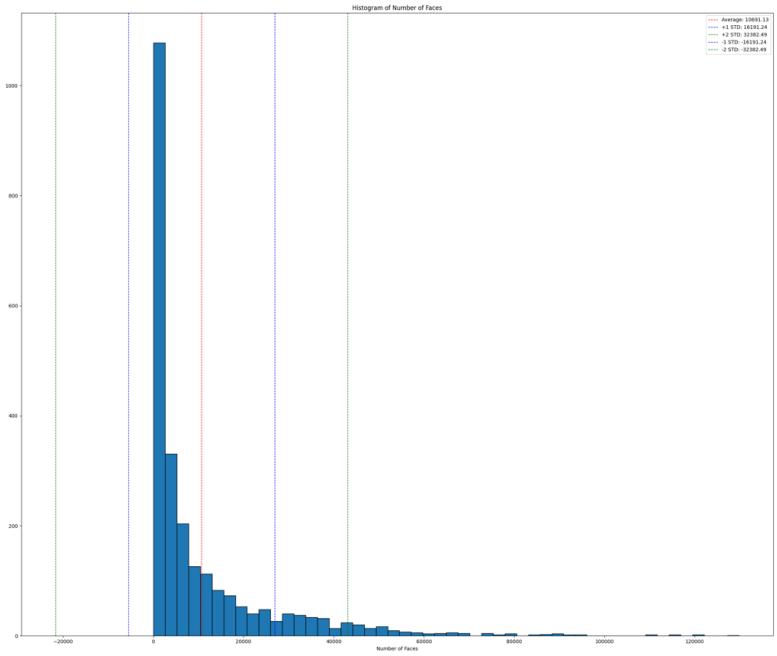


Figure 5. *The frequency of shapes in our dataset across various ranges of face counts*

Similarly, Figure 6 shows that while the average shape has aproximattely 5.6k vertices, most shapes contain fewer than 2k vertices. However, a few shapes have as many as 8k vertices. Moreover similarly, to the case of the average number of faces, the average number of vertices is also not representative of the actual data since we also have a large spread as we have a standard deviation of around 9851. Additionally, as shown in Figure 7, the most common class in our dataset is the jet, while the least common class is the sign.

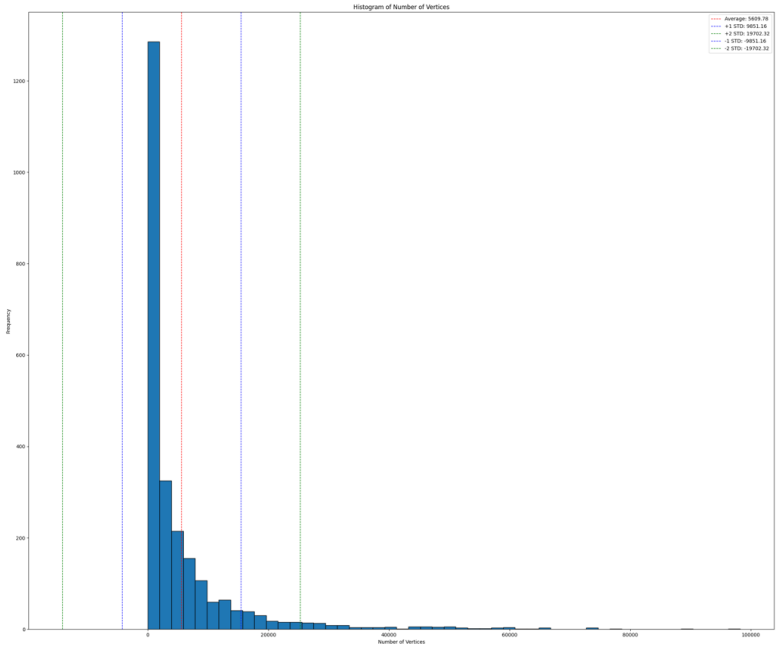


Figure 6. *The frequency of shapes in our dataset across various ranges of vertices counts*

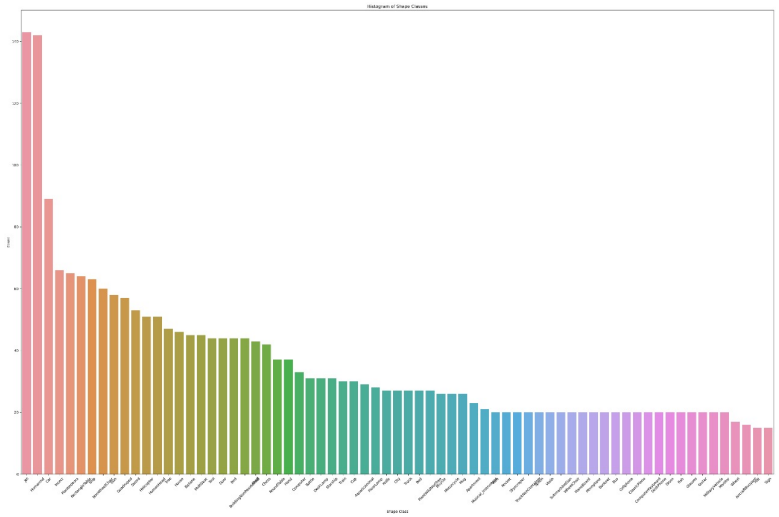
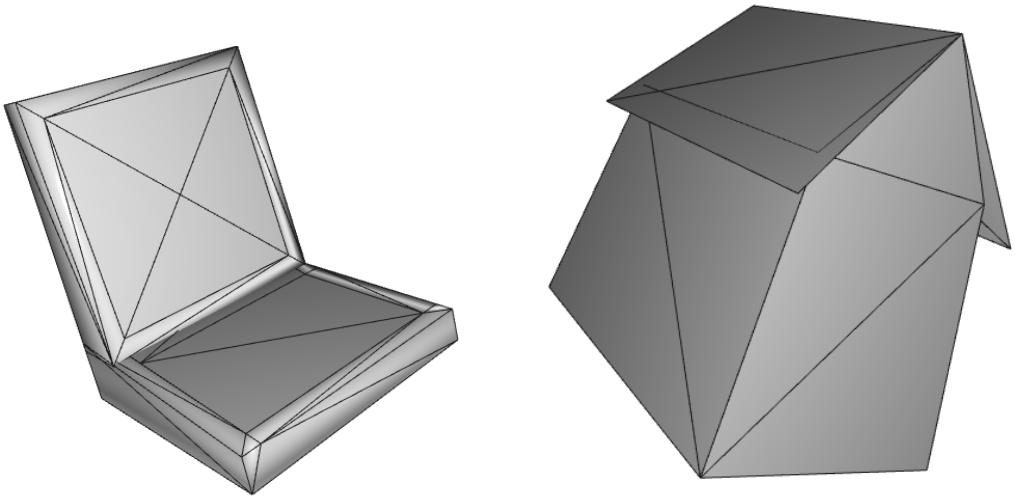


Figure 7. *The frequency of the shapes in our datataset for each class*

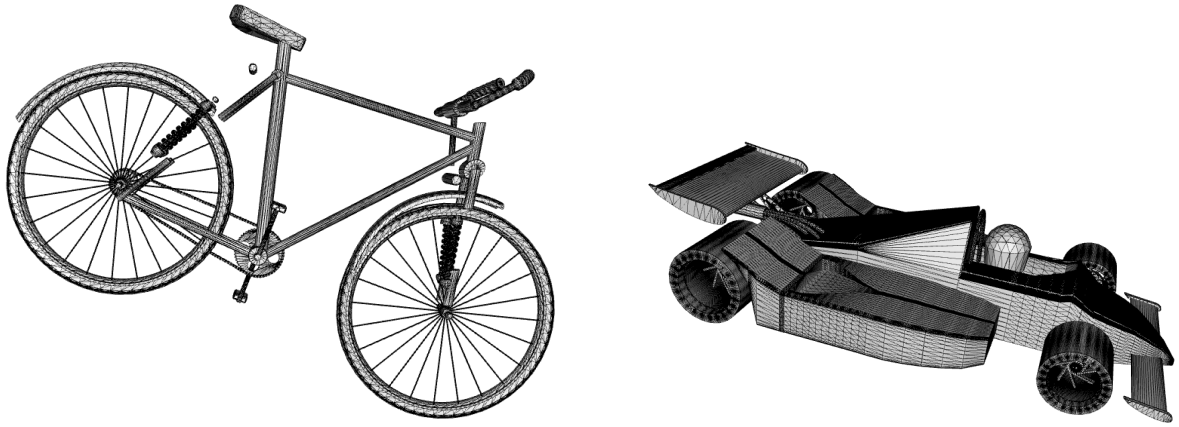
Some of the outliers with fewer vertices and faces than the average shape in our dataset are shown in Figure 8. The object on the left represents a computer, created with just 29 vertices and 43 faces, while the one on the right is a simple house consisting of 22 vertices and 26 faces. Both objects are quite basic due to their low face and vertex counts, and they are not representative of the average object in our dataset. As mentioned earlier, the average object has around 5.6k vertices and 10.6k faces.



(a) PC *(b) House*

Figure 8. *Outliers with fewer number of faces and vertices: Object from class "Computer" and object id "D01071.obj" on the left (Figure a), and object from class "House" and object id "m434.obj" on the right (Figure b)*

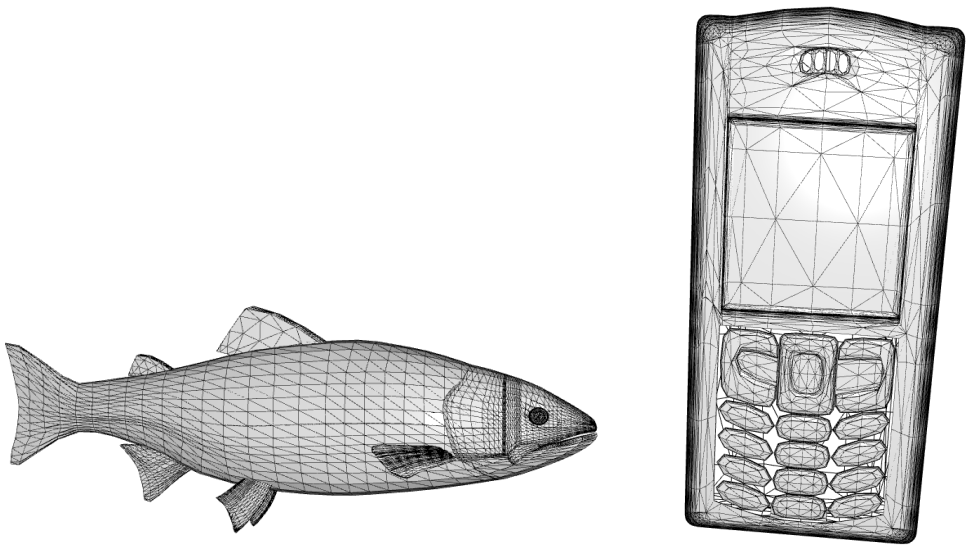
In contrast, Figure 9 presents two outliers with significantly more faces and vertices than the average. These objects, a bike on the left and a car on the right, are more complex and detailed, requiring around 50k vertices and 100k faces, about 10 times the average count in the dataset.



(a) Bike *(b) Car*

Figure 9. *Outliers with more number of faces and vertices: Object from class "Bicycle" with object id "D00462.obj" on the left (Figure a) , and object from class "Car" and object id "m1510.obj" on the right (Figure b)*

Finally, in Figure 10, we display two average shapes from the dataset: an aquatic animal and a cellphone. Both have around 5k vertices and 10k faces, making them close to the dataset's average. While these shapes are less detailed than the car and bicycle in Figure 9, they are still more intricate than the simpler objects shown in Figure 8.



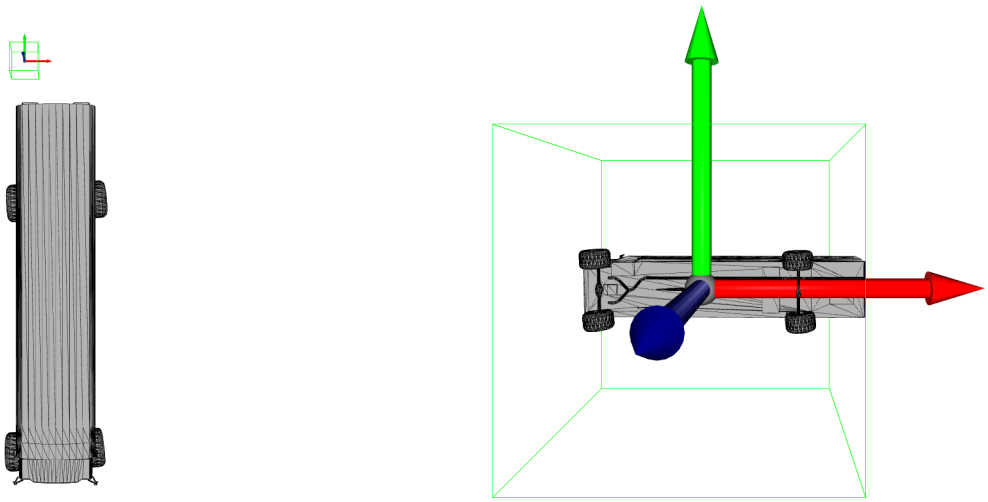
(a) AquaticAnimal *(b) Cell phone*

Figure 10. *Outliers with average number of faces and vertices: object from class "AcuaticAnimal" and object id "m56.obj" on the left (Figure a), and object from class "Cellphone" and object id "D00817.obj" on the right (Figure b)*

5.3 Normalization

In this step, we complete the preprocessing phase by introducing five different normalization processes: Remeshing, Translation, Pose (Alignment), Flipping (Mirroring) and Scaling (Size). The order of operations is crucial, as each step ensures that the following ones are not negatively affected by prior adjustments. This systematic approach preserves the shape's characteristics, ensuring that the subsequent feature extraction is both accurate and reliable.

For example we can observe the desired outcome of our normalization in Figure 11 where in Figure 11a is the original mesh and in Figure 11b we can see the mesh after utilizing the normalization function.



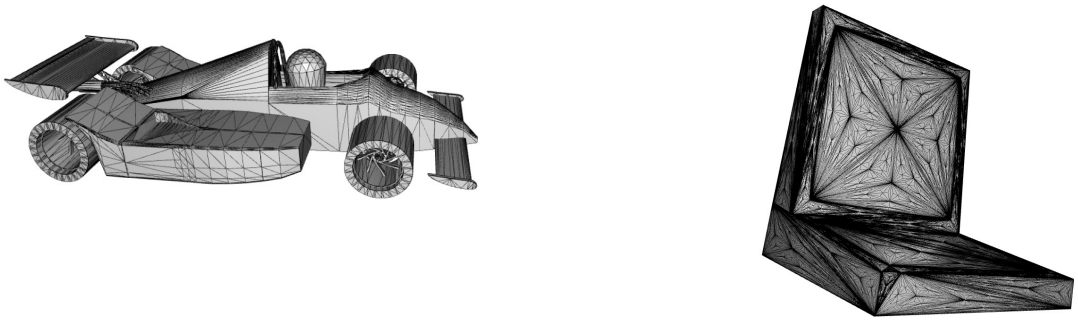
(a) Original mesh *(b) Mesh after normalization*

Figure 11. *An object from class "Bus" and object id "D00264.obj" before (Figure a) and after (Figure b) Normalization*

5.3.1 Remeshing

The first step is remeshing, which involves adding or removing vertices to achieve a finer mesh resolution or a more uniform distribution of vertices across the shape. Since this process can alter the object's barycenter, it is essential to perform Remeshing before any other normalization step. By modifying the shape's mesh structure early, we ensure that later steps, such as translation and alignment, operate on the final vertex distribution, preventing any shift in important geometric properties. To process our meshes, we used the VEDO library for data loading and manipulation[14]. We determined that maintaining around 10,000 vertices

per mesh strikes a good balance between preserving detail and improving computational performance. We acheived the goal of around 10,000 vertices per mesh by applying the decimate() function (for meshes that has more than 10,000 vertices) which reduces the vertex count by merging or removing vertices and simplifying polygons while retaining the visual quality of the mesh. By providing a parameter that represents the ratio of target vertices to the initial number of vertices in the mesh, we can accurately determine the proportion of vertices to retain. The underlying algorithm of this function is vtkQuadricDecimation which reduces the mesh by repeatedly collapsing edges until the desired size is reached. Edges are prioritized based on their "cost" to remove, which reflects how much they will affect the original shape of the mesh. This cost is measured using a quadric error measure, a mathematical representation that indicates how far each vertex is from the original surface. Each vertex has a matrix of planes related to it, and the distance from these planes to the vertex indicates the vertex's error (initially, this error is zero). When an edge is collapsed, the quadric error measures for its two endpoints are combined, creating a new optimal point for the collapse.Edges connected to this new point are then added back into the priority queue with their updated removal costs. This process continues until the mesh is reduced to the desired level or until it cannot be further reduced due to topological constraints[15]. For meshes with fewer vertices than required, we implemented a loop that repeatedly subdivides the mesh using the centroid subdivision method. This method calculates the centroid of each face to create new vertices, continuing the process until the mesh reaches the desired vertex count or a maximum of ten iterations is achieved. After subdivision, if the vertex count exceeds the target, the mesh is decimated to precisely match the target number by applying a fraction based on the current count.



(a) *Car after resampling* ***(b)*** *Computer after resampling*

Figure 12. *Shapes (that were outliers) after resampling: object from class "Car" and object id "m1510.obj" on the left (Figure a), and Object from class "Computer" and object id "D01071.obj" on the right (Figure b)*

In Figure 12a, you can see the object "car" which was previously identified as an outlier with a higher number of faces (see Figure 9b). After refining the mesh, the number of faces was reduced from 100,000 to 20,058, and the number of vertices decreased from 50,000 to 9,869, representing a significant reduction. This reduction is clearly visible in the visualization, as the mesh now has fewer wires, making it easier to view without zooming. The wires are less dense, yet the object still retains its details and shape.

Similarly, on the right of Figure 12b, we show the object "computer," which was initially an outlier with too few faces (see Figure 8a). After resampling, the number of faces and vertices increased dramatically from 43 faces and 29 vertices to 19994 faces and 10001 vertices. As expected, the computer now has far more faces, resulting in a denser wireframe. This is because the computer is a simpler structure compared to the car, so it doesn't require as much detail to represent it accurately.

5.3.2 Translation

Once remeshing is complete, the next step is translation, which centers the shape so that its barycenter coincides with the coordinate-frame origin (point [0,0,0]). This step is essential because subsequent operations (alignment, flipping, and resizing) rely on the object being centered at the origin. By translating first, we guarantee that any changes in shape orientation or size are performed symmetrically around the origin, maintaining consistency. This translation was done by loading each mesh and calculate its barycenter by averaging the positions of all

vertices of the shape, using the following formula:

$$\text{Barycenter} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{v}_i$$

Subsequently, the barycenter is subtracted from all the vertices in the mesh:

$$v_i' = v_i - \text{Barycenter}$$

where v_i is the original position of the vertex and v_i' is the new position after the translation. Thus, by applying this on all the shapes in the database, we align their center of mass (barycenter) with the coordinate-frame origin at (0,0,0).

5.3.3 Pose (alignment)

After centering the object, we will continue with aligning the object along the X and Y axis. This is done according to the PCA by computing the eigenvectors of the shape's covariance matrix and then projecting the vertices of the mesh according to e_1 and e_2 eigenvectors and X and Y axis respectively. The Z axis will be aligned according to e_3 eigenvector, which is the cross product of e_1 and e_2. We computed the alignment according to the following formula:

$$\begin{aligned}
\mathbf{x}_\mathrm{i}^{\text{updated}} &= (\mathbf{p}_\mathrm{i} - \mathbf{c}) \cdot \mathbf{e}_1 \\
\mathbf{y}_\mathrm{i}^{\text{updated}} &= (\mathbf{p}_\mathrm{i} - \mathbf{c}) \cdot \mathbf{e}_2 \\
\mathbf{z}_\mathrm{i}^{\text{updated}} &= (\mathbf{p}_\mathrm{i} - \mathbf{c}) \cdot (\mathbf{e}_1 \times \mathbf{e}_2)
\end{aligned} \tag{1}$$

In the equations, p_i refers to the i-th vertex of the mesh, with x_i, y_i, and z_i representing its updated coordinates along the principal axes. c denotes the barycenter (centroid) of the mesh, which is subtracted from each vertex to center the mesh. The transformation aligns the vertex positions with the principal eigenvectors e_1 and e_2, while z_i is computed as the projection along the cross product $\mathbf{e}_1 \times \mathbf{e}_2$ to maintain right-hand consistency.

Using equation 1 to align the mesh ensures that the mesh's pose is consistent across all shapes. The Alignment should come after translation because alignment calculations rely on the shape being centered at the origin; otherwise, rotation could lead to unpredictable movements. Additionally, if we performed alignment before translation, any shift in the mesh's barycenter would distort its orientation, complicating further steps.

5.3.4 Flipping (mirroring)

Once the shape is properly aligned, we apply the moment test to determine if the object should be flipped along its axes to ensure that the mesh's larger mass resides on the positive side of the axes. The flipping condition is assessed using the following formula:

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

Where:

- f_i is the flipping value for axis i.
- $C_{t,i}$ is the coordinate of the center of the t-th triangle along the i-th axis.
- $\text{sign}(C_{t,i})$ returns +1 or -1 based on whether the coordinate is positive or negative.
- $(C_{t,i})^2$ squares the coordinate to weight its contribution to the flipping value.

The sum across all triangles gives an overall measure for whether the mesh's mass distribution is biased toward the negative side of the axis, indicating the need for a flip.

This step must occur after alignment because the object needs to be in a standardized pose before checking whether it needs to be mirrored. Flipping before alignment could lead to incorrect orientation, as alignment could undo or distort the flip.

5.3.5 Scaling (Size)

The final normalization step is scaling, which adjusts the size of the object so that it fits within a standardized bounding box or matches a specific scale. This ensures that all objects have consistent dimensions, independent of their original sizes. Scaling is performed last because it operates on the final, fully normalized shape. If scaling were done earlier, subsequent operations like alignment and flipping could result in inconsistent scaling across different shapes, leading to inaccurate comparisons during feature extraction. In our case, we aimed to uniformly resize the meshes to fit tightly within a unit-sized cube. This normalization is achieved by applying a scaling factor, calculated using the following formula:

$$\text{Scale} \quad \text{factor} = \frac{1}{\max(\text{BoundingBoxDimension}_x, \text{BoundingBoxDimension}_y, \text{BoundingBoxDimension}_z)}$$

The scaling factor is then uniformly applied to all vertices using the scale() function provided by Open3D [16]. This ensures that the object's bounding box fits within the unit cube as required.

5.4 Validation of Normalization

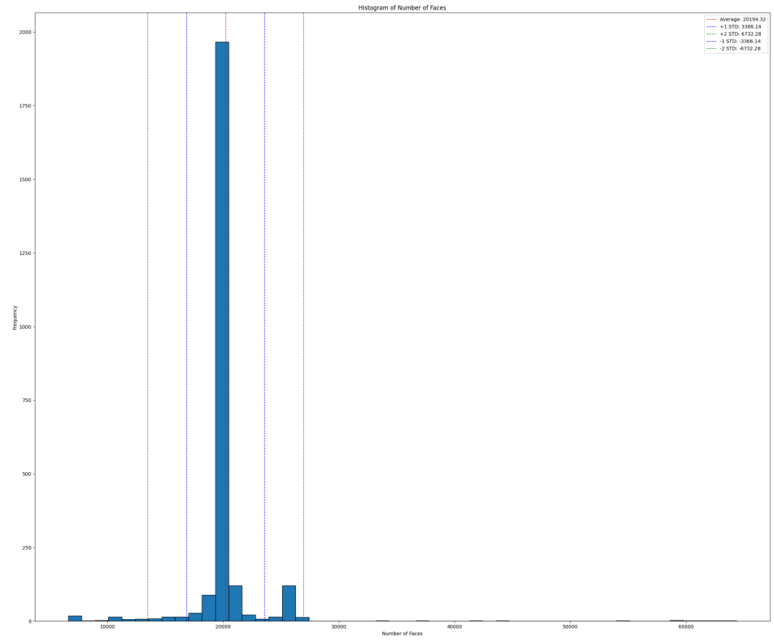
In order to validate that the normalization has been done correctly we utilized various tests.

5.4.1 Verifying Remeshing

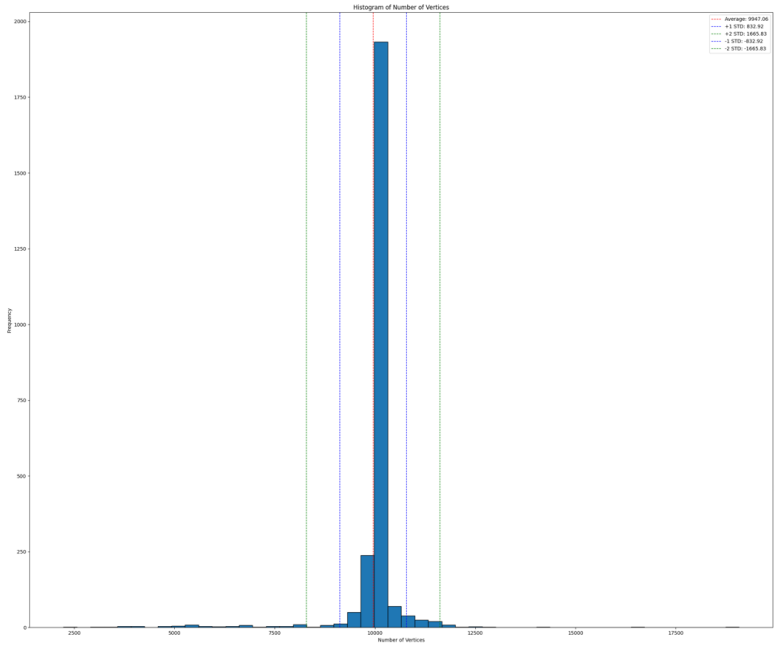
In order to verify whether the resampling was successfully applied to every mesh in our database, we used the tool implemented in step 5.2 to analyze the statistics of our data.

As shown in Figure 5, the histograms illustrate the distribution of shapes' face counts across various ranges before normalization, and in Figure 13a, the distribution of face counts after normalization. Following resampling, the face counts of meshes now range from approximately 5,000 to 30,000, a much narrower span compared to the original distribution, which ranged from 100 to 120,000 faces. Notably, most shapes, totaling around 2,000 objects, have face counts close to the average of 20194 , making them representative of the overall dataset. The uniformity of our dataset is further evident from the marked reduction in standard deviation, decreasing from 16191 to 3,366. This indicates that after resampling, the face counts of objects in our dataset are now more consistently aligned with the average, improving dataset uniformity.

Similarly, Figure 6 presents the distribution of vertex counts before normalization, and Figure 13b shows the distribution after normalization. We observe that most meshes have around 10,000 vertices, aligning with our target of 10,000 vertices set during resampling. The average vertex count is 9,947, slightly below the target but very close to the majority, indicating that the resampling process successfully centered most meshes around the desired vertex count. This consistency is further reflected in the standard deviation, which dropped significantly from 9,851 to 832, indicating that the average vertex count is now more representative of the dataset, as the spread of vertex counts across objects has become much narrower.



(a) *The frequency of shapes in our dataset across various ranged of face counts, after resampling*



(b) *The frequency of shapes in our dataset across various ranged of vertices counts, after resampling*

Figure 13. *Histograms of faces and vertices count in the dataset*

5.4.2 Validating Translation

To validate the success of the translation for all meshes in our database, we computed the standard deviation of the distances from the barycenter to the origin, both before and after normalization. Thus, before normalization, the standard deviation of the distribution of the distances from the origin was approximately 4.088584×10^6. After translation, this value reduced to 0.000104, with a mean of 1.185×10^{-5}, which is very close to zero. This confirms that the translation was successfully applied to all meshes.

For further and more visual validation of the translation process, we plotted the distributions of the distances of the barycenter from the coordinate-frame origin (see Figure 14). As observed in Figure 14, all distances are smaller than 0.002, with the vast majority being smaller than 0.00001. From this, we can conclude that the barycenter of all the meshes is very close to the origin.

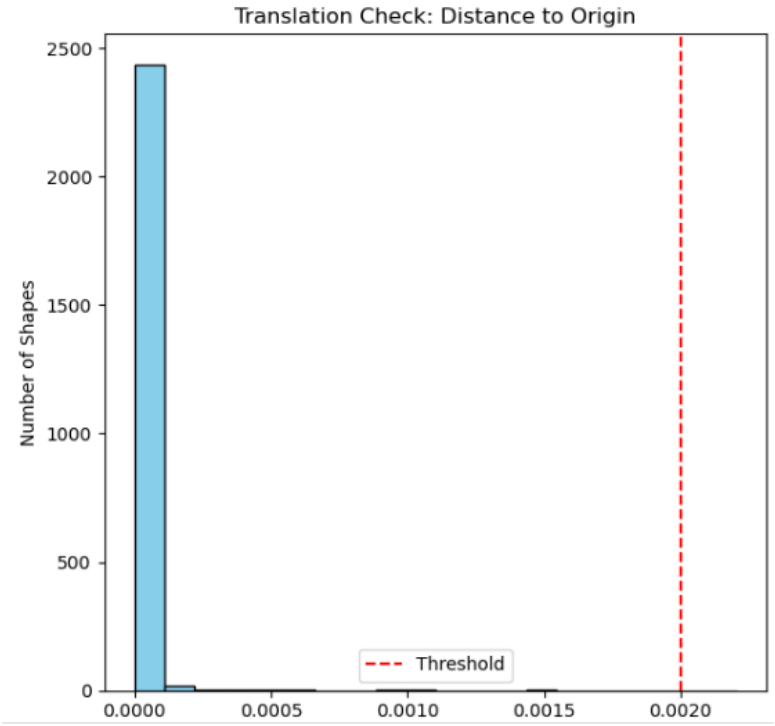


Figure 14. *Distance between the Barycenter and the origin (0,0,0) after Translation*

5.4.3 Verifying Alignment

To validate the effectiveness of the alignment normalization, we analyzed the consistency in orientation of the principal axes of the shapes before and after normalization. The goal of alignment normalization is to orient all shapes consistently along the coordinate axes, ensuring they face the same direction in 3D space. We computed the variance in the orientation distribution of each shape's principal axes, with a significant reduction observed after alignment. Specifically, the standard deviation of each axis's orientation angle (measured in degrees) dropped from approximately 39.90° (X-axis), 37.61° (Y-axis), and 38.34° (Z-axis) before alignment to minimal values of 0.04° (X-axis), 0.07° (Y-axis), and 0.06° (Z-axis) after alignment. This drastic reduction in variance indicates a consistent and precise alignment across all shapes. Additionally, the means for the X, Y, and Z axes were adjusted from 47.03°, 51.93°, and 39.09° before alignment to 0.00°, 0.01°, and 0.01° after alignment, further confirming the effectiveness of the alignment process.

Further validation was performed using a dot product test between each shape's eigenvectors and the coordinate system axes (x, y, and z). We established a threshold of 1×10^{-4} to verify that each shape's eigenvector aligned closely with the coordinate axes. As shown in Figure 15, after normalization, the dot product values for all shapes are around 1, with no values exceeding the threshold of 1×10^{-4}, confirming accurate alignment.

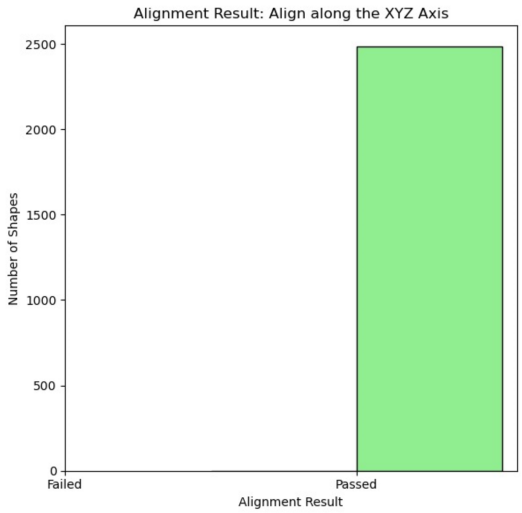


Figure 15. *Dot product values of each shape's principal axes with the coordinate system axes, showing that all shapes pass the alignment test after normalization.*

5.4.4 Verifying Flipping

To ensure that the flipping normalization works correctly, we implemented a validation method based on the mesh's geometry. Specifically, we checked whether the shape's barycenter (center of mass) is properly aligned in the positive half-space of the coordinate system after flipping. The procedure involves calculating the center of each triangle in the mesh and summing up their coordinates. If the sum of the center coordinates is positive along all axes (i.e., the mesh resides primarily in the positive half-space), the mesh is considered correctly flipped. The validation function `verify_flipping()` works by iterating through all triangles in the mesh, computing the centroid (center) of each triangle, and adding it to a cumulative sum of the coordinates for all triangles. The flipping values are then checked to ensure that they are non-negative along all axes. If all flipping values are greater than or equal to zero, it confirms that the mesh has been correctly flipped. The flipping values are calculated using the following formula:

$$\text{Flipping Value} = \sum_{i=1}^{N} \left(\frac{v0_i + v1_i + v2_i}{3} \right)$$

where $v0, v1, v2$ are the vertices of each triangle, and the sum represents the centroid coordinates.

Further validation is presented in Figure 16, which shows that all of the shapes pass the flipping validation test and are correctly aligned in the positive half-space.

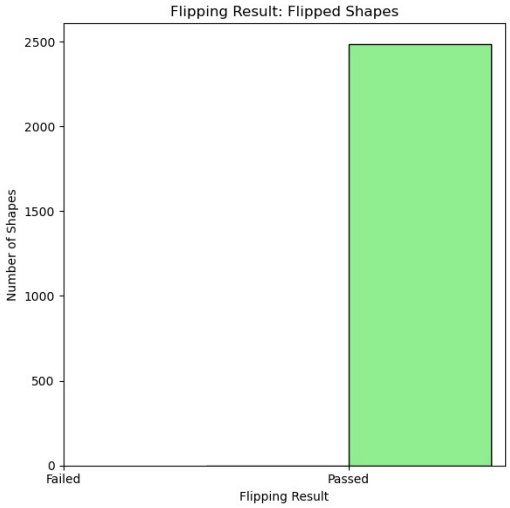


Figure 16. *Validation of flipping normalization, showing that all shapes pass the flipping test and are correctly aligned in the positive half-space.*

5.4.5 validating Scaling (Size)

To validate the successful application of scaling to all meshes, we calculated the standard deviation of the bounding box diagonal distances.Before scaling, the standard deviation was approximately 1.0238006×10^7, while after scaling, it reduced to around 0.1715, with a mean diagonal of 1.261.This suggests that the scaling was applied as intended.

For further and more visual validation of the scaling process, we plotted the distribution of the diagonal length of the meshes' bounding boxes (see Figure 17). We noticed that the majority of bounding box diagonal lengths range from 1 to approximately $\sqrt{3} \approx 1.73$, which is the diagonal of a unit cube. To be more precise, we conducted an additional test to verify that all axis lengths (in the x, y, and z directions) of the meshes are smaller than $1 + 1 \times 10^{-4}$ and that the largest axis has a length greater than $1 - 1 \times 10^{-4}$. All meshes successfully passed this test, confirming that they were all properly scaled to fit tightly within a unit cube.

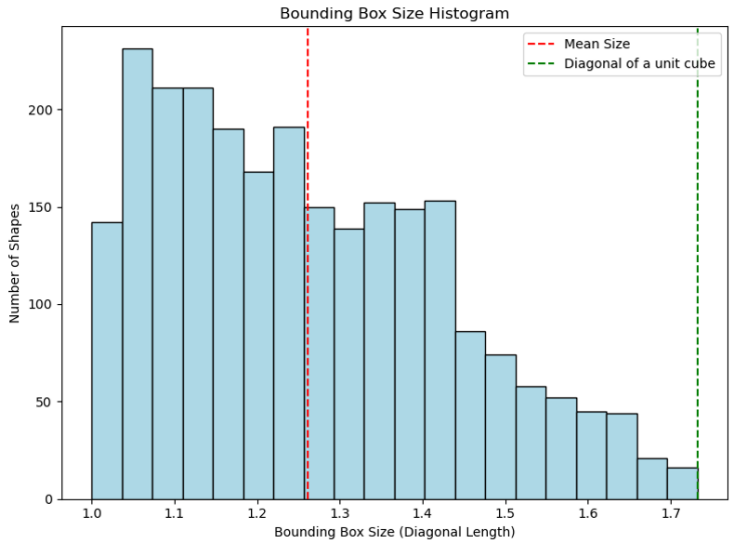


Figure 17. *Distribution of the diagonal length of the meshes' bounding boxes*

6 Step 3: Feature extraction

The purpose of this step is to define and extract elementary and shape property descriptors, from 3D objects. These descriptors, including both global shape descriptors (e.g., surface area, compactness) and shape property descriptors (e.g., angular distributions, vertex distances), allow us to characterize and compare various shapes more effectively. By using these extracted features, we can differentiate objects based on their geometric properties and later query these objects for specific attributes, which will be introduced in Section7.

6.1 Step 3.1: Elementary Descriptors

The first features calculated for each mesh in our dataset were the area and volume. The area, representing the surface of the mesh's faces, was computed using the area function from the Vedo library. The feature volume refers to the total space enclosed by the mesh's surface. For watertight triangular meshes, the volume is commonly calculated using the following formula:

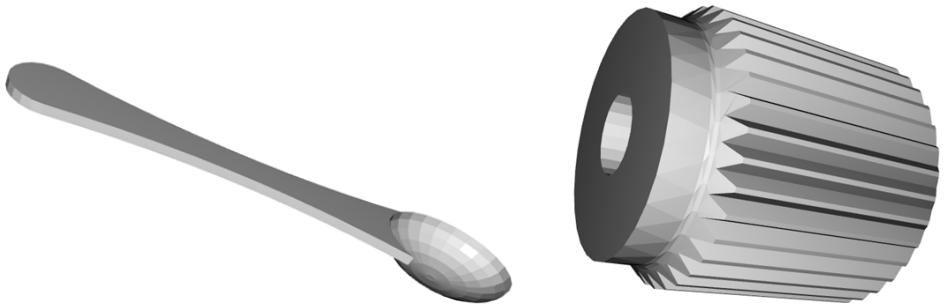
$$V = \frac{1}{6} \sum_{i=1}^{n} \vec{v}_1 \cdot (\vec{v}_2 \times \vec{v}_3)$$

Where $\vec{v}_1$,$\vec{v}_2$,$\vec{v}_3$ are the vertices of each triangle in the mesh. Although we used the fill_holes function provided by the Vedo library to try and make the mesh watertight, we were unable to close all gaps. As a result, because our shapes still contain some holes and are not fully watertight, we used Vedo's volume function. This function offers a simplified approximation by assuming the shape is mostly watertight, allowing it to estimate the volume based on that assumption.

To ensure that the volume function from Vedo did not overestimate the results, we compared the volumes by converting all the shapes into their convex hulls. A convex hull is the smallest convex shape that fully encloses the original shape. We then calculated the volume of each convex hull.

After this comparison, we observed that for almost every shape, the calculated volume from Vedo was smaller than the volume of the corresponding convex hull. This shows that although the volume function assumes the shapes are watertight, it does not produce exaggerated results and gives more accurate volumes than the convex hull approximation.

As shown in Figure 18a, the spoon, which has a relatively slim shape, has a volume of 0.001643, which is significantly smaller than the volume of the wheel in Figure 18b, measured at 0.480939085. These results align with expectations, as volume represents the total amount of space enclosed by a surface.



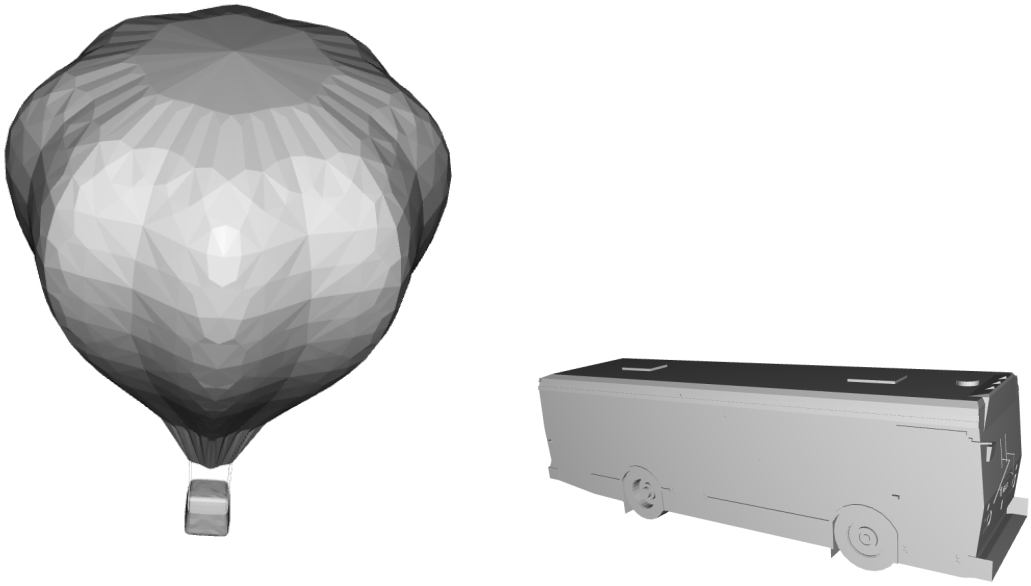
(a) *Spoon with volume 0.001643* ***(b)*** *Wheel with volume 0.480939085*

Figure 18. *Spoon (object id "D00085.obj") which has relatively slim shape compared to wheel (object id "m741.obj") which is more thick.*

Next, we calculated the feature compactness for each shape in our dataset, which essentially indicates the shape's proximity to a spherical form. Higher compactness values suggest a shape closer to a perfect sphere. Compactness is calculated using previously computed values for each shape's surface area and volume. The formula used is:

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

Where S represents the shape's surface area and V its volume. In Figure 19a, we see an aircraft buoyant shape with a compactness value of 170176, which is significantly larger than that of the bus shape in Figure 19b, which has a compactness of 10.10451. Which again the values are as expected since aircraft buoyant is more spherical than a buss.



(a) AircraftBuoyant (with object id "m1345.obj") with Compactness 170176 and rectangularity 0.002484

(b) Buss (with object id "D00516.obj ")with Compactness 10.10451 and rectangularity 1.062982

Figure 19. *Visualization of a spherical balloon and a rectangular bus shape for comparison.*

We also derived rectangularity as a feature, which measures how closely a shape resembles a rectangular form; higher rectangularity values indicate a shape closer to rectangular. To calculate rectangularity, we divide the actual volume of the shape's by the volume of the oriented bounding box (OBB). The actual volume is as previously calculated, while the OBB volume is computed separately. As observed in Figure 19, the bus shape, which is closer to a rectangular form, has a higher rectangularity value than the aircraft buoyant shape, which is closer to spherical.

Furthermore, we computed convexity for each mesh to determine if the mesh is curved or flat. We derived convexity by first converting the mesh into its convex hull using the ConvexHull function provided by VEDO, calculating its volume, and then dividing the original volume by the convex hull volume.

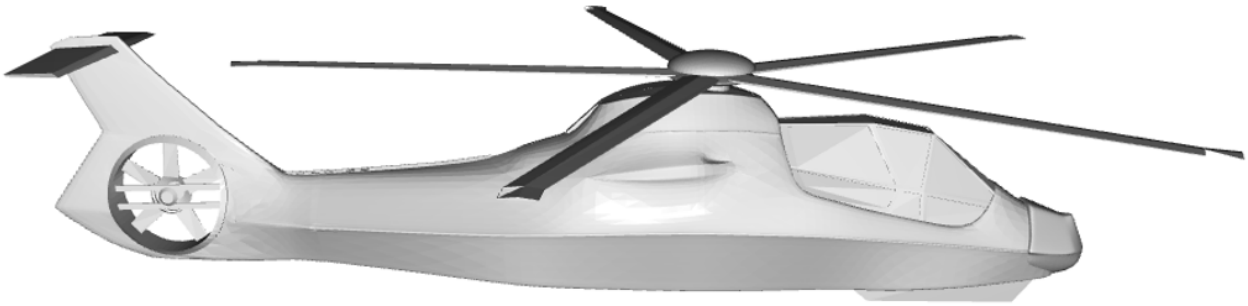
Next, we calculated the diameter of each shape by measuring the distances between all shape points and selecting the maximum distance, which represents the shape's diameter.

Lastly, we determined eccentricity, which measures how stretched a shape is. A high eccentricity value indicates an elongated shape, while a low value suggests a more rounded shape. To calculate eccentricity, we found the ratio of the largest to the smallest eigenvalues of the covariance matrix for each shape.

As shown in Figure 20, the object in question is a sword, which can be considered relatively slim and long. This suggests we would expect a high eccentricity value, representing how stretched the object is, and a low volume value, indicating its thickness. As observed from the results, the eccentricity is indeed large, with a value of 1449.461318, while the volume is relatively small, with a value of 0.000251563.

Figure 20. *Visualization of a "sword" with object id "D00540.obj" with eccentricity 1449.461318 and volume of 0.000251563.*

In Figure 21, we present a helicopter mesh, which is visually more complex than the previously shown meshes, along with all its fundamental descriptors.



Object	Area	Volume	Compuctness	Rectangular.	Convexity	Diameter	Eccentricity
HELICOPTER	0.77	0.011	32.68	0.054	0.14	1.04	48.36

Figure 21. *The values of the elementary descriptors of Helicopter mesh (D00751.obj).*

6.2 Step 3.2: Shape Property Descriptors

In the previous step, we calculated simple global descriptors that produce a single real value. In this step, we'll enrich our feature vector by calculating shape-property descriptors for each mesh in our dataset. The first descriptor, called A3, is the angle between three randomly selected vertices in a mesh. We calculate this angle 10,000 times per mesh to gather a variety of angles. The process involves selecting three random vertices, p1, p2, and p3, constructing vectors for the line segments p1p2 and p2p3, and using their dot product to find the cosine of the angle between them. After taking the inverse cosine and converting to degrees, we store the result as an instance of the A3 descriptor.

The next descriptor, D1, measures the Euclidean distance between the barycentre of the mesh and a random vertex. We calculate this distance for as many random points as there are vertices in the current mesh. Similarly, D2 is calculated as the distance between two random points, using 10,000 random pairs. Following that, we calculate D3, which represents the area of a triangle formed by three random points. To calculate the area of the triangle defined by points p1, p2, and p3, we first create two vectors: AB from p1 to p2, and AC from p1 to p3, representing the two sides of the triangle. By computing the cross product of these vectors, we obtain a new vector perpendicular to both AB and AC. The magnitude of this cross product represents the area of the parallelogram formed by the two vectors. To calculate the area of the triangle, which is half the area of this parallelogram, we take half of the magnitude of the cross product, i.e., $0.5 \times$norm of the cross product.

If we extract this feature 10,000 times as described previously, it results in 10,000 combinations of 3 points uniformly distributed across the space of possibilities. This corresponds to approximately $10,000^{\frac{1}{3}} = 21$ vertices being considered, which is clearly too low. Therefore, we decided to extract D3 100,000 times, ensuring that around 46 vertices are considered.

For the final descriptor, D4, we compute the volume of a tetrahedron formed by four random points in space. To do this, we construct a matrix where each row represents a point with three coordinates, followed by an additional column of ones. The inclusion of this extra column enables us to calculate the volume of the tetrahedron using the determinant method. The determinant is a scalar value derived from a square matrix. The volume V of the tetrahedron formed by the four points can be calculated using the formula:

$$V = \frac{1}{6} \left| \det \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{pmatrix} \right|$$

In simpler terms, the volume is the absolute value of the determinant of this matrix, divided by 6. Throughout the descriptor calculations, vertices are not excluded from repeated selection, so it's possible to calculate the same angles, distances, or areas multiple times within a mesh. After obtaining all descriptors, we end up with an array of values for each descriptor in each mesh. We then convert these arrays into histograms. To ensure consistency, we use the same number of bins for each descriptor's histogram and calculate bins based on the global minimum and maximum values across all meshes[1]. Each histogram is created with 100 bins spanning

[1]We determined the range of bins for each descriptor's histogram individually based on its global minimum and maximum values, calculated across all meshes for each Shape Property descriptor.

this global range, and these values are saved for future use in querying new objects.

Figure 22 illustrates three objects from our dataset: two that are highly similar and belong to the same class, Humanoid, and one that is distinctly different, belonging to the AircraftBuoyant class. To assess the effectiveness of our descriptor extraction, we present the histograms of all descriptors for these three objects. As anticipated, the histograms for the first object (Humanoid, represented by the orange line) and the second object (Humanoid, represented by the green line) are quite similar, reflecting their shared class. In contrast, the histograms for the third object (AircraftBuoyant, represented by the blue line) show significant differences across all descriptors (see Figure 23).

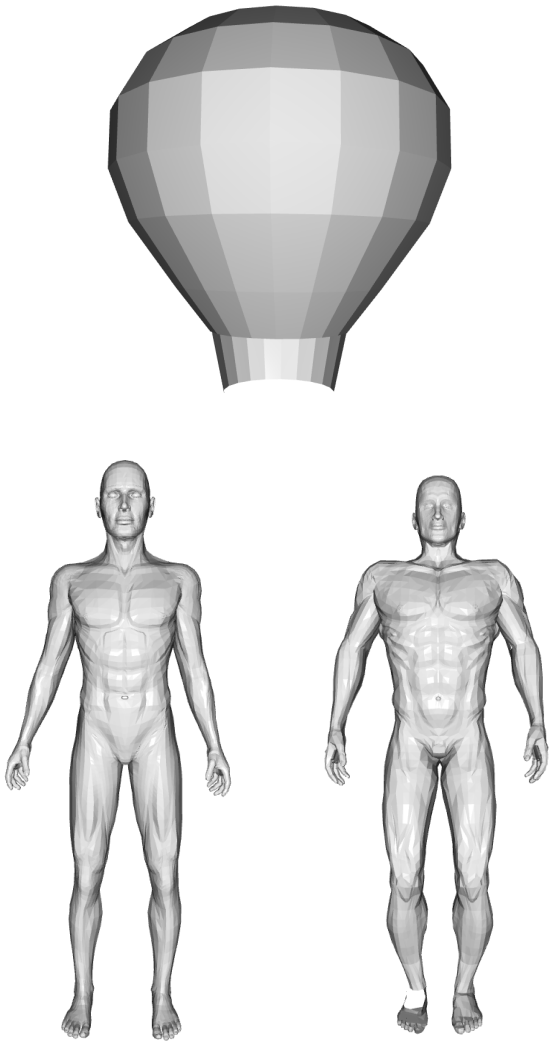


Figure 22. *Visualization one object from the class AircraftBuoyant (m1337.obj) and two object of the same class "Humanoid" (left: "m156.obj", right: "m140.obj")*

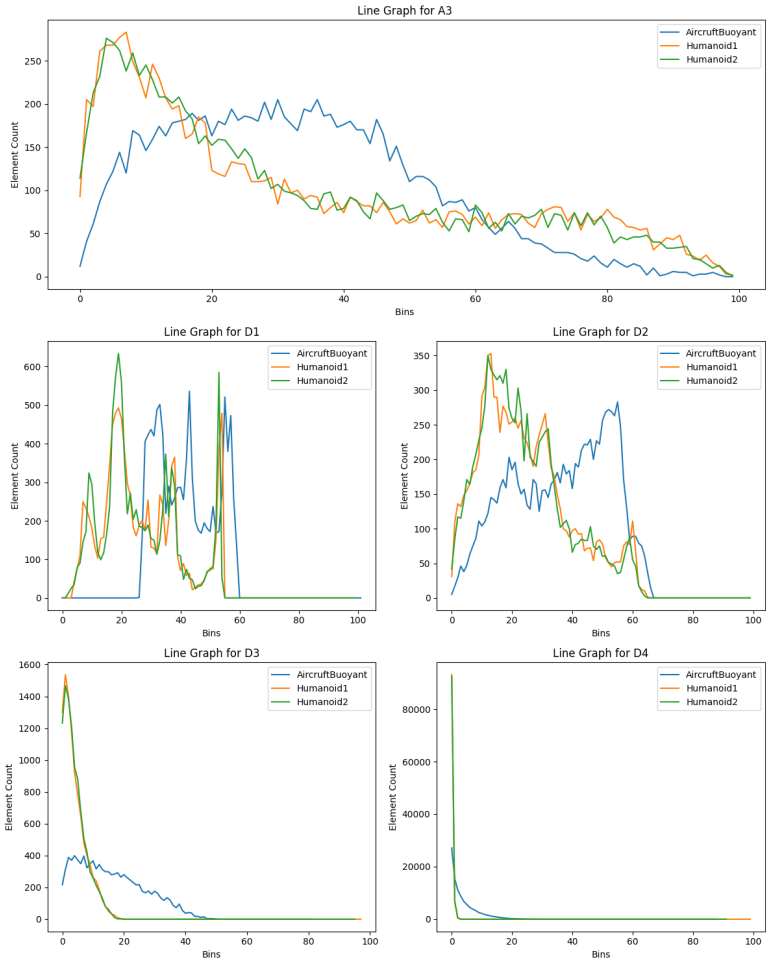


Figure 23. *Visualization of five line graphs, each representing a descriptor and displaying the histogram values for three objects: AircraftBuoyant (blue line with object id m1337.obj), humanoid1 (orange line with object id "m156.obj"), and humanoid2 (green line with object id "m140.obj").*

A plausible feature for shape retrieval should exhibit similar values for objects with similar shapes and different values for objects with distinct shapes. As previously shown in Figure 23, where we compared dissimilar objects (such as humanoid and aircraft), we observed that objects within the same class tend to have similar descriptor histograms, while objects from different classes exhibit distinct histograms. In this analysis, we aim to examine the intra-class homogeneity and inter-class heterogeneity across a variety of classes in our dataset. Intra-class homogeneity refers to the similarity of histograms within the same class, while inter-class heterogeneity refers to the dissimilarity of histograms across different classes.

To visualize this, Figure 24, presents the histograms of all descriptors for objects across six different classes in our dataset. For descriptors A3, D3, and D4, we observe that the histograms within each class have a similar distribution, indicating that objects within the same class share similar values. For descriptor D2, while we still observe some similarity within classes, there is noticeable noise, particularly in the "Bird" and "Humanoid" classes, where the histograms show more variability compared to other classes and descriptors. Finally, for descriptor D1, the histograms exhibit more evident noise, likely due to drastic fluctuations in the element count of the bins. These fluctuations suggest that the values of the bins vary significantly in this descriptor. Additionally, the presence of noise may arise from the fact that we visualized descriptors across all objects within each class, where object-to-object differences can contribute to the variability in the histograms.

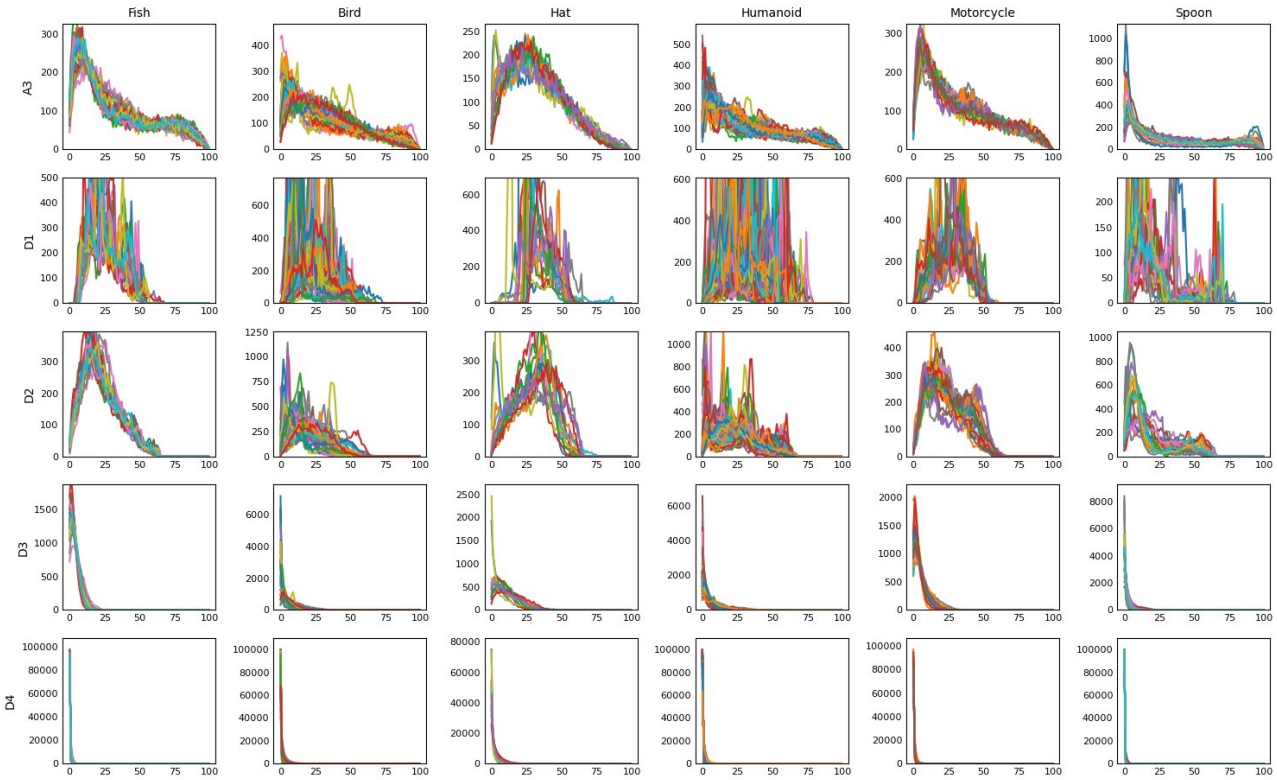


Figure 24. *Visualization of the descriptor histograms for different classes in our dataset*

7 Step 4: Querying

In this step, we implemented an end-to-end querying system with a graphical user interface (GUI) using Flask. This interactive interface allows users to upload a 3D mesh object and retrieve the k most similar shapes from our database.

First, we normalized the query shape following the process described in Step 2, ensuring that the user-provided object has the same structure as the data in our dataset. This allows for accurate comparisons based on the features we extracted in Step 3. We extract the same features from the query mesh to maintain consistency with the dataset.

Our features are divided into two categories: elementary descriptors, represented as single values for each object, and property descriptors, represented as histograms where each bin reflects the element count. We apply different standardization methods to each category for consistent value ranges across the dataset, including the query object.

For elementary features, we applied Z-score standardization, resulting in data with a mean of zero and a standard deviation of one. The formula for Z-score is:

$$Z = \frac{X - \mu}{\sigma}$$

Where X is the data point, μ is the mean, and σ is the standard deviation. High absolute Z-scores indicate potential outliers.

For histogram features, we normalized by dividing by the histogram's area (the total bin count), which scales values from 0 to 1 within each histogram. Unlike elementary feature standardization, this normalization is applied independently for each histogram.

Next, we calculated the distance between the query object and each object in the dataset. For elementary features, we used Euclidean distance, which is calculated as:

$$d(\mathbf{Q}, \mathbf{P}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

where Q and P are the elementary features of the query and the dataset point, respectively. For histogram features, we used Earth Mover's Distance (EMD) to compute a single similarity measure between histograms. EMD quantifies the effort required to transform one histogram into another, ideal for comparing feature vectors when individual feature identities are less critical. The formula for EMD is:

$$\text{EMD}(P, Q) = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} f_{i,j} d_{i,j}}{\sum_{i=1}^{m} \sum_{j=1}^{n} f_{i,j}}$$

Where Q and P represent the elementary features of the query and the dataset object, respectively, $f_{i,j}$ denotes the amount of "flow" (mass being transported) from point i in the distribution P to point j in the distribution Q, and $d_{i,j}$ is the distance between points i and j.

To account for differences in the range between elementary and histogram features, we standardized all computed distances for the histogram features using Z-score normalization. For instance, with the D1 feature, we first calculated the distances between the query and each shape in our database, then standardized these distances based on their spread. This approach was applied consistently across all histogram features, ensuring that even small variations for example in D1 have a comparable impact to larger variations in elementary features. We then combined the standardized EMD distances of histogram features with the Euclidean distances for elementary features by summing them. Finally, the k most similar objects are returned, based on the smallest combined distances from our dataset, excluding the query object.

As shown in Figure 25, our GUI allows users to upload any mesh object and receive the k most similar objects from our dataset (see Figure 26). First, we display the query object, followed by the ranked results with the most similar shape at the top. Each result displays the distance from the query, and users can interact with the shapes for a closer examination.

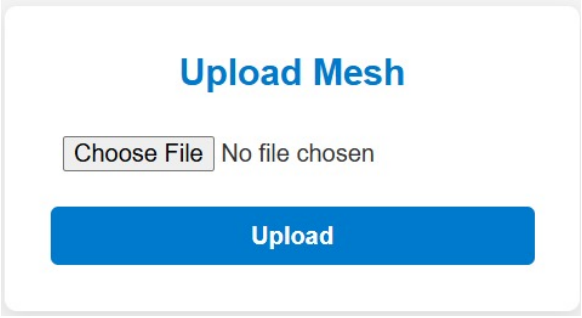


Figure 25. *Upload button allowing the user to upload a mesh for querying.*

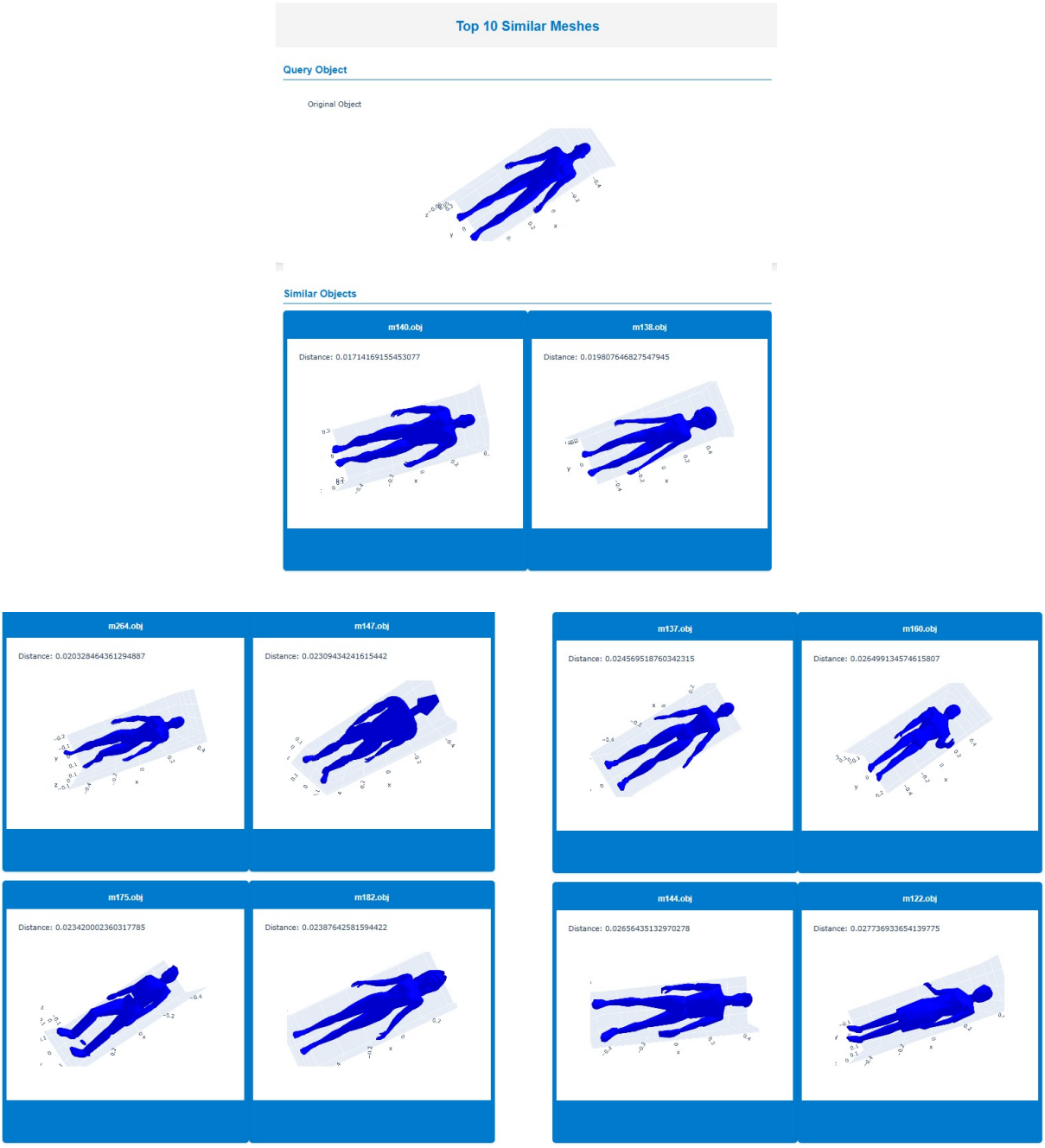


Figure 26. *Visualization of the result for step 4 using as a query the object with id "m156.obj" from class "Humanoid"*

8 Step 5: Scalability

In this step, we focused on enhancing the scalability of the Content-Based Shape Retrieval (CBSR) system by optimizing the query engine to efficiently handle larger shape databases. Additionally, we implemented a dimensionality reduction (DR) engine to effectively visualize shape similarity based on the feature vectors.

8.1 Querying with ANN

To optimize the querying engine, instead of using our custom distance function and comparing every pair as done in step 4, we implemented an approximate nearest neighbor (ANN) search using a kd-tree structure.

To achieve this, we utilized the KDTree function from the scipy.spatial library. The KDTree function allows efficient spatial searches by organizing data points in a k-dimensional tree structure. This binary tree partitions the space recursively, with each node representing an axis-aligned hyperrectangle. The tree's construction involves choosing an axis and a splitting point according to the "sliding midpoint" rule, which prevents highly

imbalanced divisions and ensures the resulting cells are not overly elongated.

Once constructed, the kd-tree can quickly return the k approximate nearest neighbors for any query point, along with their distances. By default, KDTree uses Euclidean distance for these calculations.

Figure 27 shows the output of the approximate nearest neighbors (ANN) search visualized in our GUI. As seen, this approach produces slightly less accurate results compared to the custom method used in step 4 (see Figure 26), but it is computationally faster. While the ANN approach sacrifices some accuracy, its speed may offer benefits in certain applications. Further analysis of its trade-offs will be conducted in step 6.

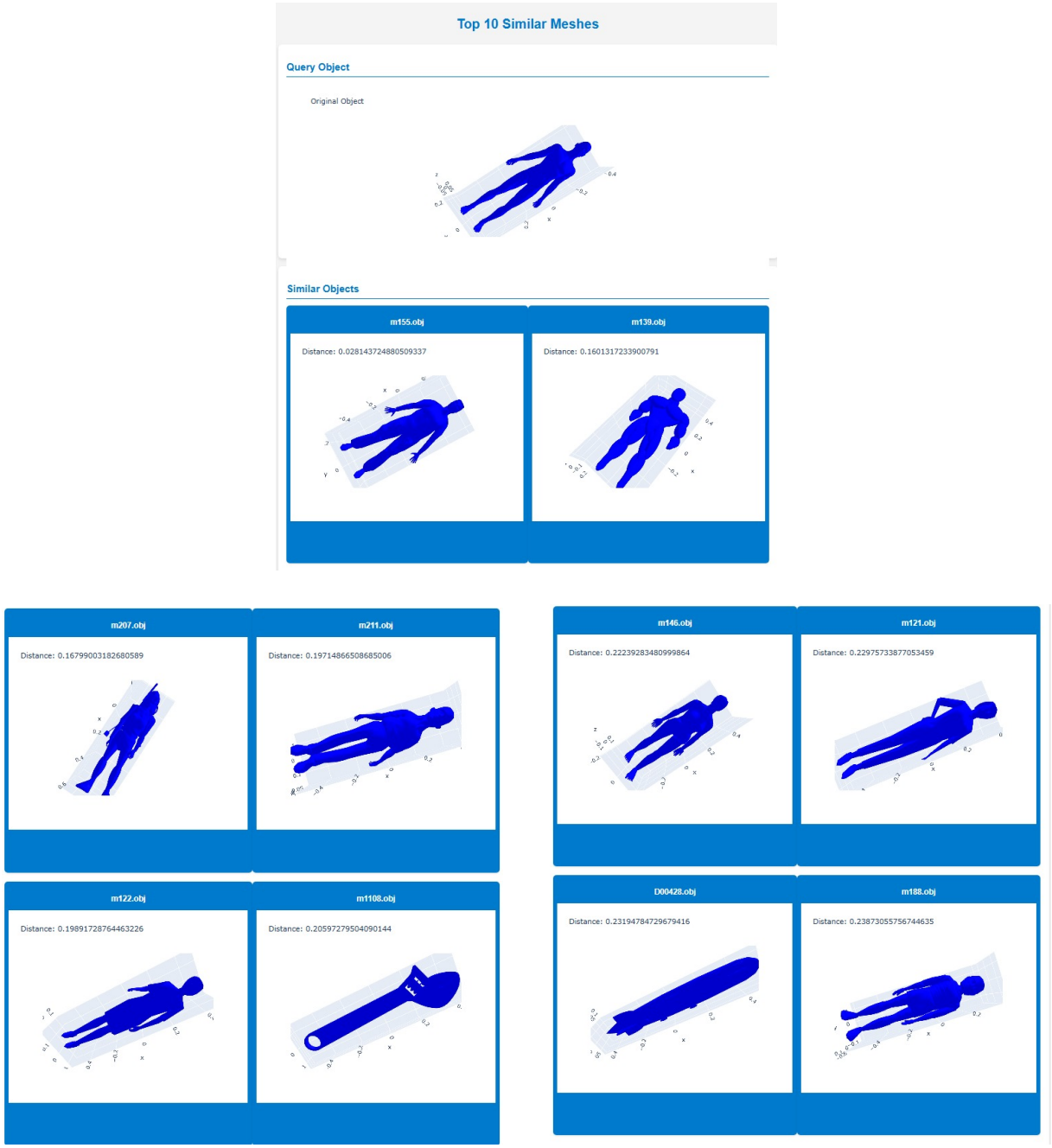


Figure 27. *Visualization of the result for step 5 using as a query the object with id "m156.obj" from class "Humanoid"*

8.2 Dimensionality Reduction with TSNE

In this step, we aim to perform dimensionality reduction with a target dimension of 2, reducing our feature space to a 2D space. To achieve this, we use t-SNE, which visualizes how the feature vectors are distributed in the reduced-dimensional space.

T-SNE is a technique used for visualizing high-dimensional data by transforming the similarities between data points into joint probabilities. It aims to minimize the Kullback-Leibler divergence between the joint probabilities of the lower-dimensional representation and the original high-dimensional data. A key feature of t-SNE is its non-convex cost function, which means that different initializations can lead to different results [17]. The cost function is the sum of the Kullback-Leibler divergences over all data points and is computed as follows:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{ij} \log \frac{P_{ij}}{q_{ij}}$$

Where P_i represents the conditional probability distribution over all other data points given data point x_i in the high-dimensional space, Q_i represents the conditional probability distribution over all other map points given map point y_i in the low-dimensional space, while p_{ij} and q_{ij} the conditional probabilities between the high-dimensional data points x_i , x_j and low-dimensional data points y_i , y_j respectively [17]. Consequently to minimize the cost function aims to find low-dimensional data where $p_{ij} = q_{ij}$ and therefore $\log \frac{P_{ij}}{q_{ij}} = 0$.

8.2.1 Hyperparameters Tuning and Result

To optimize the t-SNE algorithm for our visualization, we conducted a grid search to fine-tune its hyperparameters, evaluating each configuration using the silhouette score.[2]
The following parameters were considered for tuning:

1. **n_components**: 2 - The number of dimensions to represent the dissimilarities.

2. **Perplexity** - Optimal value: 5.
   We considered values from 5 to 60, incrementing by 5. Perplexity is a parameter that influences the optimization process and the resulting embedding and controls the effective number of neighbors that each data point considers during the optimization process.

3. **Learning Rate** - Optimal value: 50.
   We considered the following values: [12, 15, 20, 30, 50, 100]. The learning rate determines the pace of convergence; a low learning rate can lead to slow convergence, while a high learning rate may cause instability.

4. **init**: - Optimal value: "PCA".
   We considered two values ["PCA","Random"] parameter that determines the initialization strategy for the optimization algorithm. It specifies how the algorithm should initialize the embedding of the data points in the low-dimensional space before optimizing their positions.

5. **n_iter** - Optimal value: 5,000.
   We considered the following values: [500, 800, 1,000, 3,000, 5,000, 10,000]. The number of iterations (*n_iter*) the optimization algorithm will run.

6. **Metric** - Optimal value: "euclidean" .
   We tested multiple distance metrics, including ["euclidean", "cosine", "manhattan", "chebyshev"]. The metric is responsible for how our algorithm computes pairwise distances between points.

7. **Early Exaggeration** -Optimal value: 12.
   we considered the following values: [4, 6, 10, 12, 18, 20, 50]. Early exaggeration controls the strength of attraction between points in the early stages of optimization. A higher early exaggeration helps separate clusters more distinctly in the initial phase of optimization, which can aid in forming clearer clusters.

[2]The silhouette score is a ratio scale between -1 and 1, representing the dissimilarity of a sample to its own cluster compared to other clusters; a score closer to 1 indicates well-defined clusters, while a score closer to -1 indicates that the sample is closer to another cluster [18].

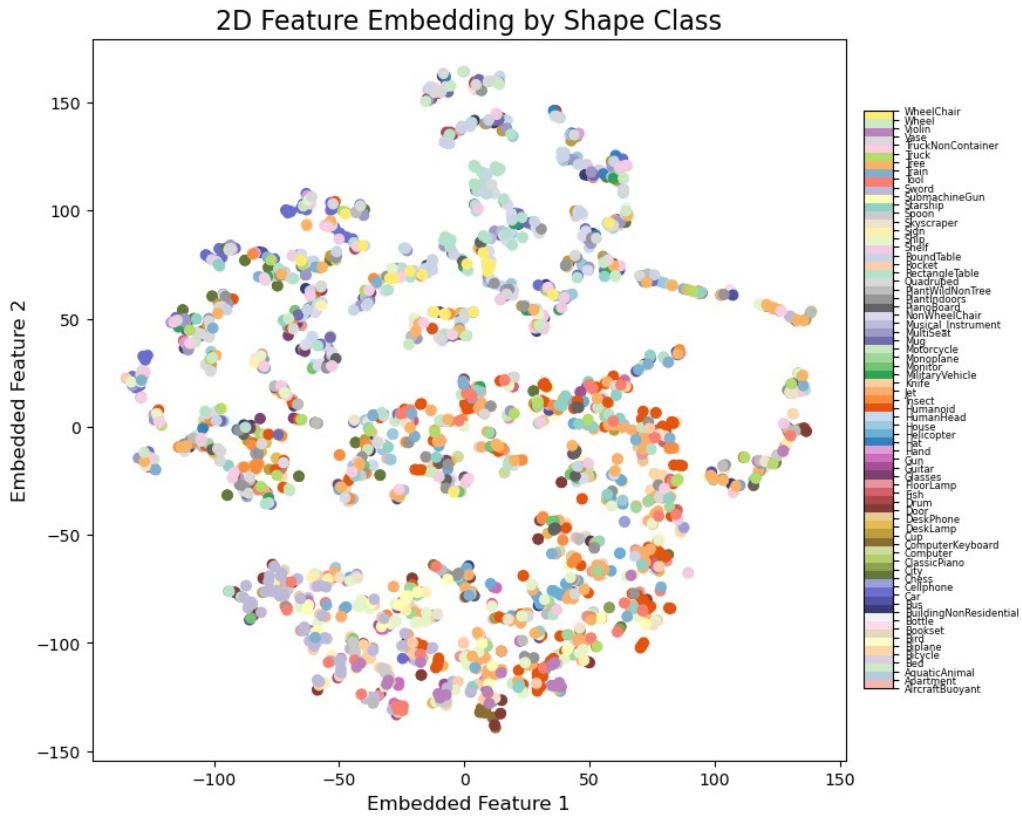


Figure 28. *TSNE scatter-plot*

With these results, we achieved an optimal configuration for our t-SNE. Figure 28 shows a scatterplot of our data, where each point represents an object in the dataset, and each color corresponds to a specific shape class. Despite extensive tuning of the t-SNE parameters, the results are still not as promising as expected. This could be attributed to several factors. First, the data collection process may not have captured the full range of variability within the shapes, leading to limited discriminative power. Second, the high dimensionality of the feature vectors resulting from a very large vector space may also be affecting the performance of t-SNE, as the algorithm struggles to capture meaningful structures in such high-dimensional data. These factors combined may explain why the resulting visualizations and clustering are less effective than anticipated.

9 Step 6: Evaluation

In this section, we evaluate and compare the performance of the retrieval systems constructed in the previous steps. The first model, developed in Step 4, takes as input a mesh object and calculates the distance between the query object and every object in the dataset, returning the K most similar objects. The second model, developed in Step 5, follows the same logic as the first but uses the KDTree structure to calculate approximate nearest neighbours. This approach was chosen to optimize retrieval time for large shape datasets.

Finally, the third model is an enhancement of the initial one, where we apply weighted descriptors to improve retrieval accuracy. Specifically, A3, D1, and D2 contribute 70% of the total weight for histogram features, while D3 and D4 each contribute 15%. This weighting scheme was based on observations of the histogram features in Figure 24, where A3, D1, and D2 show greater heterogeneity between the 24 classes compared to D3 and D4. However, this observation was qualitative, and a quantitative assessment of inter-class heterogeneity is planned for future work due to the high time complexity.

To evaluate the models, we queried each object in our dataset, setting the number of objects returned by the model to match the class size of the query object. This approach helps control for false positives and false negatives. If we were to ask the model to return more objects than the actual class size, it could increase false positives by including objects from other classes that are incorrectly identified as similar. Conversely, setting the model to return fewer objects than the class size could increase false negatives by omitting objects from the same class that should be considered similar. This balance ensures the retrieval process remains as accurate as possible.

9.1 Metrics

For each model, we evaluated performance using multiple metrics, including accuracy, precision, recall, and F1 score.

9.1.1 Precision

Motivation: Precision assesses the quality of positive predictions, ensuring that when the model predicts a positive outcome, it is likely to be correct. High precision indicates that the model is not retrieving shapes from other classes incorrectly, thus minimizing false positives. This metric is particularly crucial when minimizing false positive results is more important than retrieving all possible positive cases.

Implementation: Precision is calculated as the ratio of True Positives (correctly retrieved shapes) to the total number of shapes returned by the model (True Positives + False Positives):

$$\text{Precision} = \frac{TP}{TP + FP}$$

9.1.2 Recall

Motivation: Recall measures how effectively the model identifies all relevant shapes within the dataset. High recall ensures that most or all relevant shapes are retrieved, which is critical when the cost of missing a positive case (false negative) is high.

Implementation: Recall is calculated by comparing the number of True Positives to the total number of relevant shapes (True Positives + False Negatives):

$$\text{Recall} = \frac{TP}{TP + FN}$$

9.1.3 F1 Score

Motivation: The F1 Score is the harmonic mean of Precision and Recall, providing a balanced assessment of both the correctness (precision) and completeness (recall) of retrieval. It is particularly useful in cases where improving precision may reduce recall, or vice versa. The F1 Score is ideal when both retrieving relevant shapes and minimizing irrelevant shapes are equally importan.

Implementation: The F1 Score is computed as:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

9.1.4 Accuracy

Motivation: Accuracy provides an overall measure of the system's performance by calculating the proportion of correct predictions, both relevant and irrelevant shapes. It offers a high-level overview of how well the model is functioning in general, but may be less informative in imbalanced datasets.

Implementation: Accuracy is computed by dividing the sum of True Positives and True Negatives by the total number of predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

9.2 Result

While accuracy is a common measure of model performance, it can be misleading in cases of imbalanced class distributions. When one class significantly outnumbers another, especially in our case where the query result matches the query class size, this imbalance can lead to fewer true negatives (TN) for the largest class, inflating the accuracy score. This skew makes accuracy a less reliable indicator of the model's true performance. To obtain a fairer assessment, alternative metrics such as Precision, Recall, and F1 Score are more suitable, as they

account for class imbalance and provide a more accurate evaluation of model effectiveness.

For each model, we evaluated performance using the metrics previously outlined. As shown in Table 1, precision and recall are identical for each model. This is due to the fact that, in our case, false positives (FP) and false negatives (FN) are equal. Specifically, false positives represent instances that our model misclassified as the query class, while false negatives are instances of the query class that our model failed to identify. This balance between false positives and false negatives results in identical precision and recall values across all models. Since precision and recall are equal in our case, we can derive from the formula of the F1 score (Equation of F1 described in subsection 9.1.3) that Precision = Recall = F1. Therefore, we will focus exclusively on the F1 score moving forward, as it combines both precision and recall, making it more appropriate for our class distribution, especially since accuracy alone is not a reliable metric in this context.

Model	Precision	Recall	F1 Score	Accuracy
First Model	0.161663	0.161663	0.161663	0.976049
Second Model	0.153608	0.153608	0.153608	0.975902
Third Model	0.180180	0.180180	0.180180	0.976606

Table 1. *Comparison of Model Metrics*

9.2.1 F1-Score for each class

As shown in Figure 29, which visualizes the distribution of F1 scores for each class across our three models, each dot represents the average F1 score for a class, while the line indicates the standard deviation. Notably, the best-performing classes in our dataset are the Bicycle class for Model 1 with an F1 score of 0.396, the HumanHead class for Model 2 with a score of 0.385, and again the Bicycle class for Model 3 with an F1 score of 0.440. On the other hand, the worst-performing classes are the BuildingNonResidential class for Model 1 with an F1 score of 0.053, the FloorLamp class for Model 2 with a score of 0.052, and the Train class for Model 3 with an F1 score of 0.073.

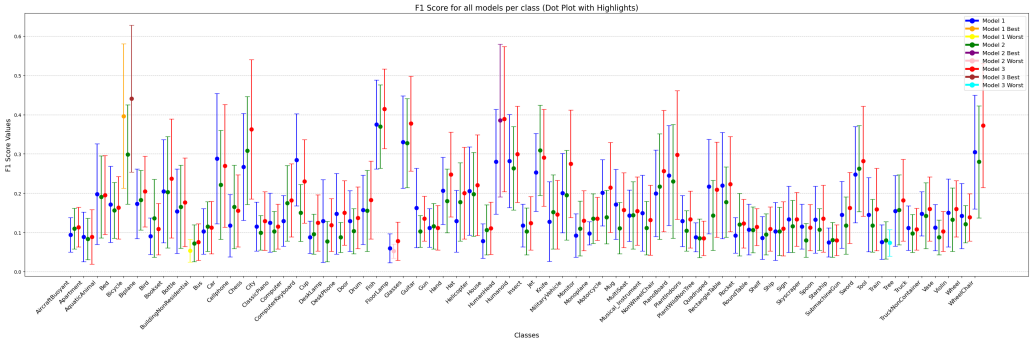


Figure 29. *Average standardized F1-Score values for each class across the models, with error bars representing the standard deviation of variance.*

9.3 Discussion

Now, we will compare the F1 scores of our three models. As shown in Table 1, the enhanced model (Model 3) achieved the highest performance with an F1 score of 18%, while the implementation using the ANN had the lowest F1 score at around 15%. However, the ANN performs significantly faster, requiring only 0.004 seconds to process a query, compared to around 3 seconds for the other models. This makes the ANN approximately 750 times faster. The reason for this performance difference is that both of the other models calculate the distance between the query and every object in the dataset, while the ANN uses a kd-tree structure, which efficiently traverses a binary tree without checking every object. Although the ANN only approximates the nearest neighbours, this trade-off results in lower F1 score.

In contrast, our models, which use custom distance functions, check every possible pair, ensuring a more accurate comparison. Additionally, both of our models use Earth Mover's Distance (EMD), which is better suited for histogram features, contributing to their improved accuracy. However, we observe that all the models have very low F1 scores. This could be attributed to the noise present in the descriptors, as indicated in Figure 24,

and the significant heterogeneity in the appearance of shapes within many classes. In other words, shapes from the same class often exhibited substantial visual dissimilarity (see Figure 30).

Ultimately, the choice of the best model depends on the trade-off between performance and time complexity. However, in our case, while the performance differences among the three models are relatively small, the difference in query time is substantial.

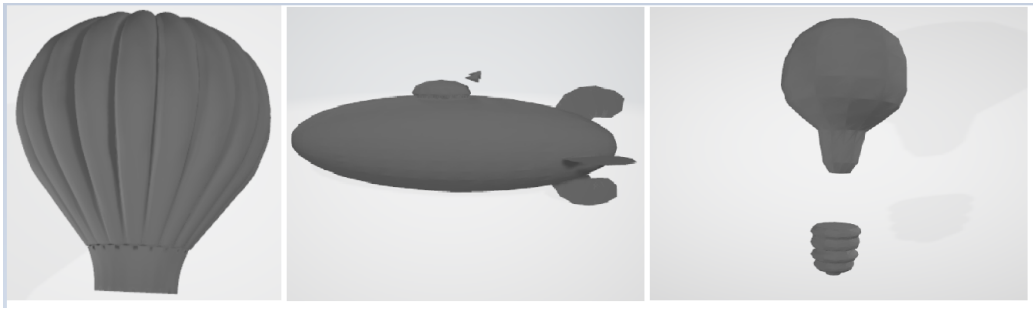


Figure 30. *Objects with IDs "m1338.obj", "m1346.obj", and "m1341.obj" from class "AircraftBuoyant"*

10 Conclusion

In this project, we developed a 3D shape retrieval system that relies on content-based techniques to compare and retrieve shapes based on their geometric properties. The system was built using libraries like Open3D, PyMeshLab, and VEDO to facilitate preprocessing, normalization, feature extraction, and querying. The pipeline included steps to clean and resample shapes, normalize them by aligning and scaling, and extract both elementary descriptors (e.g., surface area, volume) and histogram-based shape property descriptors.

While the system was functional, its performance highlighted several limitations. The retrieval process, based on custom distance measures and histogram comparison, achieved moderate accuracy, with the enhanced weighted model showing slightly better performance than simpler approaches. However, the overall retrieval precision and F1 scores remained low, indicating room for improvement in feature representation and matching algorithms.

Querying using Approximate Nearest Neighbor (ANN) techniques significantly reduced computation time but at the cost of retrieval accuracy. Although ANN was faster and scalable for larger datasets, it struggled to maintain consistency with the more accurate, distance-based approach. The dimensionality reduction using t-SNE provided limited insights, as the high-dimensional feature space proved challenging to visualize effectively.

Despite these challenges, the project successfully implemented a basic framework for content-based 3D shape retrieval, offering insights into the preprocessing, feature extraction, and querying processes.

Future work could involve refining descriptor weighting further, optimizing the ANN search, and investigating alternative dimensionality reduction methods to improve visual interpretability. Additionally, a quantitative analysis of class-wise histogram variability could provide deeper insights into feature effectiveness, guiding the system towards higher retrieval accuracy and scalability across diverse 3D shape classes by more accurately weighting the features.

References

[1] Foundation, P. S. Python programming language. `https://www.python.org/` (2023). Accessed: 2024-11-14.

[2] Harris, C. R., Millman, K. J., van der Walt, S. J. *et al. NumPy: Array processing for numbers, strings, records, and objects* (2020). URL `https://numpy.org/`. Version 1.26.3.

[3] Zhou, Q.-Y., Park, J. & Koltun, V. *Open3D: A Modern Library for 3D Data Processing* (2018). URL `http://www.open3d.org/`. Version 0.18.0.

[4] pandas development team, T. *pandas: Python Data Analysis Library* (2023). URL `https://pandas.pydata.org/`. Version 2.1.4.

[5] Cignoni, P. *et al.* *PyMeshLab: Python Binding for MeshLab* (2023). URL https://pymeshlab.readthedocs.io/. Version 2023.12.post2.

[6] Waskom, M. L. *et al.* *Seaborn: Statistical Data Visualization* (2023). URL https://seaborn.pydata.org/. Version 0.12.2.

[7] Hunter, J. D. *et al.* *Matplotlib: Python plotting* (2007). URL https://matplotlib.org/. Version 3.8.0.

[8] Pedregosa, F., Varoquaux, G. *et al.* *scikit-learn: Machine Learning in Python* (2011). URL https://scikit-learn.org/stable/. Version 1.2.2.

[9] Virtanen, P. *et al.* *SciPy: Open-source software for mathematics, science, and engineering* (2023). URL https://scipy.org/. Version 1.11.4.

[10] Grinberg, M. *Flask: Web development framework* (2010). URL https://flask.palletsprojects.com/. Version 2.2.5.

[11] Inc., P. T. *Plotly: Interactive graphing library* (2023). URL https://plotly.com/python/. Version 5.9.0.

[12] Ngrok, I. *pyngrok: Ngrok Tunnel Manager* (2023). URL https://pyngrok.com/. Version 7.2.1.

[13] Safadi, B., Derbas, N. & Quénot, G. Descriptor optimization for multimedia indexing and retrieval. *Multimedia Tools Appl.* **74**, 1267–1290 (2015). URL http://dblp.uni-trier.de/db/journals/mta/mta74.html#SafadiDQ15.

[14] Musy, M. vedo (2024). URL https://pypi.org/project/vedo/.

[15] VTK. vtkquadricdecimation: Mesh decimation using quadric error metrics. https://vtk.org/doc/nightly/html/classvtkQuadricDecimation.html.

[16] Contributors, O. Open3d: A modern library for 3d data processing (2020). URL http://www.open3d.org/.

[17] learn Developers, S. t-sne: t-distributed stochastic neighbor embedding (2024). URL https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.

[18] Rousseeuw, P. J. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* **20**, 53–65 (1987).